OpenShift Container Platform 4.11

CI/CD

Contains information on builds for OpenShift Container Platform

OpenShift Container Platform 4.11 CI/CD

Contains information on builds for OpenShift Container Platform

Legal Notice

Abstract

Builds for the OpenShift Container Platform

Table of Contents

CHAPTER 1. OPENSHIFT CONTAINER PLATFORM CI/CD OVERVIEW

OpenShift Container Platform is an enterprise-ready Kubernetes platform for developers, which enables organizations to automate the application delivery process through DevOps practices, such as continuous integration (CI) and continuous delivery (CD). To meet your organizational needs, the OpenShift Container Platform provides the following CI/CD solutions:

- OpenShift Builds

- OpenShift Pipelines

- OpenShift GitOps

1.1. OPENSHIFT BUILDS

With OpenShift Builds, you can create cloud-native apps by using a declarative build process. You can define the build process in a YAML file that you use to create a BuildConfig object. This definition includes attributes such as build triggers, input parameters, and source code. When deployed, the BuildConfig object typically builds a runnable image and pushes it to a container image registry.

OpenShift Builds provides the following extensible support for build strategies:

- Docker build

- Source-to-image (S2I) build

- Custom build

For more information, see Understanding image builds

1.2. OPENSHIFT PIPELINES

OpenShift Pipelines provides a Kubernetes-native CI/CD framework to design and run each step of the CI/CD pipeline in its own container. It can scale independently to meet the on-demand pipelines with predictable outcomes.

For more information, see Understanding OpenShift Pipelines.

1.3. OPENSHIFT GITOPS

OpenShift GitOps is an Operator that uses Argo CD as the declarative GitOps engine. It enables GitOps workflows across multicluster OpenShift and Kubernetes infrastructure. Using OpenShift GitOps, administrators can consistently configure and deploy Kubernetes-based infrastructure and applications across clusters and development lifecycles.

For more information, see About Red Hat OpenShift GitOps .

1.4. JENKINS

Jenkins automates the process of building, testing, and deploying applications and projects. OpenShift Developer Tools provides a Jenkins image that integrates directly with the OpenShift Container Platform. Jenkins can be deployed on OpenShift by using the Samples Operator templates or certified Helm chart.

CHAPTER 2. BUILDS

2.1. UNDERSTANDING IMAGE BUILDS

2.1.1. Builds

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

OpenShift Container Platform uses Kubernetes by creating containers from build images and pushing them to a container image registry.

Build objects share common characteristics including inputs for a build, the requirement to complete a build process, logging the build process, publishing resources from successful builds, and publishing the final status of the build. Builds take advantage of resource restrictions, specifying limitations on resources such as CPU usage, memory usage, and build or pod execution time.

The OpenShift Container Platform build system provides extensible support for build strategies that are based on selectable types specified in the build API. There are three primary build strategies available:

- Docker build

- Source-to-image (S2I) build

- Custom build

By default, docker builds and S2I builds are supported.

The resulting object of a build depends on the builder used to create it. For docker and S2I builds, the resulting objects are runnable images. For custom builds, the resulting objects are whatever the builder image author has specified.

Additionally, the pipeline build strategy can be used to implement sophisticated workflows:

- Continuous integration

- Continuous deployment

2.1.1.1. Docker build

OpenShift Container Platform uses Buildah to build a container image from a Dockerfile. For more information on building container images with Dockerfiles, see the Dockerfile reference documentation.

TIP

If you set Docker build arguments by using the **buildArgs** array, see Understand how ARG and FROM interact in the Dockerfile reference documentation.

2.1.1.2. Source-to-image build

Source-to-image (S2I) is a tool for building reproducible container images. It produces ready-to-run images by injecting application source into a container image and assembling a new image. The new image incorporates the base image, the builder, and built source and is ready to use with the **buildah**

run command. S2I supports incremental builds, which re-use previously downloaded dependencies, previously built artifacts, and so on.

2.1.1.3. Custom build

The custom build strategy allows developers to define a specific builder image responsible for the entire build process. Using your own builder image allows you to customize your build process.

A custom builder image is a plain container image embedded with build process logic, for example for building RPMs or base images.

Custom builds run with a high level of privilege and are not available to users by default. Only users who can be trusted with cluster administration permissions should be granted access to run custom builds.

2.1.1.4. Pipeline build

IMPORTANT

The Pipeline build strategy is deprecated in OpenShift Container Platform 4. Equivalent and improved functionality is present in the OpenShift Container Platform Pipelines based on Tekton.

Jenkins images on OpenShift Container Platform are fully supported and users should follow Jenkins user documentation for defining their **jenkinsfile** in a job or store it in a Source Control Management system.

The Pipeline build strategy allows developers to define a Jenkins pipeline for use by the Jenkins pipeline plugin. The build can be started, monitored, and managed by OpenShift Container Platform in the same way as any other build type.

Pipeline workflows are defined in a **jenkinsfile**, either embedded directly in the build configuration, or supplied in a Git repository and referenced by the build configuration.

2.2. UNDERSTANDING BUILD CONFIGURATIONS

The following sections define the concept of a build, build configuration, and outline the primary build strategies available.

2.2.1. BuildConfigs

A build configuration describes a single build definition and a set of triggers for when a new build is created. Build configurations are 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.

A build configuration, or **BuildConfig**, is characterized by a build strategy and one or more sources. The strategy determines the process, while the sources provide its input.

Depending on how you choose to create your application using OpenShift Container Platform, a **BuildConfig** is typically generated automatically for you if you use the web console or CLI, and it can be edited at any time. Understanding the parts that make up a **BuildConfig** and their available options can help if you choose to manually change your configuration later.

The following example **BuildConfig** results in a new build every time a container image tag or the source code changes:

BuildConfig object definition

```
kind: BuildConfig
apiVersion: build.openshift.io/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
    git:
      uri: "https://github.com/openshift/ruby-hello-world"
  strategy: 5
    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 creates 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 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. See the documentation for each source type for details.

[5] The **strategy** section describes the build strategy used to execute the build. You can specify a **Source** , **Docker**, or **Custom** strategy here. This example uses the **ruby-20-centos7** container image that Source-to-image (S2I) uses for the application build.

[6] After the container image is successfully built, it is pushed into the repository described in the **output** section.

[7] The **postCommit** section defines an optional build hook.

2.3. CREATING BUILD INPUTS

Use the following sections for an overview of build inputs, instructions on how to use inputs to provide source content for builds to operate on, and how to use build environments and create secrets.

2.3.1. Build inputs

A build input provides source content for builds to operate on. You can use the following build inputs to provide sources in OpenShift Container Platform, listed in order of precedence:

- Inline Dockerfile definitions

- Content extracted from existing images

- Git repositories

- Binary (Local) inputs

- Input secrets

- External artifacts

You can combine multiple inputs in a single build. However, as the inline Dockerfile takes precedence, it can overwrite any other file named Dockerfile provided by another input. Binary (local) input and Git repositories are mutually exclusive inputs.

You can use input secrets when you do not want certain resources or credentials used during a build to be available in the final application image produced by the build, or want to consume a value that is defined in a secret resource. External artifacts can be used to pull in additional files that are not available as one of the other build input types.

When you run a build:

1. A working directory is constructed and all input content is placed in the working directory. For example, the input Git repository is cloned into the working directory, and files specified from input images are copied into the working directory using the target path.

2. The build process changes directories into the **contextDir**, if one is defined.

3. The inline Dockerfile, if any, is written to the current directory.

4. The content from the current directory is provided to the build process for reference by the Dockerfile, custom builder logic, or **assemble** script. This means any input content that resides outside the **contextDir** is ignored by the build.

The following example of a source definition 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
    ref: "master"
  images:
  - from:
      kind: ImageStreamTag
```

```
    name: myinputimage:latest
    namespace: mynamespace
  paths:
  - destinationDir: app/dir/injected/dir ❷
    sourcePath: /usr/lib/somefile.jar
contextDir: "app/dir" ❸
dockerfile: "FROM centos:7\nRUN yum install -y httpd" ❹
```

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

❷ **/usr/lib/somefile.jar** from **myinputimage** is stored in **<workingdir>/app/dir/injected/dir**.

❸ The working directory for the build becomes **<original_workingdir>/app/dir**.

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

2.3.2. Dockerfile source

When you supply a **dockerfile** value, the content of this field is written to disk as a file named **dockerfile**. This is done after other input sources are processed, so if the input source repository contains a Dockerfile in the root directory, it is overwritten with this content.

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

```
source:
  dockerfile: "FROM centos:7\nRUN yum install -y httpd" ❶
```

❶ The **dockerfile** field contains an inline Dockerfile that is built.

Additional resources

- The typical use for this field is to provide a Dockerfile to a docker strategy build.

2.3.3. Image source

You can add additional files to the build process with 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 is loaded and the indicated files and directories are copied into the context directory of the build process. This is the same directory into which the source repository content is cloned. If the source path ends in /**.** then the content of the directory is copied, but the directory itself is not 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
    ref: "master"
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**
>
> If your cluster uses an **ImageContentSourcePolicy** object to configure repository mirroring, you can use only global pull secrets for mirrored registries. You cannot add a pull secret to a project.

Images that require pull secrets

When using an input image that requires a pull secret, you can link the pull secret to the service account used by the build. By default, builds use the **builder** service account. The pull secret is automatically added to the build if the secret contains a credential that matches the repository hosting the input image. To link a pull secret to the service account used by the build, run:

```
$ oc secrets link builder dockerhub
```

> **NOTE**
>
> This feature is not supported for builds using the custom strategy.

Images on mirrored registries that require pull secrets

When using an input image from a mirrored registry, if you get a **build error: failed to pull image** message, you can resolve the error by using either of the following methods:

- Create an input secret that contains the authentication credentials for the builder image's repository and all known mirrors. In this case, create a pull secret for credentials to the image registry and its mirrors.

- Use the input secret as the pull secret on the **BuildConfig** object.

2.3.4. Git source

When specified, source code is fetched from the supplied location.

If you supply an inline Dockerfile, it overwrites the Dockerfile in the **contextDir** of the Git repository.

