OpenShift Container Platform 3.4 Developer Guide

OpenShift Container Platform 3.4 Developer Reference

Red Hat OpenShift Documentation Team

OpenShift Container Platform 3.4 Developer Guide

OpenShift Container Platform 3.4 Developer Reference

Legal Notice

Abstract

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

Table of Contents

CHAPTER 1. OVERVIEW

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

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

CHAPTER 2. APPLICATION LIFE CYCLE MANAGEMENT

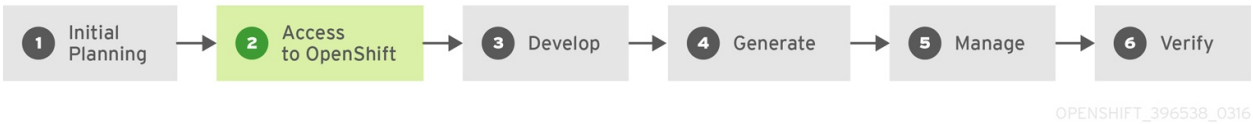
2.1. PLANNING YOUR DEVELOPMENT PROCESS

2.1.1. Overview

OpenShift Container Platform is designed for building and deploying applications. Depending on how much you want to involve OpenShift Container Platform in your development process, you can choose to:

- focus your development within a OpenShift Container Platform project, using it to build an application from scratch then continuously develop and manage its lifecycle, or
- bring an application (e.g., binary, container image, source code) you have already developed in a separate environment and deploy it onto OpenShift Container Platform.

2.1.2. Using OpenShift Container Platform as Your Development Environment



You can begin your application's development from scratch using OpenShift Container Platform directly. Consider the following steps when planning this type of development process:

Initial Planning

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

Access to OpenShift Container Platform

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

Develop

- Using your editor or IDE of choice, create a basic skeleton of an application. It should be developed enough to tell OpenShift Container Platform what kind of application it is.
- Push the code to your Git repository.

Generate

- Create a basic application using the **`oc new-app`** command. OpenShift Container Platform generates build and deployment configurations.

Manage

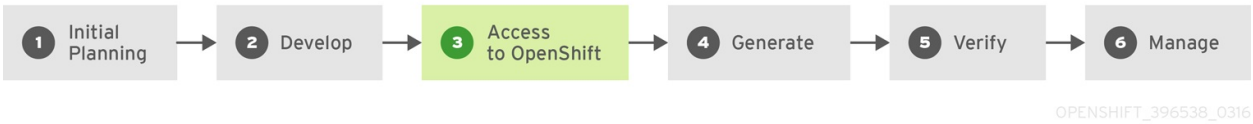
- Start developing your application code.
- Ensure your application builds successfully.
- Continue to locally develop and polish your code.

- Push your code to a Git repository.
- Is any extra configuration needed? Explore the Developer Guide for more options.

Verify

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

2.1.3. Bringing an Application to Deploy on OpenShift Container Platform



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

Initial Planning

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

Develop

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

Access to OpenShift Container Platform

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

Generate

- Create a basic application using the **`oc new-app`** command. OpenShift Container Platform generates build and deployment configurations.

Verify

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

Manage

- Continue to develop your application code until you are happy with the results.
- Rebuild your application in OpenShift Container Platform to accept any newly pushed code.
- Is any extra configuration needed? Explore the Developer Guide for more options.

2.2. CREATING NEW APPLICATIONS

2.2.1. Overview

You can create a new OpenShift Container Platform application from components including source or binary code, images and/or templates by using either the OpenShift CLI or web console.

2.2.2. Creating an Application Using the CLI

2.2.2.1. Creating an Application From Source Code

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

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

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

Note

If using a local Git repository, the repository should have a remote named **origin** that points to a URL accessible by the OpenShift Container Platform cluster. If there is no recognised remote, **new-app** will create a binary build.

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

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

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

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

The **new-app** command creates a build configuration, which itself creates a new application image from your source code. The **new-app** command typically also creates a deployment configuration to deploy the new image, and a service to provide load-balanced access to the deployment running your image.

OpenShift Container Platform automatically detects whether the **Docker** or **Source** build strategy should be used, and in the case of **Source** builds, detects an appropriate language builder image.

Build Strategy Detection

If a ***Dockerfile*** is in the repository when creating a new application, OpenShift Container Platform generates a **Docker** build strategy. Otherwise, it generates a **Source** build strategy.

You can override the build strategy by setting the **--strategy** flag to either **docker** or **source**.

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

Language Detection

If using the **Source** build strategy, **new-app** attempts to determine the language builder to use by the presence of certain files in the root or specified context directory of the repository:

Table 2.1. Languages Detected by new-app

Language	Files
dotnet	***project.json***, ****.csproj***
jee	***pom.xml***
nodejs	***app.json***, ***package.json***
perl	***cpanfile***, ***index.pl***
php	***composer.json***, ***index.php***
python	***requirements.txt***, ***setup.py***
ruby	***Gemfile***, ***Rakefile***, ***config.ru***
scala	***build.sbt***

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

You can override the image the builder uses for a particular source repository by specifying the image (either an image stream or container specification) and the repository, with a **~** as a separator. Note that if this is done, build strategy detection and language detection are not carried out.

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

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

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

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

2.2.2.2. Creating an Application From an Image

You can deploy an application from an existing image. Images can come from image streams in the OpenShift Container Platform server, images in a specific registry or Docker Hub registry, or images in the local Docker server.

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

Note

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

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

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

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

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

Note

If the registry containing the image is not secured with SSL, cluster administrators must ensure that the Docker daemon on the OpenShift Container Platform node hosts is run with the **--insecure-registry** flag pointing to that registry. You must also tell **new-app** that the image comes from an insecure registry with the **--insecure-registry** flag.

You can create an application from an existing image stream and optional image stream tag:

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

2.2.2.3. Creating an Application From a Template

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

To create an application from a stored template:

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

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

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

Template Parameters

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

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

2.2.2.4. Further Modifying Application Creation

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

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

Table 2.2. new-app Output Objects

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

Object	Description
DeploymentConfig	A **DeploymentConfig** is created either to deploy the output of a build, or a specified image. The **new-app** command creates **EmptyDir** volumes for all Docker volumes that are specified in containers included in the resulting **DeploymentConfig**.
Service	The **new-app** command attempts to detect exposed ports in input images. It uses the lowest numeric exposed port to generate a service that exposes that port. In order to expose a different port, after **new-app** has completed, simply use the **oc expose** command to generate additional services.
Other	Other objects may be generated when instantiating templates, according to the template.

2.2.2.4.1. Specifying Environment Variables

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

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

2.2.2.4.2. Specifying Labels

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

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

2.2.2.4.3. Viewing the Output Without Creation

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

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

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

```
    -o yaml > myapp.yaml
$ vi myapp.yaml
$ oc create -f myapp.yaml
```

2.2.2.4.4. Creating Objects With Different Names

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

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

2.2.2.4.5. Creating Objects in a Different Project

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

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

2.2.2.4.6. Creating Multiple Objects

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

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

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

Note

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

2.2.2.4.7. Grouping Images and Source in a Single Pod

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

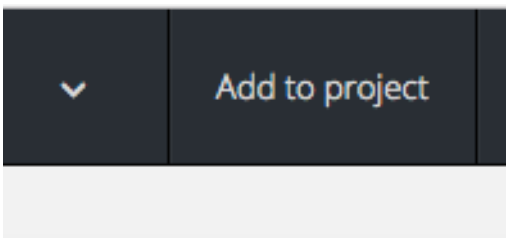
```
$ oc new-app nginx+mysql
```

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

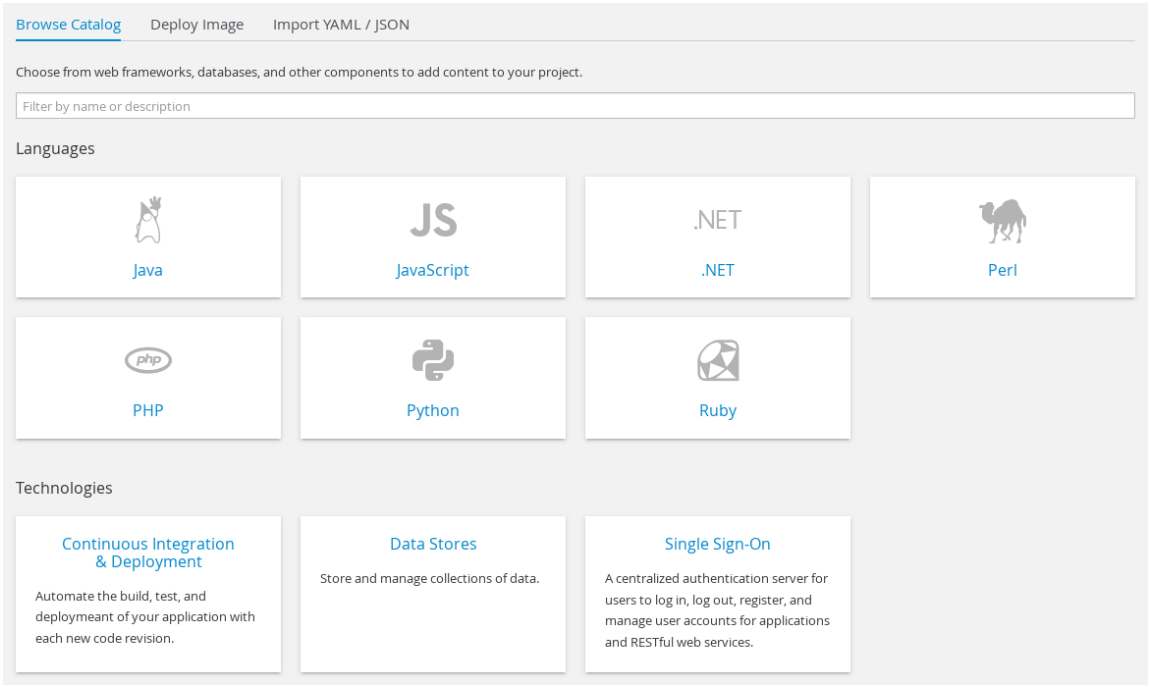
```
$ oc new-app \
    ruby~https://github.com/openshift/ruby-hello-world \
    mysql \
    --group=ruby+mysql
```

2.2.3. Creating an Application Using the Web Console

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



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



Note

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

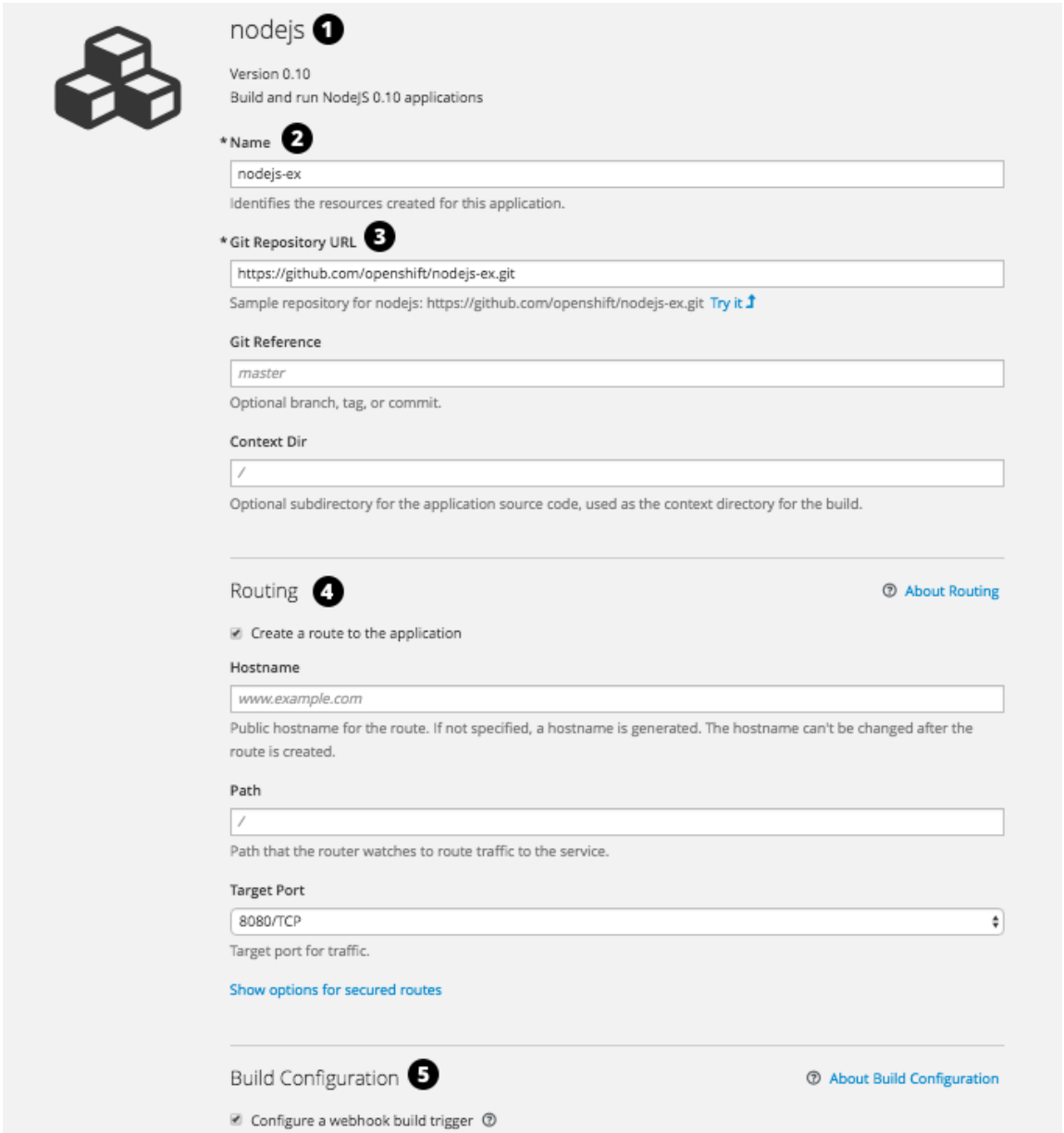
```
kind: "ImageStream"
apiVersion: "v1"
metadata:
  name: "ruby"
  creationTimestamp: null
spec:
```

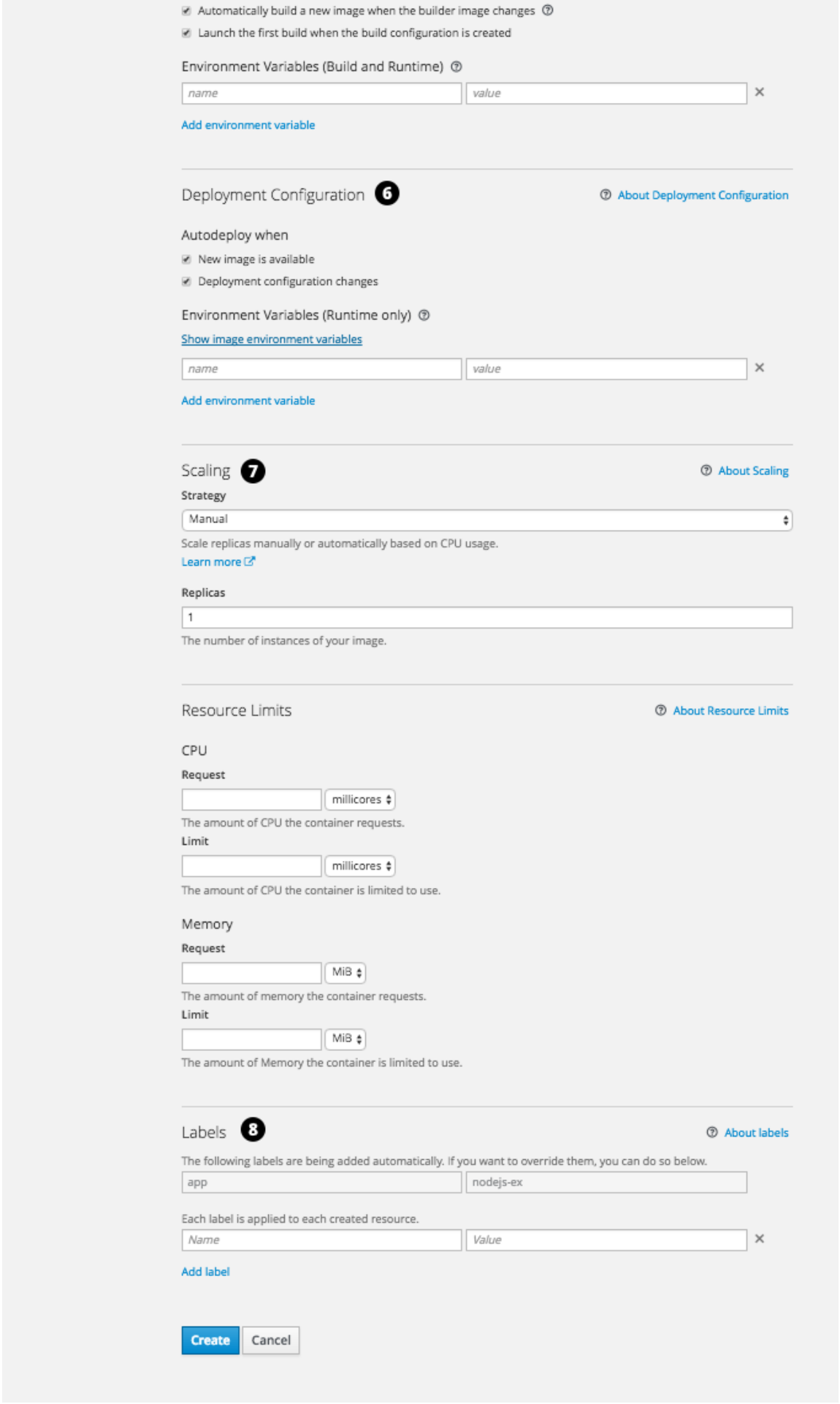
```
  dockerImageRepository:
"registry.access.redhat.com/openshift3/ruby-20-rhel7"
  tags:
    -
      name: "2.0"
      annotations:
        description: "Build and run Ruby 2.0 applications"
        iconClass: "icon-ruby"
        tags: "builder,ruby" 1
        supports: "ruby:2.0,ruby"
        version: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

1

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

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




Automatically build a new image when the builder image changes
Launch the first build when the build configuration is created
Environment Variables (Build and Runtime)
name
value
Add environment variable
Deployment Configuration 6
About Deployment Configuration
Autodeploy when
New image is available
Deployment configuration changes
Environment Variables (Runtime only)
Show image environment variables
name
value
Add environment variable
Scaling 7
About Scaling
Strategy
Manual
Scale replicas manually or automatically based on CPU usage.
Learn more
Replicas
1
The number of instances of your image.
Resource Limits
About Resource Limits
CPU
Request
millicores
The amount of CPU the container requests.
Limit
millicores
The amount of CPU the container is limited to use.
Memory
Request
MiB
The amount of memory the container requests.
Limit
MiB
The amount of Memory the container is limited to use.
Labels 8
About labels
The following labels are being added automatically. If you want to override them, you can do so below.
app
nodejs-ex
Each label is applied to each created resource.
Name
Value
Add label
Create
Cancel

	The builder image name and description.
	The application name used for the generated OpenShift Container Platform objects.
	The Git repository URL, reference, and context directory for your source code.
	Routing configuration section for making this application publicly accessible.
	Build configuration section for customizing build triggers.
	Deployment configuration section for customizing deployment triggers and image environment variables.
	Replica scaling section for configuring the number of running instances of the application.
	The labels to assign to all items generated for the application. You can add and edit labels for all objects here.

Note

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

2.3. PROMOTING APPLICATIONS ACROSS ENVIRONMENTS

2.3.1. Overview

Application promotion means moving an application through various runtime environments, typically with an increasing level of maturity. For example, an application might start out in a development environment, then be promoted to a stage environment for further testing, before finally being promoted into a production environment. As changes are introduced in the application, again the changes will start in development and be promoted through stage and production.

The "application" today is more than just the source code written in Java, Perl, Python, etc. It is more now than the static web content, the integration scripts, or the associated configuration for the language specific runtimes for the application. It is more than the application specific archives consumed by those language specific runtimes.

In the context of OpenShift Container Platform and its combined foundation of Kubernetes and Docker, additional application artifacts include:

- *Docker container images* with their rich set of metadata and associated tooling.
- *Environment variables* that are injected into containers for application use.
- *API objects* (also known as resource definitions; see Core Concepts) of OpenShift Container Platform, which:
  - are injected into containers for application use.
  - dictate how OpenShift Container Platform manages containers and pods.

In examining how to promote applications in OpenShift Container Platform, this topic will:

- Elaborate on these new artifacts introduced to the application definition.
- Describe how you can demarcate the different environments for your application promotion pipeline.
- Discuss methodologies and tools for managing these new artifacts.
- Provide examples that apply the various concepts, constructs, methodologies, and tools to application promotion.

2.3.2. Application Components

2.3.2.1. API Objects

With regard to OpenShift Container Platform and Kubernetes resource definitions (the items newly introduced to the application inventory), there are a couple of key design points for these API objects that are relevant to revisit when considering the topic of application promotion.

First, as highlighted throughout OpenShift Container Platform documentation, every API object can be expressed via either JSON or YAML, making it easy to manage these resource definitions via traditional source control and scripting.

Also, the API objects are designed such that there are portions of the object which specify the desired state of the system, and other portions which reflect the status or current state of the system. This can be thought of as inputs and outputs. The input portions, when expressed in JSON or YAML, in particular are items that fit naturally as source control managed (SCM) artifacts.

Note

Remember, the input or specification portions of the API objects can be totally static or dynamic in the sense that variable substitution via template processing is possible on instantiation.

The result of these points with respect to API objects is that with their expression as JSON or YAML files, you can treat the configuration of the application as code.

Conceivably, almost any of the API objects may be considered an application artifact by your organization. Listed below are the objects most commonly associated with deploying and managing an application:

BuildConfigs

This is a special case resource in the context of application promotion. While a **`BuildConfig`** is certainly a part of the application, especially from a developer's perspective, typically the **`BuildConfig`** is not promoted through the pipeline. It produces the **`Image`** that is promoted (along with other items) through the pipeline.

Templates

In terms of application promotion, **`Templates`** can serve as the starting point for setting up resources in a given staging environment, especially with the parameterization capabilities. Additional post-instantiation modifications are very conceivable though when applications move through a promotion pipeline. See Scenarios and Examples for more on this.

Routes

These are the most typical resources that differ stage to stage in the application promotion pipeline, as tests against different stages of an application access that application via its **`Route`**. Also, remember that you have options with regard to manual specification or auto-generation of host names, as well as the HTTP-level security of the **`Route`**.

Services

If reasons exist to avoid **`Routers`** and **`Routes`** at given application promotion stages (perhaps for simplicity's sake for individual developers at early stages), an application can be accessed via the **`Cluster`** IP address and port. If used, some management of the address and port between stages could be warranted.

Endpoints

Certain application-level services (e.g., database instances in many enterprises) may not be managed by OpenShift Container Platform. If so, then creating those **`Endpoints`** yourself, along with the necessary modifications to the associated **`Service`** (omitting the selector field on the **`Service`**) are activities that are either duplicated or shared between stages (based on how you delineate your environment).

Secrets

The sensitive information encapsulated by **`Secrets`** are shared between staging environments when the corresponding entity (either a **`Service`** managed by OpenShift Container Platform or an external service managed outside of OpenShift Container Platform) the information pertains to is shared. If there are different versions of the said entity in different stages of your application promotion pipeline, it may be necessary to maintain a distinct **`Secret`** in each stage of the pipeline or to make modifications to it as it traverses through the pipeline. Also, take care that if you are storing the **`Secret`** as JSON or YAML in an SCM, some form of encryption to protect the sensitive information may be warranted.

DeploymentConfigs

This object is the primary resource for defining and scoping the environment for a given application promotion pipeline stage; it controls how your application starts up. While there are aspects of it that will be common across all the different stage, undoubtedly there will be modifications to this object as it progresses through your application promotion pipeline to reflect differences in the environments for each stage, or changes in behavior of the system to facilitate testing of the different scenarios your application must support.

ImageStreams, ImageStreamTags, and ImageStreamImage

Detailed in the Images and Image Streams sections, these objects are central to the OpenShift Container Platform additions around managing container images.

ServiceAccounts and RoleBindings

Management of permissions to other API objects within OpenShift Container Platform, as well as the external services, are intrinsic to managing your application. Similar to **`Secrets`**, the **`ServiceAccounts`** and **`RoleBindingscan`** objects vary in how they are shared between the different stages of your application promotion pipeline based on your needs to share or isolate those different environments.

PersistentVolumeClaims

Relevant to stateful services like databases, how much these are shared between your different application promotion stages directly correlates to how your organization shares or isolates the copies of your application data.

ConfigMaps

A useful decoupling of **Pod** configuration from the **Pod** itself (think of an environment variable style configuration), these can either be shared by the various staging environments when consistent **Pod** behavior is desired. They can also be modified between stages to alter **Pod** behavior (usually as different aspects of the application are vetted at different stages).

2.3.2.2. Images

As noted earlier, container images are now artifacts of your application. In fact, of the new applications artifacts, images and the management of images are the key pieces with respect to application promotion. In some cases, an image might encapsulate the entirety of your application, and the application promotion flow consists solely of managing the image.

Images are not typically managed in a SCM system, just as application binaries were not in previous systems. However, just as with binaries, installable artifacts and corresponding repositories (that is, RPMs, RPM repositories, Nexus, etc.) arose with similar semantics to SCMs, similar constructs and terminology around image management that are similar to SCMs have arisen:

- Image registry == SCM server
- Image repository == SCM repository

As images reside in registries, application promotion is concerned with ensuring the appropriate image exists in a registry that can be accessed from the environment that needs to run the application represented by that image.

Rather than reference images directly, application definitions typically abstract the reference into an image stream. This means the image stream will be another API object that makes up the application components. For more details on image streams, see Core Concepts.

2.3.2.3. Summary

Now that the application artifacts of note, images and API objects, have been detailed in the context of application promotion within OpenShift Container Platform, the notion of *where* you run your application in the various stages of your promotion pipeline is next the point of discussion.

2.3.3. Deployment Environments

A deployment environment, in this context, describes a distinct space for an application to run during a particular stage of a CI/CD pipeline. Typical environments include **development**, **test**, **stage**, and **production**, for example. The boundaries of an environment can be defined in different ways, such as:

- Via labels and unique naming within a single project.
- Via distinct projects within a cluster.
- Via distinct clusters.

And it is conceivable that your organization leverages all three.

2.3.3.1. Considerations

Typically, you will consider the following heuristics in how you structure the deployment environments:

- How much resource sharing the various stages of your promotion flow allow
- How much isolation the various stages of your promotion flow require
- How centrally located (or geographically dispersed) the various stages of your promotion flow are

Also, some important reminders on how OpenShift Container Platform clusters and projects relate to image registries:

- Multiple project in the same cluster can access the same image streams.
- Multiple clusters can access the same external registries.
- Clusters can only share a registry if the OpenShift Container Platform internal image registry is exposed via a route.

2.3.3.2. Summary

After deployment environments are defined, promotion flows with delineation of stages within a pipeline can be implemented. The methods and tools for constructing those promotion flow implementations are the next point of discussion.

2.3.4. Methods and Tools

Fundamentally, application promotion is a process of moving the aforementioned application components from one environment to another. The following subsections outline tools that can be used to move the various components by hand, before advancing to discuss holistic solutions for automating application promotion.

Note

There are a number of insertion points available during both the build and deployment processes. They are defined within **`BuildConfig`** and **`DeploymentConfig`** API objects. These hooks allow for the invocation of custom scripts which can interact with deployed components such as databases, and with the OpenShift Container Platform cluster itself.

Therefore, it is possible to use these hooks to perform component management operations that effectively move applications between environments, for example by performing an image tag operation from within a hook. However, the various hook points are best suited to managing an application's lifecycle within a given environment (for example, using them to perform database schema migrations when a new version of the application is deployed), rather than to move application components between environments.

2.3.4.1. Managing API Objects

Resources, as defined in one environment, will be exported as JSON or YAML file content in preparation for importing it into a new environment. Therefore, the expression of API objects as JSON or YAML serves as the unit of work as you promote API objects through your application pipeline. The **`oc`** CLI is used to export and import this content.

Tip

While not required for promotion flows with OpenShift Container Platform, with the JSON or YAML stored in files, you can consider storing and retrieving the content from a SCM system. This allows you to leverage the versioning related capabilities of the SCM, including the creation of branches, and the assignment of and query on various labels or tags associated to versions.

2.3.4.1.1. Exporting API Object State

API object specifications should be captured with **`oc export`**. This operation removes environment specific data from the object definitions (e.g., current namespace or assigned IP addresses), allowing them to be recreated in different environments (unlike **`oc get`** operations, which output an unfiltered state of the object).

Use of **`oc label`**, which allows for adding, modifying, or removing labels on API objects, can prove useful as you organize the set of object collected for promotion flows, because labels allow for selection and management of groups of pods in a single operation. This makes it easier to export the correct set of objects and, because the labels will carry forward when the objects are created in a new environment, they also make for easier management of the application components in each environment.

Note

API objects often contain references such as a **`DeploymentConfig`** that references a **`Secret`**. When moving an API object from one environment to another, you must ensure that such references are also moved to the new environment.

Similarly, API objects such as a **`DeploymentConfig`** often contain references to **`ImageStreams`** that reference an external registry. When moving an API object from one environment to another, you must ensure such references are resolvable within the new environment, meaning that the reference must be resolvable and the **`ImageStream`** must reference an accessible registry in the new environment. See Moving Images and Promotion Caveats for more detail.

2.3.4.1.2. Importing API Object State

2.3.4.1.2.1. Initial Creation

The first time an application is being introduced into a new environment, it is sufficient to take the JSON or YAML expressing the specifications of your API objects and run **`oc create`** to create them in the appropriate environment. When using **`oc create`**, keep the **`--save-config`** option in mind. Saving configuration elements on the object in its annotation list facilitates the later use of **`oc apply`** to modify the object.

2.3.4.1.2.2. Iterative Modification

After the various staging environments are initially established, as promotion cycles commence and the application moves from stage to stage, the updates to your application can include modification of the API objects that are part of the application. Changes in these API objects are conceivable since they represent the configuration for the OpenShift Container Platform system. Motivations for such changes include:

- Accounting for environmental differences between staging environments.
- Verifying various scenarios your application supports.

Transfer of the API objects to the next stage's environment is accomplished via use of the **`oc`** CLI. While a rich set of **`oc`** commands which modify API objects exist, this topic focuses on **`oc apply`**, which computes and applies differences between objects.

Specifically, you can view **`oc apply`** as a three-way merge that takes in files or stdin as the input along with an existing object definition. It performs a three-way merge between:

1. the input into the command,
2. the current version of the object, and
3. the most recent user specified object definition stored as an annotation in the current object.

The existing object is then updated with the result.

If further customization of the API objects is necessary, as in the case when the objects are not expected to be identical between the source and target environments, **`oc`** commands such as **`oc set`** can be used to modify the object after applying the latest object definitions from the upstream environment.

Some specific usages are cited in Scenarios and Examples.

2.3.4.2. Managing Images and Image Streams

Images in OpenShift Container Platform are managed via a series of API objects as well. However, managing images are so central to application promotion that discussion of the tools and API objects most directly tied to images warrant separate discussion. Both manual and automated forms exist to assist you in managing image promotion (the propagation of images through your pipeline).

2.3.4.2.1. Moving Images

Note

For all the detailed caveats around managing images, refer to the Managing Images topic.

2.3.4.2.1.1. When Staging Environments Share a Registry

When your staging environments share the same OpenShift Container Platform registry, for example if they are all on the same OpenShift Container Platform cluster, there are two operations that are the basic means of *moving* your images between the stages of your application promotion pipeline:

1. First, analogous to **`docker tag`** and **`git tag`**, the **`oc tag`** command allows you to update an OpenShift Container Platform image stream with a reference to a specific image. It also allows you to copy references to specific versions of an image from one image stream to another, even across different projects in a cluster.
2. Second, the **`oc import-image`** serves as a bridge between external registries and image streams. It imports the metadata for a given image from the registry and stores it into the image stream as an image stream tag. Various **`BuildConfigs`** and **`DeploymentConfigs`** in your project can reference those specific images.

2.3.4.2.1.2. When Staging Environments Use Different Registries

More advanced usage occurs when your staging environments leverage different OpenShift Container Platform registries. Accessing the Internal Registry spells out the steps in detail, but in summary you can:

1. Use the **`docker`** command in conjunction which obtaining the OpenShift Container Platform access token to supply into your **`docker login`** command.
2. After being logged into the OpenShift Container Platform registry, use **`docker pull`**, **`docker tag`** and **`docker push`** to transfer the image.
3. After the image is available in the registry of the next environment of your pipeline, use **`oc tag`** as needed to populate any image streams.

2.3.4.2.2. Deploying

Whether changing the underlying application image or the API objects that configure the application, a deployment is typically necessary to pick up the promoted changes. If the images for your application change (for example, due to an **`oc tag`** operation or a **`docker push`** as part of

promoting an image from an upstream environment), **`ImageChangeTriggers`** on your **`DeploymentConfig`** can trigger the new deployment. Similarly, if the **`DeploymentConfig`** API object itself is being changed, a **`ConfigChangeTrigger`** can initiate a deployment when the API object is updated by the promotion step (for example, **`oc apply`**).

Otherwise, the **`oc`** commands that facilitate manual deployment include:

- **`oc deploy`**: The original method to view, start, cancel, or retry deployments.
- **`oc rollout`**: The new approach to manage deployments, including pause and resume semantics and richer features around managing history.
- **`oc rollback`**: Allows for reversion to a previous deployment; in the promotion scenario, if testing of a new version encounters issues, confirming it still works with the previous version could be warranted.

2.3.4.2.3. Automating Promotion Flows with Jenkins

After you understand the components of your application that need to be moved between environments when promoting it and the steps required to move the components, you can start to orchestrate and automate the workflow. OpenShift Container Platform provides a Jenkins image and plug-ins to help with this process.

The OpenShift Container Platform Jenkins image is detailed in Using Images, including the set of OpenShift Container Platform-centric plug-ins that facilitate the integration of Jenkins, and Jenkins Pipelines. Also, the Pipeline build strategy facilitates the integration between Jenkins Pipelines and OpenShift Container Platform. All of these focus on enabling various aspects of CI/CD, including application promotion.

When moving beyond manual execution of application promotion steps, the Jenkins-related features provided by OpenShift Container Platform should be kept in mind:

- OpenShift Container Platform provides a Jenkins image that is heavily customized to greatly ease deployment in an OpenShift Container Platform cluster.
- The Jenkins image contains the OpenShift Pipeline plug-in, which provides building blocks for implementing promotion workflows. These building blocks include the triggering of Jenkins jobs as image streams change, as well as the triggering of builds and deployments within those jobs.
- **`BuildConfigs`** employing the OpenShift Container Platform Jenkins Pipeline build strategy enable execution of Jenkinsfile-based Jenkins Pipeline jobs. Pipeline jobs are the strategic direction within Jenkins for complex promotion flows and can leverage the steps provided by the OpenShift Pipeline Plug-in.

2.3.4.2.4. Promotion Caveats

2.3.4.2.4.1. API Object References

API objects can reference other objects. A common use for this is to have a **`DeploymentConfig`** that references an image stream, but other reference relationships may also exist.

When copying an API object from one environment to another, it is critical that all references can still be resolved in the target environment. There are a few reference scenarios to consider:

- The reference is "local" to the project. In this case, the referenced object resides in the same project as the object that references it. Typically the correct thing to do is to ensure that you copy the referenced object into the target environment in the same project as the object referencing it.

- The reference is to an object in another project. This is typical when an image stream in a shared project is used by multiple application projects (see Managing Images). In this case, when copying the referencing object to the new environment, you must update the reference as needed so it can be resolved in the target environment. That may mean:
  - Changing the project the reference points to, if the shared project has a different name in the target environment.
  - Moving the referenced object from the shared project into the local project in the target environment and updating the reference to point to the local project when moving the primary object into the target environment.
  - Some other combination of copying the referenced object into the target environment and updating references to it.

In general, the guidance is to consider objects referenced by the objects being copied to a new environment and ensure the references are resolvable in the target environment. If not, take appropriate action to fix the references and make the referenced objects available in the target environment.

2.3.4.2.4.2. Image Registry References

Image streams point to image repositories to indicate the source of the image they represent. When an image stream is moved from one environment to another, it is important to consider whether the registry and repository reference should also change:

- If different image registries are used to assert isolation between a test environment and a production environment.
- If different image repositories are used to separate test and production-ready images.

If either of these are the case, the image stream must be modified when it is copied from the source environment to the target environment so that it resolves to the correct image. This is in addition to performing the steps described in Scenarios and Examples to copy the image from one registry and repository to another.

2.3.4.3. Summary

At this point, the following have been defined:

- New application artifacts that make up a deployed application.
- Correlation of application promotion activities to tools and concepts provided by OpenShift Container Platform.
- Integration between OpenShift Container Platform and the CI/CD pipeline engine Jenkins.

Putting together examples of application promotion flows within OpenShift Container Platform is the final step for this topic.

2.3.5. Scenarios and Examples

Having defined the new application artifact components introduced by the Docker, Kubernetes, and OpenShift Container Platform ecosystems, this section covers how to promote those components between environments using the mechanisms and tools provided by OpenShift Container Platform.

Of the components making up an application, the image is the primary artifact of note. Taking that

premise and extending it to application promotion, the core, fundamental application promotion pattern is image promotion, where the unit of work is the image. The vast majority of application promotion scenarios entails management and propagation of the image through the promotion pipeline.

Simpler scenarios solely deal with managing and propagating the image through the pipeline. As the promotion scenarios broaden in scope, the other application artifacts, most notably the API objects, are included in the inventory of items managed and propagated through the pipeline.

This topic lays out some specific examples around promoting images as well as API objects, using both manual and automated approaches. But first, note the following on setting up the environment(s) for your application promotion pipeline.

2.3.5.1. Setting up for Promotion

After you have completed development of the initial revision of your application, the next logical step is to package up the contents of the application so that you can transfer to the subsequent staging environments of your promotion pipeline.

1. First, group all the API objects you view as transferable and apply a common **`label`** to them:

   ```
   labels:
     promotion-group: <application_name>
   ```

   As previously described, the **`oc label`** command facilitates the management of labels with your various API objects.

   Tip

   If you initially define your API objects in a OpenShift Container Platform template, you can easily ensure all related objects have the common label you will use to query on when exporting in preparation for a promotion.

2. You can leverage that label on subsequent queries. For example, consider the following set of **oc** command invocations that would then achieve the transfer of your application's API objects:

   ```
   $ oc login <source_environment>
   $ oc project <source_project>
   $ oc export dc,is,svc,route,secret,sa -l promotion-group=
   <application_name> -o yaml > export.yaml
   $ oc login <target_environment>
   $ oc new-project <target_project> 1
   $ oc create -f export.yaml
   ```

   1

   Alternatively, **`oc project <target_project>`** if it already exists.

Note

On the **`oc export`** command, whether or not you include the **`is`** type for image streams depends on how you choose to manage images, image streams, and registries across the different environments in your pipeline. The caveats around this are discussed below. See also the Managing Images topic.

3. You must also get any tokens necessary to operate against each registry used in the different staging environments in your promotion pipeline. For each environment:

    a. Log in to the environment:

    ```
    $ oc login <each_environment_with_a_unique_registry>
    ```

    b. Get the access token with:

    ```
    $ oc whoami -t
    ```

    c. Copy and paste the token value for later use.

2.3.5.2. Repeatable Promotion Process

After the initial setup of the different staging environments for your pipeline, a set of repeatable steps to validate each iteration of your application through the promotion pipeline can commence. These basic steps are taken each time the image or API objects in the source environment are changed:

Move updated images → Move updated API objects → Apply environment specific customizations

1. Typically, the first step is promoting any updates to the image(s) associated with your application to the next stage in the pipeline. As noted above, the key differentiator in promoting images is whether the OpenShift Container Platform registry is shared or not between staging environments.

    a. If the registry is shared, simply leverage **`oc tag`**:

    ```
    $ oc tag
    <project_for_stage_N>/<imagestream_name_for_stage_N>:
    <tag_for_stage_N>
    <project_for_stage_N+1>/<imagestream_name_for_stage_N+1>:
    <tag_for_stage_N+1>
    ```

    b. If the registry is not shared, you can leverage the access tokens for each of your promotion pipeline registries as you log into both the source and destination registries, pulling, tagging, and pushing your application images accordingly:

        i. Log in to the source environment registry:

        ```
        $ docker login -u <username> -e <any_email_address> -
        p <token_value> <src_env_registry_ip>:<port>
        ```

        ii. Pull your application's image:

```
$ docker pull <src_env_registry_ip>:
<port>/<namespace>/<image name>:<tag>
```

iii. Tag your application's image to the destination registry's location, updating namespace, name, and tag as needed to conform to the destination staging environment:

```
$ docker tag <src_env_registry_ip>:
<port>/<namespace>/<image name>:<tag>
<dest_env_registry_ip>:<port>/<namespace>/<image
name>:<tag>
```

iv. Log into the destination staging environment registry:

```
$ docker login -u <username> -e <any_email_address> -
p <token_value> <dest_env_registry_ip>:<port>
```

v. Push the image to its destination:

```
$ docker push <dest_env_registry_ip>:
<port>/<namespace>/<image name>:<tag>
```

Tip

To automatically import new versions of an image from an external registry, the **`oc tag`** command has a **`--scheduled`** option. If used, the image the **`ImageStreamTag`** references will be periodically pulled from the registry hosting the image.

2. Next, there are the cases where the evolution of your application necessitates fundamental changes to your API objects or additions and deletions from the set of API objects that make up the application. When such evolution in your application's API objects occurs, the OpenShift Container Platform CLI provides a broad range of options to transfer to changes from one staging environment to the next.

   a. Start in the same fashion as you did when you initially set up your promotion pipeline:

```
$ oc login <source_environment>
$ oc project <source_project>
$ oc export dc,is,svc,route,secret,sa -l template=
<application_template_name> -o yaml > export.yaml
$ oc login <target_environment>
$ oc <target_project>
```

   b. Rather than simply creating the resources in the new environment, update them. You can do this a few different ways:

      i. The more conservative approach is to leverage **`oc apply`** and merge the new changes to each API object in the target environment. In doing so, you can **`--dry-run=true`** option and examine the resulting objects prior to actually changing the objects:

```
$ oc apply -f export.yaml --dry-run=true
```

If satisfied, actually run the **apply** command:

```
$ oc apply -f export.yaml
```

The **apply** command optionally takes additional arguments that help with more complicated scenarios. See **oc apply --help** for more details.

ii. Alternatively, the simpler but more aggressive approach is to leverage **oc replace** (which is also synonymous with **oc update**). There is no dry run with this update and replace. In the most basic form, this involves executing:

```
$ oc replace -f export.yaml
```

As with **apply**, **replace** optionally takes additional arguments for more sophisticated behavior. See **oc replace --help** for more details.

3. The previous steps automatically handle new API objects that were introduced, but if API objects were deleted from the source environment, they must be manually deleted from the target environment using **oc delete**.

4. Tuning of the environment variables cited on any of the API objects may be necessary as the desired values for those may differ between staging environments. For this, use **oc set env**:

```
$ oc set env <api_object_type>/<api_object_ID> <env_var_name>=
<env_var_value>
```

5. Finally, trigger a new deployment of the updated application using the **oc rollout** command or one of the other mechanisms discussed in the Deployments section above.

2.3.5.3. Repeatable Promotion Process Using Jenkins

The OpenShift Sample job defined in the Jenkins Docker Image for OpenShift Container Platform is an example of image promotion within OpenShift Container Platform within the constructs of Jenkins. Setup for this sample is located in the OpenShift Origin source repository.

This sample includes:

- Use of **Jenkins as the CI/CD engine**.
- Use of the **OpenShift Pipeline plug-in for Jenkins**. This plug-in provides a subset of the functionality provided by the **oc** CLI for OpenShift Container Platform packaged as Jenkins Freestyle and DSL Job steps. Note that the **oc** binary is also included in the Jenkins Docker Image for OpenShift Container Platform, and can also be used to interact with OpenShift Container Platform in Jenkins jobs.
- The OpenShift Container Platform-provided **templates for Jenkins**. There is a template for both ephemeral and persistent storage.

- A **sample application**: defined in the OpenShift Origin source repository, this application leverages **`ImageStreams`**, **`ImageChangeTriggers`**, **`ImageStreamTags`**, **`BuildConfigs`**, and separate **`DeploymentConfigs`** and **`Services`** corresponding to different stages in the promotion pipeline.

The following examines the various pieces of the OpenShift Sample job in more detail:

1. The first step is the equivalent of an **`oc scale dc frontend --replicas=0`** call. This step is intended to bring down any previous versions of the application image that may be running.
2. The second step is the equivalent of an **`oc start-build frontend`** call.
3. The third step is the equivalent of an **`oc deploy frontend --latest`** or **`oc rollout latest dc/frontend`** call.
4. The fourth step is the "test" for this sample. It ensures that the associated service for this application is in fact accessible from a network perspective. Under the covers, a socket connection is attempted against the IP address and port associated with the OpenShift Container Platform service. Of course, additional tests can be added (if not via OpenShift Pipepline plug-in steps, then via use of the Jenkins Shell step to leverage OS-level commands and scripts to test your application).
5. The fifth step commences under that assumption that the testing of your application passed and hence intends to mark the image as "ready". In this step, a new **prod** tag is created for the application image off of the **latest** image. With the **frontend `DeploymentConfig`** having an **`ImageChangeTrigger`** defined for that tag, the corresponding "production" deployment is launched.
6. The sixth and last step is a verification step, where the plug-in confirms that OpenShift Container Platform launched the desired number of replicas for the "production" deployment.

CHAPTER 3. AUTHENTICATION

3.1. WEB CONSOLE AUTHENTICATION

When accessing the web console from a browser at **<master_public_addr>:8443**, you are automatically redirected to a login page.

Review the browser versions and operating systems that can be used to access the web console.

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

3.2. CLI AUTHENTICATION

You can authenticate from the command line using the CLI command **`oc login`**. You can get started with the CLI by running this command without any options:

```
$ oc login
```

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

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

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

The following table describes these common options:

Table 3.1. Common CLI Configuration Options

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

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

CLI configuration files allow you to easily manage multiple CLI profiles.

Note

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

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

CHAPTER 4. PROJECTS

4.1. OVERVIEW

A project allows a community of users to organize and manage their content in isolation from other communities.

4.2. CREATING A PROJECT

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

To create a new project using the CLI:

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

For example:

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

Note

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

4.3. VIEWING PROJECTS

When viewing projects, you are restricted to seeing only the projects you have access to view based on the authorization policy.

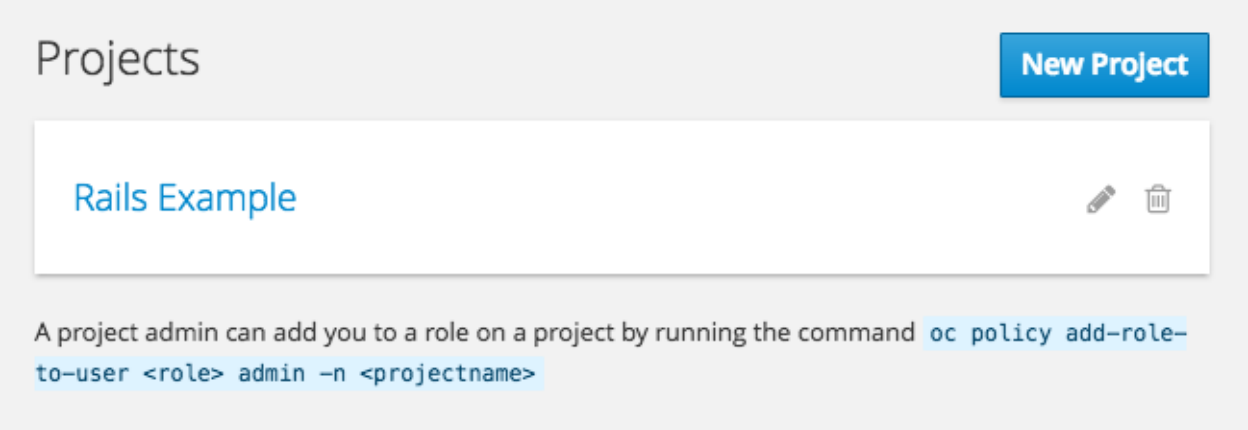
To view a list of projects:

```
$ oc get projects
```

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

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

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



If you use the CLI to create a new project, you can then refresh the page in the browser to see the new project.

Selecting a project brings you to the project overview for that project.

4.4. CHECKING PROJECT STATUS

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

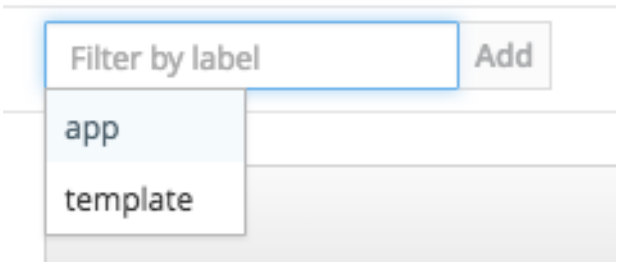
```
$ oc status
```

4.5. FILTERING BY LABELS

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

To filter by labels:

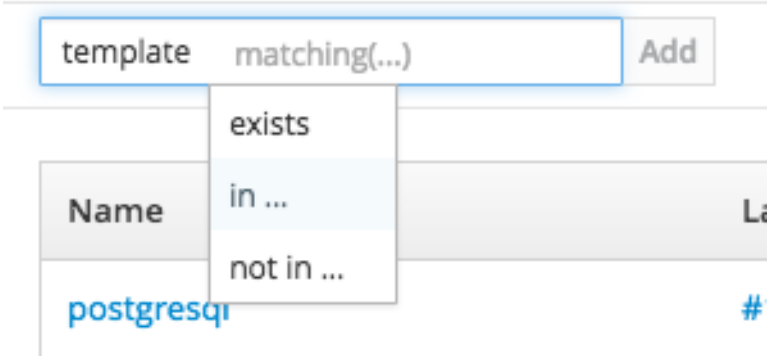
1. Select a label type:



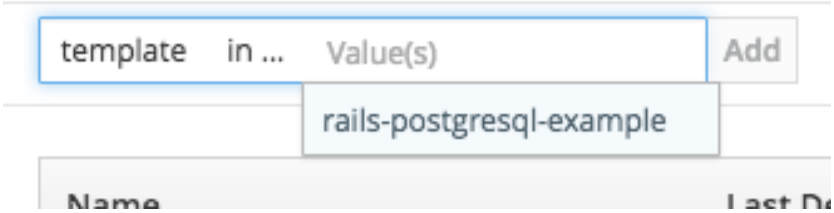
2. Select one of the following:

exists	Verify that the label name exists, but ignore its value.

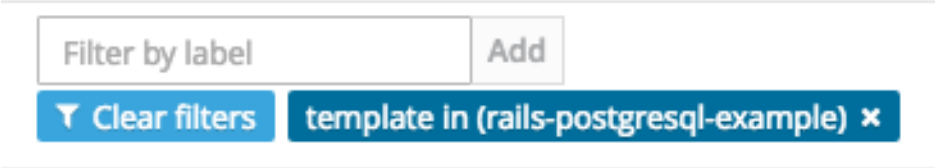
in	Verify that the label name exists and is equal to one of the selected values.
not in	Verify that the label name does not exist, or is not equal to any of the selected values.



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



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



4.6. BOOKMARKING PAGE STATES

The OpenShift Container Platform web console now bookmarks page states, which is helpful in saving label filters and other settings.

When you do something to change the page's state, like switching between tabs, the URL in the browser's navigation bar is automatically updated.

4.7. DELETING A PROJECT

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

To delete a project using the CLI:

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

CHAPTER 5. MIGRATING APPLICATIONS

5.1. OVERVIEW

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

Note

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

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

The general migration procedure is:

1. Back up the v2 application.
   - Web cartridge: The source code can be backed up to a Git repository such as by pushing to a repository on GitHub.
   - Database cartridge: The database can be backed up using a dump command (**`mongodump`**, **`mysqldump`**, **`pg_dump`**) to back up the database.
   - Web and database cartridges: **`rhc`** client tool provides snapshot ability to back up multiple cartridges:

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

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

5.2. MIGRATING DATABASE APPLICATIONS

5.2.1. Overview

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

5.2.2. Supported Databases

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

5.2.3. MySQL

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

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

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

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

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

```
$ oc new-app mysql-persistent -p \
   MYSQL_USER=<your_V2_mysql_username> -p \
   MYSQL_PASSWORD=<your_v2_mysql_password> -p MYSQL_DATABASE=
<your_v2_database_name>
```

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

```
$ oc get pods
```

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

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

6. Restore the database in the v3 running pod:

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

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

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

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

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

7. Remove the dump directory from the pod:

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

Supported MySQL Environment Variables

v2	v3
OPENSHIFT_MYSQL_DB_HOST	**[service_name]_SERVICE_HOST**
OPENSHIFT_MYSQL_DB_PORT	**[service_name]_SERVICE_PORT**
OPENSHIFT_MYSQL_DB_USERNAME	**MYSQL_USER**
OPENSHIFT_MYSQL_DB_PASSWORD	**MYSQL_PASSWORD**
OPENSHIFT_MYSQL_DB_URL	
OPENSHIFT_MYSQL_DB_LOG_DIR	
OPENSHIFT_MYSQL_VERSION	

v2	v3
OPENSHIFT_MYSQL_DIR	
OPENSHIFT_MYSQL_DB_SOCKET	
OPENSHIFT_MYSQL_IDENT	
OPENSHIFT_MYSQL_AIO	MYSQL_AIO
OPENSHIFT_MYSQL_MAX_ALLOWED_PACKET	MYSQL_MAX_ALLOWED_PACKET
OPENSHIFT_MYSQL_TABLE_OPEN_CACHE	MYSQL_TABLE_OPEN_CACHE
OPENSHIFT_MYSQL_SORT_BUFFER_SIZE	MYSQL_SORT_BUFFER_SIZE
OPENSHIFT_MYSQL_LOWER_CASE_TABLE_NAMES	MYSQL_LOWER_CASE_TABLE_NAMES
OPENSHIFT_MYSQL_MAX_CONNECTIONS	MYSQL_MAX_CONNECTIONS
OPENSHIFT_MYSQL_FT_MIN_WORD_LEN	MYSQL_FT_MIN_WORD_LEN
OPENSHIFT_MYSQL_FT_MAX_WORD_LEN	MYSQL_FT_MAX_WORD_LEN
OPENSHIFT_MYSQL_DEFAULT_STORAGE_ENGINE	
OPENSHIFT_MYSQL_TIMEZONE	
	MYSQL_DATABASE

v2	v3
	MYSQL_ROOT_PASSWORD
	MYSQL_MASTER_USER
	MYSQL_MASTER_PASSWORD

5.2.4. PostgreSQL

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

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

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

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

Note

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

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

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

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

```
$ oc get pods
```

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

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

6. Remotely access the pod:

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

7. Restore the database:

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

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

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

Supported PostgreSQL Environment Variables

v2	v3
OPENSHIFT_POSTGRESQL_DB_HOST	**[service_name]_SERVICE_HOST**
OPENSHIFT_POSTGRESQL_DB_PORT	**[service_name]_SERVICE_PORT**
OPENSHIFT_POSTGRESQL_DB_USERNAME	**POSTGRESQL_USER**
OPENSHIFT_POSTGRESQL_DB_PASSWORD	**POSTGRESQL_PASSWORD**
OPENSHIFT_POSTGRESQL_DB_LOG_DIR	
OPENSHIFT_POSTGRESQL_DB_PID	
OPENSHIFT_POSTGRESQL_DB_SOCKET_DIR	
OPENSHIFT_POSTGRESQL_DB_URL	
OPENSHIFT_POSTGRESQL_VERSION	
OPENSHIFT_POSTGRESQL_SHARED_BUFFERS	
OPENSHIFT_POSTGRESQL_MAX_CONNECTIONS	

v2	v3
OPENSHIFT_POSTGRESQL_MAX_PREPARED_ TRANSACTIONS	
OPENSHIFT_POSTGRESQL_DATESTYLE	
OPENSHIFT_POSTGRESQL_LOCALE	
OPENSHIFT_POSTGRESQL_CONFIG	
OPENSHIFT_POSTGRESQL_SSL_ENABLED	
	POSTGRESQL_DATABASE
	POSTGRESQL_ADMIN_PASSWORD

5.2.5. MongoDB

Note

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7. Check if the database is restored:

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

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

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

Supported MongoDB Environment Variables

v2	v3
OPENSHIFT_MONGODB_DB_HOST	[service_name]_SERVICE_HOST
OPENSHIFT_MONGODB_DB_PORT	[service_name]_SERVICE_PORT
OPENSHIFT_MONGODB_DB_USERNAME	MONGODB_USER
OPENSHIFT_MONGODB_DB_PASSWORD	MONGODB_PASSWORD
OPENSHIFT_MONGODB_DB_URL	
OPENSHIFT_MONGODB_DB_LOG_DIR	
	MONGODB_DATABASE
	MONGODB_ADMIN_PASSWORD
	MONGODB_NOPREALLOC
	MONGODB_SMALLFILES
	MONGODB_QUIET
	MONGODB_REPLICA_NAME
	MONGODB_KEYFILE_VALUE

5.3. MIGRATING WEB FRAMEWORK APPLICATIONS

5.3.1. Overview

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

5.3.2. Python

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

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

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

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

3. Ensure that all important files such as ***setup.py***, ***wsgi.py***, ***requirements.txt***, and ***etc*** are pushed to new repository.
   - Ensure all required packages for your application are included in ***requirements.txt***.
   - S2I-python does not support **mod_wsgi** anymore. However, **gunicorn** is supported and it is an alternative for **mod_wsgi**. So, add **gunicorn** package to ***requirements.txt***.
4. Use the **oc** command to launch a new Python application from the builder image and source code:

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

Supported Python Versions

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

5.3.3. Ruby

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

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

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

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

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

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

Note

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

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

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

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

Supported Ruby Versions

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

5.3.4. PHP

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

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

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

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

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

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

Supported PHP Versions

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

5.3.5. Perl

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

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

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

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

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

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

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

Edit cpanfile, as it has a slightly different format:

format of cpanfile	format of cpan.txt
requires ‘cpan::mod’;	cpan::mod
requires ‘Dancer’;	Dancer
requires ‘YAML’;	YAML

b. Remove ***.openshift*** directory

Note

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

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

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

Supported Perl Versions

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

5.3.6. Node.js

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

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

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

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

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

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

    Note

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

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

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

    Note

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

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

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

Supported Node.js Versions

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

5.3.7. JBoss EAP

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

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

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

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

3. If the repository includes pre-built ***.war*** files, they need to reside in the ***deployments*** directory off the root directory of the repository.

4. Create the new application using the JBoss EAP 6 builder image (jboss-eap64-openshift) and the source code repository from GitHub:

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

5.3.8. JBoss WS (Tomcat)

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

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

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

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

3. If the repository includes pre-built ***.war*** files, they need to reside in the ***deployments*** directory off the root directory of the repository.

4. Create the new application using the JBoss Web Server 3 (Tomcat 7) builder image (jboss-webserver30-tomcat7) and the source code repository from GitHub:

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

5.3.9. JBoss AS (Wildfly 10)

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

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

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

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

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

    a. Remove ***.openshift*** directory.

Note

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

b. Add the ***deployments*** directory to the root of the source repository. Move the ***.war*** files to 'deployments' directory.

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

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

Note

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

5.3.10. Supported JBoss/XPaas Versions

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

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

5.4. QUICKSTART EXAMPLES

5.4.1. Overview

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

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

- template: cakephp-mysql-example

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

5.4.2. Workflow

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

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

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

3. Commit and push your source code to GitHub:

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

4. Create a new application in v3:

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

1

Only applicable for MongoDB.

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

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

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

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

5.5. CONTINUOUS INTEGRATION AND DEPLOYMENT (CI/CD)

5.5.1. Overview

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

5.5.2. Jenkins

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

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

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

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

5.6. WEBHOOKS AND ACTION HOOKS

5.6.1. Overview

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

5.6.2. Webhooks

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

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

   This will output a webhook GitHub URL that looks like:

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

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

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

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

Note

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

5.6.3. Action Hooks

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

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

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

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

or:

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

Also, environment variables can be added or changed using:

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

5.7. S2I TOOL

5.7.1. Overview

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

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

5.7.2. Creating a Container Image

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

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

   Note

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

3. Test the built image with the Docker daemon:

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

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

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

5.8. SUPPORT GUIDE

5.8.1. Overview

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

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

5.8.2. Supported Databases

See the Supported Databases section of the Database Applications topic.

5.8.3. Supported Languages

- PHP
- Python
- Perl
- Node.js
- Ruby
- JBoss/xPaaS

5.8.4. Supported Frameworks

Table 5.1. Supported Frameworks

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

5.8.5. Supported Markers

Table 5.2. Python

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

Table 5.3. Ruby

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

Table 5.4. Perl

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

Table 5.5. PHP

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

Table 5.6. Node.js

v2	v3
NODEJS_VERSION	N/A

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

Table 5.7. JBoss EAP, JBoss WS, WildFly

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

Table 5.8. Jenkins

v2	v3
enable_debugging	N/A

Table 5.9. All

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

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

5.8.6. Supported Environment Variables

- MySQL
- MongoDB
- PostgreSQL

CHAPTER 6. APPLICATION TUTORIALS

6.1. OVERVIEW

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

6.2. QUICKSTART TEMPLATES

6.2.1. Overview

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

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

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

6.2.2. Web Framework Quickstart Templates

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

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

- Rails: a Ruby web framework (includes a PostgreSQL database)
  - Template definition
  - Source repository

6.3. RUBY ON RAILS

6.3.1. Overview

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

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

For this guide you will need:

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

6.3.2. Local Workstation Setup

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

6.3.2.1. Setting Up the Database

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

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

Next you need to initialize the database with:

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

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

Start the database by typing:

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

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

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

Note that the user we created has no password.

6.3.3. Writing Your Application

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

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

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

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

Then change into your new application directory.

```
$ cd rails-app
```

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

```
gem 'pg'
```

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

```
$ bundle install
```

In addition to using the **postgresql** database with the **pg** gem, you'll also need to ensure the **config/database.yml** is using the **postgresql** adapter.

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

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

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

```
$ rake db:create
```

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

6.3.3.1. Creating a Welcome Page

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

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

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

Rails offers a generator that will do all this necessary steps for you.

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

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

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

Run the rails server to verify the page is available.

```
$ rails server
```

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

6.3.3.2. Configuring the Application for OpenShift Container Platform

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

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

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

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

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

6.3.3.3. Storing Your Application in Git

OpenShift Container Platform requires git, if you don't have it installed you will need to install it.

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

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

```
$ ls -1
app
bin
config
config.ru
db
Gemfile
Gemfile.lock
lib
log
public
Rakefile
README.rdoc
test
tmp
vendor
```

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

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

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

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

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

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

```
$ git push
```

6.3.4. Deploying Your Application to OpenShift Container Platform

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

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

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

Deploying your application in OpenShift Container Platform involves three steps:

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

6.3.4.1. Creating the Database Service

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

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

- POSTGRESQL_DATABASE
- POSTGRESQL_USER
- POSTGRESQL_PASSWORD

Setting these variables ensures:

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

For example:

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

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

```
-e POSTGRESQL_ADMIN_PASSWORD=admin_pw
```

To watch the progress of this command:

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

6.3.4.2. Creating the Frontend Service

To bring your application to OpenShift Container Platform, you need to specify a repository in which your application lives, using once again the oc new-app command, in which you will need to specify database related environment variables we setup in the Creating the Database Service:

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

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

You can verify the environment variables have been added by viewing the JSON document of the **rails-app** DeploymentConfig:

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

You should see the following section:

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

To check the build process, use the build-logs command:

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

Once the build is complete, you can look at the running pods in OpenShift Container Platform:

```
$ oc get pods
```

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

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

- Manually from the running frontend container:

First you need to exec into frontend container with rsh command:

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

Run the migration from inside the container:

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

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

- By adding pre-deployment lifecycle hooks in your template. For example check the hooks example in our Rails example application.

6.3.4.3. Creating a Route for Your Application

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

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

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

- Routes
- Configuring a Highly-available Routing Service

6.4. SETTING UP A NEXUS MIRROR FOR MAVEN

6.4.1. Introduction

While developing your application with Java and Maven, you will most likely be building many times. In order to shorten the build times of your pods, Maven dependencies can be cached in a local Nexus repository. This tutorial will guide you through creating a Nexus repository on your cluster.

This tutorial assumes that you are working with a project that is already set up for use with Maven. If you are interested in using Maven with your Java project, it is highly recommended that you look at their guide.

In addition, be sure to check your application's image for Maven mirror capabilities. Many images that use Maven have a **`MAVEN_MIRROR_URL`** environment variable that you can use to simplify this process. If it does not have this capability, read the Nexus documentation to configure your build properly.

Furthermore, make sure that you give each pod enough resources to function. You may have to edit the pod template in the Nexus deployment configuration to request more resources.

6.4.2. Setting up Nexus

1. Download and deploy the official Nexus container image:

   ```
   oc new-app sonatype/nexus
   ```

2. Create a route by exposing the newly created Nexus service:

```
oc expose svc/nexus
```

3. Use **oc get routes** to find the pod's new external address.

```
oc get routes
```

The output should resemble:

```
NAME      HOST/PORT                                   PATH
SERVICES   PORT       TERMINATION
nexus     nexus-myproject.192.168.1.173.xip.io             nexus
8081-tcp
```

4. Confirm that Nexus is running by navigating your browser to the URL under **HOST/PORT**. To sign in to Nexus, the default administrator username is **admin**, and the password is **admin123**.

Note

Nexus comes pre-configured for the Central Repository, but you may need others for your application. For many Red Hat images, it is recommended to add the **jboss-ga** repository at Maven repository.

6.4.2.1. Using Probes to Check for Success

This is a good time to set up readiness and liveness probes. These will periodically check to see that Nexus is running properly.

```
$ oc set probe dc/nexus \
 --liveness \
 --failure-threshold 3 \
 --initial-delay-seconds 30 \
 -- echo ok
$ oc set probe dc/nexus \
 --readiness \
 --failure-threshold 3 \
 --initial-delay-seconds 30 \
 --get-url=http://:8081/nexus/content/groups/public
```

6.4.2.2. Adding Persistence to Nexus

Note

If you do not want persistent storage, continue to Connecting to Nexus. However, your cached dependencies and any configuration customization will be lost if the pod is restarted for any reason.

Create a persistent volume claim (PVC) for Nexus, so that the cached dependencies are not lost when the pod running the server terminates. PVCs require available persistent volumes (PV) in the cluster. If there are no PVs available and you do not have administrator access on your cluster, ask

your system administrator to create a Read/Write Persistent Volume for you. Otherwise, see Persistent Storage in OpenShift Container Platform for instructions on creating a persistent volume.

Add a PVC to the Nexus deployment configuration.

```
$ oc volumes dc/nexus --add \
 --name 'nexus-volume-1' \
 --type 'pvc' \
 --mount-path '/sonatype-work/' \
 --claim-name 'nexus-pv' \
 --claim-size '1G' \
 --overwrite
```

This removes the previous **emptyDir** volume for the deployment config and adds a claim for one gigabyte of persistent storage mounted at **`/sonatype-work`**, which is where the dependencies will be stored. Due to the change in configuration, the Nexus pod will be redeployed automatically.

To verify that Nexus is running, refresh the Nexus page in your browser. You can monitor the deployment's progress using:

```
$ oc get pods -w
```

6.4.3. Connecting to Nexus

The next steps demonstrate defining a build that uses the new Nexus repository. The rest of the tutorial uses this example repository with **wildfly-100-centos7** as a builder, but these changes should work for any project.

The example builder image supports **`MAVEN_MIRROR_URL`** as part of its environment, so we can use this to point our builder image to our Nexus repository. If your image does not support consuming an environment variable to configure a Maven mirror, you may need to modify the builder image to provide the correct Maven settings to point to the Nexus mirror.