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

```
source:
  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 Uniform Resource Identifier (URI) to the remote Git repository of the source code. You must specify the value of the **ref** field to check out a specific Git reference. A valid **ref** can be a SHA1 tag or a branch name. The default value of the **ref** field is **master**.

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.

If the **ref** field denotes a pull request, the system uses a **git fetch** operation and then checkout **FETCH_HEAD**.

When no **ref** value is provided, OpenShift Container Platform performs a shallow clone (**--depth=1**). In this case, only the files associated with the most recent commit on the default branch (typically **master**) are downloaded. This results in repositories downloading faster, but without the full commit history. To perform a full **git clone** of the default branch of a specified repository, set **ref** to the name of the default branch (for example **main**).

> **WARNING**
>
> Git clone operations that go through a proxy that is performing man in the middle (MITM) TLS hijacking or reencrypting of the proxied connection do not work.

2.3.4.1. 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 build configuration. You can configure both an HTTP and HTTPS proxy to use. Both fields are optional. Domains for which no proxying should be performed can also be specified in the **NoProxy** field.

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

```
source:
  git:
    uri: "https://github.com/openshift/ruby-hello-world"
    ref: "master"
    httpProxy: http://proxy.example.com
    httpsProxy: https://proxy.example.com
    noProxy: somedomain.com, otherdomain.com
```

> **NOTE**
>
> For Pipeline strategy builds, given the current restrictions with the Git plugin for Jenkins, any Git operations through the Git plugin do not leverage the HTTP or HTTPS proxy defined in the **BuildConfig**. The Git plugin only uses the proxy configured in the Jenkins UI at the Plugin Manager panel. This proxy is then used for all git interactions within Jenkins, across all jobs.

Additional resources

- You can find instructions on how to configure proxies through the Jenkins UI at JenkinsBehindProxy.

2.3.4.2. Source Clone Secrets

Builder pods require access to any Git repositories defined as source for a build. Source clone secrets are used to provide the builder pod with access it would not normally have access to, such as private repositories or repositories with self-signed or untrusted SSL certificates.

The following source clone secret configurations are supported:

- .gitconfig File

- Basic Authentication

- SSH Key Authentication

- Trusted Certificate Authorities
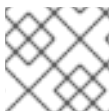
> **NOTE**
>
> You can also use combinations of these configurations to meet your specific needs.

2.3.4.2.1. Automatically adding a source clone secret to a build configuration

When a **BuildConfig** is created, OpenShift Container Platform can automatically populate its source clone secret reference. This behavior allows the resulting builds to automatically use the credentials stored in the referenced secret to authenticate to a remote Git repository, without requiring further configuration.

To use this functionality, a secret containing the Git repository credentials must exist in the namespace in which the **BuildConfig** is later created. This secrets must include one or more annotations prefixed with **build.openshift.io/source-secret-match-uri-**. The value of each of these annotations is a Uniform Resource Identifier (URI) pattern, which is defined as follows. When a **BuildConfig** is created without a source clone secret reference and its Git source URI matches a URI pattern in a secret annotation, OpenShift Container Platform automatically inserts a reference to that secret in the **BuildConfig**.

Prerequisites

A URI pattern must consist of:

- A valid scheme: ***://**, **git://**, **http://**, **https://** or **ssh://**

- A host: ***`** or a valid hostname or IP address optionally preceded by ***.**

- A path: **/*** or **/** followed by any characters optionally including ***** characters

In all of the above, a ***** character is interpreted as a wildcard.

> **IMPORTANT**
>
> URI patterns must match Git source URIs which are conformant to [RFC3986](). Do not include a username (or password) component in a URI pattern.
>
> For example, if you use **ssh://git@bitbucket.atlassian.com:7999/ATLASSIAN jira.git** for a git repository URL, the source secret must be specified as **ssh://bitbucket.atlassian.com:7999/*** (and not **ssh://git@bitbucket.atlassian.com:7999/***).
>
> ```
> $ oc annotate secret mysecret \
>     'build.openshift.io/source-secret-match-uri-1=ssh://bitbucket.atlassian.com:7999/*'
> ```

Procedure

If multiple secrets match the Git URI of a particular **BuildConfig**, OpenShift Container Platform selects the secret with the longest match. This allows for basic overriding, as in the following example.

The following fragment shows two partial source clone secrets, the first matching any server in the domain **mycorp.com** accessed by HTTPS, and the second overriding access to servers **mydev1.mycorp.com** and **mydev2.mycorp.com**:

```
kind: Secret
apiVersion: v1
metadata:
  name: matches-all-corporate-servers-https-only
  annotations:
    build.openshift.io/source-secret-match-uri-1: https://*.mycorp.com/*
data:
  ...
---
kind: Secret
```

```
apiVersion: v1
metadata:
  name: override-for-my-dev-servers-https-only
  annotations:
    build.openshift.io/source-secret-match-uri-1: https://mydev1.mycorp.com/*
    build.openshift.io/source-secret-match-uri-2: https://mydev2.mycorp.com/*
data:
  ...
```

- Add a **build.openshift.io/source-secret-match-uri-** annotation to a pre–existing secret using:

  ```
  $ oc annotate secret mysecret \
      'build.openshift.io/source-secret-match-uri-1=https://*.mycorp.com/*'
  ```

2.3.4.2.2. Manually adding a source clone secret

Source clone secrets can be added manually to a build configuration by adding a **sourceSecret** field to the **source** section inside the **BuildConfig** and setting it to the name of the secret that you created. In this example, it is the **basicsecret**.

```
apiVersion: "build.openshift.io/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"
  strategy:
    sourceStrategy:
      from:
        kind: "ImageStreamTag"
        name: "python-33-centos7:latest"
```

Procedure

You can also use the **oc set build-secret** command to set the source clone secret on an existing build configuration.

- To set the source clone secret on an existing build configuration, enter the following command:

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

2.3.4.2.3. Creating a secret from a .gitconfig file

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

Procedure

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

```
$ oc create secret generic <secret_name> --from-file=<path/to/.gitconfig>
```

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

2.3.4.2.4. Creating a secret from a .gitconfig file for secured Git

If your Git server is secured with two-way SSL and user name with password, you must add the certificate files to your source build and add references to the certificate files in the **.gitconfig** file.

Prerequisites

- You must have Git credentials.

Procedure

Add the certificate files to your source build and add references to the certificate files in the **.gitconfig** file.

1. Add the **client.crt**, **cacert.crt**, and **client.key** files to the **/var/run/secrets/openshift.io/source/** folder in the application source code.

2. In the **.gitconfig** file for the server, add the **[http]** section shown in the following example:

```
# cat .gitconfig
```

   Example output

```
[user]
        name = <name>
        email = <email>
[http]
        sslVerify = false
        sslCert = /var/run/secrets/openshift.io/source/client.crt
        sslKey = /var/run/secrets/openshift.io/source/client.key
        sslCaInfo = /var/run/secrets/openshift.io/source/cacert.crt
```

3. Create the secret:

```
$ oc create secret generic <secret_name> \
--from-literal=username=<user_name> \     1
--from-literal=password=<password> \       2
--from-file=.gitconfig=.gitconfig \
```

```
--from-file=client.crt=/var/run/secrets/openshift.io/source/client.crt \
--from-file=cacert.crt=/var/run/secrets/openshift.io/source/cacert.crt \
--from-file=client.key=/var/run/secrets/openshift.io/source/client.key
```

1 The user's Git user name.

2 The password for this user.
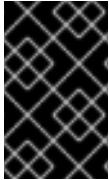
> **IMPORTANT**
>
> To avoid having to enter your password again, be sure to specify the source-to-image (S2I) image in your builds. However, if you cannot clone the repository, you must still specify your user name and password to promote the build.

Additional resources

- **/var/run/secrets/openshift.io/source/** folder in the application source code.

2.3.4.2.5. Creating a secret from source code basic authentication

Basic authentication requires either a combination of **--username** and **--password**, or a token to authenticate against the software configuration management (SCM) server.

Prerequisites

- User name and password to access the private repository.

Procedure

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

   ```
   $ oc create secret generic <secret_name> \
       --from-literal=username=<user_name> \
       --from-literal=password=<password> \
       --type=kubernetes.io/basic-auth
   ```

2. Create a basic authentication secret with a token:

   ```
   $ oc create secret generic <secret_name> \
       --from-literal=password=<token> \
       --type=kubernetes.io/basic-auth
   ```

2.3.4.2.6. Creating a secret from source code 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.

Procedure

1. Generate SSH key credentials:

```
$ ssh-keygen -t ed25519 -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.

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

```
$ oc create secret generic <secret_name> \
    --from-file=ssh-privatekey=<path/to/ssh/private/key> \
    --from-file=<path/to/known_hosts> \ ❶
    --type=kubernetes.io/ssh-auth
```

❶ Optional: Adding this field enables strict server host key check.

> **WARNING**
>
> Skipping the **known_hosts** file while creating the secret makes the build vulnerable to a potential man-in-the-middle (MITM) attack.

> **NOTE**
>
> Ensure that the **known_hosts** file includes an entry for the host of your source code.

2.3.4.2.7. Creating a secret from source code trusted certificate authorities

The set of Transport Layer Security (TLS) certificate authorities (CA) 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 can create a secret that contains the certificate or disable TLS verification.

If you create a secret for the CA certificate, OpenShift Container Platform uses it to access your Git server during the Git clone operation. Using this method is significantly more secure than disabling Git SSL verification, which accepts any TLS certificate that is presented.

Procedure

Create a secret with a CA certificate file.

1. If your CA uses Intermediate Certificate Authorities, combine the certificates for all CAs in a **ca.crt** file. Enter the following command:

```
$ cat intermediateCA.crt intermediateCA.crt rootCA.crt > ca.crt
```

a. Create the secret:

```
$ oc create secret generic mycert --from-file=ca.crt=</path/to/file> ❶
```

❶   You must use the key name **ca.crt**.

2.3.4.2.8. Source secret combinations

You can combine the different methods for creating source clone secrets for your specific needs.

2.3.4.2.8.1. Creating a SSH-based authentication secret with a .**gitconfig** file

You can combine the different methods for creating source clone secrets for your specific needs, such as a SSH-based authentication secret with a **.gitconfig** file.

Prerequisites

- SSH authentication

- .gitconfig file

Procedure

- To create a SSH-based authentication secret with a **.gitconfig** file, run:

```
$ oc create secret generic <secret_name> \
    --from-file=ssh-privatekey=<path/to/ssh/private/key> \
    --from-file=<path/to/.gitconfig> \
    --type=kubernetes.io/ssh-auth
```

2.3.4.2.8.2. Creating a secret that combines a .gitconfig file and CA certificate

You can combine the different methods for creating source clone secrets for your specific needs, such as a secret that combines a **.gitconfig** file and certificate authority (CA) certificate.

Prerequisites

- .gitconfig file

- CA certificate

Procedure

- To create a secret that combines a **.gitconfig** file and CA certificate, run:

```
$ oc create secret generic <secret_name> \
    --from-file=ca.crt=<path/to/certificate> \
    --from-file=<path/to/.gitconfig>
```

2.3.4.2.8.3. Creating a basic authentication secret with a CA certificate

You can combine the different methods for creating source clone secrets for your specific needs, such as a secret that combines a basic authentication and certificate authority (CA) certificate.

Prerequisites

- Basic authentication credentials

- CA certificate

Procedure

- Create a basic authentication secret with a CA certificate, run:

    ```
    $ oc create secret generic <secret_name> \
        --from-literal=username=<user_name> \
        --from-literal=password=<password> \
        --from-file=ca-cert=</path/to/file> \
        --type=kubernetes.io/basic-auth
    ```

2.3.4.2.8.4. Creating a basic authentication secret with a .gitconfig file

You can combine the different methods for creating source clone secrets for your specific needs, such as a secret that combines a basic authentication and **.gitconfig** file.

Prerequisites

- Basic authentication credentials

- **.gitconfig** file

Procedure

- To create a basic authentication secret with a **.gitconfig** file, run:

    ```
    $ oc create secret generic <secret_name> \
        --from-literal=username=<user_name> \
        --from-literal=password=<password> \
        --from-file=</path/to/.gitconfig> \
        --type=kubernetes.io/basic-auth
    ```

2.3.4.2.8.5. Creating a basic authentication secret with a .gitconfig file and CA certificate

You can combine the different methods for creating source clone secrets for your specific needs, such as a secret that combines a basic authentication, **.gitconfig** file, and certificate authority (CA) certificate.

Prerequisites

- Basic authentication credentials

- **.gitconfig** file

- CA certificate

Procedure

- To create a basic authentication secret with a **.gitconfig** file and CA certificate, run:

```
$ oc create secret generic <secret_name> \
    --from-literal=username=<user_name> \
    --from-literal=password=<password> \
    --from-file=</path/to/.gitconfig> \
    --from-file=ca-cert=</path/to/file> \
    --type=kubernetes.io/basic-auth
```

2.3.5. Binary (local) source

Streaming content 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 these 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, like an image change trigger, is not possible. This is 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. You can also specify a URL to a file. Then, the builder 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. Then, the builder extracts the contents of the archive within the build context directory. With **--from-dir**, you can also specify a URL to an archive, which is extracted.

- **--from-archive**: The archive you specify is sent to the builder, where it is extracted within the build context directory. 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 previously listed cases:

- If your **BuildConfig** already has a **Binary** source type defined, it is effectively 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 a 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. 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** has 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.

2.3.6. Input secrets and config maps

> **IMPORTANT**
>
> To prevent the contents of input secrets and config maps from appearing in build output container images, use build volumes in your Docker build and source-to-image build strategies.

In some scenarios, build operations require credentials or other configuration data to access dependent resources, but it is undesirable for that information to be placed in source control. You can define input secrets and input config maps for this purpose.

For example, when building a Java application with Maven, you can set up a private mirror of Maven Central or JCenter that is accessed by private keys. To download libraries from that private mirror, you have to supply the following:

1. A **settings.xml** file configured with the mirror's URL and connection settings.

2. A private key referenced in the settings file, such as ~/**.ssh**/**id_rsa**.

For security reasons, you do not want to expose your credentials in the application image.

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

2.3.6.1. What is a secret?

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 plugin or the system can use secrets to perform actions on behalf of a pod.

YAML Secret Object Definition

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
  namespace: my-namespace
type: Opaque 1
```

```
  data: 2
    username: <username> 3
    password: <password>
  stringData: 4
    hostname: myapp.mydomain.com 5
```

1	Indicates the structure of the secret's key names and values.
2	The allowable format for the keys in the **data** field must meet the guidelines in the **DNS_SUBDOMAIN** value in the Kubernetes identifiers glossary.
3	The value associated with keys in the **data** map must be base64 encoded.
4	Entries in the **stringData** map are converted to base64 and the entry are then moved to the **data** map automatically. This field is write-only. The value is only be returned by the **data** field.
5	The value associated with keys in the **stringData** map is made up of plain text strings.

2.3.6.1.1. 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.

2.3.6.1.2. Types of Secrets

The value in the **type** field indicates the structure of the secret's key names and values. The type can be used to enforce the presence of user names and keys in the secret object. If you do not want validation, use the **opaque** type, which is the default.

Specify one of the following types to trigger minimal server-side validation to ensure the presence of specific key names in the secret data:

- **kubernetes.io/service-account-token**. Uses a service account token.

- **kubernetes.io/dockercfg**. Uses the **.dockercfg** file for required Docker credentials.

- **kubernetes.io/dockerconfigjson**. Uses the **.docker/config.json** file for required Docker credentials.

- **kubernetes.io/basic-auth**. Use with basic authentication.

- **kubernetes.io/ssh-auth**. Use with SSH key authentication.

- **kubernetes.io/tls**. Use with TLS certificate authorities.

Specify **type= Opaque** if you do not want validation, which means the secret does not claim to conform to any convention for key names or values. An **opaque** secret, allows for unstructured **key:value** pairs that can contain arbitrary values.

NOTE

You can specify other arbitrary types, such as **example.com/my-secret-type**. These types are not enforced server–side, but indicate that the creator of the secret intended to conform to the key/value requirements of that type.

2.3.6.1.3. Updates to secrets

When you modify the value of a secret, the value used by an already running pod does not dynamically change. To change a secret, you must delete the original pod and create a new pod, in some cases 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, the version of the secret that is used for the pod is not 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 report this information, so that a controller could restart ones using an old **resourceVersion**. In the interim, do not update the data of existing secrets, but create new ones with distinct names.

2.3.6.2. 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 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.

Procedure

- Use the create command to create a secret object from a JSON or YAML file:

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

  For example, you can create a secret from your local **.docker/config.json** file:

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

  This command generates a JSON specification of the secret named **dockerhub** and creates the object.

  ### YAML Opaque Secret Object Definition

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque 1
data:
  username: <username>
  password: <password>
```

1   Specifies an *opaque* secret.

Docker Configuration JSON File Secret Object Definition

```
apiVersion: v1
kind: Secret
metadata:
  name: aregistrykey
  namespace: myapps
type: kubernetes.io/dockerconfigjson 1
data:

  .dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2cg
YXV0aCBrZXlzCg== 2
```

1   Specifies that the secret is using a docker configuration JSON file.

2   The output of a base64-encoded the docker configuration JSON file

2.3.6.3. Using secrets

After creating secrets, you can create a pod to reference your secret, get logs, and delete the pod.

Procedure

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
   ```

Additional resources

- Example YAML files with secret data:

  YAML Secret That Will Create Four Files

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: <username>
  password: <password>
stringData:
  hostname: myapp.mydomain.com
  secret.properties: |-
    property1=valueA
    property2=valueB
```

1   File contains decoded values.

2   File contains decoded values.

3   File contains the provided string.

4   File contains the provided data.

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
```

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
```

YAML of a Build Config Populating Environment Variables with Secret Data

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: TEST_SECRET_USERNAME_ENV_VAR
        valueFrom:
          secretKeyRef:
            name: test-secret
            key: username
```

2.3.6.4. Adding input secrets and config maps

To provide credentials and other configuration data to a build without placing them in source control, you can define input secrets and input config maps.

In some scenarios, build operations require credentials or other configuration data to access dependent resources. To make that information available without placing it in source control, you can define input secrets and input config maps.

Procedure

To add an input secret, config maps, or both to an existing **BuildConfig** object:

1. Create the **ConfigMap** object, if it does not exist:

   ```
   $ oc create configmap settings-mvn \
       --from-file=settings.xml=<path/to/settings.xml>
   ```

   This creates a new config map named **settings-mvn**, which contains the plain text content of the **settings.xml** file.

You can alternatively apply the following YAML to create the config map:

```
apiVersion: core/v1
kind: ConfigMap
metadata:
  name: settings-mvn
data:
  settings.xml: |
    <settings>
    … # Insert maven settings here
    </settings>
```

2. Create the **Secret** object, if it does not exist:

```
$ oc create secret generic secret-mvn \
    --from-file=ssh-privatekey=<path/to/.ssh/id_rsa>
    --type=kubernetes.io/ssh-auth
```

This creates a new secret named **secret-mvn**, which contains the base64 encoded content of the **id_rsa** private key.

TIP

You can alternatively apply the following YAML to create the input secret:

```
apiVersion: core/v1
kind: Secret
metadata:
  name: secret-mvn
type: kubernetes.io/ssh-auth
data:
  ssh-privatekey: |
    # Insert ssh private key, base64 encoded
```

3. Add the config map and secret to the **source** section in the existing **BuildConfig** object:

```
source:
  git:
    uri: https://github.com/wildfly/quickstart.git
  contextDir: helloworld
  configMaps:
    - configMap:
        name: settings-mvn
  secrets:
    - secret:
        name: secret-mvn
```

To include the secret and config map in a new **BuildConfig** object, run the following command:

```
$ oc new-build \
    openshift/wildfly-101-centos7~https://github.com/wildfly/quickstart.git \
```

```
--context-dir helloworld --build-secret "secret-mvn" \
--build-config-map "settings-mvn"
```

During the build, the **settings.xml** and **id_rsa** files are copied into the directory where the source code is located. In 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 definition:

```
source:
  git:
    uri: https://github.com/wildfly/quickstart.git
  contextDir: helloworld
  configMaps:
    - configMap:
        name: settings-mvn
      destinationDir: ".m2"
  secrets:
    - secret:
        name: secret-mvn
      destinationDir: ".ssh"
```

You can also specify the destination directory when creating a new **BuildConfig** object:

```
$ oc new-build \
    openshift/wildfly-101-centos7~https://github.com/wildfly/quickstart.git \
    --context-dir helloworld --build-secret "secret-mvn:.ssh" \
    --build-config-map "settings-mvn:.m2"
```

In both cases, the **settings.xml** file is added to the **./.m2** directory of the build environment, and the **id_rsa** key is added to the **./.ssh** directory.

2.3.6.5. Source-to-image strategy

When using a **Source** strategy, all defined input 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 working directory of the image. The final directory in the **destinationDir** path is created if it does not exist in the builder image. All preceding directories in the **destinationDir** must exist, or an error will occur.
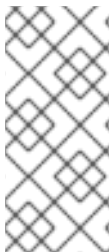
> **NOTE**
>
> Input secrets are added as world-writable, have **0666** permissions, and are truncated to size zero after executing the **assemble** script. This means that the secret files exist in the resulting image, but they are empty for security reasons.
>
> Input config maps are not truncated after the **assemble** script completes.

2.3.6.6. Docker strategy

When using a docker strategy, you can add all defined input 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 are copied into the same directory in which the Dockerfile is located. If you specify a relative path as **destinationDir**, then the secrets are 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.

Example of a Dockerfile referencing secret and config map data

```
FROM centos/ruby-22-centos7

USER root
COPY ./secret-dir /secrets
COPY ./config /

# Create a shell script that will output secrets and ConfigMaps when the image is run
RUN echo '#!/bin/sh' > /input_report.sh
RUN echo '(test -f /secrets/secret1 && echo -n "secret1=" && cat /secrets/secret1)' >>
/input_report.sh
RUN echo '(test -f /config && echo -n "relative-configMap=" && cat /config)' >> /input_report.sh
RUN chmod 755 /input_report.sh

CMD ["/bin/sh", "-c", "/input_report.sh"]
```

> **IMPORTANT**
>
> Users normally remove their input secrets from the final application image so that the secrets are not present in the container running from that image. However, the secrets still exist in the image itself in the layer where they were added. This removal is part of the Dockerfile itself.
>
> To prevent the contents of input secrets and config maps from appearing in the build output container images and avoid this removal process altogether, use build volumes in your Docker build strategy instead.

2.3.6.7. Custom strategy

When using a Custom strategy, all the defined input secrets and config maps are available in the builder container in the **/var/run/secrets/openshift.io/build** directory. The custom build image must use these secrets and config maps appropriately. With the Custom strategy, you can define secrets as described in Custom strategy options.

There is no technical difference between existing strategy secrets and the input secrets. However, your builder image can distinguish between them and use them differently, based on your build use case.

The input 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.

> **IMPORTANT**
>
> If a pull secret for the registry exists in both the namespace and the node, builds default to using the pull secret in the namespace.

2.3.7. External artifacts

It is not recommended to store binary files in a source repository. Therefore, you must 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:

.s2i/**bin**/**assemble** File

```
#!/bin/sh
APP_VERSION=1.0
wget http://repository.example.com/app/app-$APP_VERSION.jar -O app.jar
```

.s2i/**bin**/**run** File

```
#!/bin/sh
exec java -jar app.jar
```

For a Docker build strategy, you must modify the Dockerfile and invoke shell commands with the **RUN instruction**:

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 Dockerfile or **assemble** script.

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)

2.3.8. Using docker credentials for private registries

You can supply builds with a **.docker/config.json** file with valid credentials for private container registries. This allows you to push the output image into a private container image registry or pull a builder image from the private container image registry that requires authentication.

You can supply credentials for multiple repositories within the same registry, each with credentials specific to that registry path.

NOTE

For the OpenShift Container Platform container image registry, this is not required 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:
  index.docker.io/v1/: 1
    auth: "YWRfbGzhcGU6R2labnRib21ifTE=" 2
    email: "user@example.com" 3
  docker.io/my-namespace/my-user/my-image: 4
    auth: "GzhYWRGU6R2fbclabnRgbkSp="''
    email: "user@example.com"
  docker.io/my-namespace: 5
    auth: "GzhYWRGU6R2deesfrRgbkSp="''
    email: "user@example.com"
```

1 URL of the registry.

2 Encrypted password.

3 Email address for the login.

4 URL and credentials for a specific image in a namespace.

5 URL and credentials for a registry namespace.

You can define multiple container image registries or define multiple repositories in the same registry. 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 can be used to store configuration and passwords.