```
$ oc new-build openshift/wildfly-100-
centos7:latest~https://github.com/openshift/jee-ex.git \
 -e MAVEN_MIRROR_URL='http://nexus.
<Nexus_Project>:8081/nexus/content/groups/public'
$ oc logs build/jee-ex-1 --follow
```

Replace **`<Nexus_Project>`** with the project name of the Nexus repository. If it is in the same project as the application that is using it, you can remove the **`<Nexus_Project>.`**. Learn more about DNS resolution in OpenShift Container Platform.

6.4.4. Confirming Success

In your web browser, navigate to ***http://<Nexus IP>:8081/nexus/content/groups/public*** to confirm that it has stored your application's dependencies. You can also check the build logs to see if Maven is using the Nexus mirror. If successful, you should see output referencing the URL ***http://nexus:8081***.

6.4.5. Additional Resources

- Managing Volumes in OpenShift Container Platform

- Improving Build Time of Java Builds on OpenShift Container Platform
- Nexus Repository Documentation

CHAPTER 7. OPENING A REMOTE SHELL TO CONTAINERS

7.1. OVERVIEW

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

7.2. START A SECURE SHELL SESSION

Open a remote shell session to a container:

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

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

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

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

Note

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

7.3. SECURE SHELL SESSION HELP

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

```
$ oc rsh -h
```

CHAPTER 8. TEMPLATES

8.1. OVERVIEW

A template describes a set of objects that can be parameterized and processed to produce a list of objects for creation by OpenShift Container Platform. 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.

8.2. 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:

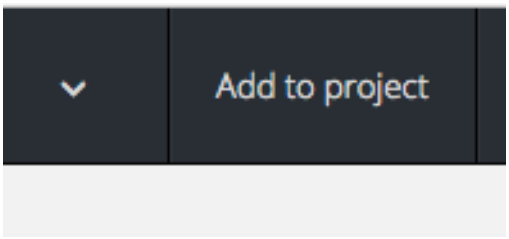
```
$ oc create -f <filename> -n <project>
```

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

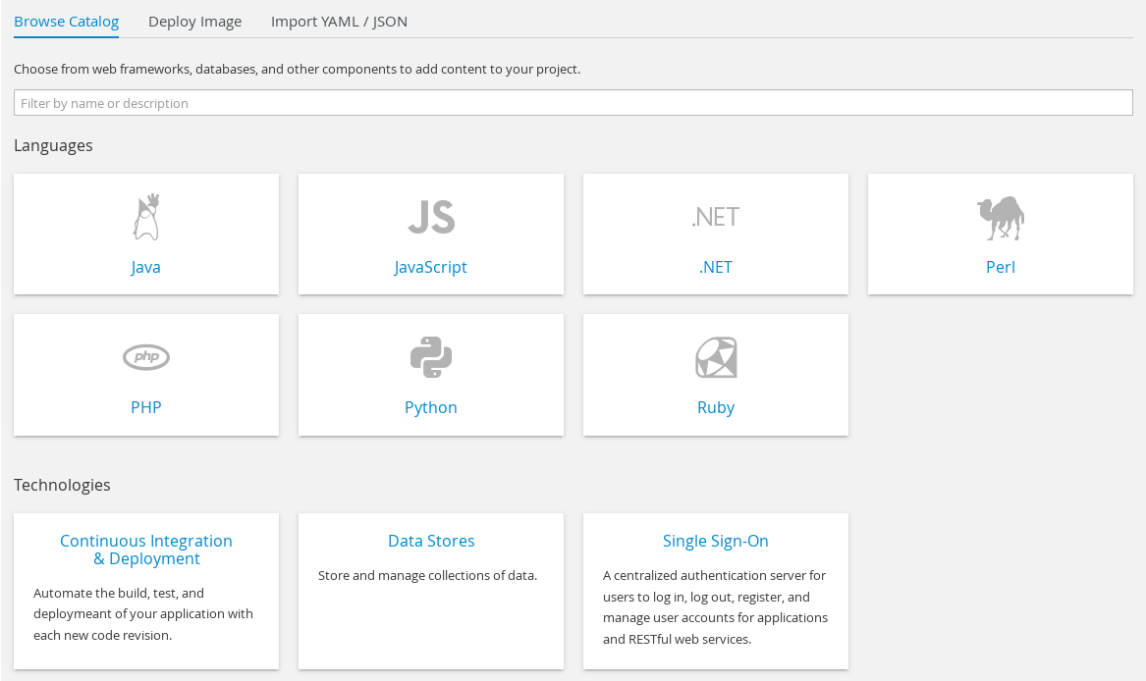
8.3. CREATING FROM TEMPLATES USING THE WEB CONSOLE

To create the objects from an uploaded template using the web console:

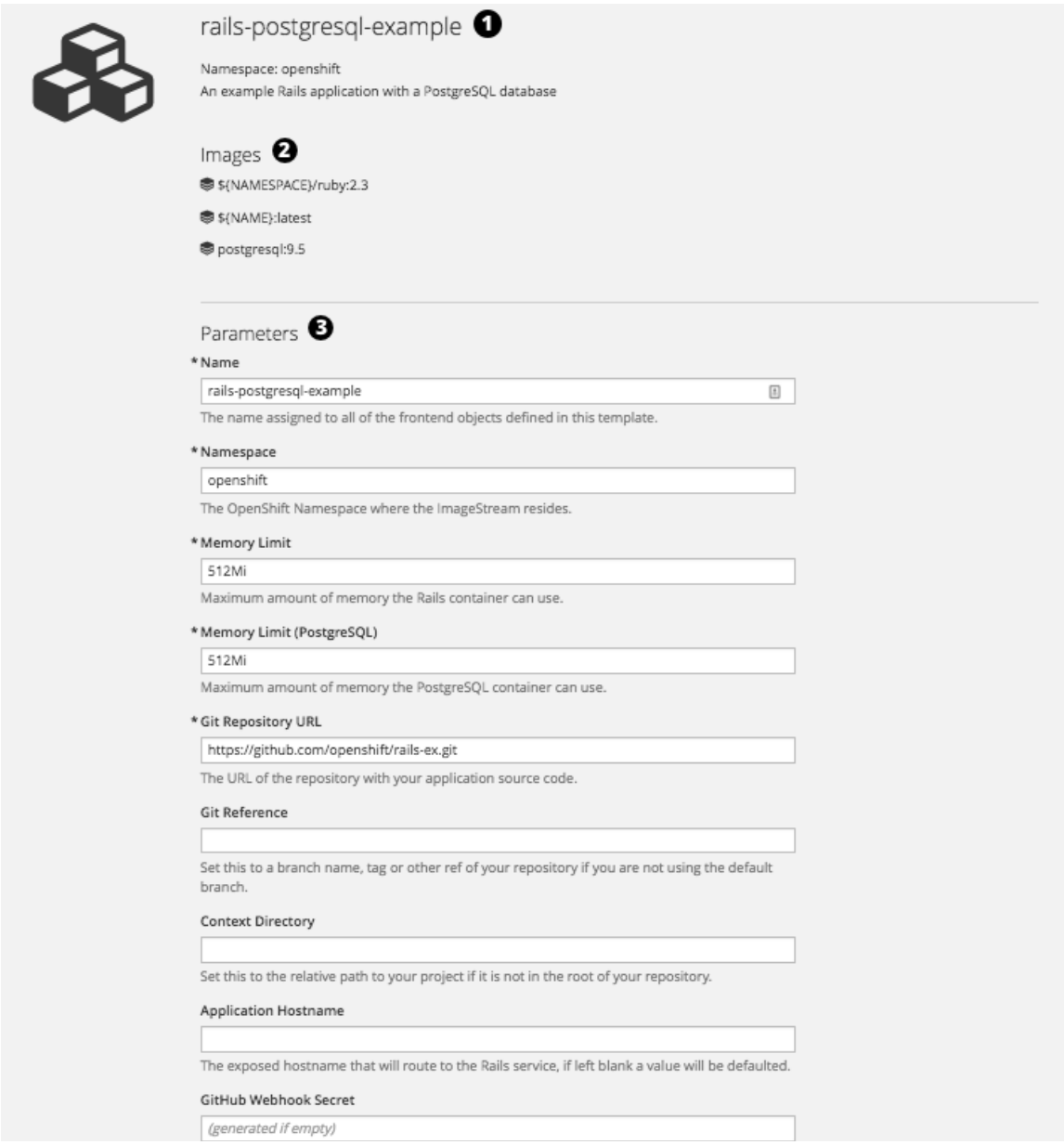
1. While in the desired project, click **Add to Project**:

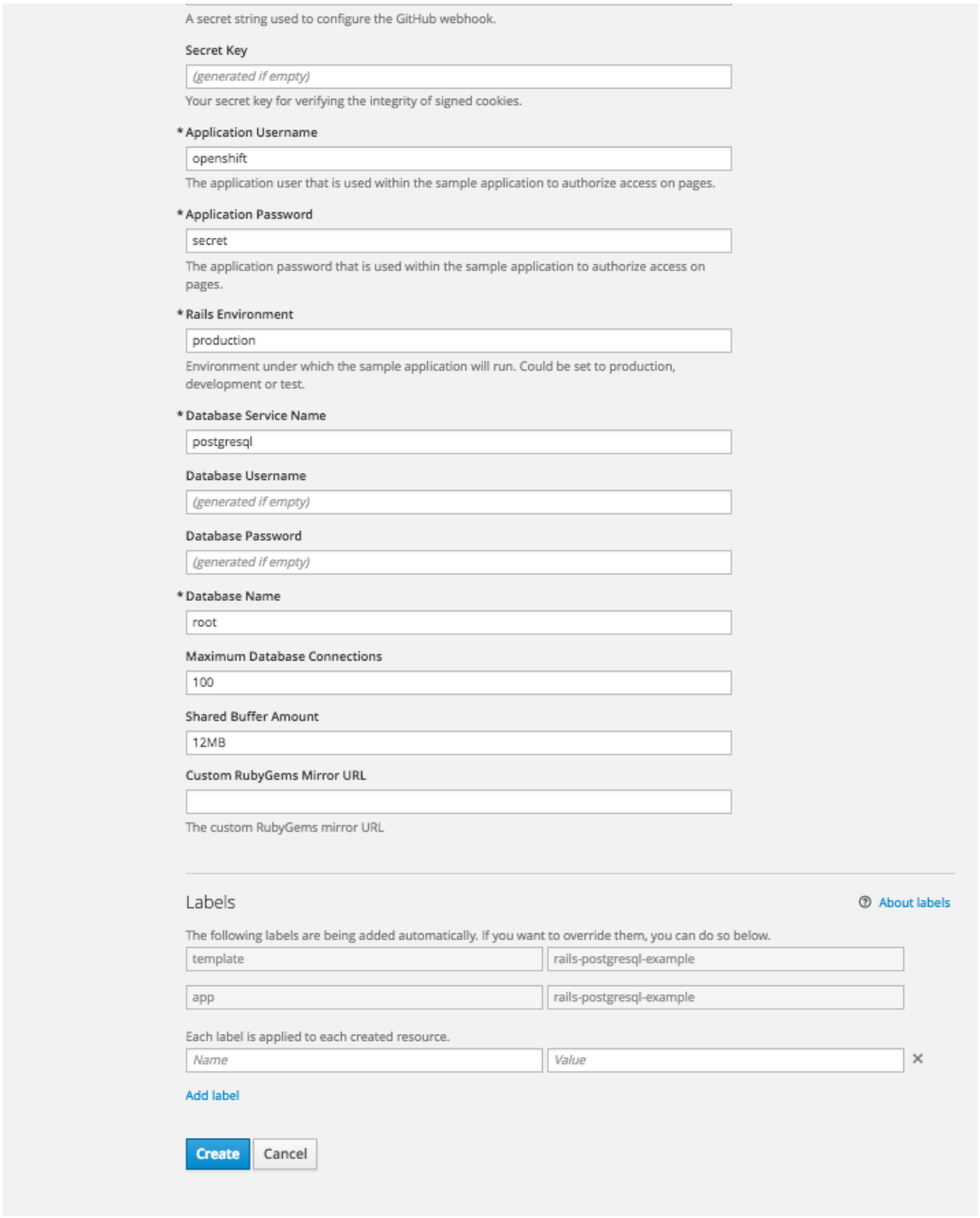


2. Select a template from the list of templates in your project, or provided by the global template library:



3. Modify template parameters in the template creation screen:





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

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

8.4. CREATING FROM TEMPLATES USING THE CLI

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

8.4.1. 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
```

8.4.2. Parameters

The list of parameters that you can override are listed in the **`parameters`** section of the template. You can list them with the CLI by using the following command and specifying the file to be used:

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

Alternatively, if the template is already uploaded:

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

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

```
$ oc process --parameters -n openshift rails-postgresql-example
NAME                     DESCRIPTION
GENERATOR           VALUE
SOURCE_REPOSITORY_URL        The URL of the repository with your
application source code
https://github.com/openshift/rails-ex.git
SOURCE_REPOSITORY_REF        Set this to a branch name, tag or other
ref of your repository if you are not using the default branch
CONTEXT_DIR                  Set this to the relative path to your
project if it is not in the root of your repository
APPLICATION_DOMAIN           The exposed hostname that will route to
the Rails service
rails-postgresql-example.openshiftapps.com
GITHUB_WEBHOOK_SECRET        A secret string used to configure the
GitHub webhook
expression          [a-zA-Z0-9]{40}
SECRET_KEY_BASE              Your secret key for verifying the
integrity of signed cookies
expression          [a-z0-9]{127}
APPLICATION_USER             The application user that is used within
the sample application to authorize access on pages
openshift
```

```
APPLICATION_PASSWORD         The application password that is used
within the sample application to authorize access on pages
secret
DATABASE_SERVICE_NAME        Database service name
postgresql
POSTGRESQL_USER              database username
expression          user[A-Z0-9]{3}
POSTGRESQL_PASSWORD          database password
expression          [a-zA-Z0-9]{8}
POSTGRESQL_DATABASE          database name
root
POSTGRESQL_MAX_CONNECTIONS   database max connections
10
POSTGRESQL_SHARED_BUFFERS    database shared buffers
12MB
```

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

8.4.3. 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>
```

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 for each **`<name>=<value>`** pair you want to override. A parameter reference may appear in any text field inside the template items.

For example, in the following the **`POSTGRESQL_USER`** and **`POSTGRESQL_DATABASE`** parameters of a template are overridden to output a configuration with customized environment variables:

Example 8.1. Creating a List of Objects from a Template

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

8.5. 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>
```

8.6. USING THE INSTANT APP AND QUICKSTART TEMPLATES

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

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

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

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

1. Fork the repository referenced by the template's default **SOURCE_REPOSITORY_URL** parameter.
2. Override the value of the **SOURCE_REPOSITORY_URL** parameter when creating from the template, specifying your fork instead of the default value.

By doing this, the build configuration created by the template will now point to your fork of the application code, and you can modify the code and rebuild the application at will. A walkthrough of this process using the web console is provided in Getting Started for Developers: Web Console.

Note

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

8.7. 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.

Example 8.2. A Simple Template Object Definition (YAML)

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

8.7.1. 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.).

Example 8.3. Template Description Metadata

```
kind: "Template"
apiVersion: "v1"
```

```
metadata:
  name: "cakephp-mysql-example" 1
  annotations:
    openshift.io/display-name: "CakePHP MySQL Example" 2
    description: "An example CakePHP application with a MySQL
database.\n\nFor more information see
https://github.com/openshift/cakephp-ex" 3
    tags: "instant-app,php,cakephp,mysql" 4
    iconClass: "icon-php" 5
message: "Your admin credentials are
${ADMIN_USERNAME}:${ADMIN_PASSWORD}" 6
```

1

The unique name of the template.

2

A brief, user-friendly name, which can be employed by user interfaces.

3

A description of the template. Include enough detail that the user will understand what is being deployed and any caveats they need to know before deploying. It should also provide links to additional information, such as a ***README*** file. Newline characters **\n** can be included to create paragraphs.

4

Tags to be associated with the template for searching and grouping. Add tags that will include it into one of the provided catalog categories. Refer to the **id** and **categoryAliases** in **CATALOG_CATEGORIES** in the console's constants file. The categories can also be customized for the whole cluster.

5

An icon to be displayed with your template in the web console. Choose from our existing logo icons when possible. You can also use icons from FontAwesome and Patternfly. Alternatively, provide icons through CSS customizations that can be added to an OpenShift Container Platform cluster that uses your template. You must specify an icon class that exists, or it will prevent falling back to the generic icon.

6

An instructional message that is displayed when this template is instantiated. This field should inform the user how to use the newly created resources. Parameter substitution is performed on the message before being displayed so that generated credentials and

other parameters can be included in the output. Include links to any next-steps documentation that users should follow.

8.7.2. 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.

Example 8.4. Template Object Labels

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

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

8.7.3. Parameters

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

- As a string value by placing values in the form **${PARAMETER_NAME}** in any string field in the template.
- As a json/yaml value by placing values in the form **${{PARAMETER_NAME}}** in place of any field in the template.

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

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

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

A default value can be provided, which is used if the user does not supply a different value:

Example 8.5. Setting an Explicit Value as the Default Value

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

Parameter values can also be generated based on rules specified in the parameter definition:

Example 8.6. Generating a Parameter Value

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

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

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

- **[\w]{10}** produces 10 alphabet characters, numbers, and underscores. This follows the PCRE standard and is equal to **[a-zA-Z0-9_]{10}**.
- **[\d]{10}** produces 10 numbers. This is equal to **[0-9]{10}**.
- **[\a]{10}** produces 10 alphabetical characters. This is equal to **[a-zA-Z]{10}**.

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

Example 8.7. A full template with parameter definitions and references

```
kind: Template
apiVersion: v1
objects:
  - kind: BuildConfig
    apiVersion: v1
    metadata:
      name: cakephp-mysql-example
      annotations:
        description: Defines how to build the application
    spec:
      source:
        type: Git
        git:
          uri: "${SOURCE_REPOSITORY_URL}" 1
          ref: "${SOURCE_REPOSITORY_REF}"
```

```
        contextDir: "${CONTEXT_DIR}"
  - kind: DeploymentConfig
    apiVersion: v1
    metadata:
      name: frontend
    spec:
      replicas: "${{REPLICA_COUNT}}" 2
parameters:
  - name: SOURCE_REPOSITORY_URL 3
    displayName: Source Repository URL 4
    description: The URL of the repository with your application source
code 5
    value: https://github.com/openshift/cakephp-ex.git 6
    required: true 7
  - name: GITHUB_WEBHOOK_SECRET
    description: A secret string used to configure the GitHub webhook
    generate: expression 8
    from: "[a-zA-Z0-9]{40}" 9
  - name: REPLICA_COUNT
    description: Number of replicas to run
    value: "2"
    required: true
message: "... The GitHub webhook secret is ${GITHUB_WEBHOOK_SECRET}
..." 10
```

1

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

2

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

3

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

4

The user-friendly name for the parameter. This will be displayed to users.

5

A description of the parameter. Provide more detailed information for the purpose of the parameter, including any constraints on the expected value. Descriptions should use complete sentences to follow the console's text standards. Don't make this a duplicate of the display name.

6

A default value for the parameter which will be used if the user does not override the value when instantiating the template. Avoid using default values for things like passwords, instead use generated parameters in combination with Secrets.

7

Indicates this parameter is required, meaning the user cannot override it with an empty value. If the parameter does not provide a default or generated value, the user must supply a value.

8

A parameter which has its value generated.

9

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

10

Parameters can be included in the template message. This informs the user about generated values.

8.7.4. 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.

8.7.5. Other Recommendations

- Group related services together in the management console by adding the **`service.alpha.openshift.io/dependencies`** annotation to the Service object in your template.

Example 8.8. Group the Frontend and Database Services Together on the Management Console Overview

```
kind: "Template"
apiVersion: "v1"
objects:
  - kind: "Service"
    apiVersion: "v1"
    metadata:
      name: "frontend"
      annotations:
        "service.alpha.openshift.io/dependencies": "[{\"name\":
\"database\", \"kind\": \"Service\"}]"
...
  - kind: "Service"
    apiVersion: "v1"
    metadata:
      name: "database"
```

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

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

8.7.6. Creating a Template from Existing Objects

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

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

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

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

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

CHAPTER 9. SERVICE ACCOUNTS

9.1. OVERVIEW

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

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

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

9.2. USER NAMES AND GROUPS

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

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

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

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

Every service account is also a member of two groups:

system:serviceaccounts

Includes all service accounts in the system.

system:serviceaccounts:<project>

Includes all service accounts in the specified project.

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

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

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

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

9.3. DEFAULT SERVICE ACCOUNTS AND ROLES

Three service accounts are automatically created in every project:

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

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

9.4. MANAGING SERVICE ACCOUNTS

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

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

9.5. 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.

9.6. 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 link --for=pull <serviceaccount-name> <secret-name>
```

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

```
$ oc secrets link --for=mount <serviceaccount-name> <secret-name>
```

Note

Limiting secrets to only the service accounts that reference them is disabled by default. This means that if **`serviceAccountConfig.limitSecretReferences`** is set to **`false`** (the default setting) in the master configuration file, mounting secrets to a service account's pods with the **`--for=mount`** option is not required. However, using the **`--for=pull`** option to enable using an image pull secret is required, regardless of the **`serviceAccountConfig.limitSecretReferences`** value.

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 link robot secret-plans --for=mount

$ oc secrets link robot 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
```

9.7. 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"
```

9.8. USING A SERVICE ACCOUNT'S CREDENTIALS EXTERNALLY

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

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

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

For example:

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

```
11e5-afdc-3c970e4b7ffe

Type: kubernetes.io/service-account-token

Data

token: eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...

$ oc login --token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...
Logged into "https://server:8443" as "system:serviceaccount:top-
secret:robot" using the token provided.

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

    $ oc new-project <projectname>

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

CHAPTER 10. BUILDS

10.1. OVERVIEW

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

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

There are four build strategies:

- Source-To-Image (S2I) (description, options)
- Docker (description, options)
- Pipeline (description, options)
- Custom (description, options)

And there are four types of build sources:

- Git
- Dockerfile
- Image
- Binary

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

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

10.2. DEFINING A BUILDCONFIG

A build configuration describes a single build definition and a set of triggers for when a new build should be created.

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

Example 10.1. BuildConfig Object Definition

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "ruby-sample-build" 1
spec:
  runPolicy: "Serial" 2
```

```
triggers: 3
  -
    type: "GitHub"
    github:
      secret: "secret101"
  - type: "Generic"
    generic:
      secret: "secret101"
  -
    type: "ImageChange"
source: 4
  type: "Git"
  git:
    uri: "https://github.com/openshift/ruby-hello-world"
  dockerfile: "FROM openshift/ruby-22-centos7\nUSER example"
strategy: 5
  type: "Source"
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "ruby-20-centos7:latest"
output: 6
  to:
    kind: "ImageStreamTag"
    name: "origin-ruby-sample:latest"
postCommit: 7
    script: "bundle exec rake test"
```

1

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

2

The **`runPolicy`** field controls whether builds created from this build configuration can be run simultaneously. The default value is **Serial**, which means new builds will run sequentially, not simultaneously.

3

You can specify a list of triggers, which cause a new build to be created.

4

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

5

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

6

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

7

The **postCommit** section defines an optional build hook.

10.3. SOURCE-TO-IMAGE STRATEGY OPTIONS

The following options are specific to the S2I build strategy.

10.3.1. Force Pull

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

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

1

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

2

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

10.3.2. Incremental Builds

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

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

1

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

2

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

Note

See the S2I Requirements topic for information on how to create a builder image supporting incremental builds.

10.3.3. Extended Builds

Note

This feature is in technology preview. This means the API may change without notice or the feature may be removed entirely. For a supported mechanism to produce application images with runtime-only content, consider using the Image Source feature and defining two builds, one which produces an image containing the runtime artifacts and a second build which consumes the runtime artifacts from that image and adds them to a runtime-only image.

For compiled languages (Go, C, C++, Java, etc.) the dependencies necessary for compilation might increase the size of the image or introduce vulnerabilities that can be exploited.

To avoid these problems, S2I (Source-to-Image) introduces a two-image build process that allows an application to be built via the normal flow in a builder image, but then injects the resulting application artifacts into a runtime-only image for execution.

To offer flexibility in this process, S2I executes an **`assemble-runtime`** script inside the runtime image that allows further customization of the resulting runtime image.

More information about this can be found in the official S2I extended builds documents.

This feature is available only for the source strategy.

```
strategy:
  type: "Source"
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "builder-image:latest"
    runtimeImage: 1
      kind: "ImageStreamTag"
      name: "runtime-image:latest"
    runtimeArtifacts: 2
      - sourcePath: "/path/to/source"
        destinationDir: "path/to/destination"
```

1

The runtime image that the artifacts should be copied to. This is the final image that the application will run on. This image should contain the minimum application dependencies to run the injected content from the builder image.

2

The runtime artifacts are a mapping of artifacts produced in the builder image that should be injected into the runtime image. **`sourcePath`** can be the full path to a file or directory inside the builder image. **`destinationDir`** must be a directory inside the runtime image where the artifacts will be copied. This directory is relative to the specified **WORKDIR** inside that image.

Note

In the current implementation, you cannot have incremental extended builds thus, the **incremental** option is not valid with **`runtimeImage`**.

If the runtime image needs authentication to be pulled across OpenShift projects or from another private registry, the details can be specified within the image pull secret configuration.

10.3.3.1. Testing your Application

Extended builds offer two ways of running tests against your application.

The first option is to install all test dependencies and run the tests inside your **builder image** since that image, in the context of extended builds, will not be pushed to a registry. This can be done as a part of the **`assemble`** script for the builder image.

The second option is to specify a script via the postcommit hook. This is executed in an ephemeral container based on the runtime image, thus it is not committed to the image.

10.3.4. Overriding Builder Image Scripts

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

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

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

1

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

Note

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

10.3.5. 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.

10.3.5.1. Environment Files

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

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

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

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

10.3.5.2. 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"
```

You can also manage environment variables defined in the **`BuildConfig`** with the **`oc set env`** command.

10.3.6. Adding Secrets to Source Strategy Build Configurations from the Web Console

Add a secret to your build configuration so that it can access a private repository.

1. Create a new OpenShift Container Platform project.
2. Create a secret that contains credentials for accessing a private source code repository.
3. Create a Source-to-Image (S2I) build configuration.
4. On the build configuration editor page or in the **fromimage** page of the web console, set the **Source Secret**.
5. Click the **Save** button.

10.3.6.1. Enabling Pulling and Pushing

Enable pulling to a private registry by setting the **`Pull Secret`** in the build configuration and enable pushing by setting the **`Push Secret`**.

10.4. DOCKER STRATEGY OPTIONS

The following options are specific to the Docker build strategy.

10.4.1. FROM Image

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

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

10.4.2. Dockerfile Path

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

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

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

10.4.3. No Cache

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

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

10.4.4. Force Pull

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

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

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.

10.4.5. Environment Variables

To make environment variables available to the Docker build process and resulting image, you can add environment variables to the **`dockerStrategy`** definition of the **`BuildConfig`**.

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

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

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

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

Cluster administrators can also configure global build settings using Ansible.

You can also manage environment variables defined in the **`BuildConfig`** with the **`oc set env`** command.

10.4.6. Adding Secrets to Docker Strategy Build Configurations from the Web Console

Add a secret to your build configuration so that it can access a private repository.

1. Create a new OpenShift Container Platform project.
2. Create a secret that contains credentials for accessing a private source code repository.
3. Create a docker build configuration.
4. On the build configuration editor page or in the **fromimage** page of the web console, set the **Source Secret**.
5. Click the **Save** button.

10.4.6.1. Enabling Pulling and Pushing

Enable pulling to a private registry by setting the **`Pull Secret`** in the build configuration and enable pushing by setting the **`Push Secret`**.

10.5. CUSTOM STRATEGY OPTIONS

The following options are specific to the Custom build strategy.

10.5.1. FROM Image

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

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

10.5.2. Exposing the Docker Socket

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

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

10.5.3. Secrets

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

Each secret can be mounted at a specific location:

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

1

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

2

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

10.5.3.1. Adding Secrets to Custom Strategy Build Configurations from the Web Console

Add a secret to your build configuration so that it can access a private repository.

1. Create a new OpenShift Container Platform project.
2. Create a secret that contains credentials for accessing a private source code repository.
3. Create a custom build configuration.
4. On the build configuration editor page or in the **fromimage** page of the web console, set the **Source Secret**.
5. Click the **Save** button.

10.5.3.2. Enabling Pulling and Pushing

Enable pulling to a private registry by setting the **`Pull Secret`** in the build configuration and enable pushing by setting the **`Push Secret`**.

10.5.4. Force Pull

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

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

1

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

10.5.5. Environment Variables

To make environment variables available to the Custom build process, you can add environment variables to the **`customStrategy`** definition of the **`BuildConfig`**.

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

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

```
customStrategy:
...
  env:
```

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

Cluster administrators can also configure global build settings using Ansible.

You can also manage environment variables defined in the **BuildConfig** with the **oc set env** command.

10.6. PIPELINE STRATEGY OPTIONS

The following options are specific to the Pipeline build strategy.

10.6.1. Providing the Jenkinsfile

You can provide the Jenkinsfile in one of two ways:

1. Embed the Jenkinsfile in the build configuration.
2. Include in the build configuration a reference to the Git repository that contains the Jenkinsfile.

Example 10.2. Embedded Definition

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "sample-pipeline"
spec:
  strategy:
    type: "JenkinsPipeline"
    jenkinsPipelineStrategy:
      jenkinsfile: "node('agent') {\nstage
'build'\nopenshiftBuild(buildConfig: 'ruby-sample-build',
showBuildLogs: 'true')\nstage
'deploy'\nopenshiftDeploy(deploymentConfig: 'frontend')\n}"
```

Example 10.3. Reference to Git Repository

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "sample-pipeline"
spec:
  source:
    type: "Git"
    git:
      uri: "https://github.com/openshift/ruby-hello-world"
```

```
strategy:
  type: "JenkinsPipeline"
  jenkinsPipelineStrategy:
    jenkinsfilePath: some/repo/dir/filename 1
```

1

The optional **`jenkinsfilePath`** field specifies the name of the file to use, relative to the source **`contextDir`**. If **`contextDir`** is omitted, it defaults to the root of the repository. If **`jenkinsfilePath`** is omitted, it defaults to ***Jenkinsfile***.

10.7. BUILD INPUTS

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

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

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

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

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

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

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

1

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

2

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

3

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

4

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

10.7.1. Git Source

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

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

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

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

1

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

a branch name.

2

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

3

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

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

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

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

10.7.1.1. Using a Proxy

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. Domains for which no proxying should be performed can also be specified via the **NoProxy** field.

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
    noProxy: somedomain.com, otherdomain.com
```

Cluster administrators can also configure a global proxy for Git cloning using Ansible.

10.7.1.2. Source Secrets

10.7.1.2.1. Overview

Source secrets are used to provide the builder pod with access to Git repositories that it would not normally have access to, such as private repositories or repositories with self-signed or untrusted SSL certificates.

The following source secret configurations are supported:

- Gitconfig File
- Basic Authentication
- SSH Key Authentication
- Trusted Certificate Authorities

Note

You can also use combinations of the these configurations to meet your specific needs.

All source secrets must be linked to the builder account and added to the build configuration using the following instructions:

Note

Limiting secrets to only the service accounts that reference them is disabled by default. This means that if **`serviceAccountConfig.limitSecretReferences`** is set to **`false`** (the default setting) in the master configuration file, linking secrets to a service is not required.

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

```
$ oc secrets link builder basicsecret
```

2. Add a **`sourceSecret`** field to the **`source`** section inside the **`BuildConfig`** and set it to the name of the **`secret`** that you created (**`basicsecret`**, in this example).

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "sample-image:latest"
  source:
    git:
      uri: "https://github.com/user/app.git"
    sourceSecret:
      name: "basicsecret"
    type: "Git"
```

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

Note

You can also use the **`oc set build-secret`** command to set the secret on the existing build configuration:

```
$ oc set build-secret --source bc/sample-build basicsecret
```

Defining Secrets in the BuildConfig provides more information on this topic.

10.7.1.2.2. .Gitconfig File

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

To create a secret from a ***.gitconfig*** file:

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

Note

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

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

10.7.1.2.3. Basic Authentication

Basic authentication requires either a combination of **`username`** and **`password`**, or a **`token`** to authenticate against the SCM server.

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

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

To create a basic authentication secret with a token:

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

10.7.1.2.4. SSH Key Authentication

SSH key based authentication requires a private SSH key.

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

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

Note

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

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

Before using the SSH key to access the private repository, create the secret first:

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

You can also use the **`oc set build-secret`** command to set the secret on the existing build configuration:

```
$ oc set build-secret --source bc/sample-build sshsecret
```

10.7.1.2.5. Trusted Certificate Authorities

The set of TLS certificate authorities that are trusted during a **`git clone`** operation are built into the OpenShift Container Platform infrastructure images. If your Git server uses a self-signed certificate or one signed by an authority not trusted by the image, you have several options.

1. Create a secret with a CA certificate file (recommended).

   A secret containing a **`CA certificate`** in a key named **`ca.crt`** will automatically be used by Git to trust your self-signed or otherwise un-trusted TLS certificate during the **`git clone`** operation. Using this method is significantly more secure than disabling Git's SSL verification, which accepts any TLS certificate that is presented.

   ```
   # the key name ca.crt MUST be used
   $ oc secrets new mycert ca.crt=FILENAME
   ```

2. Disable Git TLS verification.

You can disable Git's TLS verification by setting the **GIT_SSL_NO_VERIFY** environment variable to **true** in the appropriate strategy section of your build configuration. You can use the **oc set env** command to manage **BuildConfig** environment variables.

10.7.1.2.6. Combinations

Below are several examples of how you can combine the above methods for creating source secrets for your specific needs.

a. To create an SSH-based authentication secret with a ***.gitconfig*** file:

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

b. To create a secret that combines a ***.gitconfig*** file and CA certificate:

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

c. To create a basic authentication secret with a CA certificate file:

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

d. To create a basic authentication secret with a ***.gitconfig*** file:

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

e. To create a basic authentication secret with a ***.gitconfig*** file and CA certificate file:

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

You can also use the **oc set build-secret** command to set the secret on the existing build configuration:

```
$ oc set build-secret --source bc/sample-build mysecret
```

10.7.2. 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.

10.7.3. Binary Source

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

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

Note

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

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

- **--from-file**: The contents of the file you specify are sent as a binary stream to the builder. The builder then stores the data in a file with the same name at the top of the build context.
- **--from-dir** and **--from-repo**: The contents are archived and sent as a binary stream to the builder. The builder then extracts the contents of the archive within the build context directory.
- **--from-archive**: The archive you specify is sent to the builder, where it is extracted within the build context directory. Please note that this option behaves the same as **--from-dir**, an archive is created on your host first whenever the argument to these options is a directory.

In each of the above cases:

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

Instead of a file name, you can pass URL with http or https schema to **--from-file** and **--from-archive**. When using **--from-file** with a URL, the name of the file in the builder image is determined by the **Content-Disposition** header sent by the web server, or the last component of the URL path if the header is not present. Please note that no form of authentication is supported and it is not possible to use custom TLS certificate or disable certificate validation.