Prerequisites

- You must have a **.docker/config.json** file.

Procedure

1. Create the secret from your local **.docker/config.json** file:

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

   This generates a JSON specification of the secret named **dockerhub** and creates the object.

2. 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 previous example is **dockerhub**:

   ```
   spec:
     output:
       to:
   ```

```
    kind: "DockerImage"
    name: "private.registry.com/org/private-image:latest"
  pushSecret:
    name: "dockerhub"
```

You can 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
```

You can also link the push secret to the service account used by the build instead of specifying the **pushSecret** field. By default, builds use the **builder** service account. The push secret is automatically added to the build if the secret contains a credential that matches the repository hosting the build's output image.

```
$ oc secrets link builder dockerhub
```

3. Pull the builder container image from a private container image 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"
```

You can 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
```
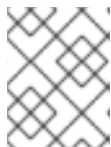
> **NOTE**
>
> This example uses **pullSecret** in a Source build, but it is also applicable in Docker and Custom builds.

You can also link the pull secret to the service account used by the build instead of specifying the **pullSecret** field. By default, builds use the **builder** service account. The pull secret is automatically added to the build if the secret contains a credential that matches the repository hosting the build's input image. To link the pull secret to the service account used by the build instead of specifying the **pullSecret** field, run:

```
$ oc secrets link builder dockerhub
```

> **NOTE**
>
> You must specify a **from** image in the **BuildConfig** spec to take advantage of this feature. Docker strategy builds generated by **oc new-build** or **oc new-app** may not do this in some situations.

2.3.9. Build environments

As with pod environment variables, build environment variables can be defined in terms of references to other resources or variables using the Downward API. There are some exceptions, which are noted.

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

> **NOTE**
>
> Referencing container resources using **valueFrom** in build environment variables is not supported as the references are resolved before the container is created.

2.3.9.1. Using build fields as environment variables

You can inject information about the build object by setting the **fieldPath** environment variable source to the **JsonPath** of the field from which you are interested in obtaining the value.

> **NOTE**
>
> Jenkins Pipeline strategy does not support **valueFrom** syntax for environment variables.

Procedure

- Set the **fieldPath** environment variable source to the **JsonPath** of the field from which you are interested in obtaining the value:

```
env:
  - name: FIELDREF_ENV
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
```

2.3.9.2. Using secrets as environment variables

You can make key values from secrets available as environment variables using the **valueFrom** syntax.
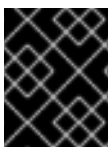
> **IMPORTANT**
>
> This method shows the secrets as plain text in the output of the build pod console. To avoid this, use input secrets and config maps instead.

Procedure

- To use a secret as an environment variable, set the **valueFrom** syntax:

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
       - name: MYVAL
         valueFrom:
```

```
      secretKeyRef:
        key: myval
        name: mysecret
```

Additional resources

- [Input secrets and config maps](#)

2.3.10. Service serving certificate secrets

Service serving certificate secrets are intended to support complex middleware applications that need out-of-the-box certificates. It has the same settings as the server certificates generated by the administrator tooling for nodes and masters.

Procedure

To secure communication to your service, have the cluster generate a signed serving certificate/key pair into a secret in your namespace.

- Set the **service.beta.openshift.io/serving-cert-secret-name** annotation on your service with the value set to the name you want to use for your secret.
  Then, your **PodSpec** can mount that secret. When it is available, your pod runs. The certificate is 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. The certificate/key pair is automatically replaced when it gets close to expiration. View the expiration date in the **service.beta.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 certificate authority (CA) bundle in the **/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt** file that is automatically mounted in their pod.

The signature algorithm for this feature is **x509.SHA256WithRSA**. To manually rotate, delete the generated secret. A new certificate is created.

2.3.11. Secrets 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.

2.4. MANAGING BUILD OUTPUT

Use the following sections for an overview of and instructions for managing build output.

2.4.1. Build output

Builds that use the docker or source-to-image (S2I) strategy result in the creation of a new container image. The image is then pushed to the container image 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 image registry and tagged in the specified imagestream. 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.

Output to an ImageStreamTag

```
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "sample-image:latest"
```

Output to a docker Push Specification

```
spec:
  output:
    to:
      kind: "DockerImage"
      name: "my-registry.mycompany.com:5000/myimages/myimage:tag"
```

2.4.2. Output image environment variables

docker and source-to-image (S2I) strategy 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

Additionally, any user-defined environment variable, for example those configured with S2I] or docker strategy options, will also be part of the output image environment variable list.

2.4.3. Output image labels

docker and source-to-image (S2I)` 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 build configuration.

Custom Labels to be Applied to Built Images

```
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "my-image:latest"
    imageLabels:
    - name: "vendor"
      value: "MyCompany"
    - name: "authoritative-source-url"
      value: "registry.mycompany.com"
```

2.5. USING BUILD STRATEGIES

The following sections define the primary supported build strategies, and how to use them.

2.5.1. Docker build

OpenShift Container Platform uses Buildah to build a container image from a Dockerfile. For more information on building container images with Dockerfiles, see the Dockerfile reference documentation.

TIP

If you set Docker build arguments by using the **buildArgs** array, see Understand how ARG and FROM interact in the Dockerfile reference documentation.

2.5.1.1. Replacing Dockerfile FROM image

You can replace the **FROM** instruction of the Dockerfile with the **from** of the **BuildConfig** object. If the Dockerfile uses multi-stage builds, the image in the last **FROM** instruction will be replaced.

Procedure

To replace the **FROM** instruction of the Dockerfile with the **from** of the **BuildConfig**.

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

2.5.1.2. Using Dockerfile path

By default, docker builds use a 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 a different file name than the default Dockerfile, such as **MyDockerfile**, or a path to a Dockerfile in a subdirectory, such as **dockerfiles/app1/Dockerfile**.

Procedure

To use the **dockerfilePath** field for the build to use a different path to locate your Dockerfile, set:

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

2.5.1.3. Using docker 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 build configuration.

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.

Procedure

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/"
```

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

2.5.1.4. Adding docker build arguments

You can set docker build arguments using the **buildArgs** array. The build arguments are passed to docker when a build is started.

TIP

See Understand how ARG and FROM interact in the Dockerfile reference documentation.

Procedure

To set docker build arguments, add entries to the **buildArgs** array, which is located in the **dockerStrategy** definition of the **BuildConfig** object. For example:

```
dockerStrategy:
...
  buildArgs:
    - name: "foo"
      value: "bar"
```

NOTE

Only the **name** and **value** fields are supported. Any settings on the **valueFrom** field are ignored.

2.5.1.5. Squashing layers with docker builds

Docker builds normally create a layer representing each instruction in a Dockerfile. Setting the **imageOptimizationPolicy** to **SkipLayers** merges all instructions into a single layer on top of the base image.

Procedure

- Set the **imageOptimizationPolicy** to **SkipLayers**:

    ```
    strategy:
      dockerStrategy:
        imageOptimizationPolicy: SkipLayers
    ```

–

2.5.1.6. Using build volumes

You can mount build volumes to give running builds access to information that you don't want to persist in the output container image.

Build volumes provide sensitive information, such as repository credentials, that the build environment or configuration only needs at build time. Build volumes are different from build inputs, whose data can persist in the output container image.

The mount points of build volumes, from which the running build reads data, are functionally similar to pod volume mounts.

Prerequisites

- You have added an input secret, config map, or both to a BuildConfig object .

Procedure

- In the **dockerStrategy** definition of the **BuildConfig** object, add any build volumes to the **volumes** array. For example:

```
spec:
  dockerStrategy:
    volumes:
      - name: secret-mvn 1
        mounts:
        - destinationPath: /opt/app-root/src/.ssh 2
        source:
          type: Secret 3
          secret:
            secretName: my-secret 4
      - name: settings-mvn 5
        mounts:
        - destinationPath: /opt/app-root/src/.m2 6
        source:
          type: ConfigMap 7
          configMap:
            name: my-config 8
      - name: my-csi-volume 9
        mounts:
        - destinationPath: /opt/app-root/src/some_path 10
        source:
          type: CSI 11
          csi:
            driver: csi.sharedresource.openshift.io 12
            readOnly: true 13
            volumeAttributes: 14
              attribute: value
```

1 **5** **9** Required. A unique name.

2 **6** **10**

Required. The absolute path of the mount point. It must not contain **..** or **:** and doesn't collide with the destination path generated by the builder. The **/opt/app-root/src** is the

3 **7** **11** Required. The type of source, **ConfigMap**, **Secret**, or **CSI**.

4 **8** Required. The name of the source.

12 Required. The driver that provides the ephemeral CSI volume.

13 Required. This value must be set to **true**. Provides a read-only volume.

14 Optional. The volume attributes of the ephemeral CSI volume. Consult the CSI driver's documentation for supported attribute keys and values.
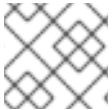
> **NOTE**
>
> The Shared Resource CSI Driver is supported as a Technology Preview feature.

2.5.2. Source-to-image build

Source-to-image (S2I) is a tool for building reproducible container images. It produces ready-to-run images by injecting application source into a container image and assembling a new image. The new image incorporates the base image, the builder, and built source and is ready to use with the **buildah run** command. S2I supports incremental builds, which re-use previously downloaded dependencies, previously built artifacts, and so on.

2.5.2.1. Performing source-to-image incremental builds

Source-to-image (S2I) can perform incremental builds, which means it reuses artifacts from previously-built images.

Procedure

- To create an incremental build, create a with the following modification to the strategy definition:

```
strategy:
  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.

Additional resources

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

2.5.2.2. Overriding source-to-image builder image scripts

You can override the **assemble**, **run**, and **save-artifacts** source-to-image (S2I) scripts provided by the builder image.

Procedure

To override the **assemble**, **run**, and **save-artifacts** S2I scripts provided by the builder image, either:

- Provide an **assemble**, **run**, or **save-artifacts** script in the **.s2i/bin** directory of your application source repository.