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

The **`dockerfile`** and **`contextDir`** source options have special meaning with binary builds.

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

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

10.7.4. Image Source

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

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

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

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

1

An array of one or more input images and files.

2

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

3

An array of source/destination paths.

4

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

5

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

6

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

Note

This feature is not supported for builds using the Custom Strategy.

10.8. USING SECRETS DURING A BUILD

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

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

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

10.8.1. Defining Secrets in the BuildConfig

1. Create the **`Secret`**:

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

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

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

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

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

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

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

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

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

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

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

10.8.2. Source-to-Image Strategy

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

process.

Note

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

10.8.3. Docker Strategy

When using a **Docker** strategy, you can add all defined source secrets into your container image using the ADD and COPY instructions in your ***Dockerfile***. If you do not specify the **destinationDir** for a secret, then the files will be copied into the same directory in which the ***Dockerfile*** is located. If you specify a relative path as **destinationDir**, then the secrets will be copied into that directory, relative to your ***Dockerfile*** location. This makes the secret files available to the Docker build operation as part of the context directory used during the build.

Note

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

10.8.4. Custom Strategy

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

10.9. STARTING A BUILD

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

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

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

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

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

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

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

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

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

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

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

Note

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

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

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

10.10. CANCELING A BUILD

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

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

Cancel multiple builds at the same time:

```
$ oc cancel-build <build1_name> <build2_name> <build3_name>
```

Cancel all builds created from the build configuration:

```
$ oc cancel-build bc/<buildconfig_name>
```

Cancel all builds in a given state (for example, **new** or **pending**), ignoring the builds in other states:

```
$ oc cancel-build bc/<buildconfig_name>  --state=<state>
```

10.11. DELETING A BUILDCONFIG

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

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

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

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

10.12. VIEWING BUILD DETAILS

You can view build details with the web console or by using the **`oc describe`** CLI command:

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

This displays information such as:

- The build source
- The strategy
- The output destination
- How the build was created

If the build uses the Docker or Source strategy, the **`oc describe`** output also includes information about the source revision used for the build, including the commit ID, author, committer, and message.

10.13. ACCESSING BUILD LOGS

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

To stream the logs using the build directly:

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

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

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

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

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

Log Verbosity

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

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

1

Adjust this value to the desired log level.

Note

A platform administrator can set the default build verbosity for the entire OpenShift Container Platform instance by configuring **env/BUILD_LOGLEVEL** for the **BuildDefaults** admission controller. This default can be overridden by specifying **BUILD_LOGLEVEL** in a given **BuildConfig**. You can specify a higher priority override on the command line for non-binary builds by passing **--build-loglevel** to **oc start-build**.

Available log levels for Source builds are as follows:

Level 0	Produces output from containers running the ***assemble*** script and all encountered errors. This is the default.
Level 1	Produces basic information about the executed process.
Level 2	Produces very detailed information about the executed process.
Level 3	Produces very detailed information about the executed process, and a listing of the archive contents.
Level 4	Currently produces the same information as level 3.

Level 5	Produces everything mentioned on previous levels and additionally provides docker push messages.

10.14. SETTING MAXIMUM DURATION

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

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

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

```
spec:
  completionDeadlineSeconds: 1800
```

10.15. 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

10.15.1. Webhook Triggers

Webhook triggers allow you to trigger a new build by sending a request to the OpenShift Container Platform API endpoint. You can define these triggers using GitHub webhooks or Generic webhooks.

GitHub Webhooks

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

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

Note

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

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

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

To configure a GitHub Webhook:

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

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

2. Copy the webhook URL.
3. Follow the GitHub setup instructions to paste the webhook URL into your GitHub repository settings.

Note

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

Given a file containing a valid JSON payload, you can manually trigger the webhook via **`curl`**:

```
$ curl -H "X-GitHub-Event: push" -H "Content-Type: application/json" -k
-X POST --data-binary @github_payload_file.json
https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildc
onfigs/<name>/webhooks/<secret>/github
```

The **`-k`** argument is only necessary if your API server does not have a properly signed certificate.

Generic Webhooks

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

```
type: "Generic"
generic:
  secret: "secret101"
  allowEnv: true 1
```

1

Set to **true** to allow a generic webhook to pass in environment variables.

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

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

The caller must invoke the webhook as a **POST** operation.

To invoke the webhook manually you can use **curl**:

```
$ curl -X POST -k
https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildc
onfigs/<name>/webhooks/<secret>/generic
```

The HTTP verb must be set to **POST**. The insecure **-k** flag is specified to ignore certificate validation. This second flag is not necessary if your cluster has properly signed certificates.

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>"
env: 1
   - name: "<variable name>"
     value: "<variable value>"
```

1

Similar to the **BuildConfig** environment variables, the environment variables defined here are made available to your build. If these variables collide with the **BuildConfig** environment variables, these variables take precedence. By default, environment variables passed via webhook are ignored. Set the **allowEnv** field to **true** on the webhook definition to enable this behavior.

To pass this payload using **curl**, define it in a file named ***payload_file.yaml*** and run:

```
$ curl -H "Content-Type: application/yaml" --data-binary
@payload_file.yaml -X POST -k
https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildc
onfigs/<name>/webhooks/<secret>/generic
```

The arguments are the same as the previous example with the addition of a header and a payload. The **-H** argument sets the **Content-Type** header to **application/yaml** or **application/json** depending on your payload format. The **--data-binary** argument is used to send a binary payload with newlines intact with the **POST** request.

Note

OpenShift Container Platform permits builds to be triggered via the generic webhook even if an invalid request payload is presented (for example, invalid content type, unparsable or invalid content, and so on). This behavior is maintained for backwards compatibility. If an invalid request payload is presented, OpenShift Container Platform returns a warning in JSON format as part of its **HTTP 200 OK** response.

Note

OpenShift Container Platform permits builds to be triggered via the generic webhook even if an invalid request payload is presented (for example, invalid content type, unparsable or invalid content, and so on). This behavior is maintained for backwards compatibility. If an invalid request payload is presented, OpenShift Container Platform returns a warning in JSON format as part of its **HTTP 200 OK** response.

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.

10.15.2. 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 container image repository located at ***<system-registry>/<namespace>/*ruby-20-centos7**. The ***<system-registry>*** is defined as a service with the name **`docker-registry`** running in OpenShift Container Platform.

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

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

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

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

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

1

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

2

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

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

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

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

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

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

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

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

Note

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

10.15.3. 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 YAML within the **`BuildConfig`**:

```
  type: "ConfigChange"
```

Note

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

10.16. BUILD HOOKS

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

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

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

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

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

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

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

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

- Shell script:

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

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

  Note

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

- Command as the image entry point:

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

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

- Pass arguments to the default entry point:

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

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

- Shell script with arguments:

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

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

- Command with arguments:

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

This form is equivalent to appending the arguments to **`command`**.

Note

Providing both **`script`** and **`command`** simultaneously creates an invalid build hook.

10.16.1. Using the Command Line

The **`oc set build-hook`** command can be used to set the build hook for a build configuration.

To set a command as the post-commit build hook:

```
$ oc set build-hook bc/mybc --post-commit --command -- bundle exec rake
test --verbose
```

To set a script as the post-commit build hook:

```
$ oc set build-hook bc/mybc --post-commit --script="bundle exec rake
test --verbose"
```

10.17. USING DOCKER CREDENTIALS FOR PUSHING AND PULLING IMAGES

Supply the ***.docker/config.json*** 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 Container Platform Docker Registry, you don't have to do this because **`secrets`** are generated automatically for you by OpenShift Container Platform.

The ***.docker/config.json*** file is found in your home directory by default and has the following format:

```
auths:
  https://index.docker.io/v1/: 1
    auth: "YWRfbGzhcGU6R2labnRib21ifTE=" 2
    email: "user@example.com" 3
```

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 objects, which are used to store your configuration and passwords.

1. Create the **`secret`** from your local ***.docker/config.json*** file:

```
$ oc secrets new dockerhub ~/.docker/config.json
```

   This generates a JSON specification of the **`secret`** named **dockerhub** and creates the object.

2. Once the **`secret`** is created, add it to the builder service account. Each build is run with the **`builder`** role, so you need to give it access your secret with the following command:

```
$ oc secrets link builder dockerhub
```

3. Add a **`pushSecret`** field into the **`output`** section of the **`BuildConfig`** and set it to the name of the **`secret`** that you created, which in the above example is **`dockerhub`**:

```
spec:
  output:
    to:
      kind: "DockerImage"
      name: "private.registry.com/org/private-image:latest"
    pushSecret:
      name: "dockerhub"
```

   You can also use the **`oc set build-secret`** command to set the push secret on the build configuration:

```
$ oc set build-secret --push bc/sample-build dockerhub
```

4. Pull the builder container image from a private Docker registry by specifying the **`pullSecret`** field, which is part of the build strategy definition:

```
strategy:
  sourceStrategy:
    from:
      kind: "DockerImage"
      name: "docker.io/user/private_repository"
    pullSecret:
      name: "dockerhub"
  type: "Source"
```

You can also use the **`oc set build-secret`** command to set the pull secret on the build configuration:

```
$ oc set build-secret --pull bc/sample-build dockerhub
```

Note

This example uses **`pullSecret`** in a Source build, but it is also applicable in Docker and Custom builds.

10.18. BUILD RUN POLICY

The build run policy describes the order in which the builds created from the build configuration should run. This can be done by changing the value of the **`runPolicy`** field in the **`spec`** section of the **`Build`** specification.

It is also possible to change the **`runPolicy`** value for existing build configurations.

- Changing **`Parallel`** to **`Serial`** or **`SerialLatestOnly`** and triggering a new build from this configuration will cause the new build to wait until all parallel builds complete as the serial build can only run alone.
- Changing **`Serial`** to **`SerialLatestOnly`** and triggering a new build will cause cancellation of all existing builds in queue, except the currently running build and the most recently created build. The newest build will execute next.

10.18.1. Serial Run Policy

Setting the **`runPolicy`** field to **`Serial`** will cause all new builds created from the **`Build`** configuration to be run sequentially. That means there will be only one build running at a time and every new build will wait until the previous build completes. Using this policy will result in consistent and predictable build output. This is the default **`runPolicy`**.

Triggering three builds from the **sample-build** configuration, using the **`Serial`** policy will result in:

```
NAME             TYPE      FROM          STATUS    STARTED
DURATION
sample-build-1   Source    Git@e79d887   Running   13 seconds ago
13s
sample-build-2   Source    Git           New
sample-build-3   Source    Git           New
```

When the **sample-build-1** build completes, the **sample-build-2** build will run:

```
NAME              TYPE      FROM          STATUS    STARTED
DURATION
sample-build-1    Source    Git@e79d887   Completed 43 seconds ago
34s
sample-build-2    Source    Git@1aa381b   Running   2 seconds ago    2s
sample-build-3    Source    Git           New
```

10.18.2. SerialLatestOnly Run Policy

Setting the **`runPolicy`** field to **`SerialLatestOnly`** will cause all new builds created from the **`Build`** configuration to be run sequentially, same as using the **`Serial`** run policy. The difference is that when a currently running build completes, the next build that will run is the latest build created. In other words, you do not wait for the queued builds to run, as they are skipped. Skipped builds are marked as **Cancelled**. This policy can be used for fast, iterative development.

Triggering three builds from the **sample-build** configuration, using the **`SerialLatestOnly`** policy will result in:

```
NAME              TYPE      FROM          STATUS    STARTED
DURATION
sample-build-1    Source    Git@e79d887   Running   13 seconds ago
13s
sample-build-2    Source    Git           Cancelled
sample-build-3    Source    Git           New
```

The **sample-build-2** build will be canceled (skipped) and the next build run after **sample-build-1** completes will be the **sample-build-3** build:

```
NAME              TYPE      FROM          STATUS    STARTED
DURATION
sample-build-1    Source    Git@e79d887   Completed 43 seconds ago
34s
sample-build-2    Source    Git           Cancelled
sample-build-3    Source    Git@1aa381b   Running   2 seconds ago    2s
```

10.18.3. Parallel Run Policy

Setting the **`runPolicy`** field to **`Parallel`** causes all new builds created from the **`Build`** configuration to be run in parallel. This can produce unpredictable results, as the first created build can complete last, which will replace the pushed container image produced by the last build which completed earlier.

Use the parallel run policy in cases where you do not care about the order in which the builds will complete.

Triggering three builds from the **sample-build** configuration, using the **`Parallel`** policy will result in three simultaneous builds:

```
NAME              TYPE      FROM          STATUS    STARTED
```

```
DURATION
sample-build-1   Source    Git@e79d887   Running   13 seconds ago
13s
sample-build-2   Source    Git@a76d881   Running   15 seconds ago   3s
sample-build-3   Source    Git@689d111   Running   17 seconds ago   3s
```

The completion order is not guaranteed:

```
NAME             TYPE      FROM          STATUS    STARTED
DURATION
sample-build-1   Source    Git@e79d887   Running   13 seconds ago
13s
sample-build-2   Source    Git@a76d881   Running   15 seconds ago   3s
sample-build-3   Source    Git@689d111   Completed 17 seconds ago   5s
```

10.19. BUILD OUTPUT

Docker and Source builds result in the creation of a new container image. The image is then pushed to the registry specified in the **`output`** section of the **`Build`** specification.

If the output kind is **`ImageStreamTag`**, then the image will be pushed to the integrated OpenShift Container Platform registry and tagged in the specified image stream. If the output is of type **`DockerImage`**, then the name of the output reference will be used as a Docker push specification. The specification may contain a registry or will default to DockerHub if no registry is specified. If the output section of the build specification is empty, then the image will not be pushed at the end of the build.

Example 10.4. Output to an ImageStreamTag

```
output:
  to:
    kind: "ImageStreamTag"
    name: "sample-image:latest"
```

Example 10.5. Output to a Docker Push Specification

```
output:
  to:
    kind: "DockerImage"
    name: "my-registry.mycompany.com:5000/myimages/myimage:tag"
```

10.19.1. Output Image Environment Variables

Docker and Source builds set the following environment variables on output images:

Variable	Description
OPENSHIFT_BUILD_NAME	Name of the build
OPENSHIFT_BUILD_NAMESPACE	Namespace of the build
OPENSHIFT_BUILD_SOURCE	The source URL of the build
OPENSHIFT_BUILD_REFERENCE	The Git reference used in the build
OPENSHIFT_BUILD_COMMIT	Source commit used in the build

10.19.2. Output Image Labels

Docker and Source builds set the following labels on output images:

Label	Description
io.openshift.build.commit.author	Author of the source commit used in the build
io.openshift.build.commit.date	Date of the source commit used in the build
io.openshift.build.commit.id	Hash of the source commit used in the build
io.openshift.build.commit.message	Message of the source commit used in the build
io.openshift.build.commit.ref	Branch or reference specified in the source
io.openshift.build.source-location	Source URL for the build

You can also use the **BuildConfig.spec.output.imageLabels** field to specify a list of custom labels that will be applied to each image built from the BuildConfig.

Example 10.6. Custom labels to be applied to built images

```
output:
  to:
    kind: "ImageStreamTag"
    name: "my-image:latest"
  imageLabels:
  - name: "vendor"
    value: "MyCompany"
  - name: "authoritative-source-url"
    value: "registry.mycompany.com"
```

10.20. USING EXTERNAL ARTIFACTS DURING A BUILD

It is not recommended to store binary files in a source repository. Therefore, you may find it necessary to define a build which pulls additional files (such as Java ***.jar*** dependencies) during the build process. How this is done depends on the build strategy you are using.

For a **`Source`** build strategy, you must put appropriate shell commands into the ***assemble*** script:

Example 10.7. *.s2i/bin/assemble* File

```
#!/bin/sh
APP_VERSION=1.0
wget http://repository.example.com/app/app-$APP_VERSION.jar -O app.jar
```

Example 10.8. *.s2i/bin/run* File

```
#!/bin/sh
exec java -jar app.jar
```

Note

For more information on how to control which ***assemble*** and ***run*** script is used by a Source build, see Overriding Builder Image Scripts.

For a **`Docker`** build strategy, you must modify the ***Dockerfile*** and invoke shell commands with the **`RUN`** instruction:

Example 10.9. Excerpt of `Dockerfile`

```
FROM jboss/base-jdk:8

ENV APP_VERSION 1.0
RUN wget http://repository.example.com/app/app-$APP_VERSION.jar -O
```

```
app.jar

EXPOSE 8080
CMD [ "java", "-jar", "app.jar" ]
```

In practice, you may want to use an environment variable for the file location so that the specific file to be downloaded can be customized using an environment variable defined on the **`BuildConfig`**, rather than updating the ***assemble*** script or ***Dockerfile***.

You can choose between different methods of defining environment variables:

- Using the ***.s2i/environment*** file (only for a Source build strategy)
- Setting in **`BuildConfig`**
- Providing explicitly using **`oc start-build --env`** (only for builds that are triggered manually)

10.21. BUILD RESOURCES

By default, builds are completed by pods using unbound resources, such as memory and CPU. These resources can be limited by specifying resource limits in a project's default container limits.

You can also limit resource use by specifying resource limits as part of the build configuration. In the following example, each of the **`resources`**, **`cpu`**, and **`memory`** parameters are optional:

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  resources:
    limits:
      cpu: "100m" 1
      memory: "256Mi" 2
```

1

`cpu` is in CPU units: **`100m`** represents 0.1 CPU units (100 * 1e-3).

2

`memory` is in bytes: **`256Mi`** represents 268435456 bytes (256 * 2 ^ 20).

However, if a quota has been defined for your project, one of the following two items is required:

- A **`resources`** section set with an explicit **`requests`**:

```
resources:
  requests: 1
    cpu: "100m"
    memory: "256Mi"
```

1

The **requests** object contains the list of resources that correspond to the list of resources in the quota.

- A limit range defined in your project, where the defaults from the **LimitRange** object apply to pods created during the build process.

Otherwise, build pod creation will fail, citing a failure to satisfy quota.

10.22. ASSIGNING BUILDS TO SPECIFIC NODES

Builds can be targeted to run on specific nodes by specifying labels in the **nodeSelector** field of a build configuration. The **nodeSelector** value is a set of key/value pairs that are matched to **node** labels when scheduling the **build pod**.

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  nodeSelector: 1
    key1: value1
    key2: value2
```

1

Builds associated with this build configuration will run only on nodes with the **key1=value2** and **key2=value2** labels.

The **nodeSelector** value can also be controlled by cluster-wide default and override values. Defaults will only be applied if the build configuration does not define any key/value pairs for the **nodeSelector** and also does not define an explicitly empty map value of "nodeSelector:{}". Override values will replace values in the build configuration on a key by key basis.

See configuring global build defaults and overrides for more information.

Note

If the specified **NodeSelector** cannot be matched to a node with those labels, the build still stay in the **pending** state indefinitely.

10.23. TROUBLESHOOTING

Table 10.1. Troubleshooting Guidance for Builds

Issue	Resolution
A build fails with: `requested access to the resource is denied`	You have exceeded one of the image quotas set on your project. Check your current quota and verify the limits applied and storage in use: `$ oc describe quota`

CHAPTER 11. MANAGING IMAGES

11.1. OVERVIEW

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

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

There are many ways you can interact with images and set up image streams, depending on where the images' registries are located, any authentication requirements around those registries, and how you want your builds and deployments to behave. The following sections cover a range of these topics.

11.2. TAGGING IMAGES

Before working with OpenShift Container Platform image streams and their tags, it will help to first understand image tags in the context of Docker generally.

Container images can have names added to them that make it more intuitive to determine what they contain, called a *tag*. Using a tag to specify the version of what is contained in the image is a common use case. If you have an image named **ruby**, you could have a tag named **2.0** for 2.0 version of Ruby, and another named **latest** to indicate literally the latest built image in that repository overall.

When interacting directly with images using the **`docker`** CLI, the **`docker tag`** command can add tags, which essentially adds an alias to an image that can consist of several parts. Those parts can include:

```
<registry_server>/<user_name>/<image_name>:<tag>
```

The **`<user_name>`** part in the above could also refer to a project or namespace if the image is being stored in an OpenShift Container Platform environment with an internal registry.

OpenShift Container Platform provides the **`oc tag`** command, which is similar to the **`docker tag`** command, but operates on image streams instead of directly on images.

Note

See Red Hat Enterprise Linux 7's Getting Started with Containers documentation for more about tagging images directly using the **`docker`** CLI.

11.2.1. Adding Tags to Image Streams

Keeping in mind that an image stream in OpenShift Container Platform comprises zero or more container images identified by tags, you can add tags to an image stream using the **`oc tag`** command:

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

For example, to configure the **ruby** image's **latest** tag to always refer to the current image for the tag **2.0**:

```
$ oc tag ruby:latest ruby:2.0
```

There are different types of tags available. The default behavior uses a *permanent* tag, which points to a specific image in time; even when the source changes, it will not reflect in the destination tag.

A *tracking* tag means the destination tag's metadata will be imported during the import. To ensure the destination tag is updated whenever the source tag changes, use the **`--alias=true`** flag:

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

You can also add the **`--scheduled=true`** flag to have the destination tag be refreshed (i.e., re-imported) periodically. The period is configured globally at system level. See Importing Tag and Image Metadata for more details.

Important

Avoid tagging OpenShift Container Platform-managed images (i.e., those built using an OpenShift Container Platform instance and pushed to its internal registry). There is a known issue that prevents the registry client from pulling from such a tag.

11.2.2. Tag Naming

Images evolve over time and the tag reflects this. It always points to the latest image built. If there is too much information embedded in a tag name (for example, **`v2.0.1-may-2016`**), the tag will point to just one revision of an image and will never be updated. Using default image pruning options, such an image will never be removed. Instead, if the tag is named **`v2.0`**, more image revisions are more likely. This results in longer tag history and, therefore, the image pruner will more likely remove old and unused images. Refer to pruning images for more information.

Although tag naming convention is up to you, here are a few examples:

Description	Example
Revision	**`v2.0.1`**
Architecture	**`v2.0-x86_64`**
Base image	**`v1.2-centos7`**

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

11.2.3. Removing Tags from Image Streams

To remove a tag completely from an image stream run:

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

or:

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

11.2.4. Referencing Images in Image Streams

Images can be referenced in image streams using the following reference types:

- An **ImageStreamTag** is used to reference or retrieve an image for a given image stream and tag. It uses the following convention for its name:

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

- An **ImageStreamImage** is used to reference or retrieve an image for a given image stream and image name. It uses the following convention for its name:

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

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

- A **DockerImage** is used to reference or retrieve an image for a given external registry. It uses standard Docker *pull specification* for its name, e.g.:

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

  Note

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

  You can also reference a third-party registry:

  ```
  registry.access.redhat.com/rhel7:latest
  ```

  Or an image with a digest:

  ```
  centos/ruby-22-
  centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c7
  46e8986b28e
  ```

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

This is because the **ImageStreamImage** objects are automatically created in OpenShift Container Platform whenever you import or tag an image into the image stream. You should never have to explicitly define an **ImageStreamImage** object in any image stream definition that you use to create image streams.

You can view an image's object definition by retrieving an **ImageStreamImage** definition using the image stream name and ID:

```
$ oc export isimage <image_stream_name>@<id>
```

Note

You can find valid **<id>** values for a given image stream by running:

```
$ oc describe is <image_stream_name>
```

For example, from the **ruby** image stream asking for the **ImageStreamImage** with the name and ID of **ruby@3a335d7**:

Example 11.1. Definition of an Image Object Retrieved via ImageStreamImage

```
$ oc export isimage ruby@3a335d7

apiVersion: v1
image:
  dockerImageLayers:
  - name:
sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46
d4
    size: 0
  - name:
sha256:ee1dd2cb6df21971f4af6de0f1d7782b81fb63156801cfde2bb47b4247c23c
29
    size: 196634330
  - name:
sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46
d4
    size: 0
  - name:
sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46
d4
    size: 0
  - name:
sha256:ca062656bff07f18bff46be00f40cfbb069687ec124ac0aa038fd676cfaea0
92
    size: 177723024
  - name:
sha256:63d529c59c92843c395befd065de516ee9ed4995549f8218eac6ff088bfa6b
6e
    size: 55679776
  dockerImageMetadata:
    Architecture: amd64
```

```
    Author: SoftwareCollections.org <sclorg@redhat.com>
    Config:
      Cmd:
      - /bin/sh
      - -c
      - $STI_SCRIPTS_PATH/usage
      Entrypoint:
      - container-entrypoint
      Env:
      - PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
      - STI_SCRIPTS_URL=image:///usr/libexec/s2i
      - STI_SCRIPTS_PATH=/usr/libexec/s2i
      - HOME=/opt/app-root/src
      - BASH_ENV=/opt/app-root/etc/scl_enable
      - ENV=/opt/app-root/etc/scl_enable
      - PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
      - RUBY_VERSION=2.2
      ExposedPorts:
        8080/tcp: {}
      Image:
d9c3abc5456a9461954ff0de8ae25e0e016aad35700594714d42b687564b1f51
      Labels:
        build-date: 2015-12-23
        io.k8s.description: Platform for building and running Ruby
2.2 applications
        io.k8s.display-name: Ruby 2.2
        io.openshift.builder-base-version: 8d95148
        io.openshift.builder-version:
8847438ba06307f86ac877465eadc835201241df
        io.openshift.expose-services: 8080:http
        io.openshift.s2i.scripts-url: image:///usr/libexec/s2i
        io.openshift.tags: builder,ruby,ruby22
        io.s2i.scripts-url: image:///usr/libexec/s2i
        license: GPLv2
        name: CentOS Base Image
        vendor: CentOS
      User: "1001"
      WorkingDir: /opt/app-root/src
    ContainerConfig: {}
    Created: 2016-01-26T21:07:27Z
    DockerVersion: 1.8.2-el7
    Id:
57b08d979c86f4500dc8cad639c9518744c8dd39447c055a3517dc9c18d6fccd
    Parent:
d9c3abc5456a9461954ff0de8ae25e0e016aad35700594714d42b687564b1f51
    Size: 430037130
    apiVersion: "1.0"
    kind: DockerImage
  dockerImageMetadataVersion: "1.0"
  dockerImageReference: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c74
6e8986b28e
  metadata:
    creationTimestamp: 2016-01-29T13:17:45Z
    name:
```

```
sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b2
8e
    resourceVersion: "352"
    uid: af2e7a0c-c68a-11e5-8a99-525400f25e34
kind: ImageStreamImage
metadata:
  creationTimestamp: null
  name: ruby@3a335d7
  namespace: openshift
  selflink:
/oapi/v1/namespaces/openshift/imagestreamimages/ruby@3a335d7
```

11.3. IMAGE PULL POLICY

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

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

- **`Always`** - always pull the image.
- **`IfNotPresent`** - only pull the image if it does not already exist on the node.
- **`Never`** - never pull the image.

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

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

11.4. ACCESSING THE INTERNAL REGISTRY

You can access OpenShift Container Platform's internal registry directly to push or pull images. For example, this could be helpful if you wanted to create an image stream by manually pushing an image, or just to **`docker pull`** an image directly.

The internal registry authenticates using the same tokens as the OpenShift Container Platform API. To perform a **`docker login`** against the internal registry, you can choose any user name and email, but the password must be a valid OpenShift Container Platform token.

To log into the internal registry:

1. Log in to OpenShift Container Platform:

   ```
   $ oc login
   ```

2. Get your access token:

```
$ oc whoami -t
```

3. Log in to the internal registry using the token. You must have **docker** installed on your system:

```
$ docker login -u <user_name> -e <email_address> \
    -p <token_value> <registry_server>:<port>
```

Note

Contact your cluster administrator if you do not know the registry IP or host name and port to use.

In order to pull an image, the authenticated user must have **`get`** rights on the requested **`imagestreams/layers`**. In order to push an image, the authenticated user must have **`update`** rights on the requested **`imagestreams/layers`**.

By default, all service accounts in a project have rights to pull any image in the same project, and the **builder** service account has rights to push any image in the same project.

11.5. USING IMAGE PULL SECRETS

Docker registries can be secured to prevent unauthorized parties from accessing certain images. If you are using OpenShift Container Platform's internal registry and are pulling from image streams located in the same project, then your pod's service account should already have the correct permissions and no additional action should be required.

However, for other scenarios, such as referencing images across OpenShift Container Platform projects or from secured registries, then additional configuration steps are required. The following sections detail these scenarios and their required steps.

11.5.1. Allowing Pods to Reference Images Across Projects

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

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

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

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

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

11.5.2. Allowing Pods to Reference Images from Other Secured Registries

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

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

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

```
$ oc secrets new <pull_secret_name> .dockercfg=<path/to/.dockercfg>
```

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

```
$ oc secrets new <pull_secret_name> .dockerconfigjson=
<path/to/.docker/config.json>
```

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

```
$ oc secrets new-dockercfg <pull_secret_name> \
    --docker-server=<registry_server> --docker-username=<user_name> \
    --docker-password=<password> --docker-email=<email>
```

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

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

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

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

11.6. IMPORTING TAG AND IMAGE METADATA

An image stream can be configured to import tag and image metadata from an image repository in an external Docker image registry. You can do this using a few different methods.

- You can manually import tag and image information with the **`oc import-image`** command using the **`--from`** option:

  ```
  $ oc import-image <image_stream_name>[:<tag>] --from=
  <docker_image_repo> --confirm
  ```

  For example:

  ```
  $ oc import-image my-ruby --from=docker.io/openshift/ruby-20-centos7
  --confirm
  The import completed successfully.

  Name:   my-ruby
  ```

```
Created:  Less than a second ago
Labels:   <none>
Annotations:  openshift.io/image.dockerRepositoryCheck=2016-05-
06T20:59:30Z
Docker Pull Spec: 172.30.94.234:5000/demo-project/my-ruby

Tag Spec     Created   PullSpec       Image
latest docker.io/openshift/ruby-20-centos7 Less than a second ago
docker.io/openshift/ruby-20-centos7@sha256:772c5bf9b2d1e8... <same>
```

You can also add the **`--all`** flag to import all tags for the image instead of just **latest**.

- Like most objects in OpenShift Container Platform, you can also write and save a JSON or YAML definition to a file then create the object using the CLI. Set the **`spec.dockerImageRepository`** field to the Docker pull spec for the image:

```
apiVersion: "v1"
kind: "ImageStream"
metadata:
  name: "my-ruby"
spec:
  dockerImageRepository: "docker.io/openshift/ruby-20-centos7"
```

Then create the object:

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

When you create an image stream that references an image in an external Docker registry, OpenShift Container Platform communicates with the external registry within a short amount of time to get up to date information about the image.

After the tag and image metadata is synchronized, the image stream object would look similar to the following:

```
apiVersion: v1
kind: ImageStream
metadata:
  name: my-ruby
  namespace: demo-project
  selflink: /oapi/v1/namespaces/demo-project/imagestreams/my-ruby
  uid: 5b9bd745-13d2-11e6-9a86-0ada84b8265d
  resourceVersion: '4699413'
  generation: 2
  creationTimestamp: '2016-05-06T21:34:48Z'
  annotations:
    openshift.io/image.dockerRepositoryCheck: '2016-05-06T21:34:48Z'
spec:
  dockerImageRepository: docker.io/openshift/ruby-20-centos7
  tags:
    -
      name: latest
      annotations: null
      from:
        kind: DockerImage
```

```
        name: 'docker.io/openshift/ruby-20-centos7:latest'
      generation: 2
      importPolicy: {  }
status:
  dockerImageRepository: '172.30.94.234:5000/demo-project/my-ruby'
  tags:
    -
      tag: latest
      items:
        -
          created: '2016-05-06T21:34:48Z'
          dockerImageReference: 'docker.io/openshift/ruby-20-
centos7@sha256:772c5bf9b2d1e8e80742ed75aab05820419dc4532fa6d7ad8a1efddd
a5493dc3'
          image:
'sha256:772c5bf9b2d1e8e80742ed75aab05820419dc4532fa6d7ad8a1efddda5493dc
3'
          generation: 2
```

You can set a tag to query external registries at a scheduled interval to synchronize tag and image metadata by setting the **`--scheduled=true`** flag with the **`oc tag`** command as mentioned in Adding Tags to Image Streams.

Alternatively, you can set **`importPolicy.scheduled`** to **true** in the tag's definition:

```
apiVersion: v1
kind: ImageStream
metadata:
  name: ruby
spec:
  tags:
  - from:
      kind: DockerImage
      name: openshift/ruby-20-centos7
    name: latest
    importPolicy:
      scheduled: true
```

11.6.1. Importing Images from Insecure Registries

An image stream can be configured to import tag and image metadata from insecure image registries, such as those signed with a self-signed certificate or using plain HTTP instead of HTTPS.

To configure this, add the **`openshift.io/image.insecureRepository`** annotation and set it to **true**. This setting bypasses certificate validation when connecting to the registry:

```
kind: ImageStream
apiVersion: v1
metadata:
  name: ruby
```

```
  annotations:
    openshift.io/image.insecureRepository: "true" (1)
  spec:
    dockerImageRepository: my.repo.com:5000/myimage
```

1

Set the **openshift.io/image.insecureRepository** annotation to **true**

Important

The above definition only affects importing tag and image metadata. For this image to be used in the cluster (e.g., to be able to do a **docker pull**), each node must have Docker configured with the **--insecure-registry** flag. See Host Preparation for information.

Additionally, you can specify a single tag using an insecure repository. To do so, set **importPolicy.insecure** in the tag's definition to **true**:

```
kind: ImageStream
apiVersion: v1
metadata:
  name: ruby
  tags:
  - from:
      kind: DockerImage
      name: my.repo.com:5000/myimage
    name: mytag
    importPolicy:
      insecure: true (1)
```

1

Set tag **mytag** to use insecure connection to that registry.

11.6.2. Importing Images from Private Registries

An image stream can be configured to import tag and image metadata from private image registries, requiring authentication.

To configure this, you need to create a secret which is used to store your credentials.

Create the secret first, before importing the image from the private repository:

```
$ oc secrets new-dockercfg <secret_name> \
    --docker-server=<docker_registry_server> \
    --docker-username=<docker_user> \
    --docker-password=<docker_password> \
    --docker-email=<docker_email>
```

For more options, see:

```
$ oc secrets new-dockercfg --help
```

After the secret is configured, proceed with creating the new image stream or using the **oc import-image** command. During the import process, OpenShift Container Platform will pick up the secrets and provide them to the remote party.

11.6.3. Importing Images Across Projects

An image stream can be configured to import tag and image metadata from the internal registry, but from a different project. The recommended method for this is to use the **oc tag** command as shown in Adding Tags to Image Streams:

```
$ oc tag <source_project>/<image_stream>:<tag> <new_image_stream>:
<new_tag>
```

Another method is to import the image from the other project manually using the pull spec:

Warning

The following method is strongly discouraged and should be used only if the former using **oc tag** is insufficient.

1. First, add the necessary policy to access the other project:

   ```
   $ oc policy add-role-to-group \
       system:image-puller \
       system:serviceaccounts:<destination_project> \
       -n <source_project>
   ```

   This allows **<destination_project>** to pull images from **<source_project>**.

2. With the policy in place, you can import the image manually:

   ```
   $ oc import-image <new_image_stream> --confirm \
       --from=<docker_registry>/<source_project>/<image_stream>
   ```

11.6.4. Creating an Image Stream by Manually Pushing an Image

An image stream can also be automatically created by manually pushing an image to the internal registry. This is only possible when using an OpenShift Container Platform internal registry.

Before performing this procedure, the following must be satisfied:

- The destination project you push to must already exist.
- The user must be authorized to **{get, update} "imagestream/layers"** in that project. The **system:image-pusher** role can be added to a user to provide these permissions. If you are a project administrator, then you would also have these permissions.

To create an image stream by manually pushing an image:

1. First, log in to the internal registry.
2. Then, tag your image using the appropriate internal registry location. For example, if you had already pulled the **docker.io/centos:centos7** image locally:

```
$ docker tag docker.io/centos:centos7 172.30.48.125:5000/test/my-
image
```

3. Finally, push the image to your internal registry. For example:

```
$ docker push 172.30.48.125:5000/test/my-image
The push refers to a repository [172.30.48.125:5000/test/my-
image] (len: 1)
c8a648134623: Pushed
2bf4902415e3: Pushed
latest: digest:
sha256:be8bc4068b2f60cf274fc216e4caba6aa845fff5fa29139e6e7497bb57
e48d67 size: 6273
```

4. Verify that the image stream was created:

```
$ oc get is
NAME       DOCKER REPO                          TAGS      UPDATED
my-image   172.30.48.125:5000/test/my-image   latest    3
seconds ago
```

11.7. WRITING IMAGE STREAMS FOR S2I BUILDERS

Image streams for S2I builders that are displayed in the management console's catalog page require additional metadata to provide the best experience for end users.

Example 11.2. Definition of an Image Stream Object with Catalog Metadata

```
apiVersion: v1
kind: ImageStream
metadata:
  name: ruby
  annotations:
    openshift.io/display-name: Ruby 1
spec:
  tags:
    - name: '2.0' 2
      annotations:
        openshift.io/display-name: Ruby 2.0 3
        description: >- 4
          Build and run Ruby 2.0 applications on CentOS 7. For more
information
```

```
          about using this builder image, including OpenShift
considerations,
          see
          https://github.com/sclorg/s2i-ruby-
container/tree/master/2.0/README.md.
        iconClass: icon-ruby 5
        sampleRepo: 'https://github.com/openshift/ruby-ex.git' 6
        tags: 'builder,ruby' 7
        version: '2.0' 8
      from:
        kind: DockerImage
        name: 'openshift/ruby-20-centos7:latest'
```

1

A brief, user-friendly name for the whole image stream.

2

The tag is referred to as the version. Tags appear in a drop-down menu.

3

A user-friendly name for this tag within the image stream. This should be brief and include version information when appropriate.

4

A description of the tag, which includes enough detail for users to understand what the image is providing. It can include links to additional instructions. Limit the description to a few sentences.

5

The icon to show for this tag. Pick from our existing logo icons when possible. Icons from FontAwesome and Patternfly can also be used. Alternatively, provide icons through CSS customizations that can be added to an OpenShift Container Platform cluster that uses your image stream. You must specify an icon class that exists, or it will prevent falling back to the generic icon.

6

A URL to a source repository that works with this builder image tag and results in a sample running application.

7

Categories that the image stream tag is associated with. The builder tag is required for it to show up in the catalog. Add tags that will associate it with one of the provided catalog categories. Refer to the **`id`** and **`categoryAliases`** in **`CATALOG_CATEGORIES`** in the console's constants file. The categories can also be customized for the whole cluster.

8

Version information for this tag.

CHAPTER 12. QUOTAS AND LIMIT RANGES

12.1. OVERVIEW

Using quotas and limit ranges, cluster administrators can set constraints to limit the number of objects or amount of compute resources that are used in your project. This helps cluster administrators better manage and allocate resources across all projects, and ensure that no projects are using more than is appropriate for the cluster size.

As a developer, you can also set requests and limits on compute resources at the pod and container level.

The following sections help you understand how to check on your quota and limit range settings, what sorts of things they can constrain, and how you can request or limit compute resources in your own pods and containers.

12.2. QUOTAS

A resource quota, defined by a **`ResourceQuota`** object, provides constraints that limit aggregate resource consumption per project. It can limit the quantity of objects that can be created in a project by type, as well as the total amount of compute resources and storage that may be consumed by resources in that project.

Note

Quotas are set by cluster administrators and are scoped to a given project.

12.2.1. Viewing Quotas

You can view usage statistics related to any hard limits defined in a project's quota by navigating in the web console to the project's **Settings** tab.

You can also use the CLI to view quota details:

1. First, get the list of quotas defined in the project. For example, for a project called **demoproject**:

```
$ oc get quota -n demoproject
NAME                AGE
besteffort          11m
compute-resources   2m
core-object-counts  29m
```

2. Then, describe the quota you are interested in, for example the **core-object-counts** quota:

```
$ oc describe quota core-object-counts -n demoproject
Name:   core-object-counts
Namespace:  demoproject
Resource  Used Hard
--------  ---- ----
```

```
configmaps  3 10
persistentvolumeclaims 0 4
replicationcontrollers 3 20
secrets   9 10
services  2 10
```

Full quota definitions can be viewed by running **`oc export`** on the object. The following show some sample quota definitions:

Example 12.1. ***object-counts.yaml***

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: core-object-counts
spec:
  hard:
    configmaps: "10" 1
    persistentvolumeclaims: "4" 2
    replicationcontrollers: "20" 3
    secrets: "10" 4
    services: "10" 5
```

1 The total number of **`ConfigMap`** objects that can exist in the project.

2 The total number of persistent volume claims (PVCs) that can exist in the project.

3 The total number of replication controllers that can exist in the project.

4 The total number of secrets that can exist in the project.

5 The total number of services that can exist in the project.

Example 12.2. ***openshift-object-counts.yaml***

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: openshift-object-counts
spec:
  hard:
    openshift.io/imagestreams: "10" (1)
```

(1)

The total number of image streams that can exist in the project.

Example 12.3. *compute-resources.yaml*

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4" (1)
    requests.cpu: "1" (2)
    requests.memory: 1Gi (3)
    limits.cpu: "2" (4)
    limits.memory: 2Gi (5)
```

(1)

The total number of pods in a non-terminal state that can exist in the project.

(2)

Across all pods in a non-terminal state, the sum of CPU requests cannot exceed 1 core.

(3)

Across all pods in a non-terminal state, the sum of memory requests cannot exceed 1Gi.

(4)

Across all pods in a non-terminal state, the sum of CPU limits cannot exceed 2 cores.

(5)

Across all pods in a non-terminal state, the sum of memory limits cannot exceed 2Gi.

Example 12.4. ***besteffort.yaml***

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort
spec:
  hard:
    pods: "1" 1
  scopes:
  - BestEffort 2
```

1

The total number of pods in a non-terminal state with **BestEffort** quality of service that can exist in the project.

2

Restricts the quota to only matching pods that have **BestEffort** quality of service for either memory or CPU.

Example 12.5. ***compute-resources-long-running.yaml***

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-long-running
spec:
  hard:
    pods: "4" 1
    limits.cpu: "4" 2
    limits.memory: "2Gi" 3
  scopes:
  - NotTerminating 4
```

1

The total number of pods in a non-terminal state.

2

Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.

3

Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.

4

Restricts the quota to only matching pods where **`spec.activeDeadlineSeconds is nil`**. For example, this quota would not charge for build or deployer pods.

Example 12.6. *compute-resources-time-bound.yaml*

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-time-bound
spec:
  hard:
    pods: "2" 1
    limits.cpu: "1" 2
    limits.memory: "1Gi" 3
  scopes:
  - Terminating 4
```

1

The total number of pods in a non-terminal state.

2

Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.

3

Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.

4

Restricts the quota to only matching pods where **`spec.activeDeadlineSeconds >=0`**. For example, this quota would charge for build or deployer pods, but not long running pods like a web server or database.

Example 12.7. storage-consumption.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage-consumption
spec:
  hard:
    persistentvolumeclaims: "10" (1)
    requests.storage: "50Gi" (2)
```

(1) The total number of persistent volume claims in a project

(2) Across all persistent volume claims in a project, the sum of storage requested cannot exceed this value.

12.2.2. Resources Managed by Quota

The following describes the set of compute resources and object types that may be managed by a quota.

Table 12.1. Compute Resources Managed by Quota

Resource Name	Description
`cpu`	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value. **`cpu`** and **`requests.cpu`**are the same value and can be used interchangeably.
`memory`	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value. **`memory`** and **`requests.memory`** are the same value and can be used interchangeably.
`requests.cpu`	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value. **`cpu`** and **`requests.cpu`**are the same value and can be used interchangeably.

Resource Name	Description
`requests.memory`	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value. **`memory`** and **`requests.memory`** are the same value and can be used interchangeably.
`requests.storage`	Across all persistent volume claim in any state, the sum of storage requests cannot exceed this value. **`storage`** and **`requests.storage`** are the same value and can be used interchangeably.
`limits.cpu`	Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
`limits.memory`	Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
`limits.storage`	Across all persistent volume claims, the sum of storage limits cannot exceed this value.

Table 12.2. Object Counts Managed by Quota

Resource Name	Description
`pods`	The total number of pods in a non-terminal state that can exist in the project. A pod is in a terminal state if **`status.phase in (Failed, Succeeded)`** is true.
`replicationcontrollers`	The total number of replication controllers that can exist in the project.
`resourcequotas`	The total number of resource quotas that can exist in the project.
`services`	The total number of services that can exist in the project.
`secrets`	The total number of secrets that can exist in the project.

Resource Name	Description
`configmaps`	The total number of `ConfigMap` objects that can exist in the project.
`persistentvolumeclaims`	The total number of persistent volume claims that can exist in the project.
`openshift.io/imagestreams`	The total number of image streams that can exist in the project.

12.2.3. Quota Scopes

Each quota can have an associated set of *scopes*. A quota will only measure usage for a resource if it matches the intersection of enumerated scopes.

When a scope is added to a quota, it limits the number of resources it supports to those that pertain to the scope. Resources specified on the quota outside of the allowed set results in a validation error.

Scope	Description
Terminating	Match pods where `spec.activeDeadlineSeconds >= 0`
NotTerminating	Match pods where `spec.activeDeadlineSeconds is nil`
BestEffort	Match pods that have best effort quality of service for either `cpu` or `memory`.
NotBestEffort	Match pods that do not have best effort quality of service for `cpu` and `memory`.

A **BestEffort** scope restricts a quota to limit the following resources:

- `pods`

A **Terminating**, **NotTerminating**, and **NotBestEffort** scope restricts a quota to tracking the following resources:

- `pods`
- `memory`

- **`requests.memory`**
- **`limits.memory`**
- **`cpu`**
- **`requests.cpu`**
- **`limits.cpu`**

12.2.4. Quota Enforcement

After a resource quota for a project is first created, the project restricts the ability to create any new resources that may violate a quota constraint until it has calculated updated usage statistics.

After a quota is created and usage statistics are updated, the project accepts the creation of new content. When you create or modify resources, your quota usage is incremented immediately upon the request to create or modify the resource.

When you delete a resource, your quota use is decremented during the next full recalculation of quota statistics for the project. If project modifications exceed a quota usage limit, the server denies the action. An appropriate error message is returned explaining the quota constraint violated, and what your currently observed usage stats are in the system.

12.2.5. Requests vs Limits

When allocating compute resources, each container may specify a request and a limit value for either CPU or memory. The quota can be configured to quota either value.

If the quota has a value specified for **`requests.cpu`** or **`requests.memory`**, then it requires that every incoming container makes an explicit request for those resources. If the quota has a value specified for **`limits.cpu`** or **`limits.memory`**, then it requires that every incoming container specifies an explicit limit for those resources.

See Compute Resources for more on setting requests and limits in pods and containers.

12.3. LIMIT RANGES

A limit range, defined by a **`LimitRange`** object, enumerates compute resource constraints in a project at the pod, container, image, image stream, and persistent volume claim level, and specifies the amount of resources that a pod, container, image, image stream, or persistent volume claim can consume.

All resource create and modification requests are evaluated against each **`LimitRange`** object in the project. If the resource violates any of the enumerated constraints, then the resource is rejected. If the resource does not set an explicit value, and if the constraint supports a default value, then the default value is applied to the resource.

Note

Limit ranges are set by cluster administrators and are scoped to a given project.

12.3.1. Viewing Limit Ranges

You can view any limit ranges defined in a project by navigating in the web console to the project's **Settings** tab.

You can also use the CLI to view limit range details:

1. First, get the list of limit ranges defined in the project. For example, for a project called **demoproject**:

```
$ oc get limits -n demoproject
NAME              AGE
resource-limits   6d
```

2. Then, describe the limit range you are interested in, for example the **resource-limits** limit range:

```
$ oc describe limits resource-limits
Name:                   limits
Namespace:              default
Type                    Resource                Min  Max
Request Limit Limit/Request
----                    --------                ---  --- --
----- ----- -------------
Pod                     memory                  6Mi  1Gi -
-     -
Pod                     cpu                     200m  2  -
-     -
Container               cpu                     100m  2
200m    300m  10
Container               memory                  4Mi  1Gi
100Mi   200Mi -
openshift.io/Image      storage                 -    1Gi -
-     -
openshift.io/ImageStream  openshift.io/image-tags  -    10  -
-     -
openshift.io/ImageStream  openshift.io/images     -    12  -
-     -
```

Full limit range definitions can be viewed by running **`oc export`** on the object. The following shows an example limit range definition:

Example 12.8. Limit Range Object Definition

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "core-resource-limits" 1
spec:
  limits:
    - type: "Pod"
      max:
        cpu: "2" 2
```

```
      memory: "1Gi" 3
    min:
      cpu: "200m" 4
      memory: "6Mi" 5
  - type: "Container"
    max:
      cpu: "2" 6
      memory: "1Gi" 7
    min:
      cpu: "100m" 8
      memory: "4Mi" 9
    default:
      cpu: "300m" 10
      memory: "200Mi" 11
    defaultRequest:
      cpu: "200m" 12
      memory: "100Mi" 13
    maxLimitRequestRatio:
      cpu: "10" 14
```

1

The name of the limit range document.

2

The maximum amount of CPU that a pod can request on a node across all containers.

3

The maximum amount of memory that a pod can request on a node across all containers.

4

The minimum amount of CPU that a pod can request on a node across all containers.

5

The minimum amount of memory that a pod can request on a node across all containers.

6

The maximum amount of CPU that a single container in a pod can request.

7

The maximum amount of memory that a single container in a pod can request.

8

The minimum amount of CPU that a single container in a pod can request.

9

The minimum amount of memory that a single container in a pod can request.

10

The default amount of CPU that a container will be limited to use if not specified.

11

The default amount of memory that a container will be limited to use if not specified.

12

The default amount of CPU that a container will request to use if not specified.

13

The default amount of memory that a container will request to use if not specified.

14

The maximum amount of CPU burst that a container can make as a ratio of its limit over request.

12.3.2. Container Limits

Supported Resources:

- CPU
- Memory

Supported Constraints:

Per container, the following must hold true if specified:

Table 12.3. Container

Constraint	Behavior
`Min`	`Min[resource]` less than or equal to `container.resources.requests[resource]` (required) less than or equal to `container/resources.limits[resource]` (optional) If the configuration defines a `min` CPU, then the request value must be greater than the CPU value. A limit value does not need to be specified.
`Max`	`container.resources.limits[resource]` (required) less than or equal to `Max[resource]` If the configuration defines a `max` CPU, then you do not need to define a request value, but a limit value does need to be set that satisfies the maximum CPU constraint.
`MaxLimitRequestRatio`	`MaxLimitRequestRatio[resource]` less than or equal to (`container.resources.limits[resource]` / `container.resources.requests[resource]`) If a configuration defines a `maxLimitRequestRatio` value, then any new containers must have both a request and limit value. Additionally, OpenShift Container Platform calculates a limit to request ratio by dividing the limit by the request. For example, if a container has `cpu: 500` in the `limit` value, and `cpu: 100` in the `request` value, then its limit to request ratio for `cpu` is `5`. This ratio must be less than or equal to the `maxLimitRequestRatio`.

Supported Defaults:

`Default[resource]`

Defaults `container.resources.limit[resource]` to specified value if none.

`Default Requests[resource]`

Defaults `container.resources.requests[resource]` to specified value if none.

12.3.3. Pod Limits

Supported Resources:

- CPU
- Memory

Supported Constraints:

Across all containers in a pod, the following must hold true:

Table 12.4. Pod

Constraint	Enforced Behavior
`Min`	`Min[resource]` less than or equal to `container.resources.requests[resource]` (required) less than or equal to `container.resources.limits[resource]` (optional)
`Max`	`container.resources.limits[resource]` (required) less than or equal to `Max[resource]`
`MaxLimitRequestRatio`	`MaxLimitRequestRatio[resource]` less than or equal to (`container.resources.limits[resource]` / `container.resources.requests[resource]`)

12.4. COMPUTE RESOURCES

Each container running on a node consumes compute resources, which are measurable quantities that can be requested, allocated, and consumed.

When authoring a pod configuration file, you can optionally specify how much CPU and memory (RAM) each container needs in order to better schedule pods in the cluster and ensure satisfactory performance.

CPU is measured in units called millicores. Each node in a cluster inspects the operating system to determine the amount of CPU cores on the node, then multiplies that value by 1000 to express its total capacity. For example, if a node has 2 cores, the node's CPU capacity would be represented as 2000m. If you wanted to use 1/10 of a single core, it would be represented as 100m.

Memory is measured in bytes. In addition, it may be used with SI suffices (E, P, T, G, M, K, m) or their power-of-two-equivalents (Ei, Pi, Ti, Gi, Mi, Ki).

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - image: nginx
    name: nginx
    resources:
      requests:
        cpu: 100m 1
        memory: 200Mi 2
      limits:
        cpu: 200m 3
        memory: 400Mi 4
```

1

The container requests 100m cpu.

2

The container requests 200Mi memory.

3

The container limits 200m cpu.

4

The container limits 400Mi memory.

12.4.1. CPU Requests

Each container in a pod can specify the amount of CPU it requests on a node. The scheduler uses CPU requests to find a node with an appropriate fit for a container.

The CPU request represents a minimum amount of CPU that your container may consume, but if there is no contention for CPU, it can use all available CPU on the node. If there is CPU contention on the node, CPU requests provide a relative weight across all containers on the system for how much CPU time the container may use.

On the node, CPU requests map to Kernel CFS shares to enforce this behavior.

12.4.2. Viewing Compute Resources

To view compute resources for a pod:

```
$ oc describe pod nginx-tfjxt
Name:       nginx-tfjxt
Namespace:      default
Image(s):     nginx
Node:       /
Labels:       run=nginx
Status:       Pending
Reason:
Message:
IP:
Replication Controllers:  nginx (1/1 replicas created)
Containers:
  nginx:
    Container ID:
    Image:    nginx
    Image ID:
    QoS Tier:
      cpu:  Burstable
```

```
      memory: Burstable
    Limits:
      cpu:  200m
      memory: 400Mi
    Requests:
      cpu:    100m
      memory:   200Mi
    State:    Waiting
    Ready:    False
    Restart Count:  0
    Environment Variables:
```

12.4.3. CPU Limits

Each container in a pod can specify the amount of CPU it is limited to use on a node. CPU limits control the maximum amount of CPU that your container may use independent of contention on the node. If a container attempts to exceed the specified limit, the system will throttle the container. This allows the container to have a consistent level of service independent of the number of pods scheduled to the node.

12.4.4. Memory Requests

By default, a container is able to consume as much memory on the node as possible. In order to improve placement of pods in the cluster, specify the amount of memory required for a container to run. The scheduler will then take available node memory capacity into account prior to binding your pod to a node. A container is still able to consume as much memory on the node as possible even when specifying a request.

12.4.5. Memory Limits

If you specify a memory limit, you can constrain the amount of memory the container can use. For example, if you specify a limit of 200Mi, a container will be limited to using that amount of memory on the node. If the container exceeds the specified memory limit, it will be terminated and potentially restarted dependent upon the container restart policy.

12.4.6. Quality of Service Tiers

A compute resource is classified with a *quality of service* (QoS) based on the specified request and limit value.

Quality of Service	Description
BestEffort	Provided when a request and limit are not specified.
Burstable	Provided when a request is specified that is less than an optionally specified limit.

Quality of Service	Description
Guaranteed	Provided when a limit is specified that is equal to an optionally specified request.

A container may have a different quality of service for each compute resource. For example, a container can have **Burstable** CPU and **Guaranteed** memory qualities of service.

The quality of service has an impact on what happens if the resource is compressible or not. CPU is a compressible resource, whereas memory is an incompressible resource.

With CPU Resources:

- A **BestEffort** CPU container is able to consume as much CPU as is available on a node with the lowest priority.
- A **Burstable** CPU container is guaranteed to get the minimum amount of CPU requested, but it may or may not get additional CPU time. Excess CPU resources are distributed based on the amount requested across all containers on the node.
- A **Guaranteed** CPU container is guaranteed to get the amount requested and no more, even if there are additional CPU cycles available. This provides a consistent level of performance independent of other activity on the node.

With Memory Resources:

- A **BestEffort** memory container is able to consume as much memory as is available on the node, but the scheduler is subject to placing that container on a node with too few memory to meet a need. In addition, a **BestEffort** memory container has the greatest chance of being killed if there is an out of memory event on the node.
- A **Burstable** memory container is scheduled on the node to get the amount of memory requested, but it may consume more. If there is an out of memory event on the node, **Burstable** containers are killed after **BestEffort** containers when attempting to recover memory.
- A **Guaranteed** memory container gets the amount of memory requested, but no more. In the event of an out of memory event, it will only be killed if there are no more **BestEffort** or **Burstable** memory containers on the system.

12.4.7. Specifying Compute Resources via CLI

To specify compute resources via the CLI:

```
$ oc run nginx --image=nginx --limits=cpu=200m,memory=400Mi --
requests=cpu=100m,memory=200Mi
```

12.5. PROJECT RESOURCE LIMITS

Resource limits can be set per-project by cluster administrators. Developers do not have the ability to create, edit, or delete these limits, but can view them for projects they have access to.

CHAPTER 13. DEPLOYMENTS

13.1. HOW DEPLOYMENTS WORK

13.1.1. What Is a Deployment?

OpenShift Container Platform deployments provide fine-grained management over common user applications. They are described using three separate API objects:

- A deployment configuration, which describes the desired state of a particular component of the application as a pod template.
- One or more replication controllers, which contain a point-in-time record of the state of a deployment configuration as a pod template.
- One or more pods, which represent an instance of a particular version of an application.

Important

Users do not need to manipulate replication controllers or pods owned by deployment configurations. The deployment system ensures changes to deployment configurations are propagated appropriately. If the existing deployment strategies are not suited for your use case and you have the need to run manual steps during the lifecycle of your deployment, then you should consider creating a custom strategy.

When you create a deployment configuration, a replication controller is created representing the deployment configuration's pod template. If the deployment configuration changes, a new replication controller is created with the latest pod template, and a deployment process runs to scale down the old replication controller and scale up the new replication controller.

Instances of your application are automatically added and removed from both service load balancers and routers as they are created. As long as your application supports graceful shutdown when it receives the **TERM** signal, you can ensure that running user connections are given a chance to complete normally.

Features provided by the deployment system:

- A deployment configuration, which is a template for running applications.
- Triggers that drive automated deployments in response to events.
- User-customizable strategies to transition from the previous version to the new version. A strategy runs inside a pod commonly referred as the deployment process.
- A set of hooks for executing custom behavior in different points during the lifecycle of a deployment.
- Versioning of your application in order to support rollbacks either manually or automatically in case of deployment failure.
- Manual replication scaling and autoscaling.

13.1.2. Creating a Deployment Configuration

Deployment configurations are **deploymentConfig** OpenShift Container Platform API resources which can be managed with the **oc** command like any other resource. The following is an example of a **deploymentConfig** resource:

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "frontend"
spec:
  template: 1
    metadata:
      labels:
        name: "frontend"
    spec:
      containers:
        - name: "helloworld"
          image: "openshift/origin-ruby-sample"
          ports:
            - containerPort: 8080
              protocol: "TCP"
  replicas: 5 2
  triggers:
    - type: "ConfigChange" 3
    - type: "ImageChange" 4
      imageChangeParams:
        automatic: true
        containerNames:
          - "helloworld"
        from:
          kind: "ImageStreamTag"
          name: "origin-ruby-sample:latest"
  strategy: 5
    type: "Rolling"
  paused: false 6
  revisionHistoryLimit: 2 7
  minReadySeconds: 0 8
```

1

The pod template of the **frontend** deployment configuration describes a simple Ruby application.

2

There will be 5 replicas of **frontend**.

3

A configuration change trigger causes a new replication controller to be created any time the

pod template changes.

4

An image change trigger trigger causes a new replication controller to be created each time a new version of the **`origin-ruby-sample:latest`** image stream tag is available.

5

The Rolling strategy is the default way of deploying your pods. May be omitted.

6

Pause a deployment configuration. This disables the functionality of all triggers and allows for multiple changes on the pod template before actually rolling it out.

7

Revision history limit is the limit of old replication controllers you want to keep around for rolling back. May be omitted. If omitted, old replication controllers will not be cleaned up.

8

Minimum seconds to wait (after the readiness checks succeed) for a pod to be considered available. The default value is 0.

13.2. BASIC DEPLOYMENT OPERATIONS

13.2.1. Starting a Deployment

You can start a new deployment process manually using the web console, or from the CLI:

```
$ oc deploy --latest dc/<name>
```

Note

If a deployment process is already in progress, the command will display a message and a new replication controller will not be deployed.

13.2.2. Viewing a Deployment

To get basic information about all the available revisions of your application:

```
$ oc rollout history dc/<name>
```

This will show details about all recently created replication controllers for the provided deployment configuration, including any currently running deployment process.

You can view details specific to a revision by using the **`--revision`** flag:

```
$ oc rollout history dc/<name> --revision=1
```

For more detailed information about a deployment configuration and its latest revision:

```
$ oc describe dc <name>
```

Note

The web console shows deployments in the **Browse** tab.

13.2.3. Canceling a Deployment

To cancel a running or stuck deployment process:

```
$ oc deploy --cancel dc/<name>
```

Warning

The cancellation is a best-effort operation, and may take some time to complete. The replication controller may partially or totally complete its deployment before the cancellation is effective. When canceled, the deployment configuration will be automatically rolled back by scaling up the previous running replication controller.

13.2.4. Retrying a Deployment

If the current revision of your deployment configuration failed to deploy, you can restart the deployment process with:

```
$ oc deploy --retry dc/<name>
```

If the latest revision of it was deployed successfully, the command will display a message and the deployment process will not be retried.

Note

Retrying a deployment restarts the deployment process and does not create a new deployment revision. The restarted replication controller will have the same configuration it had when it failed.

13.2.5. Rolling Back a Deployment

Rollbacks revert an application back to a previous revision and can be performed using the REST API, the CLI, or the web console.

To rollback to the last successful deployed revision of your configuration:

```
$ oc rollout undo dc/<name>
```

The deployment configuration's template will be reverted to match the deployment revision specified in the undo command, and a new replication controller will be started. If no revision is specified with **`--to-revision`**, then the last successfully deployed revision will be used.

Image change triggers on the deployment configuration are disabled as part of the rollback to prevent accidentally starting a new deployment process soon after the rollback is complete. To re-enable the image change triggers:

```
$ oc set triggers dc/<name> --auto
```

Note

Deployment configurations also support automatically rolling back to the last successful revision of the configuration in case the latest deployment process fails. In that case, the latest template that failed to deploy stays intact by the system and it is up to users to fix their configurations.

13.2.6. Executing Commands Inside a Container

You can add a command to a container, which modifies the container's startup behavior by overruling the image's **`ENTRYPOINT`**. This is different from a lifecycle hook, which instead can be run once per deployment at a specified time.

Add the **`command`** parameters to the **`spec`** field of the deployment configuration. You can also add an **`args`** field, which modifies the **`command`** (or the **`ENTRYPOINT`** if **`command`** does not exist).

```
...
spec:
  containers:
    -
    name: <container_name>
    image: 'image'
    command:
      - '<command>'
    args:
      - '<argument_1>'
      - '<argument_2>'
      - '<argument_3>'
...
```

For example, to execute the **`java`** command with the **`-jar`** and ***/opt/app-root/springboots2idemo.jar*** arguments:

```
...
spec:
  containers:
    -
```

```
    name: example-spring-boot
    image: 'image'
    command:
      - java
    args:
      - '-jar'
      - /opt/app-root/springboots2idemo.jar
...
```

13.2.7. Viewing Deployment Logs

To stream the logs of the latest revision for a given deployment configuration:

```
$ oc logs -f dc/<name>
```

If the latest revision is running or failed, **`oc logs`** will return the logs of the process that is responsible for deploying your pods. If it is successful, **`oc logs`** will return the logs from a pod of your application.

You can also view logs from older failed deployment processes, if and only if these processes (old replication controllers and their deployer pods) exist and have not been pruned or deleted manually:

```
$ oc logs --version=1 dc/<name>
```

For more options on retrieving logs see:

```
$ oc logs --help
```

13.2.8. Setting Deployment Triggers

A deployment configuration can contain triggers, which drive the creation of new deployment processes in response to events inside the cluster.

Warning

If no triggers are defined on a deployment configuration, a **`ConfigChange`** trigger is added by default. If triggers are defined as an empty field, deployments must be started manually.

13.2.8.1. Configuration Change Trigger

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

Note

If a **`ConfigChange`** trigger is defined on a deployment configuration, the first replication controller will be automatically created soon after the deployment configuration itself is created and it is not paused.

Example 13.1. A ConfigChange Trigger

```
triggers:
  - type: "ConfigChange"
```

13.2.8.2. ImageChange Trigger

The **ImageChange** trigger results in a new replication controller whenever the content of an image stream tag changes (when a new version of the image is pushed).

Example 13.2. An ImageChange Trigger

```
triggers:
  - type: "ImageChange"
    imageChangeParams:
      automatic: true 1
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
        namespace: "myproject"
      containerNames:
        - "helloworld"
```

1 If the **imageChangeParams.automatic** field is set to **false**, the trigger is disabled.

With the above example, when the **latest** tag value of the **origin-ruby-sample** image stream changes and the new image value differs from the current image specified in the deployment configuration's **helloworld** container, a new replication controller is created using the new image for the **helloworld** container.

Note

If an **ImageChange** trigger is defined on a deployment configuration (with a **ConfigChange** trigger and **automatic=false**, or with **automatic=true**) and the **ImageStreamTag** pointed by the **ImageChange** trigger does not exist yet, then the initial deployment process will automatically start as soon as an image is imported or pushed by a build to the **ImageStreamTag**.

13.2.8.2.1. Using the Command Line

The **oc set triggers** command can be used to set a deployment trigger for a deployment configuration. For the example above, you can set the **ImageChangeTrigger** by using the following command:

```
$ oc set triggers dc/frontend --from-image=myproject/origin-ruby-
sample:latest -c helloworld
```

For more information, see:

```
$ oc set triggers --help
```

13.2.9. Setting Deployment Resources

A deployment is completed by a pod that consumes resources (memory and CPU) on a node. By default, pods consume unbounded node resources. However, if a project specifies default container limits, then pods consume resources up to those limits.

You can also limit resource use by specifying resource limits as part of the deployment strategy. Deployment resources can be used with the Recreate, Rolling, or Custom deployment strategies.

In the following example, each of **`resources`**, **`cpu`**, and **`memory`** is optional:

```
type: "Recreate"
resources:
  limits:
    cpu: "100m" 1
    memory: "256Mi" 2
```

1

`cpu` is in CPU units: **`100m`** represents 0.1 CPU units (100 * 1e-3).

2

`memory` is in bytes: **`256Mi`** represents 268435456 bytes (256 * 2 ^ 20).

However, if a quota has been defined for your project, one of the following two items is required:

- A **`resources`** section set with an explicit **`requests`**:

```
type: "Recreate"
resources:
  requests: 1
    cpu: "100m"
    memory: "256Mi"
```

1

The **`requests`** object contains the list of resources that correspond to the list of resources in the quota.

- A limit range defined in your project, where the defaults from the **`LimitRange`** object apply to pods created during the deployment process.

Otherwise, deploy pod creation will fail, citing a failure to satisfy quota.

13.2.10. Manual Scaling

In addition to rollbacks, you can exercise fine-grained control over the number of replicas from the web console, or by using the **`oc scale`** command. For example, the following command sets the replicas in the deployment configuration **`frontend`** to 3.