- Provide a URL of a directory containing the scripts as part of the strategy definition. For example:

```
strategy:
  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 scripts provided in the image.

> **NOTE**
>
> Files located at the **scripts** URL take precedence over files located in **.s2i/bin** of the source repository.

2.5.2.3. Source-to-image 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. Variables provided will be present during the build process and in the output image.

2.5.2.3.1. Using source-to-image 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 output image.

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

Procedure

For example, to disable assets compilation for your Rails application during the build:

- Add **DISABLE_ASSET_COMPILATION=true** in the **.s2i/environment** file.

In addition to builds, the specified environment variables are also available in the running application itself. For example, to cause the Rails application to start in **development** mode instead of **production**:

- Add **RAILS_ENV=development** to the **.s2i/environment** file.

The complete list of supported environment variables is available in the using images section for each image.

2.5.2.3.2. Using source-to-image build configuration environment

You can add environment variables to the **sourceStrategy** definition of the build configuration. 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.

Procedure

- For example, to disable assets compilation for your Rails application:

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

Additional resources

- The build environment section provides more advanced instructions.

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

2.5.2.4. Ignoring source-to-image source files

Source-to-image (S2I) supports a **.s2iignore** file, which contains a list of file patterns that should be ignored. Files in the build working directory, as provided by the various input sources, that match a pattern found in the **.s2iignore** file will not be made available to the **assemble** script.

2.5.2.5. Creating images from source code with source-to-image

Source-to-image (S2I) is a framework that makes it easy to write images that take application source code as an input and produce a new image that runs the assembled application as output.

The main advantage of using S2I for building reproducible container images is the ease of use for developers. As a builder image author, you must understand two basic concepts in order for your images to provide the best S2I performance, the build process and S2I scripts.

2.5.2.5.1. Understanding the source-to-image build process

The build process consists of the following three fundamental elements, which are combined into a final container image:

- Sources

- Source-to-image (S2I) scripts

- Builder image

S2I generates a Dockerfile with the builder image as the first **FROM** instruction. The Dockerfile generated by S2I is then passed to Buildah.

2.5.2.5.2. How to write source-to-image scripts

You can write source-to-image (S2I) scripts in any programming language, as long as the scripts are executable inside the builder image. S2I supports multiple options providing **assemble/run/save-artifacts** scripts. All of these locations are checked on each build in the following order:

1. A script specified in the build configuration.

2. A script found in the application source **.s2i/bin** directory.

3. A script found at the default image URL with the **io.openshift.s2i.scripts-url** label.

Both the **io.openshift.s2i.scripts-url** label specified in the image and the script specified in a build configuration can take one of the following forms:

- **image:///path_to_scripts_dir**: absolute path inside the image to a directory where the S2I scripts are located.

- **file:///path_to_scripts_dir**: relative or absolute path to a directory on the host where the S2I scripts are located.

- **http(s)://path_to_scripts_dir**: URL to a directory where the S2I scripts are located.

Table 2.1. S2I scripts

Script	Description
assemble	The **assemble** script builds the application artifacts from a source and places them into appropriate directories inside the image. This script is required. The workflow for this script is: 1. Optional: Restore build artifacts. If you want to support incremental builds, make sure to define **save-artifacts** as well. 2. Place the application source in the desired location. 3. Build the application artifacts. 4. Install the artifacts into locations appropriate for them to run.
run	The **run** script executes your application. This script is required.
save-artifacts	The **save-artifacts** script gathers all dependencies that can speed up the build processes that follow. This script is optional. For example: - For Ruby, **gems** installed by Bundler. - For Java, **.m2** contents. These dependencies are gathered into a **tar** file and streamed to the standard output.

Script	Description
usage	The **usage** script allows you to inform the user how to properly use your image. This script is optional.
test/run	The **test/run** script allows you to create a process to check if the image is working correctly. This script is optional. The proposed flow of that process is: 1. Build the image. 2. Run the image to verify the **usage** script. 3. Run **s2i build** to verify the **assemble** script. 4. Optional: Run **s2i build** again to verify the **save-artifacts** and **assemble** scripts save and restore artifacts functionality. 5. Run the image to verify the test application is working. **NOTE** The suggested location to put the test application built by your **test/run** script is the **test/test-app** directory in your image repository.

Example S2I scripts

The following example S2I scripts are written in Bash. Each example assumes its **tar** contents are unpacked into the **/tmp/s2i** directory.

assemble script:

```bash
#!/bin/bash

# restore build artifacts
if [ "$(ls /tmp/s2i/artifacts/ 2>/dev/null)" ]; then
    mv /tmp/s2i/artifacts/* $HOME/.
fi

# move the application source
mv /tmp/s2i/src $HOME/src

# build application artifacts
pushd ${HOME}
make all

# install the artifacts
make install
popd
```

run script:

```bash
#!/bin/bash
```

```
# run the application
/opt/application/run.sh
```

save-artifacts script:

```
#!/bin/bash

pushd ${HOME}
if [ -d deps ]; then
    # all deps contents to tar stream
    tar cf - deps
fi
popd
```

usage script:

```
#!/bin/bash

# inform the user how to use the image
cat <<EOF
This is a S2I sample builder image, to use it, install
https://github.com/openshift/source-to-image
EOF
```

Additional resources

- S2I Image Creation Tutorial

2.5.2.6. Using build volumes

You can mount build volumes to give running builds access to information that you don't want to persist in the output container image.

Build volumes provide sensitive information, such as repository credentials, that the build environment or configuration only needs at build time. Build volumes are different from build inputs, whose data can persist in the output container image.

The mount points of build volumes, from which the running build reads data, are functionally similar to pod volume mounts.

Prerequisites

- You have added an input secret, config map, or both to a BuildConfig object .

Procedure

- In the **sourceStrategy** definition of the **BuildConfig** object, add any build volumes to the **volumes** array. For example:

  ```
  spec:
    sourceStrategy:
      volumes:
        - name: secret-mvn ❶
  ```

```
      mounts:
      - destinationPath: /opt/app-root/src/.ssh ②
      source:
        type: Secret ③
        secret:
          secretName: my-secret ④
    - name: settings-mvn ⑤
      mounts:
      - destinationPath: /opt/app-root/src/.m2 ⑥
      source:
        type: ConfigMap ⑦
        configMap:
          name: my-config ⑧
    - name: my-csi-volume ⑨
      mounts:
      - destinationPath: /opt/app-root/src/some_path ⑩
      source:
        type: CSI ⑪
        csi:
          driver: csi.sharedresource.openshift.io ⑫
          readOnly: true ⑬
          volumeAttributes: ⑭
            attribute: value
```

① ⑤ ⑨ Required. A unique name.

② ⑥ ⑩ Required. The absolute path of the mount point. It must not contain **..** or **:** and doesn't collide with the destination path generated by the builder. The **/opt/app-root/src** is the default home directory for many Red Hat S2I-enabled images.

③ ⑦ ⑪ Required. The type of source, **ConfigMap**, **Secret**, or **CSI**.

④ ⑧ Required. The name of the source.

⑫ Required. The driver that provides the ephemeral CSI volume.

⑬ Required. This value must be set to **true**. Provides a read-only volume.

⑭ Optional. The volume attributes of the ephemeral CSI volume. Consult the CSI driver's documentation for supported attribute keys and values.

> **NOTE**
>
> The Shared Resource CSI Driver is supported as a Technology Preview feature.

2.5.3. Custom build

The custom build strategy allows developers to define a specific builder image responsible for the entire build process. Using your own builder image allows you to customize your build process.

A custom builder image is a plain container image embedded with build process logic, for example for building RPMs or base images.

Custom builds run with a high level of privilege and are not available to users by default. Only users who can be trusted with cluster administration permissions should be granted access to run custom builds.

2.5.3.1. Using FROM image for custom builds

You can use the **customStrategy.from** section to indicate the image to use for the custom build

Procedure

- Set the **customStrategy.from** section:

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

2.5.3.2. Using secrets in custom builds

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.

Procedure

- To mount each secret at a specific location, edit the **secretSource** and **mountPath** fields of the **strategy** YAML file:

```yaml
strategy:
  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.

2.5.3.3. Using environment variables for custom builds

To make environment variables available to the custom build process, you can add environment variables to the **customStrategy** definition of the build configuration.

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

Procedure

1. Define a custom HTTP proxy to be used during build:

   ```yaml
   customStrategy:
   ```

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

2. To manage environment variables defined in the build configuration, enter the following command:

```
$ oc set env <enter_variables>
```

2.5.3.4. Using custom builder images

OpenShift Container Platform's custom build strategy enables you to define a specific builder image responsible for the entire build process. When you need a build to produce individual artifacts such as packages, JARs, WARs, installable ZIPs, or base images, use a custom builder image using the custom build strategy.

A custom builder image is a plain container image embedded with build process logic, which is used for building artifacts such as RPMs or base container images.

Additionally, the custom builder allows implementing any extended build process, such as a CI/CD flow that runs unit or integration tests.

2.5.3.4.1. Custom builder image

Upon invocation, a custom builder image receives the following environment variables with the information needed to proceed with the build:

Table 2.2. Custom Builder Environment Variables

Variable Name	Description
BUILD	The entire serialized JSON of the **Build** object definition. If you must use a specific API version for serialization, you can set the **buildAPIVersion** parameter in the custom strategy specification of the build configuration.
SOURCE_REPOSITORY	The URL of a Git repository with source to be built.
SOURCE_URI	Uses the same value as **SOURCE_REPOSITORY**. Either can be used.
SOURCE_CONTEXT_DIR	Specifies the subdirectory of the Git repository to be used when building. Only present if defined.
SOURCE_REF	The Git reference to be built.
ORIGIN_VERSION	The version of the OpenShift Container Platform master that created this build object.
OUTPUT_REGISTRY	The container image registry to push the image to.

Variable Name	Description
OUTPUT_IMAGE	The container image tag name for the image being built.
PUSH_DOCKERCFG _PATH	The path to the container registry credentials for running a **podman push** operation.

2.5.3.4.2. Custom builder workflow

Although custom builder image authors have flexibility in defining the build process, your builder image must adhere to the following required steps necessary for running a build inside of OpenShift Container Platform:

1. The **Build** object definition contains all the necessary information about input parameters for the build.

2. Run the build process.

3. If your build produces an image, push it to the output location of the build if it is defined. Other output locations can be passed with environment variables.

2.5.4. Pipeline build

> **IMPORTANT**
>
> The Pipeline build strategy is deprecated in OpenShift Container Platform 4. Equivalent and improved functionality is present in the OpenShift Container Platform Pipelines based on Tekton.
>
> Jenkins images on OpenShift Container Platform are fully supported and users should follow Jenkins user documentation for defining their **jenkinsfile** in a job or store it in a Source Control Management system.

The Pipeline build strategy allows developers to define a Jenkins pipeline for use by the Jenkins pipeline plugin. The build can be started, monitored, and managed by OpenShift Container Platform in the same way as any other build type.

Pipeline workflows are defined in a **jenkinsfile**, either embedded directly in the build configuration, or supplied in a Git repository and referenced by the build configuration.

2.5.4.1. Understanding OpenShift Container Platform pipelines

> **IMPORTANT**
>
> The Pipeline build strategy is deprecated in OpenShift Container Platform 4. Equivalent and improved functionality is present in the OpenShift Container Platform Pipelines based on Tekton.
>
> Jenkins images on OpenShift Container Platform are fully supported and users should follow Jenkins user documentation for defining their **jenkinsfile** in a job or store it in a Source Control Management system.

Pipelines give you control over building, deploying, and promoting your applications on OpenShift Container Platform. Using a combination of the Jenkins Pipeline build strategy, **jenkinsfiles**, and the OpenShift Container Platform Domain Specific Language (DSL) provided by the Jenkins Client Plugin, you can create advanced build, test, deploy, and promote pipelines for any scenario.

OpenShift Container Platform Jenkins Sync Plugin

The OpenShift Container Platform Jenkins Sync Plugin keeps the build configuration and build objects in sync with Jenkins jobs and builds, and provides the following:

- Dynamic job and run creation in Jenkins.

- Dynamic creation of agent pod templates from image streams, image stream tags, or config maps.

- Injection of environment variables.

- Pipeline visualization in the OpenShift Container Platform web console.

- Integration with the Jenkins Git plugin, which passes commit information from OpenShift Container Platform builds to the Jenkins Git plugin.

- Synchronization of secrets into Jenkins credential entries.

OpenShift Container Platform Jenkins Client Plugin

The OpenShift Container Platform Jenkins Client Plugin is a Jenkins plugin which aims to provide a readable, concise, comprehensive, and fluent Jenkins Pipeline syntax for rich interactions with an OpenShift Container Platform API Server. The plugin uses the OpenShift Container Platform command line tool, **oc**, which must be available on the nodes executing the script.

The Jenkins Client Plugin must be installed on your Jenkins master so the OpenShift Container Platform DSL will be available to use within the **jenkinsfile** for your application. This plugin is installed and enabled by default when using the OpenShift Container Platform Jenkins image.

For OpenShift Container Platform Pipelines within your project, you will must use the Jenkins Pipeline Build Strategy. This strategy defaults to using a **jenkinsfile** at the root of your source repository, but also provides the following configuration options:

- An inline **jenkinsfile** field within your build configuration.

- A **jenkinsfilePath** field within your build configuration that references the location of the **jenkinsfile** to use relative to the source **contextDir**.

> **NOTE**
>
> 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**.

2.5.4.2. Providing the Jenkins file for pipeline builds

> **IMPORTANT**
>
> The Pipeline build strategy is deprecated in OpenShift Container Platform 4. Equivalent and improved functionality is present in the OpenShift Container Platform Pipelines based on Tekton.
>
> Jenkins images on OpenShift Container Platform are fully supported and users should follow Jenkins user documentation for defining their **jenkinsfile** in a job or store it in a Source Control Management system.

The **jenkinsfile** uses the standard groovy language syntax to allow fine grained control over the configuration, build, and deployment of your application.

You can supply the **jenkinsfile** in one of the following ways:

- A file located within your source code repository.

- Embedded as part of your build configuration using the **jenkinsfile** field.

When using the first option, the **jenkinsfile** must be included in your applications source code repository at one of the following locations:

- A file named **jenkinsfile** at the root of your repository.

- A file named **jenkinsfile** at the root of the source **contextDir** of your repository.

- A file name specified via the **jenkinsfilePath** field of the **JenkinsPipelineStrategy** section of your BuildConfig, which is relative to the source **contextDir** if supplied, otherwise it defaults to the root of the repository.

The **jenkinsfile** is run on the Jenkins agent pod, which must have the OpenShift Container Platform client binaries available if you intend to use the OpenShift Container Platform DSL.

Procedure

To provide the Jenkins file, you can either:

- Embed the Jenkins file in the build configuration.

- Include in the build configuration a reference to the Git repository that contains the Jenkins file.

Embedded Definition

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "sample-pipeline"
spec:
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        node('agent') {
          stage 'build'
          openshiftBuild(buildConfig: 'ruby-sample-build', showBuildLogs: 'true')
          stage 'deploy'
          openshiftDeploy(deploymentConfig: 'frontend')
        }
```

Reference to Git Repository

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

2.5.4.3. Using environment variables for pipeline builds

> **IMPORTANT**
>
> The Pipeline build strategy is deprecated in OpenShift Container Platform 4. Equivalent and improved functionality is present in the OpenShift Container Platform Pipelines based on Tekton.
>
> Jenkins images on OpenShift Container Platform are fully supported and users should follow Jenkins user documentation for defining their **jenkinsfile** in a job or store it in a Source Control Management system.

To make environment variables available to the Pipeline build process, you can add environment variables to the **jenkinsPipelineStrategy** definition of the build configuration.

Once defined, the environment variables will be set as parameters for any Jenkins job associated with the build configuration.

Procedure

- To define environment variables to be used during build, edit the YAML file:

```
jenkinsPipelineStrategy:
...
  env:
    - name: "FOO"
      value: "BAR"
```

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

2.5.4.3.1. Mapping between BuildConfig environment variables and Jenkins job parameters

When a Jenkins job is created or updated based on changes to a Pipeline strategy build configuration, any environment variables in the build configuration are mapped to Jenkins job parameters definitions, where the default values for the Jenkins job parameters definitions are the current values of the associated environment variables.

After the Jenkins job's initial creation, you can still add additional parameters to the job from the Jenkins console. The parameter names differ from the names of the environment variables in the build configuration. The parameters are honored when builds are started for those Jenkins jobs.

How you start builds for the Jenkins job dictates how the parameters are set.

- If you start with **oc start-build**, the values of the environment variables in the build configuration are the parameters set for the corresponding job instance. Any changes you make to the parameters' default values from the Jenkins console are ignored. The build configuration values take precedence.

- If you start with **oc start-build -e**, the values for the environment variables specified in the **-e** option take precedence.
  - If you specify an environment variable not listed in the build configuration, they will be added as a Jenkins job parameter definitions.

  - Any changes you make from the Jenkins console to the parameters corresponding to the environment variables are ignored. The build configuration and what you specify with **oc start-build -e** takes precedence.

- If you start the Jenkins job with the Jenkins console, then you can control the setting of the parameters with the Jenkins console as part of starting a build for the job.

> **NOTE**
>
> It is recommended that you specify in the build configuration all possible environment variables to be associated with job parameters. Doing so reduces disk I/O and improves performance during Jenkins processing.

2.5.4.4. Pipeline build tutorial
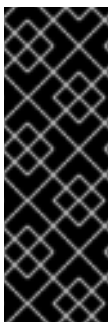
> **IMPORTANT**
>
> The Pipeline build strategy is deprecated in OpenShift Container Platform 4. Equivalent and improved functionality is present in the OpenShift Container Platform Pipelines based on Tekton.
>
> Jenkins images on OpenShift Container Platform are fully supported and users should follow Jenkins user documentation for defining their **jenkinsfile** in a job or store it in a Source Control Management system.

This example demonstrates how to create an OpenShift Container Platform Pipeline that will build, deploy, and verify a **Node.js/MongoDB** application using the **nodejs-mongodb.json** template.

Procedure

1. Create the Jenkins master:

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

Select the project that you want to use or create a new project with **oc new-project <project_name>**.

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

If you want to use persistent storage, use **jenkins-persistent** instead.

2. Create a file named **nodejs-sample-pipeline.yaml** with the following content:
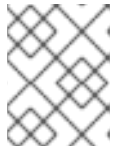
> **NOTE**
>
> This creates a **BuildConfig** object that employs the Jenkins pipeline strategy to build, deploy, and scale the **Node.js/MongoDB** example application.

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "nodejs-sample-pipeline"
spec:
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: <pipeline content from below>
    type: JenkinsPipeline
```

3. After you create a **BuildConfig** object with a **jenkinsPipelineStrategy**, tell the pipeline what to do by using an inline **jenkinsfile**:

> **NOTE**
>
> This example does not set up a Git repository for the application.
>
> The following **jenkinsfile** content is written in Groovy using the OpenShift Container Platform DSL. For this example, include inline content in the **BuildConfig** object using the YAML Literal Style, though including a **jenkinsfile** in your source repository is the preferred method.

```
def templatePath = 'https://raw.githubusercontent.com/openshift/nodejs-
ex/master/openshift/templates/nodejs-mongodb.json' ❶
def templateName = 'nodejs-mongodb-example' ❷
pipeline {
  agent {
    node {
      label 'nodejs' ❸
    }
  }
  options {
    timeout(time: 20, unit: 'MINUTES') ❹
  }
  stages {
    stage('preamble') {
      steps {
        script {
          openshift.withCluster() {
```

```
                openshift.withProject() {
                    echo "Using project: ${openshift.project()}"
                }
            }
        }
    }
}
stage('cleanup') {
  steps {
    script {
        openshift.withCluster() {
            openshift.withProject() {
              openshift.selector("all", [ template : templateName ]).delete()    5
              if (openshift.selector("secrets", templateName).exists()) {    6
                openshift.selector("secrets", templateName).delete()
              }
            }
        }
    }
  }
}
stage('create') {
  steps {
    script {
        openshift.withCluster() {
            openshift.withProject() {
              openshift.newApp(templatePath)    7
            }
        }
    }
  }
}
stage('build') {
  steps {
    script {
        openshift.withCluster() {
            openshift.withProject() {
              def builds = openshift.selector("bc", templateName).related('builds')
              timeout(5) {    8
                builds.untilEach(1) {
                  return (it.object().status.phase == "Complete")
                }
              }
            }
        }
    }
  }
}
stage('deploy') {
  steps {
    script {
        openshift.withCluster() {
            openshift.withProject() {
              def rm = openshift.selector("dc", templateName).rollout()
              timeout(5) {    9
```

```
                    openshift.selector("dc", templateName).related('pods').untilEach(1) {
                      return (it.object().status.phase == "Running")
                    }
                  }
                }
              }
            }
          }
        }
      }
    stage('tag') {
      steps {
        script {
          openshift.withCluster() {
            openshift.withProject() {
              openshift.tag("${templateName}:latest", "${templateName}-staging:latest") 10
            }
          }
        }
      }
    }
  }
}
```

1 Path of the template to use.

1 **2** Name of the template that will be created.

3 Spin up a **node.js** agent pod on which to run this build.

4 Set a timeout of 20 minutes for this pipeline.

5 Delete everything with this template label.

6 Delete any secrets with this template label.

7 Create a new application from the **templatePath**.

8 Wait up to five minutes for the build to complete.

9 Wait up to five minutes for the deployment to complete.

10 If everything else succeeded, tag the **$ {templateName}:latest** image as **$ {templateName}-staging:latest**. A pipeline build configuration for the staging environment can watch for the **$ {templateName}-staging:latest** image to change and then deploy it to the staging environment.

> **NOTE**
>
> The previous example was written using the declarative pipeline style, but the older scripted pipeline style is also supported.

4. Create the Pipeline **BuildConfig** in your OpenShift Container Platform cluster:

```
$ oc create -f nodejs-sample-pipeline.yaml
```

a. If you do not want to create your own file, you can use the sample from the Origin repository by running:

```
$ oc create -f
https://raw.githubusercontent.com/openshift/origin/master/examples/jenkins/pipeline/nodejs-
sample-pipeline.yaml
```

5. Start the Pipeline:

```
$ oc start-build nodejs-sample-pipeline
```

> **NOTE**
>
> Alternatively, you can start your pipeline with the OpenShift Container Platform web console by navigating to the Builds → Pipeline section and clicking **Start Pipeline**, or by visiting the Jenkins Console, navigating to the Pipeline that you created, and clicking **Build Now**.

Once the pipeline is started, you should see the following actions performed within your project:

- A job instance is created on the Jenkins server.

- An agent pod is launched, if your pipeline requires one.

- The pipeline runs on the agent pod, or the master if no agent is required.

  - Any previously created resources with the **template=nodejs-mongodb-example** label will be deleted.

  - A new application, and all of its associated resources, will be created from the **nodejs-mongodb-example** template.

  - A build will be started using the **nodejs-mongodb-example BuildConfig**.

    - The pipeline will wait until the build has completed to trigger the next stage.

  - A deployment will be started using the **nodejs-mongodb-example** deployment configuration.

    - The pipeline will wait until the deployment has completed to trigger the next stage.

  - If the build and deploy are successful, the **nodejs-mongodb-example:latest** image will be tagged as **nodejs-mongodb-example:stage**.

- The agent pod is deleted, if one was required for the pipeline.

> **NOTE**
>
> The best way to visualize the pipeline execution is by viewing it in the OpenShift Container Platform web console. You can view your pipelines by logging in to the web console and navigating to Builds → Pipelines.

2.5.5. Adding secrets with web console

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

Procedure

To add a secret to your build configuration so that it can access a private repository from the OpenShift Container Platform web console:

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 build configuration.

4. On the build configuration editor page or in the **create app from builder image** page of the web console, set the **Source Secret**.

5. Click **Save**.

2.5.6. Enabling pulling and pushing

You can enable pulling to a private registry by setting the pull secret and pushing by setting the push secret in the build configuration.

Procedure

To enable pulling to a private registry:

- Set the pull secret in the build configuration.

To enable pushing:

- Set the push secret in the build configuration.

2.6. CUSTOM IMAGE BUILDS WITH BUILDAH

With OpenShift Container Platform 4.11, a docker socket will not be present on the host nodes. This means the *mount docker socket* option of a custom build is not guaranteed to provide an accessible docker socket for use within a custom build image.