```
$ oc scale dc frontend --replicas=3
```

The number of replicas eventually propagates to the desired and current state of the deployment configured by the deployment configuration **`frontend`**.

Note

Pods can also be autoscaled using the **`oc autoscale`** command. See Pod Autoscaling for more details.

13.2.11. Assigning Pods to Specific Nodes

You can use node selectors in conjunction with labeled nodes to control pod placement.

Note

OpenShift Container Platform administrators can assign labels during an advanced installation, or added to a node after installation.

Cluster administrators can set the default node selector for your project in order to restrict pod placement to specific nodes. As an OpenShift Container Platform developer, you can set a node selector on a pod configuration to restrict nodes even further.

To add a node selector when creating a pod, edit the pod configuration, and add the **`nodeSelector`** value. This can be added to a single pod configuration, or in a pod template:

```
apiVersion: v1
kind: Pod
spec:
  nodeSelector:
    disktype: ssd
...
```

Pods created when the node selector is in place are assigned to nodes with the specified labels.

The labels specified here are used in conjunction with the labels added by a cluster administrator. For example, if a project has the **`type=user-node`** and **`region=east`** labels added to a project by the cluster administrator, and you add the above **`disktype: ssd`** label to a pod, the pod will only ever be scheduled on nodes that have all three labels.

Note

Labels can only be set to one value, so setting a node selector of **`region=west`** in a pod configuration that has **`region=east`** as the administrator-set default, results in a pod that will never be scheduled.

13.2.12. Running a Pod with a Different Service Account

You can run a pod with a service account other than the default:

1. Edit the deployment configuration:

```
$ oc edit dc/<deployment_config>
```

2. Add the **`serviceAccount`** and **`serviceAccountName`** parameters to the **`spec`** field, and specify the service account you want to use:

```
spec:
  securityContext: {}
  serviceAccount: <service_account>
  serviceAccountName: <service_account>
```

13.2.13. Adding Secrets to Deployment Configurations from the Web Console

Add a secret to your deployment configuration so that it can access a private repository.

1. Create a new OpenShift Container Platform project.
2. Create a secret that contains credentials for accessing a private image repository.
3. Create a deployment configuration.
4. On the deployment configuration editor page or in the **fromimage** page of the web console, set the **Pull Secret**.
5. Click the **Save** button.

13.3. DEPLOYMENT STRATEGIES

13.3.1. What Are Deployment 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 Container Platform 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 configuration will retry to run the pod until it times out. The default timeout is **`10m`**, a value set in **`TimeoutSeconds`** in **`dc.spec.strategy.*params`**.

The Rolling strategy is the default strategy used if no strategy is specified on a deployment configuration.

13.3.2. Rolling Strategy

A rolling deployment slowly replaces instances of the previous version of an application with instances of the new version of the application. A rolling deployment typically waits for new pods to become **ready** via a **readiness check** before scaling down the old components. If a significant issue occurs, the rolling deployment can be aborted.

13.3.2.1. Canary Deployments

All rolling deployments in OpenShift Container Platform are *canary* deployments; a new version (the canary) is tested before all of the old instances are replaced. If the readiness check never succeeds, the canary instance is removed and the deployment configuration will be automatically rolled back. The readiness check is part of the application code, and may be as sophisticated as necessary to ensure the new instance is ready to be used. If you need to implement more complex checks of the application (such as sending real user workloads to the new instance), consider implementing a custom deployment or using a blue-green deployment strategy.

13.3.2.2. When to Use a Rolling Deployment

- When you want to take no downtime during an application update.
- When your application supports having old code and new code running at the same time.

A rolling deployment means you to have both old and new versions of your code running at the same time. This typically requires that your application handle N-1 compatibility, that data stored by the new version can be read and handled (or gracefully ignored) by the old version of the code. This can take many forms — data stored on disk, in a database, in a temporary cache, or that is part of a user's browser session. While most web applications can support rolling deployments, it is important to test and design your application to handle it.

The following is an example of the Rolling strategy:

```
strategy:
  type: Rolling
  rollingParams:
    timeoutSeconds: 120 1
    maxSurge: "20%" 2
    maxUnavailable: "10%" 3
    pre: {} 4
    post: {}
```

1

How long to wait for a scaling event before giving up. Optional; the default is 120.

2

maxSurge is optional and defaults to **25%** if not specified; see below.

3

maxUnavailable is optional and defaults to **25%** if not specified; see below.

4

pre and **post** are both lifecycle hooks.

The Rolling strategy will:

1. Execute any **pre** lifecycle hook.
2. Scale up the new replication controller based on the surge count.
3. Scale down the old replication controller based on the max unavailable count.
4. Repeat this scaling until the new replication controller has reached the desired replica count and the old replication controller 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 process will eventually time out and result in a deployment failure.

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.

Generally, if you want fast rollouts, use **maxSurge**. If you need to take into account resource quota and can accept partial unavailability, use **maxUnavailable**.

13.3.2.3. Rolling Example

Rolling deployments are the default in OpenShift Container Platform. To see a rolling update, follow these steps:

1. Create an application based on the example deployment images found in DockerHub:

```
$ oc new-app openshift/deployment-example
```

   If you have the router installed, make the application available via a route (or use the service IP directly)

```
$ oc expose svc/deployment-example
```

   Browse to the application at **deployment-example.<project>.<router_domain>** to verify you see the **v1** image.

2. Scale the deployment configuration up to three replicas:

```
$ oc scale dc/deployment-example --replicas=3
```

3. Trigger a new deployment automatically by tagging a new version of the example as the **latest** tag:

```
$ oc tag deployment-example:v2 deployment-example:latest
```

4. In your browser, refresh the page until you see the **v2** image.

5. If you are using the CLI, the following command will show you how many pods are on version 1 and how many are on version 2. In the web console, you should see the pods slowly being added to v2 and removed from v1.

```
$ oc describe dc deployment-example
```

During the deployment process, the new replication controller is incrementally scaled up. Once the new pods are marked as **ready** (by passing their readiness check), the deployment process will continue. If the pods do not become ready, the process will abort, and the deployment configuration will be rolled back to its previous version.

13.3.3. Recreate Strategy

The Recreate strategy has basic rollout behavior and supports lifecycle hooks for injecting code into the deployment process.

The following is an example of the Recreate strategy:

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

1

recreateParams are optional.

2

pre, **mid**, and **post** are lifecycle hooks.

The Recreate strategy will:

1. Execute any **pre** lifecycle hook.
2. Scale down the previous deployment to zero.
3. Execute any **mid** lifecycle hook.
4. Scale up the new deployment.
5. Execute any **post** lifecycle hook.

Important

During scale up, if the replica count of the deployment is greater than one, the first replica of the deployment will be validated for readiness before fully scaling up the deployment. If the validation of the first replica fails, the deployment will be considered a failure.

13.3.3.1. When to Use a Recreate Deployment

- When you must run migrations or other data transformations before your new code starts.
- When you do not support having new and old versions of your application code running at the same time.
- When you want to use a RWO volume, which is not supported being shared between multiple replicas.

A recreate deployment incurs downtime because, for a brief period, no instances of your application are running. However, your old code and new code do not run at the same time.

13.3.4. Custom Strategy

The Custom strategy allows you to provide your own deployment behavior.

The following is an example of the Custom strategy:

```
strategy:
  type: Custom
  customParams:
    image: organization/strategy
    command: [ "command", "arg1" ]
    environment:
      - name: ENV_1
        value: VALUE_1
```

In the above example, the **organization/strategy** container image provides the deployment behavior. The optional **command** array overrides any **CMD** directive specified in the image's ***Dockerfile***. The optional environment variables provided are added to the execution environment of

the strategy process.

Additionally, OpenShift Container Platform provides the following environment variables to the deployment process:

Environment Variable	Description
OPENSHIFT_DEPLOYMENT_NAME	The name of the new deployment (a replication controller).
OPENSHIFT_DEPLOYMENT_NAMESPACE	The name space 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.

Learn more about advanced deployment strategies.

Alternatively, use **customParams** to inject the custom deployment logic into the existing deployment strategies. Provide a custom shell script logic and call the **openshift-deploy** binary. Users do not have to supply their custom deployer container image, but the default OpenShift Container Platform deployer image will be used instead:

```
strategy:
  type: Rolling
  customParams:
    command:
    - /bin/sh
    - -c
    - |
      set -e
      openshift-deploy --until=50%
      echo Halfway there
      openshift-deploy
      echo Complete
```

This will result in following deployment:

```
Started deployment #2
--> Scaling up custom-deployment-2 from 0 to 2, scaling down custom-
deployment-1 from 2 to 0 (keep 2 pods available, don't exceed 3 pods)
    Scaling custom-deployment-2 up to 1
--> Reached 50% (currently 50%)
Halfway there
--> Scaling up custom-deployment-2 from 1 to 2, scaling down custom-
deployment-1 from 2 to 0 (keep 2 pods available, don't exceed 3 pods)
    Scaling custom-deployment-1 down to 1
```

```
    Scaling custom-deployment-2 up to 2
    Scaling custom-deployment-1 down to 0
--> Success
Complete
```

If the custom deployment strategy process requires access to the OpenShift Container Platform API or the Kubernetes API the container that executes the strategy can use the service account token available inside the container for authentication.

13.3.5. Lifecycle Hooks

The Recreate and Rolling strategies support lifecycle hooks, which allow behavior to be injected into the deployment process at predefined points within the strategy:

The following is an example of a **`pre`** lifecycle hook:

```
pre:
  failurePolicy: Abort
  execNewPod: {} 1
```

1

`execNewPod` is a pod-based lifecycle hook.

Every hook has a **`failurePolicy`**, which defines the action the strategy should take when a hook failure is encountered:

`Abort`	The deployment process will be considered a failure if the hook fails.
`Retry`	The hook execution should be retried until it succeeds.
`Ignore`	Any hook failure should be ignored and the deployment should proceed.

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.

13.3.5.1. Pod-based Lifecycle Hook

Pod-based lifecycle hooks execute hook code in a new pod derived from the template in a deployment configuration.

The following simplified example deployment configuration uses the Rolling strategy. Triggers and some other minor details are omitted for brevity:

```
kind: DeploymentConfig
```

```
apiVersion: v1
metadata:
  name: frontend
spec:
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: helloworld
          image: openshift/origin-ruby-sample
  replicas: 5
  selector:
    name: frontend
  strategy:
    type: Rolling
    rollingParams:
      pre:
        failurePolicy: Abort
        execNewPod:
          containerName: helloworld (1)
          command: [ "/usr/bin/command", "arg1", "arg2" ] (2)
          env: (3)
            - name: CUSTOM_VAR1
              value: custom_value1
          volumes:
            - data (4)
```

1

The **helloworld** name refers to **spec.template.spec.containers[0].name**.

2

This **command** overrides any **ENTRYPOINT** defined by the **openshift/origin-ruby-sample** image.

3

env is an optional set of environment variables for the hook container.

4

volumes is an optional set of volume references for the hook container.

In this example, the **pre** hook will be executed in a new pod using the **openshift/origin-ruby-sample** image from the **helloworld** container. The hook pod will have the following properties:

- The hook command will be **/usr/bin/command arg1 arg2**.

- The hook container will have the **`CUSTOM_VAR1=custom_value1`** environment variable.
- The hook failure policy is **`Abort`**, meaning the deployment process will fail if the hook fails.
- The hook pod will inherit the **`data`** volume from the deployment configuration pod.

13.3.5.2. Using the Command Line

The **`oc set deployment-hook`** command can be used to set the deployment hook for a deployment configuration. For the example above, you can set the pre-deployment hook with the following command:

```
$ oc set deployment-hook dc/frontend --pre -c helloworld -e
CUSTOM_VAR1=custom_value1 \
  -v data --failure-policy=abort -- /usr/bin/command arg1 arg2
```

13.4. ADVANCED DEPLOYMENT STRATEGIES

13.4.1. Blue-Green Deployment

Blue-green deployments involve running two versions of an application at the same time and moving production traffic from the old version to the new version. There are several ways to implement a blue-green deployment in OpenShift Container Platform.

13.4.1.1. When to Use a Blue-Green Deployment

Use a blue-green deployment when you want to test a new version of your application in a production environment before moving traffic to it.

Blue-green deployments make switching between two different versions of your application easy. However, since many applications depend on persistent data, you will need to have an application that supports N-1 compatibility if you share a database, or implement a live data migration between your database, store, or disk if you choose to create two copies of your data layer.

13.4.1.2. Blue-Green Deployment Example

In order to maintain control over two distinct groups of instances (old and new versions of the code), the blue-green deployment is best represented with multiple deployment configurations.

13.4.1.2.1. Using a Route and Two Services

A route points to a service, and can be changed to point to a different service at any time. As a developer, test the new version of your code by connecting to the new service before your production traffic is routed to it. Routes are intended for web (HTTP and HTTPS) traffic, so this technique is best suited for web applications.

1. Create two copies of the example application:

```
$ oc new-app openshift/deployment-example:v1 --name=example-green
$ oc new-app openshift/deployment-example:v2 --name=example-blue
```

This will create two independent application components: one running the **v1** image under the **example-green** service, and one using the **v2** image under the **example-blue** service.

2. Create a route that points to the old service:

```
$ oc expose svc/example-green --name=bluegreen-example
```

3. Browse to the application at **bluegreen-example.<project>.<router_domain>** to verify you see the **v1** image.

Note

On versions of OpenShift Container Platform older than v3.0.1, this command will generate a route at **example-green.<project>.<router_domain>**, not the above location.

4. Edit the route and change the service name to **example-blue**:

```
$ oc patch route/bluegreen-example -p '{"spec":{"to":
{"name":"example-blue"}}}'
```

5. In your browser, refresh the page until you see the **v2** image.

13.4.2. A/B Deployment

A/B deployments generally imply running two (or more) versions of the application code or application configuration at the same time for testing or experimentation purposes.

The simplest form of an A/B deployment is to divide production traffic between two or more distinct **shards** — a single group of instances with homogeneous configuration and code.

More complicated A/B deployments may involve a specialized proxy or load balancer that assigns traffic to specific shards based on information about the user or application (all "test" users get sent to the B shard, but regular users get sent to the A shard).

A/B deployments can be considered similar to A/B testing, although an A/B deployment implies multiple versions of code and configuration, where as A/B testing often uses one code base with application specific checks.

13.4.2.1. When to Use an A/B Deployment

- When you want to test multiple versions of code or configuration, but are not planning to roll one out in preference to the other.
- When you want to have different configuration in different regions.

An A/B deployment groups different configuration and code — multiple shards — together under a single logical endpoint. Generally, these deployments, if they access persistent data, should properly deal with N-1 compatibility (the more shards you have, the more possible versions you have running). Use this pattern when you need separate internal configuration and code, but end users should not be aware of the changes.

13.4.2.2. A/B Deployment Example

All A/B deployments are composite deployment types consisting of multiple deployment configurations.

13.4.2.2.1. One Service, Multiple Deployment Configurations

OpenShift Container Platform, through labels and deployment configurations, supports multiple simultaneous shards being exposed through the same service. To the consuming user, the shards are invisible. An example of the simplest possible sharding is described below:

1. Create the first shard of the application based on the example deployment images:

```
$ oc new-app openshift/deployment-example --name=ab-example-a --
labels=ab-example=true SUBTITLE="shard A"
```

2. Edit the newly created shard to set a label **`ab-example=true`** that will be common to all shards:

```
$ oc edit dc/ab-example-a
```

   In the editor, add the line **`ab-example: "true"`** underneath **`spec.selector`** and **`spec.template.metadata.labels`** alongside the existing **`deploymentconfig=ab-example-a`** label. Save and exit the editor.

3. Trigger a re-deployment of the first shard to pick up the new labels:

```
$ oc deploy ab-example-a --latest
```

4. Create a service that uses the common label:

```
$ oc expose dc/ab-example-a --name=ab-example --selector=ab-
example=true
```

   If you have the router installed, make the application available via a route (or use the service IP directly):

```
$ oc expose svc/ab-example
```

   Browse to the application at **`ab-example.<project>.<router_domain>`** to verify you see the **`v1`** image.

5. Create a second shard based on the same source image as the first shard but different tagged version, and set a unique value:

```
$ oc new-app openshift/deployment-example:v2 --name=ab-example-b
--labels=ab-example=true SUBTITLE="shard B" COLOR="red"
```

6. Edit the newly created shard to set a label **`ab-example=true`** that will be common to all shards:

```
$ oc edit dc/ab-example-b
```

In the editor, add the line **`ab-example: "true"`** underneath **`spec.selector`** and **`spec.template.metadata.labels`** alongside the existing **`deploymentconfig=ab-example-b`** label. Save and exit the editor.

7. Trigger a re-deployment of the second shard to pick up the new labels:

```
$ oc deploy ab-example-b --latest
```

8. At this point, both sets of pods are being served under the route. However, since both browsers (by leaving a connection open) and the router (by default, through a cookie) will attempt to preserve your connection to a back-end server, you may not see both shards being returned to you. To force your browser to one or the other shard, use the scale command:

```
$ oc scale dc/ab-example-a --replicas=0
```

Refreshing your browser should show **v2** and **shard B** (in red).

```
$ oc scale dc/ab-example-a --replicas=1; oc scale dc/ab-example-
b --replicas=0
```

Refreshing your browser should show **v1** and **shard A** (in blue).

If you trigger a deployment on either shard, only the pods in that shard will be affected. You can easily trigger a deployment by changing the **`SUBTITLE`** environment variable in either deployment config **`oc edit dc/ab-example-a`** or **`oc edit dc/ab-example-b`**. You can add additional shards by repeating steps 5-7.

Note

These steps will be simplified in future versions of OpenShift Container Platform.

13.4.3. Proxy Shard / Traffic Splitter

In production environments, you can precisely control the distribution of traffic that lands on a particular shard. When dealing with large numbers of instances, you can use the relative scale of individual shards to implement percentage based traffic. That combines well with a **proxy shard**, which forwards or splits the traffic it receives to a separate service or application running elsewhere.

In the simplest configuration, the proxy would forward requests unchanged. In more complex setups, you can duplicate the incoming requests and send to both a separate cluster as well as to a local instance of the application, and compare the result. Other patterns include keeping the caches of a DR installation warm, or sampling incoming traffic for analysis purposes.

While an implementation is beyond the scope of this example, any TCP (or UDP) proxy could be run under the desired shard. Use the **`oc scale`** command to alter the relative number of instances serving requests under the proxy shard. For more complex traffic management, consider customizing the OpenShift Container Platform router with proportional balancing capabilities.

13.4.4. N-1 Compatibility

Applications that have new code and old code running at the same time must be careful to ensure that data written by the new code can be read by the old code. This is sometimes called *schema evolution* and is a complex problem.

For some applications, the period of time that old code and new code is running side by side is short, so bugs or some failed user transactions are acceptable. For others, the failure pattern may result in the entire application becoming non-functional.

One way to validate N-1 compatibility is to use an A/B deployment. Run the old code and new code at the same time in a controlled way in a test environment, and verify that traffic that flows to the new deployment does not cause failures in the old deployment.

13.4.5. Graceful Termination

OpenShift Container Platform and Kubernetes give application instances time to shut down before removing them from load balancing rotations. However, applications must ensure they cleanly terminate user connections as well before they exit.

On shutdown, OpenShift Container Platform will send a **TERM** signal to the processes in the container. Application code, on receiving **SIGTERM**, should stop accepting new connections. This will ensure that load balancers route traffic to other active instances. The application code should then wait until all open connections are closed (or gracefully terminate individual connections at the next opportunity) before exiting.

After the graceful termination period expires, a process that has not exited will be sent the **KILL** signal, which immediately ends the process. The **`terminationGracePeriodSeconds`** attribute of a pod or pod template controls the graceful termination period (default 30 seconds) and may be customized per application as necessary.

13.5. KUBERNETES DEPLOYMENTS SUPPORT

13.5.1. New Object Type: Deployments

In the upstream Kubernetes project, a new first-class object type called *deployments* was added in version 1.2. This object type (referred to here as *Kubernetes deployments* for distinction) serves as a descendant of the deployment configuration object type that has been in OpenShift Container Platform since 3.0. Starting in OpenShift Container Platform 3.4, support for Kubernetes deployments is available as a Technology Preview feature.

Like deployment configurations, Kubernetes deployments describe the desired state of a particular component of an application as a pod template. Kubernetes deployments create *replica sets* (an iteration of replication controllers), which orchestrate pod lifecycles.

For example, this definition of a Kubernetes deployment creates a replica set to bring up three nginx pods:

Example Kubernetes Deployment Definition *nginx-deployment.yaml*

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
```

```
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

After saving the definition to a local file, you could then use it to create a Kubernetes deployment:

```
$ oc create -f nginx-deployment.yaml
```

You can use the CLI to inspect and operate on Kubernetes deployments and replica sets like other object types, as described in Common Operations, like **`get`** and **`describe`**. For the object type, use **`deployments`** or **`deploy`** for Kubernetes deployments and **`replicasets`** or **`rs`** for replica sets.

See the Kubernetes documentation for more details about Deployments and Replica Sets, substituting **`oc`** for **`kubectl`** in CLI usage examples.

13.5.2. Kubernetes Deployments vs Deployment Configurations

Because deployment configurations existed in OpenShift Container Platform prior to deployments being added in Kubernetes 1.2, the latter object type naturally diverges slightly from the former. The long-term goal in OpenShift Container Platform is to reach full feature parity in Kubernetes deployments and switch to using them as a single object type that provides fine-grained management over applications.

Kubernetes deployments are supported to ensure upstream projects and examples that use the new object type can run smoothly on OpenShift Container Platform. Given the current feature set of Kubernetes deployments, you may want to use them instead of deployment configurations in OpenShift Container Platform if you do not plan to use any of the following in particular:

- image streams
- lifecycle hooks
- Custom deployment strategies

The following sections go into more details on the differences between the two object types to further help you decide when you might want to use Kubernetes deployments over deployment configurations.

13.5.2.1. Deployment Configuration-Specific Features

13.5.2.1.1. Automatic Rollbacks

Kubernetes deployments do not support automatically rolling back to the last successfully deployed replica set in case of a failure. This feature should be added soon.

13.5.2.1.2. Triggers

Kubernetes deployments have an implicit **`ConfigChange`** trigger in that every change in the pod template of a deployment automatically triggers a new rollout. If you do not want new rollouts on pod template changes, pause the deployment:

```
$ oc rollout pause deployments/<name>
```

At the moment, Kubernetes deployments do not support **`ImageChange`** triggers. A generic triggering mechanism has been proposed upstream, but it is unknown if and when it may be accepted. Eventually, a OpenShift Container Platform-specific mechanism could be implemented to layer on top of Kubernetes deployments, but it would be more desirable for it to exist as part of the Kubernetes core.

13.5.2.1.3. Lifecycle Hooks

Kubernetes deployments do not support any lifecycle hooks.

13.5.2.1.4. Custom Strategies

Kubernetes deployments do not yet support user-specified Custom deployment strategies yet.

13.5.2.1.5. Canary Deployments

Kubernetes deployments do not yet run canaries as part of a new rollout.

13.5.2.1.6. Test Deployments

Kubernetes deployments do not support running test tracks.

13.5.2.2. Kubernetes Deployment-Specific Features

13.5.2.2.1. Rollover

The deployment process for Kubernetes deployments is driven by a controller loop, in contrast to deployment configurations which use deployer pods for every new rollout. This means that a Kubernetes deployment can have as many active replica sets as possible, and eventually the deployment controller will scale down all old replica sets and scale up the newest one.

Deployment configurations can have at most one deployer pod running, otherwise multiple deployers end up fighting with each other trying to scale up what they think should be the newest replication controller. Because of this, only two replication controllers can be active at any point in time. Ultimately, this translates to faster rapid rollouts for Kubernetes deployments.

13.5.2.2.2. Proportional Scaling

Because the Kubernetes deployment controller is the sole source of truth for the sizes of new and old replica sets owned by a deployment, it is able to scale ongoing rollouts. Additional replicas are distributed proportionally based on the size of each replica set.

Deployment configurations cannot be scaled when a rollout is ongoing because the deployment configuration controller will end up fighting with the deployer process about the size of the new replication controller.

13.5.2.2.3. Pausing Mid-rollout

Kubernetes deployments can be paused at any point in time, meaning you can also pause ongoing rollouts. On the other hand, you cannot pause deployer pods currently, so if you try to pause a deployment configuration in the middle of a rollout, the deployer process will not be affected and will continue until it finishes.

CHAPTER 14. GETTING TRAFFIC INTO THE CLUSTER

14.1. OVERVIEW

There are many ways to access the cluster. This section describes some commonly used approaches.

The recommendation is:

- If you have HTTP/HTTPS, use the router.
- If you have a TLS-encrypted protocol other than HTTPS (for example, TLS with the SNI header), use the router.
- Otherwise, use Load Balancer, ExternalIP, or NodePort.

TCP or UDP offers several approaches:

- Use the non-cloud Load Balancer. This limits you to a single edge router IP (which can be a virtual IP (VIP), but still is a single machine for initial load balancing). It simplifies the administrator's job, but uses one IP per service.
- Manually assign ExternalIPs to the service. You can assign a set of IPs, so you can have multiple machines for the incoming load balancing. However, this requires elevated permissions to assign, and manual tracking of what IP:ports that are used.
- Use NodePorts to expose the service on *all* nodes in the cluster. This is more wasteful of scarce port resources. However, it is slightly easier to set up multiple. Again, this requires more privileges.

The router is the most common way to access the cluster. This is limited to HTTP/HTTPS(SNI)/TLS(SNI), which covers web applications.

ExternalIP or NodePort is useful when the HTTP protocol is not being used or non-standard ports are in use. There is more manual setup and monitoring involved.

The administrator must set up the external port to the cluster networking environment so that requests can reach the cluster. For example, names can be configured into DNS to point to specific nodes or other IP addresses in the cluster. The DNS wildcard feature can be used to configure a subset of names to an IP address in the cluster. This is convenient when using routers because it allows the users to set up routes within the cluster without further administrator attention.

The administrator must ensure that the local firewall on each node permits the request to reach the IP address.

The High Availability section describes how to make this highly available using replicated services.

14.2. USING A ROUTER

This is the most common way to access the cluster. A router is configured to accept external requests and proxy them based on the configured routes. This is limited to HTTP/HTTPS(SNI)/TLS(SNI), which covers web applications.

An administrator can create a wildcard DNS entry, and then set up a router. Afterward, the users can self-service the edge router without having to contact the administrators. The router has controls to allow the administrator to specify whether the users can self-provision host names, or if they must fit a pattern the administrator defines. The other solutions require the administrator to do the

provisioning, or they require that the administrator delegates a lot of privilege.

A set of routes can be created in the various projects. The overall set of routes is available to the set of routers. Each router selects from the set of routes. All routers see all routes unless restricted by labels on the router, which is called router sharding.

The High Availability section describes how to configure a router for high availability service using multiple replicas.

14.3. USING A LOAD BALANCER SERVICE

Load balancers are available on AWS and GCE clouds, and non-cloud options are also available.

The non-cloud load balancer allocates a unique IP from a configured pool. This limits you to a single edge router IP, which can be a VIP, but still will be a single machine for initial load balancing. The non-cloud load balancer simplifies the administrator's job by providing the needed IP address, but uses one IP per service.

14.4. USING A SERVICE EXTERNALIP

Administrators can assign a list of externalIPs, for which nodes in the cluster will also accept traffic for the service. These IPs are not managed by OpenShift Container Platform and administrators are responsible for ensuring that traffic arrives at a node with this IP. A common example of this is configuring a highly available service.

The supplied list of IP addresses is used for load balancing incoming requests. The service port is opened on the externalIPs on all nodes running **kube-proxy**.

Note

ExternalIPs require elevated permissions to assign, and manual tracking of the IP:ports that are in use.

Traffic from hosts that are external to the cluster that is going to the external IP address ends up on a node in the cluster. When it arrives, the service that set up the external IP redirects it to a service endpoint that it is managing. The service load balances among its endpoints.

An externally visible IP for the service can be configured in several ways:

- Manually configuring the externalIPs with a list of known external IP addresses.
- Configuring the externalIPs to a set of VIP addresses that are managed by the high avalibility service.
- In a cloud environment (AWS or GCE), by using **`type=LoadBalancer`**
- In a non-cloud environment, by configuring ingressIP range (**`ingressIPNetworkCIDR`**), **`service.type=LoadBalancer`**, and **`service.port.ingressIP`**.

The administrator must ensure the external IPs are routed to the nodes and local firewall rules on all nodes allow access to the open port.

Example External IP Configured within a Service Definition

```
$ oc create -f - <<INGRESS
apiVersion: v1
kind: Service
metadata:
  name: postgresql-ingress
spec:
  externalIPs:
  - 10.9.54.100 1
  ports:
  - port: 5432
    protocol: TCP
  selector:
    name: postgresql
  type: LoadBalancer
INGRESS
```

1

The IP address assigned to the external IP.

The external IP address is not managed by the underlying Kubernetes infrastructure and must be maintained and provided by a cluster administrator. While external IPs provide a solution for accessing services on the {product-tlte} cluster, there are several shortcomings:

- The cluster administrator user must ensure the external IP is not in any range that the cluster is configured to use. The user needs to work with the administrator that controls the network external to the cluster to assign the external IP address and ensure traffic to the IP can get to the nodes in the cluster. The cluster administrator must ensure firewall rules permit the packets to get into the nodes for the ports they want to expose.
- There are no protections in place to restrict the usage of a particular external address configured within the cluster. This allows the potential for a single address to be used by multiple services targeting the same port. When this situation occurs, the service which requested the port first is given use of the port.

Ingress IP Self-Service provides a solution.

14.4.1. Using Ingress IP Self-Service

Ingress IP Self-Service streamlines the allocation of External IPs for accessing services in the cluster. Cluster administrators can designate a range of addresses using a CIDR notation, which allows an application user to make a request against the cluster for an external IP address. When a service is configured with the type **`LoadBalancer`**, an External IP address will be automatically assigned from the designated range.

Note

Ingress IP Self-Service is only applicable in non-cloud based environments.

A common use case for Ingress IP Self-Service is the ability to provide database services, such as PostgreSQL, to clients outside of the OpenShift Container Platform cluster.

14.4.1.1. Defining the Edge Router IP Range

The ability for cluster administrators to automatically allocate External IP addresses using the edge router is enabled by default within OpenShift Container Platform and configured to use the **`172.46.0.0/16`** range. An alternate range can be specified by configuring the **`ingressIPNetworkCIDR`** parameter in the ***/etc/origin/master-config.yaml*** file:

```
networkConfig:
  ingressIPNetworkCIDR: 10.9.54.0/25
```

Restart the OpenShift Container Platform master service to apply the changes.

14.4.1.2. Deploy a Sample Application

To expose a PostgreSQL as a service for external consumption, the application must be first deployed into the cluster. Create a new project or use and existing project and instantiate one of the PostgreSQL templates.

Caution

The **postgresql-ephemeral** template does not make use of persistent storage. Once the application is scaled down or destroyed, any existing data will be lost. To use persistent storage, specify the **postgresql-persistent** template instead.

After instantiating the template, a ClusterIP-based service and **`DeploymentConfig`** is created and a new pod containing PostgreSQL will be started.

14.4.1.3. Configuring an IP Address for a Service

To allow the cluster to automatically assign an IP address for a service, create a service definition similar to the following that will create a new Ingress service:

```
$ oc create -f - <<INGRESS
apiVersion: v1
kind: Service
metadata:
  name: postgresql-ingress
spec:
  ports:
  - name: postgresql
    port: 5432
  type: LoadBalancer 1
  selector:
    name: postgresql
INGRESS
```

1

The **`LoadBalancer`** type of service will make the request for an external service on behalf of the application user.

Alternatively, the **oc expose** command can be used to create the service:

```
$ oc expose dc postgresql --type=LoadBalancer --name=postgresql-ingress
```

Once the service is created, the external IP address is automatically allocated by the cluster and can be confirmed by running:

```
$ oc get svc postgresql-ingress
```

Example oc get Output

```
NAME         CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
postgresql-ingress    172.30.74.106   10.9.54.100,10.9.54.100
5432/TCP    30s
```

Specifying the type **LoadBalancer** also configures the service with a **nodePort** value. **nodePort** exposes the service port on all nodes in the cluster. Any packet that arrives on any node in the cluster targeting the **nodePort** ends up in the service. Then, it is load balanced to the service's endpoints.

To discover the node port automatically assigned, run:

```
$ oc export svc postgresql-ingress
```

Example oc export Output

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: postgresql-persistent
    template: postgresql-persistent-template
  name: postgresql-ingress
spec:
  ports:
  - nodePort: 32439 (1)
    port: 5432
    protocol: TCP
    targetPort: 5432
  selector:
    name: postgresql
  sessionAffinity: None
  type: LoadBalancer
```

1 Automatically assigned port.

A PostgreSQL client can now be configured to connect directly to any node using the value of the assigned **`nodePort`**. A **`nodePort`** works with any IP address that allows traffic to terminate at any node in the cluster.

14.4.1.4. Configuring the Service to be Highly Available

Instead of connecting directly to individual nodes, you can use one of OpenShift Container Platform's high availability strategies by deploying the IP failover router to provide access services configured with external IP addresses. This allows cluster administrators the flexibility of defining the edge router points within a cluster, and making the service highly available.

Note

Nodes that have IP failover routers deployed to them must be in the same **Layer 2** switching domain for ARP broadcasts to communicate to switches what appropriate port the destination should flow to.

Caution

High availability is limited to a maximum of 255 VIPs. This is a limitation of the Virtual Router Redundancy Protocol (VRRP). The VIPs do not have to be sequential.

Learn more about IP failover.

14.5. USING A SERVICE NODEPORT

Use NodePorts to expose the service nodePort on all nodes in the cluster. The service exposes **`<node-name>:<nodePort>`** on all nodes in the cluster. By default, nodePorts are in the range of 30000-32767, which means a NodePort is unlikely to match a service's intended port (for example, 8080 may be exposed as 31020). This use of ports is wasteful of scarce host port resources. However, it is slightly easier to set up. Again, this requires more privileges.

The administrator must ensure the desired traffic is routed to the nodes and local firewall rules on all nodes allow access to the open port.

NodePorts and externalIP are independent and both can be used concurrently.

14.6. USING VIRTUAL IPS

High availability improves the chances that an IP address will remain active, by assigning a virtual IP address to the host in a configured pool of hosts. If the host goes down, the virtual IP address is automatically transferred to another host in the pool.

14.7. NON-CLOUD EDGE ROUTER LOAD BALANCER

In a non-cloud environment, cluster administrators can assign a unique external IP address to a service (as described here). When routed correctly, external traffic can reach the service endpoints via any TCP/UDP port the service exposes. This is simpler than having to manage the port space of a limited number of shared IP addresses, when manually assigning external IPs to services.

14.8. EDGE LOAD BALANCER

An edge load balancer can be used to accept traffic from outside networks and proxy the traffic to pods inside the cluster.

In this configuration, the internal pod network is visible to the outside.

CHAPTER 15. ROUTES

15.1. OVERVIEW

An OpenShift Container Platform route exposes a service at a host name, like *www.example.com*, so that external clients can reach it by name.