If you require this capability in order to build and push images, add the Buildah tool your custom build image and use it to build and push the image within your custom build logic. The following is an example of how to run custom builds with Buildah.
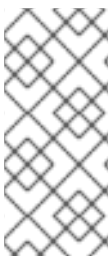
> **NOTE**
>
> Using the custom build strategy requires permissions that normal users do not have by default because it allows the user to execute arbitrary code inside a privileged container running on the cluster. This level of access can be used to compromise the cluster and therefore should be granted only to users who are trusted with administrative privileges on the cluster.

2.6.1. Prerequisites

- Review how to grant custom build permissions.

2.6.2. Creating custom build artifacts

You must create the image you want to use as your custom build image.

Procedure

1. Starting with an empty directory, create a file named **Dockerfile** with the following content:

   ```
   FROM registry.redhat.io/rhel8/buildah
   # In this example, `/tmp/build` contains the inputs that build when this
   # custom builder image is run. Normally the custom builder image fetches
   # this content from some location at build time, by using git clone as an example.
   ADD dockerfile.sample /tmp/input/Dockerfile
   ADD build.sh /usr/bin
   RUN chmod a+x /usr/bin/build.sh
   # /usr/bin/build.sh contains the actual custom build logic that will be run when
   # this custom builder image is run.
   ENTRYPOINT ["/usr/bin/build.sh"]
   ```

2. In the same directory, create a file named **dockerfile.sample**. This file is included in the custom build image and defines the image that is produced by the custom build:

   ```
   FROM registry.access.redhat.com/ubi8/ubi
   RUN touch /tmp/build
   ```

3. In the same directory, create a file named **build.sh**. This file contains the logic that is run when the custom build runs:

   ```
   #!/bin/sh
   # Note that in this case the build inputs are part of the custom builder image, but normally this
   # is retrieved from an external source.
   cd /tmp/input
   # OUTPUT_REGISTRY and OUTPUT_IMAGE are env variables provided by the custom
   # build framework
   TAG="${OUTPUT_REGISTRY}/${OUTPUT_IMAGE}"


   # performs the build of the new image defined by dockerfile.sample
   buildah --storage-driver vfs bud --isolation chroot -t ${TAG} .


   # buildah requires a slight modification to the push secret provided by the service
   # account to use it for pushing the image
   cp /var/run/secrets/openshift.io/push/.dockercfg /tmp
   (echo "{ \"auths\": " ; cat /var/run/secrets/openshift.io/push/.dockercfg ; echo "}") >
   /tmp/.dockercfg


   # push the new image to the target for the build
   buildah --storage-driver vfs push --tls-verify=false --authfile /tmp/.dockercfg ${TAG}
   ```

2.6.3. Build custom builder image

You can use OpenShift Container Platform to build and push custom builder images to use in a custom strategy.

Prerequisites

- Define all the inputs that will go into creating your new custom builder image.

Procedure

1. Define a **BuildConfig** object that will build your custom builder image:

   ```
   $ oc new-build --binary --strategy=docker --name custom-builder-image
   ```

2. From the directory in which you created your custom build image, run the build:

   ```
   $ oc start-build custom-builder-image --from-dir . -F
   ```

   After the build completes, your new custom builder image is available in your project in an image stream tag that is named **custom-builder-image:latest**.

2.6.4. Use custom builder image

You can define a **BuildConfig** object that uses the custom strategy in conjunction with your custom builder image to execute your custom build logic.

Prerequisites

- Define all the required inputs for new custom builder image.

- Build your custom builder image.

Procedure

1. Create a file named **buildconfig.yaml**. This file defines the **BuildConfig** object that is created in your project and executed:

   ```yaml
   kind: BuildConfig
   apiVersion: build.openshift.io/v1
   metadata:
     name: sample-custom-build
     labels:
       name: sample-custom-build
     annotations:
       template.alpha.openshift.io/wait-for-ready: 'true'
   spec:
     strategy:
       type: Custom
       customStrategy:
         forcePull: true
         from:
           kind: ImageStreamTag
           name: custom-builder-image:latest
           namespace: <yourproject>  1
     output:
       to:
         kind: ImageStreamTag
         name: sample-custom:latest
   ```

   1 Specify your project name.

2. Create the **BuildConfig**:

```
$ oc create -f buildconfig.yaml
```

3. Create a file named **imagestream.yaml**. This file defines the image stream to which the build will push the image:

```
kind: ImageStream
apiVersion: image.openshift.io/v1
metadata:
  name: sample-custom
spec: {}
```

4. Create the imagestream:

```
$ oc create -f imagestream.yaml
```

5. Run your custom build:

```
$ oc start-build sample-custom-build -F
```

When the build runs, it launches a pod running the custom builder image that was built earlier. The pod runs the **build.sh** logic that is defined as the entrypoint for the custom builder image. The **build.sh** logic invokes Buildah to build the **dockerfile.sample** that was embedded in the custom builder image, and then uses Buildah to push the new image to the **sample-custom image stream**.

2.7. PERFORMING AND CONFIGURING BASIC BUILDS

The following sections provide instructions for basic build operations, including starting and canceling builds, editing **BuildConfigs**, deleting **BuildConfigs**, viewing build details, and accessing build logs.

2.7.1. Starting a build

You can manually start a new build from an existing build configuration in your current project.

Procedure

To manually start a build, enter the following command:

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

2.7.1.1. Re-running a build

You can manually re-run a build using the **--from-build** flag.

Procedure

- To manually re-run a build, enter the following command:

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

2.7.1.2. Streaming build logs

You can specify the **--follow** flag to stream the build's logs in **stdout**.

Procedure

- To manually stream a build's logs in **stdout**, enter the following command:

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

2.7.1.3. Setting environment variables when starting a build

You can specify the **--env** flag to set any desired environment variable for the build.

Procedure

- To specify a desired environment variable, enter the following command:

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

2.7.1.4. Starting a build with source

Rather than relying on a Git source pull or a Dockerfile for a build, you can also start a build by directly pushing your source, which could be the contents of a Git or SVN working directory, a set of pre-built 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.

Procedure

- Start a build from a source using the following command to send the contents of a local Git repository as an archive from the tag **v2**:

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

2.7.2. Canceling a build

You can cancel a build using the web console, or with the following CLI command.

Procedure

- To manually cancel a build, enter the following command:

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

2.7.2.1. Canceling multiple builds

You can cancel multiple builds with the following CLI command.

Procedure

- To manually cancel multiple builds, enter the following command:

  ```
  $ oc cancel-build <build1_name> <build2_name> <build3_name>
  ```

2.7.2.2. Canceling all builds

You can cancel all builds from the build configuration with the following CLI command.

Procedure

- To cancel all builds, enter the following command:

  ```
  $ oc cancel-build bc/<buildconfig_name>
  ```

2.7.2.3. Canceling all builds in a given state

You can cancel all builds in a given state, such as **new** or **pending**, while ignoring the builds in other states.

Procedure

- To cancel all in a given state, enter the following command:

  ```
  $ oc cancel-build bc/<buildconfig_name>
  ```

2.7.3. Editing a BuildConfig

To edit your build configurations, you use the **Edit BuildConfig** option in the **Builds** view of the **Developer** perspective.

You can use either of the following views to edit a **BuildConfig**:

- The **Form view** enables you to edit your **BuildConfig** using the standard form fields and checkboxes.

- The **YAML view** enables you to edit your **BuildConfig** with full control over the operations.

You can switch between the **Form view** and **YAML view** without losing any data. The data in the **Form view** is transferred to the **YAML view** and vice versa.

Procedure

1. In the **Builds** view of the **Developer** perspective, click the menu ⋮ to see the **Edit BuildConfig** option.

2. Click **Edit BuildConfig** to see the **Form view** option.

3. In the **Git** section, enter the Git repository URL for the codebase you want to use to create an application. The URL is then validated.

   - Optional: Click **Show Advanced Git Options** to add details such as:

     - **Git Reference** to specify a branch, tag, or commit that contains code you want to use to build the application.

     - **Context Dir** to specify the subdirectory that contains code you want to use to build the application.

     - **Source Secret** to create a **Secret Name** with credentials for pulling your source code from a private repository.

4. In the **Build from** section, select the option that you would like to build from. You can use the following options:

   - **Image Stream tag** references an image for a given image stream and tag. Enter the project, image stream, and tag of the location you would like to build from and push to.

   - **Image Stream image** references an image for a given image stream and image name. Enter the image stream image you would like to build from. Also enter the project, image stream, and tag to push to.

   - **Docker image**: The Docker image is referenced through a Docker image repository. You will also need to enter the project, image stream, and tag to refer to where you would like to push to.

5. Optional: In the **Environment Variables** section, add the environment variables associated with the project by using the **Name** and **Value** fields. To add more environment variables, use **Add Value**, or **Add from ConfigMap** and **Secret** .

6. Optional: To further customize your application, use the following advanced options:

   Trigger

   Triggers a new image build when the builder image changes. Add more triggers by clicking **Add Trigger** and selecting the **Type** and **Secret**.

   Secrets

Adds secrets for your application. Add more secrets by clicking **Add secret** and selecting the **Secret** and **Mount point**.

Policy

Click **Run policy** to select the build run policy. The selected policy determines the order in which builds created from the build configuration must run.

Hooks

Select **Run build hooks after image is built** to run commands at the end of the build and verify the image. Add **Hook type**, **Command**, and **Arguments** to append to the command.

7. Click **Save** to save the **BuildConfig**.

2.7.4. Deleting a BuildConfig

You can delete a **BuildConfig** using the following command.

Procedure

- To delete a **BuildConfig**, enter the following command:

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

  This also deletes all builds that were instantiated from this **BuildConfig**.

- To delete a **BuildConfig** and keep the builds instatiated from the **BuildConfig**, specify the **--cascade=false** flag when you enter the following command:

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

2.7.5. Viewing build details

You can view build details with the web console or by using the **oc describe** CLI command.

This displays information including:

- The build source.

- The build strategy.

- The output destination.

- Digest of the image in the destination registry.

- 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.

Procedure

- To view build details, enter the following command:

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

2.7.6. Accessing build logs

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

Procedure

- To stream the logs using the build directly, enter the following command:

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

2.7.6.1. Accessing BuildConfig logs

You can access **BuildConfig** logs using the web console or the CLI.

Procedure

- To stream the logs of the latest build for a **BuildConfig**, enter the following command:

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

2.7.6.2. Accessing BuildConfig logs for a given version build

You can access logs for a given version build for a **BuildConfig** using the web console or the CLI.

Procedure

- To stream the logs for a given version build for a **BuildConfig**, enter the following command:

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

2.7.6.3. Enabling log verbosity

You can enable a more verbose output by passing the **BUILD_LOGLEVEL** environment variable as part of the **sourceStrategy** or **dockerStrategy** in a **BuildConfig**.

> **NOTE**
>
> An administrator can set the default build verbosity for the entire OpenShift Container Platform instance by configuring **env/BUILD_LOGLEVEL**. 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.

Procedure

- 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.

2.8. TRIGGERING AND MODIFYING BUILDS

The following sections outline how to trigger builds and modify builds using build hooks.

2.8.1. 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

2.8.1.1. 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, GitLab, Bitbucket, or Generic webhooks.

Currently, OpenShift Container Platform webhooks only support the analogous versions of the push event for each of the Git-based Source Code Management (SCM) systems. All other event types are ignored.

When the push events are processed, the OpenShift Container Platform control plane host confirms if the branch reference inside the event matches the branch reference in the corresponding **BuildConfig**. If so, it then checks out the exact commit reference noted in the webhook event on the OpenShift

Container Platform build. If they do not match, no build is triggered.

> **NOTE**
>
> **oc new-app** and **oc new-build** create GitHub and Generic webhook triggers automatically, but any other needed webhook triggers must be added manually. You can manually add triggers by setting triggers.