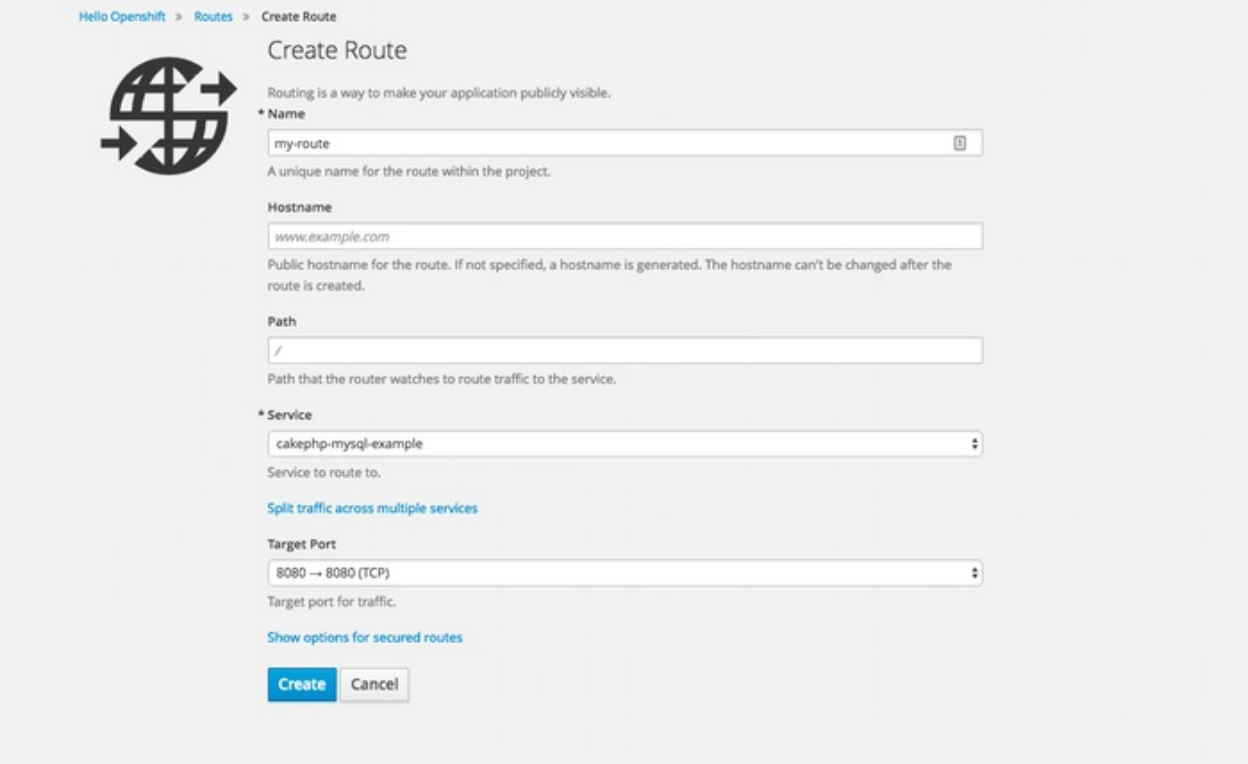
DNS resolution for a host name is handled separately from routing; your administrator may have configured a cloud domain that will always correctly resolve to the OpenShift Container Platform router, or if using an unrelated host name you may need to modify its DNS records independently to resolve to the router.

15.2. CREATING ROUTES

You can create unsecured and secured routes routes using the web console or the CLI.

Using the web console, you can navigate to the **Browse** → **Routes** page, then click **Create Route** to define and create a route in your project:

Figure 15.1. Creating a Route Using the Web Console



Using the CLI, the following example creates an unsecured route:

```
$ oc expose svc/frontend --hostname=www.example.com
```

The new route inherits the name from the service unless you specify one using the **`--name`** option.

Example 15.1. YAML Definition of the Unsecured Route Created Above

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

Unsecured routes are the default configuration, and are therefore the simplest to set up. However, secured routes offer security for connections to remain private. To create a secured HTTPS route encrypted with a key and certificate (PEM-format files which you must generate and sign separately), you can use the **`create route`** command and optionally provide certificates and a key.

Note

TLS is the replacement of SSL for HTTPS and other encrypted protocols.

```
$ oc create route edge --service=frontend \
    --cert=${MASTER_CONFIG_DIR}/ca.crt \
    --key=${MASTER_CONFIG_DIR}/ca.key \
    --ca-cert=${MASTER_CONFIG_DIR}/ca.crt \
    --hostname=www.example.com
```

Example 15.2. YAML Definition of the Secured Route Created Above

```
apiVersion: v1
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend
  tls:
    termination: edge
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

```
    caCertificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

You can create a secured route without specifying a key and certificate, in which case the router's default certificate will be used for TLS termination.

Note

TLS termination in OpenShift Container Platform 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.

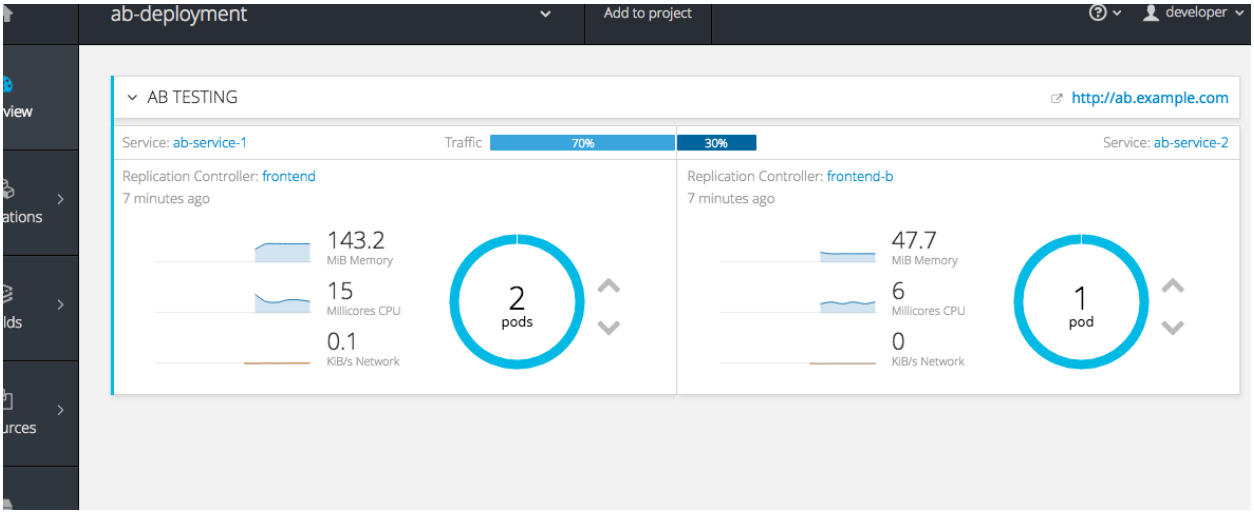
Further information on all types of TLS termination as well as path-based routing are available in the Architecture section.

15.3. LOAD BALANCING FOR A/B TESTING

You can run two versions of an application, and, entirely within OpenShift Container Platform, control the percentage of traffic to and from each application for A/B testing. A/B testing is a method of comparing two versions of an application against each other to determine which one performs better.

Previously, A/B testing only worked by adding or removing more pods of every kind (A or B). However, this was not a scalable solution because for lower B percentages, you would create a large number of pods. Starting in 3.3, the HAProxy router now supports splitting the traffic coming to a route across multiple back end services via weighting.

The web console allows users to set the weighting and show balance between them:



If you have two deployments, A and B, or more, then create respective services for the pods in those deployments and use labels.

The **Route** resource now has an **`alternateBackends`** field, which you can use to specify **Service**. Use the **`alternateBackends`** and **`To`** fields to supply the route with all of the back end

deployments grouped as services. Use the **`weight`** sub-field to specify a relative weight in integers ranging from 0 to 256. This value defaults to 100. The combined value of all the weights sets the relative proportions of traffic.

When you deploy the route, the router will balance the traffic according to the weights specified for the services.

To edit the route, run:

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

Then, update the percentage/weight of the services in the **`to`** and **`alternateBackends`** fields.

CHAPTER 16. INTEGRATING EXTERNAL SERVICES

16.1. OVERVIEW

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

16.2. EXTERNAL MYSQL DATABASE

One of the most common types of external services is an external database. To support an external database, an application needs:

1. An endpoint to communicate with.
2. A set of credentials and coordinates, including:
    a. A user name
    b. A passphrase
    c. A database name

The solution for integrating with an external database includes:

- A **`Service`** object to represent the SaaS provider as an OpenShift Container Platform 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 Container Platform 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 Container Platform 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 Container Platform router load-balance requests to the service amongst the service's endpoints.

    Services that represent an external resource do not require associated pods. Instead, leave the **`Selector`** field unset. This represents the external service, making the **`EndpointsController`** ignore the service and allows you to specify endpoints manually:

```
kind: "Service"
apiVersion: "v1"
metadata:
  name: "external-mysql-service"
spec:
  ports:
```

```
    -
      name: "mysql"
      protocol: "TCP"
      port: 3306
      targetPort: 3306
      nodePort: 0
selector: {} 1
```

1

The **selector** field to leave blank.

2. Next, create the required endpoints for the service. This gives the service proxy and router the location to send traffic directed to the service:

```
kind: "Endpoints"
apiVersion: "v1"
metadata:
  name: "external-mysql-service" 1
subsets: 2
  -
    addresses:
      -
        ip: "10.0.0.0" 3
    ports:
      -
        port: 3306 4
        name: "mysql"
```

1

The name of the **Service** instance, as defined in the previous step.

2

Traffic to the service will be load-balanced between the supplied **Endpoints** if more than one is supplied.

3

Endpoints IPs cannot be loopback (127.0.0.0/8), link-local (169.254.0.0/16), or link-local multicast (224.0.0.0/24).

4

The **port** and **name** definition must match the **port** and **name** value in the service defined in the previous step.

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: (1)
  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.

16.3. EXTERNAL SAAS PROVIDER

A common type of external service is an external SaaS endpoint. To support an external SaaS provider, an application needs:

1. An endpoint to communicate with
2. A set of credentials, such as:
    a. An API key
    b. A user name
    c. A passphrase

The following steps outline a scenario for integrating with an external SaaS provider:

1. Create an OpenShift Container Platform 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 Container Platform 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 Container Platform 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:  (1)
    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" (2)
                value: "<SaaS service API key>"
              -
                name: "SAAS_USERNAME" (3)
                value: "<SaaS service user>"
              -
                name: "SAAS_PASSPHRASE" (4)
                value: "<SaaS service passphrase>"
```

(1) (1)

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

(2) (2)

`SAAS_API_KEY`: The API key to use with the service.

(3)

`SAAS_USERNAME`: The user name to use with the service.

4

SAAS_PASSPHRASE: The passphrase to use with the service.

External SaaS Provider Environment Variables

Similarly, when using an internal service, your application is assigned environment variables for the service and the additional environment variables with the credentials described in the above steps. In the above example, the container receives the following environment variables:

- **EXAMPLE_EXTERNAL_SERVICE_SERVICE_HOST=<ip_address>**
- **EXAMPLE_EXTERNAL_SERVICE_SERVICE_PORT=<port_number>**
- **SAAS_API_KEY=<saas_api_key>**
- **SAAS_USERNAME=<saas_username>**
- **SAAS_PASSPHRASE=<saas_passphrase>**

The application reads the coordinates and credentials for the service from the environment and establishes a connection with the service.

CHAPTER 17. SECRETS

17.1. USING SECRETS

This topic discusses important properties of secrets and provides an overview on how developers can use them.

The **`Secret`** object type provides a mechanism to hold sensitive information such as passwords, OpenShift Container Platform client configuration files, **`dockercfg`** files, private source repository credentials, and so on. Secrets decouple sensitive content from the pods. You can mount secrets into containers using a volume plug-in or the system can use secrets to perform actions on behalf of a pod.

```
apiVersion: "v1"
kind: "Secret"
metadata:
  name: "mysecret"
namespace: "myns"
data: 1
  username: "dmFsdWUtMQ0K" 2
  password: "dmFsdWUtMg0KDQo="
stringData: 3
  hostname: "myapp.mydomain.com" 4
```

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.

2

The value associated with keys in the the **`data`** map must be base64 encoded.

3

Entries in the **`stringData`** map will have their values converted to base64 and the entry will then be moved to the **`data`** map automatically. This field is write-only; the value will only be returned via the **`data`** field.

4

The value associated with keys in the the **`stringData`** map is made up of plain text strings.

17.1.1. Properties of Secrets

Key properties include:

- Secret data can be referenced independently from its definition.
- Secret data volumes are backed by temporary file-storage facilities (tmpfs) and never come to rest on a node.
- Secret data can be shared within a namespace.

17.1.2. Creating Secrets

You must create a secret before creating the pods that depend on that secret.

When creating secrets:

- Create a secret object with secret data.
- Update the pod's service account to allow the reference to the secret.
- Create a pod, which consumes the secret as an environment variable or as a file (using a **`secret`** volume).

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

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

17.1.3. Updating Secrets

When you modify the value of a secret, the value (used by an already running pod) will not dynamically change. To change a secret, you must delete the original pod and create a new pod (perhaps with an identical PodSpec).

Updating a secret follows the same workflow as deploying a new container image. You can use the **`kubectl rolling-update`** command.

The **`resourceVersion`** value in a secret is not specified when it is referenced. Therefore, if a secret is updated at the same time as pods are starting, then the version of the secret will be used for the pod will not be defined.

Note

Currently, it is not possible to check the resource version of a secret object that was used when a pod was created. It is planned that pods will report this information, so that a controller could restart ones using a old **`resourceVersion`**. In the interim, do not update the data of existing secrets, but create new ones with distinct names.

17.2. SECRETS IN VOLUMES AND ENVIRONMENT VARIABLES

See examples of YAML files with secret data.

After you create a secret, you can:

1. Create the pod to reference your secret:

```
$ oc create -f <your_yaml_file>.yaml
```

2. Get the logs:

```
$ oc logs secret-example-pod
```

3. Delete the pod:

```
$ oc delete pod secret-example-pod
```

17.3. IMAGE PULL SECRETS

See Using Image Pull Secrets for more information.

17.4. SOURCE CLONE SECRETS

See Source Secrets for more information.

17.5. SERVICE SERVING CERTIFICATE SECRETS

To secure communication to your service, have the cluster generate a signed serving certificate/key pair into a secret in your namespace. To do this, set the **`service.alpha.openshift.io/serving-cert-secret-name`** to the name you want to use for your secret. Then, your **PodSpec** can mount that secret. When it is available, your pod will run. The certificate will be good for the internal service DNS name, **`<service.name>.<service.namespace>.svc`**. The certificate and key are in PEM format, stored in **`tls.crt`** and **`tls.key`** respectively. They are regenerated upon expiration. View the expiration date in the **`service.alpha.openshift.io/expiry`** annotation on the secret, which is in RFC3339 format.

Note

In most cases, the service DNS name **`<service.name>.<service.namespace>.svc`** is not externally routable. The primary use of **`<service.name>.<service.namespace>.svc`** is for intracluster or intraservice communication, and with re-encrypt routes.

Other pods can trust cluster-created certificates (which are only signed for internal DNS names), by using the CA bundle in the ***/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt*** file that is automatically mounted in their pod.

17.6. RESTRICTIONS

To use a secret, a pod needs to reference the secret. A secret can be used with a pod in three ways: - to populate environment variables for containers. - as files in a volume mounted on one or more of its containers. - by kubelet when pulling images for the pod.

Volume type secrets write data into the container as a file using the volume mechanism. **imagePullSecrets** use service accounts for the automatic injection of the secret into all pods in a namespaces.

When a template contains a secret definition, the only way for the template to use the provided secret is to ensure that the secret volume sources are validated and that the specified object reference actually points to an object of type **`Secret`**. Therefore, a secret needs to be created before any pods that depend on it. The most effective way to ensure this is to have it get injected automatically through the use of a service account.

Secret API objects reside in a namespace. They can only be referenced by pods in that same namespace.

Individual secrets are limited to 1MB in size. This is to discourage the creation of large secrets that would exhaust apiserver and kubelet memory. However, creation of a number of smaller secrets could also exhaust memory.

17.6.1. Secret Data Keys

Secret keys must be in a DNS subdomain.

17.7. EXAMPLES

Example 17.1. YAML of a Pod Populating Files in a Volume with Secret Data

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "cat /etc/secret-volume/*" ]
      volumeMounts:
          # name must match the volume name below
          - name: secret-volume
            mountPath: /etc/secret-volume
            readOnly: true
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
  restartPolicy: Never
```

Example 17.2. YAML of a Pod Populating Environment Variables with Secret Data

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
```

```
      image: busybox
      command: [ "/bin/sh", "-c", "export" ]
      env:
        - name: TEST_SECRET_USERNAME_ENV_VAR
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: username
  restartPolicy: Never
```

CHAPTER 18. CONFIGMAPS

18.1. OVERVIEW

Many applications require configuration using some combination of configuration files, command line arguments, and environment variables. These configuration artifacts should be decoupled from image content in order to keep containerized applications portable.

The **`ConfigMap`** object provides mechanisms to inject containers with configuration data while keeping containers agnostic of OpenShift Container Platform. A **`ConfigMap`** can be used to store fine-grained information like individual properties or coarse-grained information like entire configuration files or JSON blobs.

The **`ConfigMap`** API object holds key-value pairs of configuration data that can be consumed in pods or used to store configuration data for system components such as controllers. **`ConfigMap`** is similar to secrets, but designed to more conveniently support working with strings that do not contain sensitive information.

For example:

Example 18.1. ConfigMap Object Definition

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data: 1
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

1 Contains the configuration data.

Configuration data can be consumed in pods in a variety of ways. A **`ConfigMap`** can be used to:

1. Populate the value of environment variables.
2. Set command-line arguments in a container.
3. Populate configuration files in a volume.

Both users and system components may store configuration data in a **`ConfigMap`**.

18.2. CREATING CONFIGMAPS

You can use the following command to create a **ConfigMap** easily from directories, specific files, or literal values:

```
$ oc create configmap <configmap_name> [options]
```

The following sections cover the different ways you can create a **ConfigMap**.

18.2.1. Creating from Directories

Consider a directory with some files that already contain the data with which you want to populate a **ConfigMap**:

```
$ ls example-files
game.properties
ui.properties

$ cat example-files/game.properties
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30

$ cat example-files/ui.properties
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

You can use the following command to create a **ConfigMap** holding the content of each file in this directory:

```
$ oc create configmap game-config \
    --from-file=example-files/
```

When the **--from-file** option points to a directory, each file directly in that directory is used to populate a key in the **ConfigMap**, where the name of the key is the file name, and the value of the key is the content of the file.

For example, the above command creates the following **ConfigMap**:

```
$ oc describe configmaps game-config
Name:           game-config
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data
```

```
game.properties:        121 bytes
ui.properties:          83 bytes
```

You can see the two keys in the map are created from the file names in the directory specified in the command. Because the content of those keys may be large, the output of **`oc describe`** only shows the names of the keys and their sizes.

If you want to see the values of the keys, you can **`oc get`** the object with the **`-o`** option:

```
$ oc get configmaps game-config -o yaml

apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:34:05Z
  name: game-config
  namespace: default
  resourceVersion: "407"-
  selflink: /api/v1/namespaces/default/configmaps/game-config
  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

18.2.2. Creating from Files

You can also pass the **`--from-file`** option with a specific file, and pass it multiple times to the CLI. The following yields equivalent results to the Creating from Directories example:

1. Create the **`ConfigMap`** specifying a specific file:

```
$ oc create configmap game-config-2 \
    --from-file=example-files/game.properties \
    --from-file=example-files/ui.properties
```

2. Verify the results:

```
$ oc get configmaps game-config-2 -o yaml

apiVersion: v1
```

```
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config-2
  namespace: default
  resourceVersion: "516"
  selflink: /api/v1/namespaces/default/configmaps/game-config-2
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

You can also set the key to use for an individual file with the **`--from-file`** option by passing an expression of **`key=value`**. For example:

1. Create the **`ConfigMap`** specifying a key-value pair:

```
$ oc create configmap game-config-3 \
    --from-file=game-special-key=example-files/game.properties
```

2. Verify the results:

```
$ oc get configmaps game-config-3 -o yaml

apiVersion: v1
data:
  game-special-key: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  selflink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

18.2.3. Creating from Literal Values

You can also supply literal values for a **ConfigMap**. The **--from-literal** option takes a **key=value** syntax that allows literal values to be supplied directly on the command line:

1. Create the **ConfigMap** specifying a specific file:

```
$ oc create configmap special-config \
    --from-literal=special.how=very \
    --from-literal=special.type=charm
```

2. Verify the results:

```
$ oc get configmaps special-config -o yaml

apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selflink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985
```

18.3. USE CASES: CONSUMING CONFIGMAPS IN PODS

The following sections describe some uses cases when consuming **ConfigMap** objects in pods.

18.3.1. Consuming in Environment Variables

A **ConfigMap** can be used to populate the value of command line arguments. For example, consider the following **ConfigMap**:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

You can consume the keys of this **ConfigMap** in a pod using **configMapKeyRef** sections:

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

When this pod is run, its output will include the following lines:

```
SPECIAL_LEVEL_KEY=very
SPECIAL_TYPE_KEY=charm
```

18.3.2. Setting Command-line Arguments

A **`ConfigMap`** can also be used to set the value of the command or arguments in a container. This is accomplished using the Kubernetes substitution syntax **`$(VAR_NAME)`**. Consider the following **`ConfigMap`**:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

To inject values into the command line, you must consume the keys you want to use as environment variables, as in the Consuming in Environment Variables use case. Then you can refer to them in a container's command using the **`$(VAR_NAME)`** syntax.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
```

```
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY)
$(SPECIAL_TYPE_KEY)" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

When this pod is run, the output from the **test-container** container will be:

```
very charm
```

18.3.3. Consuming in Volumes

A **`ConfigMap`** can also be consumed in volumes. Returning again to the following example **`ConfigMap`**:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

You have a couple different options for consuming this **`ConfigMap`** in a volume. The most basic way is to populate the volume with files where the key is the file name and the content of the file is the value of the key:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
```

```
    - name: config-volume
      configMap:
        name: special-config
  restartPolicy: Never
```

When this pod is run, the output will be:

```
very
```

You can also control the paths within the volume where **ConfigMap** keys are projected:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/path/to/special-key" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: path/to/special-key
  restartPolicy: Never
```

When this pod is run, the output will be:

```
very
```

18.4. EXAMPLE: CONFIGURING REDIS

For a real-world example, you can configure Redis using a **ConfigMap**. To inject Redis with the recommended configuration for using Redis as a cache, the Redis configuration file should contain the following:

```
maxmemory 2mb
maxmemory-policy allkeys-lru
```

If your configuration file is located at ***example-files/redis/redis-config***, create a **ConfigMap** with it:

1. Create the **ConfigMap** specifying the configuration file:

```
$ oc create configmap example-redis-config \
    --from-file=example-files/redis/redis-config
```

2. Verify the results:

```
$ oc get configmap example-redis-config -o yaml

apiVersion: v1
data:
  redis-config: |
    maxmemory 2mb
    maxmemory-policy allkeys-lru
kind: ConfigMap
metadata:
  creationTimestamp: 2016-04-06T05:53:07Z
  name: example-redis-config
  namespace: default
  resourceVersion: "2985"
  selflink: /api/v1/namespaces/default/configmaps/example-redis-
config
  uid: d65739c1-fbbb-11e5-8a72-68f728db1985
```

Now, create a pod that uses this **`ConfigMap`**:

1. Create a pod definition like the following and save it to a file, for example ***redis-pod.yaml***:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: kubernetes/redis:v1
    env:
    - name: MASTER
      value: "true"
    ports:
    - containerPort: 6379
    resources:
      limits:
        cpu: "0.1"
    volumeMounts:
    - mountPath: /redis-master-data
      name: data
    - mountPath: /redis-master
      name: config
  volumes:
    - name: data
      emptyDir: {}
    - name: config
      configMap:
```

```
        name: example-redis-config
        items:
        - key: redis-config
          path: redis.conf
```

2. Create the pod:

```
$ oc create -f redis-pod.yaml
```

The newly-created pod has a **ConfigMap** volume that places the **redis-config** key of the **example-redis-config ConfigMap** into a file called ***redis.conf***. This volume is mounted into the ***/redis-master*** directory in the Redis container, placing our configuration file at ***/redis-master/redis.conf***, which is where the image looks for the Redis configuration file for the master.

If you **oc exec** into this pod and run the **redis-cli** tool, you can check that the configuration was applied correctly:

```
$ oc exec -it redis redis-cli
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "2097152"
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
```

18.5. RESTRICTIONS

A **ConfigMap** must be created before they are consumed in pods. Controllers can be written to tolerate missing configuration data; consult individual components configured via **ConfigMap** on a case-by-case basis.

ConfigMap objects reside in a project. They can only be referenced by pods in the same project.

The Kubelet only supports use of a **ConfigMap** for pods it gets from the API server. This includes any pods created using the CLI, or indirectly from a replication controller. It does not include pods created using the OpenShift Container Platform node's **--manifest-url** flag, its **--config** flag, or its REST API (these are not common ways to create pods).

CHAPTER 19. USING DAEMONSETS

19.1. OVERVIEW

Daemonsets are used to run a copy of a pod on specific or all nodes in an OpenShift Container Platform environment upon node creation.

Use daemonsets to create shared storage, run a logging pod on every node in your cluster, or deploy a monitoring agent on every node.

For more information on daemonsets, see the Kubernetes documentation.

19.2. CREATING DAEMONSETS

Important

Before creating daemonsets, ensure you have been given the required role by your OpenShift Container Platform administrator.

When creating Daemonsets, the **`nodeSelector`** field is used to indicate the nodes on which the Daemonset should deploy onto.

1. Define the daemonset yaml file:

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: hello-daemonset
spec:
  selector:
      matchLabels:
        name: hello-daemonset 1
  template:
    metadata:
      labels:
        name: hello-daemonset 2
    spec:
      containers:
      - image: openshift/hello-openshift
        imagePullPolicy: Always
        name: registry
        ports:
        - containerPort: 80
          protocol: TCP
        resources: {}
        terminationMessagePath: /dev/termination-log
      serviceAccount: default
      terminationGracePeriodSeconds: 10
```

1

The pod template.

2

The node selector indicating the appropriate labels. Must match the pod template above.

2. Create the daemonset object:

```
oc create -f daemonset.yaml
```

3. To verify that the pods were created, and that each node has a copy of the pod:

   a. Find the daemonset pods:

```
$ oc get pods
hello-daemonset-cx6md   1/1       Running   0          2m
hello-daemonset-e3md9   1/1       Running   0          2m
```

   b. View the pods to verify the pod has been placed onto the node:

```
$ oc describe pod/hello-daemonset-cx6md|grep Node
Node:        openshift-node01.hostname.com/10.14.20.134
$ oc describe pod/hello-daemonset-e3md9|grep Node
Node:        openshift-node02.hostname.com/10.14.20.137
```

Important

Currently, you cannot update a daemonset. You can delete a daemonset, but creating and using a new daemonset with a different template will recognize the existing pod as having matching labels dispite a mismatch in the pod template.

To update a daemonset, force a new pod to be created by deleting the pod or node.

CHAPTER 20. POD AUTOSCALING

20.1. OVERVIEW

A horizontal pod autoscaler, defined by a **`HorizontalPodAutoscaler`** object, specifies how the system should automatically increase or decrease the scale of a replication controller or deployment configuration, based on metrics collected from the pods that belong to that replication controller or deployment configuration.

Note

Horizontal pod autoscaling is supported starting in OpenShift Enterprise 3.1.1.

20.2. REQUIREMENTS FOR USING HORIZONTAL POD AUTOSCALERS

In order to use horizontal pod autoscalers, your cluster administrator must have properly configured cluster metrics.

20.3. SUPPORTED METRICS

The following metrics are supported by horizontal pod autoscalers:

Table 20.1. Metrics

Metric	Description
CPU Utilization	Percentage of the requested CPU

20.4. AUTOSCALING

You can create a horizontal pod autoscaler with the **`oc autoscale`** command and specify the minimum and maximum number of pods you want to run, as well as the CPU utilization your pods should target.

After a horizontal pod autoscaler is created, it begins attempting to query Heapster for metrics on the pods. It may take one to two minutes before Heapster obtains the initial metrics.

After metrics are available in Heapster, the horizontal pod autoscaler computes the ratio of the current metric utilization with the desired metric utilization, and scales up or down accordingly. The scaling will occur at a regular interval, but it may take one to two minutes before metrics make their way into Heapster.

For replication controllers, this scaling corresponds directly to the replicas of the replication controller. For deployment configurations, scaling corresponds directly to the replica count of the deployment configuration. Note that autoscaling applies only to the latest deployment in the **`Complete`** phase.

20.5. CREATING A HORIZONTAL POD AUTOSCALER

Use the **`oc autoscale`** command and specify at least the maximum number of pods you want to run at any given time. You can optionally specify the minimum number of pods and the average CPU utilization your pods should target, otherwise those are given default values from the OpenShift Container Platform server.

For example:

```
$ oc autoscale dc/frontend --min 1 --max 10 --cpu-percent=80
deploymentconfig "frontend" autoscaled
```

The above example creates a horizontal pod autoscaler with the following definition:

Example 20.1. Horizontal Pod Autoscaler Object Definition

```
apiVersion: extensions/v1beta1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend (1)
spec:
  scaleRef:
    kind: DeploymentConfig (2)
    name: frontend (3)
    apiVersion: v1 (4)
    subresource: scale
  minReplicas: 1 (5)
  maxReplicas: 10 (6)
  cpuUtilization:
    targetPercentage: 80 (7)
```

1

The name of this horizontal pod autoscaler object

2

The kind of object to scale

3

The name of the object to scale

4

The API version of the object to scale

5

The minimum number of replicas to which to scale down

6

The maximum number of replicas to which to scale up

7

The percentage of the requested CPU that each pod should ideally be using

20.6. VIEWING A HORIZONTAL POD AUTOSCALER

To view the status of a horizontal pod autoscaler:

```
$ oc get hpa/frontend
NAME              REFERENCE                                   TARGET
CURRENT   MINPODS        MAXPODS   AGE
frontend          DeploymentConfig/default/frontend/scale   80%
79%       1              10        8d

$ oc describe hpa/frontend
Name:                           frontend
Namespace:                      default
Labels:                         <none>
CreationTimestamp:              Mon, 26 Oct 2015 21:13:47 -0400
Reference:
DeploymentConfig/default/frontend/scale
Target CPU utilization:         80%
Current CPU utilization:        79%
Min pods:                       1
Max pods:                       10
```

CHAPTER 21. MANAGING VOLUMES

21.1. OVERVIEW

Containers are not persistent by default; on restart, their contents are cleared. Volumes are mounted file systems available to pods and their containers which may be backed by a number of host-local or network attached storage endpoints.

To ensure that the file system on the volume contains no errors and, if errors are present, to repair them when possible, OpenShift Container Platform invokes the **`fsck`** utility prior to the **`mount`** utility. This occurs when either adding a volume or updating an existing volume.

The simplest volume type is **`EmptyDir`**, which is a temporary directory on a single machine. Administrators may also allow you to request a persistent volume that is automatically attached to your pods.

Note

`EmptyDir` volume storage may be restricted by a quota based on the pods FSGroup, if enabled by your cluster administrator.

You can use the CLI command **`oc volume`** to add, update, or remove volumes and volume mounts for any object that has a pod template like replication controllers or deployment configurations. You can also list volumes in pods or any object that has a pod template.

21.2. GENERAL CLI USAGE

The **`oc volume`** command uses the following general syntax:

```
$ oc volume <object_selection> <operation> <mandatory_parameters>
<optional_parameters>
```

This topic uses the form ***`<object_type>/<name>`*** for ***`<object_selection>`*** in later examples, however you can choose one of the following options:

Table 21.1. Object Selection

Syntax	Description	Example
`<object_type> <name>`	Selects ***`<name>`*** of type ***`<object_type>`***.	**`deploymentConfig registry`**
`<object_type>/<name>`	Selects ***`<name>`*** of type ***`<object_type>`***.	**`deploymentConfig/registry`**

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

The **<*operation*>** can be one of **--add**, **--remove**, or **--list**.

Any **<*mandatory_parameters*>** or **<*optional_parameters*>** are specific to the selected operation and are discussed in later sections.

21.3. ADDING VOLUMES

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

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

Table 21.2. Supported Options for Adding Volumes

Option	Description	Default
--name	Name of the volume.	Automatically generated, if not specified.
-t, --type	Name of the volume source. Supported values: **emptyDir**, **hostPath**, **secret**, **configmap**, or **persistentVolumeClaim**.	**emptyDir**
-c, --containers	Select containers by name. It can also take wildcard **'*'** that matches any character.	**'*'**

Option	Description	Default
-m, --mount-path	Mount path inside the selected containers.	
--path	Host path. Mandatory parameter for **--type=hostPath**.	
--secret-name	Name of the secret. Mandatory parameter for **--type=secret**.	
--claim-name	Name of the persistent volume claim. Mandatory parameter for **--type=persistentVolumeClaim**.	
--source	Details of volume source as a JSON string. Recommended if the desired volume source is not supported by **--type**. See available volume sources	
-o, --output	Display the modified objects instead of updating them on the server. Supported values: **json**, **yaml**.	
--output-version	Output the modified objects with the given version.	**api-version**

Examples

Add a new volume source **emptyDir** to deployment configuration **registry**:

```
$ oc volume dc/registry --add
```

Add volume **v1** with secret **$ecret** for replication controller **r1** and mount inside the containers at ***/data***:

```
$ oc volume rc/r1 --add --name=v1 --type=secret --secret-name='$ecret'
--mount-path=/data
```

Add existing persistent volume **v1** with claim name **pvc1** to deployment configuration ***dc.json*** on disk, mount the volume on container **c1** at ***/data***, and update the deployment configuration on the server:

```
$ oc volume -f dc.json --add --name=v1 --type=persistentVolumeClaim \
  --claim-name=pvc1 --mount-path=/data --containers=c1
```

Add volume **v1** based on Git repository **https://github.com/namespace1/project1** with revision **5125c45f9f563** for all replication controllers:

```
$ oc volume rc --all --add --name=v1 \
  --source='{"gitRepo": {
                "repository":
"https://github.com/namespace1/project1",
                "revision": "5125c45f9f563"
            }}'
```

21.4. UPDATING VOLUMES

Updating existing volumes or volume mounts is the same as adding volumes, but with the **`--overwrite`** option:

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

Examples

Replace existing volume **v1** for replication controller **r1** with existing persistent volume claim **pvc1**:

```
$ oc volume rc/r1 --add --overwrite --name=v1 --
type=persistentVolumeClaim --claim-name=pvc1
```

Change deployment configuration **d1** mount point to ***/opt*** for volume **v1**:

```
$ oc volume dc/d1 --add --overwrite --name=v1 --mount-path=/opt
```

21.5. REMOVING VOLUMES

To remove a volume or volume mount from pod templates:

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

Table 21.3. Supported Options for Removing Volumes

Option	Description	Default
`--name`	Name of the volume.	

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

Some examples:

Remove a volume **v1** from deployment config **d1**:

```
$ oc volume dc/d1 --remove --name=v1
```

Unmount volume **v1** from container **c1** for deployment configuration **d1** and remove the volume **v1** if it is not referenced by any containers on **d1**:

```
$ oc volume dc/d1 --remove --name=v1 --containers=c1
```

Remove all volumes for replication controller **r1**:

```
$ oc volume rc/r1 --remove --confirm
```

21.6. LISTING VOLUMES

To list volumes or volume mounts for pods or pod templates:

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

List volume supported options:

Option	Description	Default
--name	Name of the volume.	

Option	Description	Default
`-c, --containers`	Select containers by name. It can also take wildcard `'*'` that matches any character.	`'*'`

Examples

List all volumes for pod **p1**:

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

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

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

CHAPTER 22. USING PERSISTENT VOLUMES

22.1. OVERVIEW

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

Note

The Installation and Configuration Guide provides instructions for cluster administrators on provisioning an OpenShift Container Platform cluster with persistent storage using NFS, GlusterFS, Ceph RBD, OpenStack Cinder, AWS EBS, GCE Persistent Disk, iSCSI, and Fibre Channel.

Storage can be made available to you by laying claims to the resource. You can make a request for storage resources using a **`PersistentVolumeClaim`** object; the claim is paired with a volume that generally matches your request.

22.2. REQUESTING STORAGE

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

Example 22.1. Persistent Volume Claim Object Definition

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "claim1"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "5Gi"
  volumeName: "pv0001"
```

22.3. VOLUME AND CLAIM BINDING

A **`PersistentVolume`** is a specific resource. A **`PersistentVolumeClaim`** is a request for a resource with specific attributes, such as storage size. In between the two is a process that matches a claim to an available volume and binds them together. This allows the claim to be used as a volume in a pod. OpenShift Container Platform 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
```

22.4. CLAIMS AS VOLUMES IN PODS

A **PersistentVolumeClaim** is used by a pod as a volume. OpenShift Container Platform 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 22.2. 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"
```

22.5. VOLUME AND CLAIM PRE-BINDING

If you know exactly what **PersistentVolume** you want your **PersistentVolumeClaim** to bind to, you can specify the PV in your PVC using the **volumeName** field. This method skips the normal matching and binding process. The PVC will only be able to bind to a PV that has the same name specified in **volumeName**. If such a PV with that name exists and is **Available**, the PV and PVC

will be bound regardless of whether the PV satisfies the PVC's label selector, access modes, and resource requests.

Example 22.3. Persistent Volume Claim Object Definition with volumeName

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "claim1"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "5Gi"
  volumeName: "pv0001"
```

Important

The ability to set **`claimRefs`** is a temporary workaround for the described use cases. A long-term solution for limiting who can claim a volume is in development.

Note

The cluster administrator should first consider configuring selector-label volume binding before resorting to setting **`claimRefs`** on behalf of users.

You may also want your cluster administrator to "reserve" the volume for only your claim so that nobody else's claim can bind to it before yours does. In this case, the administrator can specify the PVC in the PV using the **`claimRef`** field. The PV will only be able to bind to a PVC that has the same name and namespace specified in **`claimRef`**. The PVC's access modes and resource requests must still be satisfied in order for the PV and PVC to be bound, though the label selector is ignored.

Example 22.4. Persistent Volume Object Definition with claimRef

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  capacity:
    storage: 5Gi
  accessModes:
  - ReadWriteOnce
  nfs:
    path: /tmp
```

```
    server: 172.17.0.2
  persistentVolumeReclaimPolicy: Recycle
  claimRef:
    name: claim1
    namespace: default
```

Specifying a **volumeName** in your PVC does not prevent a different PVC from binding to the specified PV before yours does. Your claim will remain **Pending** until the PV is **Available**.

Specifying a **claimRef** in a PV does not prevent the specified PVC from being bound to a different PV. The PVC is free to choose another PV to bind to according to the normal binding process. Therefore, to avoid these scenarios and ensure your claim gets bound to the volume you want, you must ensure that both **volumeName** and **claimRef** are specified.

You can tell that your setting of **volumeName** and/or **claimRef** influenced the matching and binding process by inspecting a **Bound** PV and PVC pair for the **pv.kubernetes.io/bound-by-controller** annotation. The PVs and PVCs where you set the **volumeName** and/or **claimRef** yourself will have no such annotation, but ordinary PVs and PVCs will have it set to **"yes"**.

When a PV has its **claimRef** set to some PVC name and namespace, and is reclaimed according to a **Retain** or **Recycle** recycling policy, its **claimRef** will remain set to the same PVC name and namespace even if the PVC or the whole namespace no longer exists.

CHAPTER 23. EXECUTING REMOTE COMMANDS

23.1. OVERVIEW

You can use the CLI to execute remote commands in a container. This allows you to run general Linux commands for routine operations in the container.

Important

For security purposes, the **oc exec** command does not work when accessing privileged containers. See the CLI operations topic for more information.

23.2. BASIC USAGE

Support for remote container command execution is built into the CLI:

```
$ oc exec <pod> [-c <container>] <command> [<arg_1> ... <arg_n>]
```

For example:

```
$ oc exec mypod date
Thu Apr  9 02:21:53 UTC 2015
```

23.3. PROTOCOL

Clients initiate the execution of a remote command in a container by issuing a request to the Kubernetes API server:

```
/proxy/minions/<node_name>/exec/<namespace>/<pod>/<container>?command=
<command>
```

In the above URL:

- **<node_name>** is the FQDN of the node.
- **<namespace>** is the namespace of the target pod.
- **<pod>** is the name of the target pod.
- **<container>** is the name of the target container.
- **<command>** is the desired command to be executed.

For example:

```
/proxy/minions/node123.openshift.com/exec/myns/mypod/mycontainer?
command=date
```

Additionally, the client can add parameters to the request to indicate if:

- the client should send input to the remote container's command (stdin).
- the client's terminal is a TTY.
- the remote container's command should send output from stdout to the client.
- the remote container's command should send output from stderr to the client.

After sending an **`exec`** request to the API server, the client upgrades the connection to one that supports multiplexed streams; the current implementation uses **SPDY**.

The client creates one stream each for stdin, stdout, and stderr. To distinguish among the streams, the client sets the **`streamType`** header on the stream to one of **`stdin`**, **`stdout`**, or **`stderr`**.

The client closes all streams, the upgraded connection, and the underlying connection when it is finished with the remote command execution request.

Note

Administrators can see the Architecture guide for more information.

CHAPTER 24. COPYING FILES TO OR FROM A CONTAINER

24.1. OVERVIEW

You can use the CLI to copy local files to or from a remote directory in a container. This is a useful tool for copying database archives to and from your pods for backup and restore purposes. It can also be used to copy source code changes into a running pod for development debugging, when the running pod supports hot reload of source files.

24.2. BASIC USAGE

Support for copying local files to or from a container is built into the CLI:

```
$ oc rsync <source> <destination> [-c <container>]
```

For example, to copy a local directory to a pod directory:

```
$ oc rsync /home/user/source devpod1234:/src
```

Or to copy a pod directory to a local directory:

```
$ oc rsync devpod1234:/src /home/user/source
```

24.3. BACKING UP AND RESTORING DATABASES

Use **`oc rsync`** to copy database archives from an existing database container to a new database container's persistent volume directory.

Note

MySQL is used in the example below. Replace **`mysql|MYSQL`** with **`pgsql|PGSQL`** or **`mongodb|MONGODB`** and refer to the migration guide to find the exact commands for each of our supported database images. The example assumes an existing database container.

1. Back up the existing database from a running database pod:

```
$ oc rsh <existing db container>
# mkdir /var/lib/mysql/data/db_archive_dir
# mysqldump --skip-lock-tables -h ${MYSQL_SERVICE_HOST} -P
${MYSQL_SERVICE_PORT:-3306} \
  -u ${MYSQL_USER} --password="$MYSQL_PASSWORD" --all-databases >
/var/lib/mysql/data/db_archive_dir/all.sql
# exit
```

2. Remote sync the archive file to your local machine:

```
$ oc rsync <existing db container with db
archive>:/var/lib/mysql/data/db_archive_dir /tmp/.
```

3. Start a second MySQL pod into which to load the database archive file created above. The MySQL pod must have a unique **DATABASE_SERVICE_NAME**.

```
$ oc new-app mysql-persistent \
  -p MYSQL_USER=<archived mysql username> \
  -p MYSQL_PASSWORD=<archived mysql password> \
  -p MYSQL_DATABASE=<archived database name> \
  -p DATABASE_SERVICE_NAME='mysql2' 1
$ oc rsync /tmp/db_archive_dir new_dbpod1234:/var/lib/mysql/data
$ oc rsh new_dbpod1234
```

1

mysql is the default. In this example, **mysql2** is created.

4. Use the appropriate commands to restore the database in the new database container from the copied database archive directory:

MySQL

```
$ cd /var/lib/mysql/data/db_archive_dir
$ mysql -u root
$ source all.sql
$ GRANT ALL PRIVILEGES ON <dbname>.* TO '<your
username>'@'localhost'; FLUSH PRIVILEGES;
$ cd ../; rm -rf /var/lib/mysql/data/db_backup_dir
```

You now have two MySQL database pods running in your project with the archived database.

24.4. REQUIREMENTS

The **oc rsync** command uses the local **rsync** command if present on the client's machine. This requires that the remote container also have the **rsync** command.

If **rsync** is not found locally or in the remote container, then a tar archive will be created locally and sent to the container where **tar** will be used to extract the files. If **tar** is not available in the remote container, then the copy will fail.

The **tar** copy method does not provide the same functionality as **rsync**. For example, **rsync** creates the destination directory if it does not exist and will only send files that are different between the source and the destination.

Note

In Windows, the **cwRsync** client should be installed and added to the PATH for use with the **oc rsync** command.

24.5. SPECIFYING THE COPY SOURCE

The source argument of the **oc rsync** command must point to either a local directory or a pod directory. Individual files are not currently supported.

When specifying a pod directory the directory name must be prefixed with the pod name:

```
<pod name>:<dir>
```

Just as with standard **rsync**, if the directory name ends in a path separator (**/**), only the contents of the directory are copied to the destination. Otherwise, the directory itself is copied to the destination with all its contents.

24.6. SPECIFYING THE COPY DESTINATION

The destination argument of the **oc rsync** command must point to a directory. If the directory does not exist, but **rsync** is used for copy, the directory is created for you.

24.7. DELETING FILES AT THE DESTINATION

The **--delete** flag may be used to delete any files in the remote directory that are not in the local directory.

24.8. CONTINUOUS SYNCING ON FILE CHANGE

Using the **--watch** option causes the command to monitor the source path for any file system changes, and synchronizes changes when they occur. With this argument, the command runs forever.

Synchronization occurs after short quiet periods to ensure a rapidly changing file system does not result in continuous synchronization calls.

When using the **--watch** option, the behavior is effectively the same as manually invoking **oc rsync** repeatedly, including any arguments normally passed to **oc rsync**. Therefore, you can control the behavior via the same flags used with manual invocations of **oc rsync**, such as **--delete**.

24.9. ADVANCED RSYNC FEATURES

The **oc rsync** command exposes fewer command line options than standard **rsync**. In the case that you wish to use a standard **rsync** command line option which is not available in **oc rsync** (for example the **--exclude-from=FILE** option), it may be possible to use standard **rsync** 's **--rsh** (**-e**) option or **RSYNC_RSH** environment variable as a workaround, as follows:

```
$ rsync --rsh='oc rsh' --exclude-from=FILE SRC POD:DEST
```

or:

```
$ export RSYNC_RSH='oc rsh'
$ rsync --exclude-from=FILE SRC POD:DEST
```

Both of the above examples configure standard **rsync** to use **oc rsh** as its remote shell program to enable it to connect to the remote pod, and are an alternative to running **oc rsync**.

CHAPTER 25. PORT FORWARDING

25.1. OVERVIEW

You can use the CLI to forward one or more local ports to a pod. This allows you to listen on a given or random port locally, and have data forwarded to and from given ports in the pod.

25.2. BASIC USAGE

Support for port forwarding is built into the CLI:

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...
[<local_port_n>:]<remote_port_n>]
```

The CLI listens on each local port specified by the user, forwarding via the protocol described below.

Ports may be specified using the following formats:

5000	The client listens on port 5000 locally and forwards to 5000 in the pod.
6000:5000	The client listens on port 6000 locally and forwards to 5000 in the pod.
:5000 or **0:5000**	The client selects a free local port and forwards to 5000 in the pod.

For example, to listen on ports **5000** and **6000** locally and forward data to and from ports **5000** and **6000** in the pod, run:

```
$ oc port-forward <pod> 5000 6000
```

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

```
$ oc port-forward <pod> 8888:5000
```

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

```
$ oc port-forward <pod> :5000
```

Or, alternatively:

```
$ oc port-forward <pod> 0:5000
```

25.3. PROTOCOL

Clients initiate port forwarding to a pod by issuing a request to the Kubernetes API server:

```
/proxy/minions/<node_name>/portForward/<namespace>/<pod>
```

In the above URL:

- **<node_name>** is the FQDN of the node.
- **<namespace>** is the namespace of the target pod.
- **<pod>** is the name of the target pod.

For example:

```
/proxy/minions/node123.openshift.com/portForward/myns/mypod
```

After sending a port forward request to the API server, the client upgrades the connection to one that supports multiplexed streams; the current implementation uses **SPDY**.

The client creates a stream with the **port** header containing the target port in the pod. All data written to the stream is delivered via the Kubelet to the target pod and port. Similarly, all data sent from the pod for that forwarded connection is delivered back to the same stream in the client.

The client closes all streams, the upgraded connection, and the underlying connection when it is finished with the port forwarding request.

Note

Administrators can see the Architecture guide for more information.

CHAPTER 26. SHARED MEMORY

26.1. OVERVIEW

There are two types of shared memory objects in Linux: System V and POSIX. The containers in a pod share the IPC namespace of the pod infrastructure container and so are able to share the System V shared memory objects. This document describes how they can also share POSIX shared memory objects.

26.2. POSIX SHARED MEMORY

POSIX shared memory requires that a tmpfs be mounted at ***/dev/shm***. The containers in a pod do not share their mount namespaces so we use volumes to provide the same ***/dev/shm*** into each container in a pod. The following example shows how to set up POSIX shared memory between two containers.

shared-memory.yaml

```
---
apiVersion: v1
id: hello-openshift
kind: Pod
metadata:
  name: hello-openshift
  labels:
    name: hello-openshift
spec:
  volumes:                          1
    - name: dshm
      emptyDir:
        medium: Memory
  containers:
    - image: kubernetes/pause
      name: hello-container1
      ports:
        - containerPort: 8080
          hostPort: 6061
      volumeMounts:                 2
        - mountPath: /dev/shm
          name: dshm
    - image: kubernetes/pause
      name: hello-container2
      ports:
        - containerPort: 8081
          hostPort: 6062
      volumeMounts:                 3
        - mountPath: /dev/shm
          name: dshm
```

1

specifies the tmpfs volume **dshm**.

2

enables POSIX shared memory for **hello-container1** via **dshm**.

3

enables POSIX shared memory for **hello-container2** via **dshm**.

Create the pod using the ***shared-memory.yaml*** file:

```
$ oc create -f shared-memory.yaml
```

CHAPTER 27. APPLICATION HEALTH

27.1. OVERVIEW

In software systems, components can become unhealthy due to transient issues (such as temporary connectivity loss), configuration errors, or problems with external dependencies. OpenShift Container Platform applications have a number of options to detect and handle unhealthy containers.

27.2. CONTAINER HEALTH CHECKS USING PROBES

A probe is a Kuberbetes action that periodically performs diagnostics on a running container. Currently, two types of probes exist, each serving a different purpose:

Liveness Probe	A liveness probe checks if the container in which it is configured is still running. If the liveness probe fails, the kubelet kills the container, which will be subjected to its restart policy. Set a liveness check by configuring the **`template.spec.containers.livenessprobe`** stanza of a pod configuration.
Readiness Probe	A readiness probe determines if a container is ready to service requests. If the readiness probe fails a container, the endpoints controller ensures the container has its IP address removed from the endpoints of all services. A readiness probe can be used to signal to the endpoints controller that even though a container is running, it should not receive any traffic from a proxy. Set a readiness check by configuring the **`template.spec.containers.readinessprobe`** stanza of a pod configuration.

The exact timing of a probe is controlled by two fields, both expressed in units of seconds:

Field	Description
`initialDelaySeconds`	How long to wait after the container starts to begin the probe.
`timeoutSeconds`	How long to wait for the probe to finish (default: **`1`**). If this time is exceeded, OpenShift Container Platform considers the probe to have failed.

Both probes can be configured in three ways:

HTTP Checks

The kubelet uses a web hook to determine the healthiness of the container. The check is deemed successful if the hook returns with 200 or 399. The following is an example of a readiness check using the HTTP checks method:

Example 27.1. Readiness HTTP check

```
...
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

A HTTP check is ideal for complex applications that can return with a 200 status when completely initialized.

Container Execution Checks

The kubelet executes a command inside the container. Exiting the check with status 0 is considered a success. The following is an example of a liveness check using the container execution method:

Example 27.2. Liveness Container Execution Check

```
...
livenessProbe:
  exec:
    command:
    - cat
    - /tmp/health
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

TCP Socket Checks

The kubelet attempts to open a socket to the container. The container is only considered healthy if the check can establish a connection. The following is an example of a liveness check using the TCP socket check method:

Example 27.3. Liveness TCP Socket Check

```
...
livenessProbe:
  tcpSocket:
    port: 8080
```

```
    initialDelaySeconds: 15
    timeoutSeconds: 1
...
```

A TCP socket check is ideal for applications that do not start listening until initialization is complete.

For more information on health checks, see the Kubernetes documentation.

CHAPTER 28. EVENTS

28.1. OVERVIEW

Events in OpenShift Container Platform are modeled based on events that happen to API objects in an OpenShift Container Platform cluster. Events allow OpenShift Container Platform to record information about real-world events in a resource-agnostic manner. They also allow developers and administrators to consume information about system components in a unified way.

28.2. VIEWING EVENTS WITH THE CLI

You can get a list of events in a given project using the following command:

```
$ oc get events [-n <project>]
```

28.3. VIEWING EVENTS IN THE CONSOLE

You can see events in your project from the web console from the **Browse** → **Events** page. Many other objects, such as pods and deployments, have their own **Events** tab as well, which shows events related to that object.

28.4. COMPREHENSIVE LIST OF EVENTS

This section describes the events of OpenShift Container Platform.

Table 28.1. Configuration Events

Name	Description
`FailedValidation`	Failed pod configuration validation.

Table 28.2. Container Events

Name	Description
`BackOff`	Back-off restarting failed the container.
`Created`	Container created.

Name	Description
`Failed`	Pull/Create/Start failed.
`Killing`	Killing the container.
`Started`	Container started.

Table 28.3. Health Events

Name	Description
`Unhealthy`	Container is unhealthy.

Table 28.4. Image Events

Name	Description
`BackOff`	Back off Ctr Start, image pull.
`ErrImageNeverPull`	The image's **NeverPull Policy** is violated.
`Failed`	Failed to pull the image.
`InspectFailed`	Failed to inspect the image.
`Pulled`	Successfully pulled the image or the container image is already present on the machine.
`Pulling`	Pulling the image.

Table 28.5. Image Manager Events

Name	Description
FreeDiskSpaceFailed	Free disk space failed.
InvalidDiskCapacity	Invalid disk capacity.

Table 28.6. Node Events

Name	Description
FailedMount	Volume mount failed.
HostNetworkNotSupported	Host network not supported.
HostPortConflict	Host/port conflict.
InsufficientFreeCPU	Insufficient free CPU.
InsufficientFreeMemory	Insufficient free memory.
KubeletSetupFailed	Kubelet setup failed.
NilShaper	Undefined shaper.
NodeNotReady	Node is not ready.
NodeNotSchedulable	Node is not schedulable.

Name	Description
`NodeReady`	Node is ready.
`NodeSchedulable`	Node is schedulable.
`NodeSelectorMismatching`	Node selector mismatch.
`OutOfDisk`	Out of disk.
`Rebooted`	Node rebooted.
`Starting`	Starting kubelet.

Table 28.7. Pod Worker Events

Name	Description
`FailedSync`	Pod sync failed.

Table 28.8. System Events

Name	Description
`SystemOOM`	There is an OOM (out of memory) situation on the cluster.

CHAPTER 29. DOWNWARD API

29.1. OVERVIEW

The downward API is a mechanism that allows containers to consume information about API objects without coupling to OpenShift Container Platform. Such information includes the pod's name, namespace, and resource values. Containers can consume information from the downward API using environment variables or a volume plug-in.

29.2. SELECTING FIELDS

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

Field	Description
`fieldPath`	The path of the field to select, relative to the pod.
`apiVersion`	The API version to interpret the **`fieldPath`** selector within.

Currently, the valid selectors in the v1 API include:

Selector	Description
`metadata.name`	The pod's name. This is supported in both environment variables and volumes.
`metadata.namespace`	The pod's namespace.This is supported in both environment variables and volumes.
`metadata.labels`	The pod's labels. This is only supported in volumes and not in environment variables.
`metadata.annotations`	The pod's annotations. This is only supported in volumes and not in environment variables.
`status.podIP`	The pod's IP. This is only supported in environment variables and not volumes.

The **apiVersion** field, if not specified, defaults to the API version of the enclosing pod template.

29.3. CONSUMING THE CONTAINER VALUES USING THE DOWNWARD API

29.3.1. 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

    - 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
```

    - Create the pod from the ***pod.json*** file:

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

    - Check the container's logs for the **MY_POD_NAME** and **MY_POD_NAMESPACE** values:

```
$ oc logs -p dapi-env-test-pod
```

29.3.2. Using the Volume Plug-in

Another mechanism for consuming the downward API is using a volume plug-in. The downward API volume plug-in creates a volume with configured fields projected into files. The **`metadata`** field of the **`VolumeSource`** API object is used to configure this volume. The plug-in supports the following fields:

- Pod name
- Pod namespace
- Pod annotations
- Pod labels

Example 29.1. Downward API Volume Plug-in Configuration

```
spec:
  volumes:
    - name: podinfo
      metadata: 1
        items: 2
          - name: "labels" 3
            fieldRef:
              fieldPath: metadata.labels 4
```

1 The **`metadata`** field of the volume source configures the downward API volume.

2 The **`items`** field holds a list of fields to project into the volume.

3 The name of the file to project the field into.

4 The selector of the field to project.

For example:

1. Create a ***`volume-pod.json`*** file:

```
kind: Pod
apiVersion: v1
metadata:
  labels:
    zone: us-east-coast
    cluster: downward-api-test-cluster1
    rack: rack-123
  name: dapi-volume-test-pod
  annotations:
    annotation1: 345
    annotation2: 456
spec:
  containers:
    - name: volume-test-container
      image: gcr.io/google_containers/busybox
      command: ["sh", "-c", "cat /etc/labels /etc/annotations"]
      volumeMounts:
        - name: podinfo
          mountPath: /etc
          readOnly: false
  volumes:
    - name: podinfo
      metadata:
        items:
          - name: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - name: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
  restartPolicy: Never
```

2. Create the pod from the ***volume-pod.json*** file:

```
$ oc create -f volume-pod.json
```

3. Check the container's logs and verify the presence of the configured fields:

```
$ oc logs -p dapi-volume-test-pod
cluster=downward-api-test-cluster1
rack=rack-123
zone=us-east-coast
annotation1=345
annotation2=456
kubernetes.io/config.source=api
```

29.4. CONSUMING CONTAINER RESOURCES USING THE DOWNWARD API

When creating pods, you can use the downward API to inject information about computing resource requests and limits so that image and application authors can correctly create an image for specific environments.

You can do this using both the environment variable and volume plug-in methods.

29.4.1. Using Environment Variables

1. When creating a pod configuration, specify environment variables that correspond to the contents of the **`resources`** field in the **`spec.container`** field:

```
...
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox:1.24
      command: [ "/bin/sh", "-c", "env" ]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      env:
        - name: MY_CPU_REQUEST
          valueFrom:
            resourceFieldRef:
              resource: requests.cpu
        - name: MY_CPU_LIMIT
          valueFrom:
            resourceFieldRef:
              resource: limits.cpu
        - name: MY_MEM_REQUEST
          valueFrom:
            resourceFieldRef:
              resource: requests.memory
        - name: MY_MEM_LIMIT
          valueFrom:
            resourceFieldRef:
              resource: limits.memory
...
```

   If the resource limits are not included in the container configuration, the downward API defaults to the node's CPU and memory allocatable values.

2. Create the pod from the ***`pod.json`*** file:

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

29.4.2. Using the Volume Plug-in

1. When creating a pod configuration, use the **`spec.volumes.downwardAPI.items`** field to describe the desired resources that correspond to the **`spec.resources`** field:

```
....
spec:
  containers:
    - name: client-container
      image: gcr.io/google_containers/busybox:1.24
      command: ["sh", "-c", "while true; do echo; if [[ -e
/etc/cpu_limit ]]; then cat /etc/cpu_limit; fi; if [[ -e
/etc/cpu_request ]]; then cat /etc/cpu_request; fi; if [[ -e
/etc/mem_limit ]]; then cat /etc/mem_limit; fi; if [[ -e
/etc/mem_request ]]; then cat /etc/mem_request; fi; sleep 5;
done"]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      volumeMounts:
        - name: podinfo
          mountPath: /etc
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "cpu_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.cpu
          - path: "cpu_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.cpu
          - path: "mem_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.memory
          - path: "mem_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.memory
....
```

If the resource limits are not included in the container configuration, the downward API defaults to the node's CPU and memory allocatable values.

2. Create the pod from the ***volume-pod.json*** file:

```
$ oc create -f volume-pod.json
```

CHAPTER 30. MANAGING ENVIRONMENT VARIABLES

30.1. SETTING AND UNSETTING ENVIRONMENT VARIABLES

OpenShift Container Platform provides the **`oc set env`** command to set or unset environment variables for objects that have a pod template, such as replication controllers or deployment configurations. It can also list environment variables in pods or any object that has a pod template. This command can also be used on **`BuildConfig`** objects.

30.2. LIST ENVIRONMENT VARIABLES

To list environment variables in pods or pod templates:

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

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

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

30.3. SET ENVIRONMENT VARIABLES

To set environment variables in the pod templates:

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

Set environment options:

Option	Description
`-e, --env=<KEY>=<VAL>`	Set given key value pairs of environment variables.
`--overwrite`	Confirm updating existing environment variables.

In the following example, both commands modify environment variable **`STORAGE`** in the deployment config **`registry`**. The first adds, with value **`/data`**. The second updates, with value **`/opt`**.

```
$ oc set env dc/registry STORAGE=/data
$ oc set env dc/registry --overwrite STORAGE=/opt
```

The following example finds environment variables in the current shell whose names begin with **`RAILS`**_ and adds them to the replication controller **`r1`** on the server:

```
$ env | grep RAILS_ | oc set env rc/r1 -e -
```

The following example does not modify the replication controller defined in file **`rc.json`**. Instead, it

writes a YAML object with updated environment **STORAGE=/local** to new file **rc.yaml**.

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

30.3.1. Automatically Added Environment Variables

Table 30.1. Automatically Added Environment Variables

Variable Name
<SVCNAME>_SERVICE_HOST
<SVCNAME>_SERVICE_PORT

Example Usage

The service **KUBERNETES** which exposes TCP port 53 and has been allocated cluster IP address 10.0.0.11 produces the following environment variables:

```
KUBERNETES_SERVICE_PORT=53
MYSQL_DATABASE=root
KUBERNETES_PORT_53_TCP=tcp://10.0.0.11:53
KUBERNETES_SERVICE_HOST=10.0.0.11
```

Note

Use the **oc rsh** command to SSH into your container and run **oc set env** to list all available variables.

30.4. UNSET ENVIRONMENT VARIABLES

To unset environment variables in the pod templates:

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

Important

The trailing hyphen (**-**, U+2D) is required.

This example removes environment variables **ENV1** and **ENV2** from deployment config **d1**:

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

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

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

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

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

CHAPTER 31. JOBS

31.1. OVERVIEW

A job, in contrast to a replication controller, runs any number of pods to completion. A job tracks the overall progress of a ask and updates its status with information about active, succeeded, and failed pods. Deleting a job will clean up any pods it created. Jobs are part of the Kubernetes API, which can be managed with **oc** commands like other object types.

See the Kubernetes documentation for more information about jobs.

31.2. CREATING A JOB

A job configuration consists of the following key parts:

- A pod template, which describes the application the pod will create.
- An optional **`parallelism`** parameter, which specifies how many successful pod completions are needed to finish a job. If not specified, this defaults to the value in the **`completions`** parameter.
- An optional **`completions`** parameter, specifying how many concurrently running pods should execute a job. If not specified, this value defaults to one.

The following is an example of a **`job`** resource:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1     1
  completions: 1     2
  template:          3
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

1. Optional value for how many pods a job should run in parallel, defaults to **`completions`**.
2. Optional value for how many successful pod completions is needed to mark a job completed, defaults to one.
3. Template for the pod the controller creates.

31.3. SCALING A JOB

A job can be scaled up or down by changing the **parallelism** parameter accordingly. You can also use the **oc scale** command with the **--replicas** option, which, in the case of jobs, modifies the **parallelism** parameter.

The following command uses the example job above, and sets the **parallelism** parameter to three:

```
$ oc scale job pi --replicas=3
```

Note

Scaling replication controllers also uses the **oc scale** command with the **--replicas** option, but instead changes the **replicas** parameter of a replication controller configuration.

31.4. SETTING MAXIMUM DURATION

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

The maximum duration is counted from the time when a first pod gets scheduled in the system, and defines how long a job can be active. It tracks overall time of an execution and is irrelevant to the number of completions (number of pods needed to execute a task). After reaching the specified timeout, the job is terminated by OpenShift Container Platform.

The following example shows the part of a **Job** specifying **activeDeadlineSeconds** field for 30 minutes:

```
spec:
  activeDeadlineSeconds: 1800
```

CHAPTER 32. SCHEDULED JOBS

32.1. OVERVIEW

A *scheduled job* builds on a regular job by allowing you to specifically schedule how the job should be run. Scheduled jobs are part of the Kubernetes API, which can be managed with **oc** commands like other object types.

Note

As of OpenShift Container Platform 3.3.1, Scheduled Jobs is a feature in Technology Preview.

32.2. CREATING A SCHEDULED JOB

A scheduled job configuration consists of the following key parts:

- A schedule specified in cron format.
- A job template used when creating the next job.
- An optional deadline (in seconds) for starting the job if it misses its scheduled time for any reason. Missed jobs executions will be counted as failed ones. If not specified, there is no deadline.
- **`ConcurrencyPolicy`**: An optional concurrency policy, specifying how to treat concurrent jobs within a scheduled job. Only one of the following concurrent policies may be specified. If not specified, this defaults to allowing concurrent executions.
  - **`Allow`** allows Scheduled Jobs to run concurrently.
  - **`Forbid`** forbids concurrent runs, skipping the next run if the previous has not finished yet.
  - **`Replace`** cancels the currently running job and replaces it with a new one.
- An optional flag allowing the suspension of a scheduled job. If set to **`true`**, all subsequent executions will be suspended.

The following is an example of a **`ScheduledJob`** resource:

```
apiVersion: batch/v2alpha1
kind: ScheduledJob
metadata:
  name: pi
spec:
  schedule: */1 * * * ?     1
  jobTemplate:              2
    spec:
      template:
        spec:
          containers:
          - name: pi
```

```
            image: perl
            command: ["perl",  "-Mbignum=bpi", "-wle", "print
bpi(2000)"]
          restartPolicy: Never
```

1. Schedule for the job. In this example, a job will run every minute.
2. Job template. This is similar to the job example.

32.3. KNOWN ISSUES

32.3.1. Unable to Edit a Scheduled Job

There is a known issue when invoking **`oc edit scheduledjob`** due to an error that was already fixed in the latest version. However, due to significant code changes, this was not backported.

One possible solution is to use **`oc patch`** command instead of **`oc edit`**.

32.3.2. Unable to Change Concurrency Policy

There is a known issue when changing concurrency policy where no new jobs are created after that operation is run. The issue is still under investigation in the latest version.

CHAPTER 33. REVISION HISTORY: DEVELOPER GUIDE

33.1. THU FEB 16 2017

Affected Topic	Description of Change
Secrets	Corrected an example YAML file and added missing steps.
Using Persistent Volumes	Added a new Volume and Claim Pre-binding section
Service Serving Certificate Secrets	Added a note to the Service Serving Certificate Secrets section clarifying the use of the service DNS name.
Getting Traffic into the Cluster	Added more details about Ingress.

33.2. MON FEB 06 2017

Affected Topic	Description of Change
Service Serving Certificate Secrets	Removed Tech Preview note from the Service Serving Certificate Secrets section.

33.3. MON JAN 30 2017

Affected Topic	Description of Change
Builds	Updated the example Dockerfile path to point to a file, not a directory.
Managing Environment Variables	Removed redundant information and CLI reference material; rearranged sections to match user process.

33.4. WED JAN 25 2017

Affected Topic	Description of Change
Builds	Updated a Note box in the Accessing Build Logs section advising that the build defaults for an administrator can be overridden for non-binary builds by passing `--build-loglevel` to `oc start-build`.

33.5. WED JAN 18 2017

OpenShift Container Platform 3.4 initial release.

Affected Topic	Description of Change
Projects	Added a new Bookmarking Page States section, which discusses that OpenShift Container Platform now bookmarks page states, which is helpful in saving label filters.
Application Tutorials → Setting Up a Nexus Mirror for Maven	New topic on setting up a containerized Nexus repository for Maven dependency caching.
Templates	Added details about template writing for the best user experience.
	Added examples of a template object definition, template description metadata, template object labels, generating a parameter value, setting an explicit value as the default value, and a full template with parameter definitions and references,
Builds	Added a note box to the Generic Webhooks section explaining that OpenShift Container Platform permits builds to be triggered via the generic webhook even if an invalid request payload is presented.
	Added new sections on adding secrets to Source Strategy, Docker Strategy, and Custom Strategy build configurations from the web console.
	Added the Assigning Builds to Specific Nodes section.
	Added example for custom build image label.

Affected Topic	Description of Change
Managing Images	Added a new Tag Naming section reviewing recommended conventions and best practices when naming tags.
	Added a new Writing Image Streams for S2I Builders section.
Deployments → Basic Deployment Operations	Updated with new **`oc rollout`** commands.
	Added a new Adding Secrets to Deployment Configurations from the Web Console section.
Deployments → Kubernetes Deployments Support	New topic detailing Technology Preview support for the new Kubernetes-provided deployments object type.
Secrets	Added details about **`stringData`** for secrets.
Managing Images	Added information about the **`supports`** annotation on image streams.