For all webhooks, you must define a secret with a key named **WebHookSecretKey** and the value being the value to be supplied when invoking the webhook. The webhook definition must then reference the secret. The secret ensures the uniqueness of the URL, preventing others from triggering the build. The value of the key is compared to the secret provided during the webhook invocation.

For example here is a GitHub webhook with a reference to a secret named **mysecret**:

```
type: "GitHub"
github:
  secretReference:
    name: "mysecret"
```

The secret is then defined as follows. Note that the value of the secret is base64 encoded as is required for any **data** field of a **Secret** object.

```
- kind: Secret
  apiVersion: v1
  metadata:
    name: mysecret
    creationTimestamp:
  data:
    WebHookSecretKey: c2VjcmV0dmFsdWUx
```

2.8.1.1.1. Using 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 is part of the URL you supply to GitHub when configuring the webhook.

Example GitHub webhook definition:

```
type: "GitHub"
github:
  secretReference:
    name: "mysecret"
```

> **NOTE**
>
> The secret used in the 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 **oc describe** command (see Displaying Webhook URLs), and is structured as follows:

Example output

> https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/github

Prerequisites

- Create a **BuildConfig** from a GitHub repository.

Procedure

1. To configure a GitHub Webhook:

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

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

      This generates a webhook GitHub URL that looks like:

      #### Example output

      > <https://api.starter-us-east-
      > 1.openshift.com:443/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/github

   b. Cut and paste this URL into GitHub, from the GitHub web console.

   c. In your GitHub repository, select **Add Webhook** from **Settings → Webhooks**.

   d. Paste the URL output into the **Payload URL** field.

   e. Change the **Content Type** from GitHub's default **application/x-www-form-urlencoded** to **application/json**.

   f. Click **Add webhook**.
      You should see a message from GitHub stating that your webhook was successfully configured.

      Now, when you push a change to your GitHub repository, a new build automatically starts, and upon a successful build a new deployment starts.

      > **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 by your Gogs server as well.

2. Given a file containing a valid JSON payload, such as **payload.json**, you can manually trigger the webhook with **curl**:

   ```
   $ curl -H "X-GitHub-Event: push" -H "Content-Type: application/json" -k -X POST --data-binary @payload.json
   https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/github
   ```

The **-k** argument is only necessary if your API server does not have a properly signed certificate.

> **NOTE**
>
> The build will only be triggered if the **ref** value from GitHub webhook event matches the **ref** value specified in the **source.git** field of the **BuildConfig** resource.

Additional resources

- Gogs

2.8.1.1.2. Using GitLab webhooks

GitLab webhooks handle the call made by GitLab when a repository is updated. As with the GitHub triggers, you must specify a secret. The following example is a trigger definition YAML within the **BuildConfig**:

```
type: "GitLab"
gitlab:
  secretReference:
    name: "mysecret"
```

The payload URL is returned as the GitLab Webhook URL by the **oc describe** command, and is structured as follows:

Example output

```
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/gitlab
```

Procedure

1. To configure a GitLab Webhook:

   a. Describe the **BuildConfig** to get the webhook URL:

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

   b. Copy the webhook URL, replacing **<secret>** with your secret value.

   c. Follow the GitLab setup instructions to paste the webhook URL into your GitLab repository settings.

2. Given a file containing a valid JSON payload, such as **payload.json**, you can manually trigger the webhook with **curl**:

   ```
   $ curl -H "X-GitLab-Event: Push Hook" -H "Content-Type: application/json" -k -X POST --data-binary @payload.json
   https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/gitlab
   ```

   The **-k** argument is only necessary if your API server does not have a properly signed certificate.

2.8.1.1.3. Using Bitbucket webhooks

Bitbucket webhooks handle the call made by Bitbucket when a repository is updated. Similar to the previous triggers, you must specify a secret. The following example is a trigger definition YAML within the **BuildConfig**:

```
type: "Bitbucket"
bitbucket:
  secretReference:
    name: "mysecret"
```

The payload URL is returned as the Bitbucket Webhook URL by the **oc describe** command, and is structured as follows:

Example output

```
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/bitbucket
```

Procedure

1. To configure a Bitbucket Webhook:

   a. Describe the 'BuildConfig' to get the webhook URL:

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

   b. Copy the webhook URL, replacing **<secret>** with your secret value.

   c. Follow the Bitbucket setup instructions to paste the webhook URL into your Bitbucket repository settings.

2. Given a file containing a valid JSON payload, such as **payload.json**, you can manually trigger the webhook with **curl**:

   ```
   $ curl -H "X-Event-Key: repo:push" -H "Content-Type: application/json" -k -X POST --data-binary @payload.json
   https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/bitbucket
   ```

   The **-k** argument is only necessary if your API server does not have a properly signed certificate.

2.8.1.1.4. Using generic webhooks

Generic webhooks are invoked from any system capable of making a web request. As with the other webhooks, you must specify a secret, which is 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:
  secretReference:
    name: "mysecret"
  allowEnv: true ❶
```

1 Set to **true** to allow a generic webhook to pass in environment variables.

Procedure

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

   Example output

   ```
   https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/generic
   ```

   The caller must invoke the webhook as a **POST** operation.

2. To invoke the webhook manually you can use **curl**:

   ```
   $ curl -X POST -k
   https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<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:

   ```
   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 by webhook are ignored. Set the **allowEnv** field to **true** on the webhook definition to enable this behavior.

3. 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>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<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 by 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.

2.8.1.1.5. Displaying webhook URLs

You can use the following command to display webhook URLs associated with a build configuration. If the command does not display any webhook URLs, then no webhook trigger is defined for that build configuration.

Procedure

- To display any webhook URLs associated with a **BuildConfig**, run:

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

2.8.1.2. Using image change triggers

As a developer, you can configure your build to run automatically every time a base image changes.

You can use image change triggers to automatically invoke your build when a new version of an upstream image is available. For example, if a build is based on a RHEL image, 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.

> **NOTE**
>
> Image streams that point to container images in v1 container 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 container registries.

Procedure

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

   ```
   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 **ImageStream**:

```
    strategy:
      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 **ImageStreams**:

```
    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 is 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 is started, but the build strategy is not updated with a unique image reference.

Since this example has an image change trigger for the strategy, the resulting build is:

```
    strategy:
      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.

You can pause an image change trigger to allow multiple changes on the referenced image stream before a build is started. You can also set the **paused** attribute to true when initially adding an **ImageChangeTrigger** to a **BuildConfig** to prevent a build from being immediately triggered.

```
    type: "ImageChange"
    imageChange:
      from:
        kind: "ImageStreamTag"
        name: "custom-image:latest"
      paused: true
```

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.

Additional resources

- v1 container registries

2.8.1.3. Identifying the image change trigger of a build

As a developer, if you have image change triggers, you can identify which image change initiated the last build. This can be useful for debugging or troubleshooting builds.

Example **BuildConfig**

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: bc-ict-example
  namespace: bc-ict-example-namespace
spec:

# ...

  triggers:
  - imageChange:
      from:
        kind: ImageStreamTag
        name: input:latest
        namespace: bc-ict-example-namespace
  - imageChange:
      from:
        kind: ImageStreamTag
        name: input2:latest
        namespace: bc-ict-example-namespace
    type: ImageChange
status:
  imageChangeTriggers:
  - from:
      name: input:latest
      namespace: bc-ict-example-namespace
    lastTriggerTime: "2021-06-30T13:47:53Z"
    lastTriggeredImageID: image-registry.openshift-image-registry.svc:5000/bc-ict-example-
namespace/input@sha256:0f88ffbeb9d25525720bfa3524cb1bf0908b7f791057cf1acfae917b11266a69

  - from:
      name: input2:latest
      namespace: bc-ict-example-namespace
    lastTriggeredImageID:  image-registry.openshift-image-registry.svc:5000/bc-ict-example-
namespace/input2@sha256:0f88ffbeb9d25525720bfa3524cb2ce0908b7f791057cf1acfae917b11266a6
```

```
9

lastVersion: 1
```
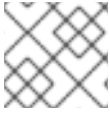
> **NOTE**
>
> This example omits elements that are not related to image change triggers.

Prerequisites

- You have configured multiple image change triggers. These triggers have triggered one or more builds.

Procedure

1. In **buildConfig.status.imageChangeTriggers** to identify the **lastTriggerTime** that has the latest timestamp.
   This **ImageChangeTriggerStatus**

   > Then you use the `name` and `namespace` from that build to find the corresponding image change trigger in `buildConfig.spec.triggers`.

2. Under **imageChangeTriggers**, compare timestamps to identify the latest

Image change triggers

In your build configuration, **buildConfig.spec.triggers** is an array of build trigger policies, **BuildTriggerPolicy**.

Each **BuildTriggerPolicy** has a **type** field and set of pointers fields. Each pointer field corresponds to one of the allowed values for the **type** field. As such, you can only set **BuildTriggerPolicy** to only one pointer field.

For image change triggers, the value of **type** is **ImageChange**. Then, the **imageChange** field is the pointer to an **ImageChangeTrigger** object, which has the following fields:

- **lastTriggeredImageID**: This field, which is not shown in the example, is deprecated in OpenShift Container Platform 4.8 and will be ignored in a future release. It contains the resolved image reference for the **ImageStreamTag** when the last build was triggered from this **BuildConfig**.

- **paused**: You can use this field, which is not shown in the example, to temporarily disable this particular image change trigger.

- **from**: You use this field to reference the **ImageStreamTag** that drives this image change trigger. Its type is the core Kubernetes type, **OwnerReference**.

The **from** field has the following fields of note: **kind: For image change triggers, the only supported value is ImageStreamTag. namespace**: You use this field to specify the namespace of the **ImageStreamTag**. ** **name**: You use this field to specify the **ImageStreamTag**.

Image change trigger status

In your build configuration, **buildConfig.status.imageChangeTriggers** is an array of **ImageChangeTriggerStatus** elements. Each **ImageChangeTriggerStatus** element includes the **from**, **lastTriggeredImageID**, and **lastTriggerTime** elements shown in the preceding example.

The **ImageChangeTriggerStatus** that has the most recent **lastTriggerTime** triggered the most recent build. You use its **name** and **namespace** to identify the image change trigger in **buildConfig.spec.triggers** that triggered the build.

The **lastTriggerTime** with the most recent timestamp signifies the **ImageChangeTriggerStatus** of the last build. This **ImageChangeTriggerStatus** has the same **name** and **namespace** as the image change trigger in **buildConfig.spec.triggers** that triggered the build.

Additional resources

- v1 container registries

2.8.1.4. 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.

2.8.1.4.1. Setting triggers manually

Triggers can be added to and removed from build configurations with **oc set triggers**.

Procedure

- To set a GitHub webhook trigger on a build configuration, use:

  ```
  $ oc set triggers bc <name> --from-github
  ```

- To set an imagechange trigger, use:

  ```
  $ oc set triggers bc <name> --from-image='<image>'
  ```

- To remove a trigger, add **--remove**:

  ```
  $ oc set triggers bc <name> --from-bitbucket --remove
  ```
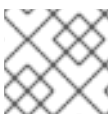
> **NOTE**
>
> When a webhook trigger already exists, adding it again regenerates the webhook secret.

For more information, consult the help documentation with by running:

```
$ oc set triggers --help
```

2.8.2. Build hooks

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

The **postCommit** field of a **BuildConfig** object runs commands inside a temporary container that is running the build output image. The hook is run 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 contains 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 running the hook cannot affect the final image. This behavior allows for, among other uses, the installation and usage of test dependencies that are automatically discarded and are not present in the final image.

2.8.2.1. Configuring post commit build hooks

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

Procedure

- 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.

- Command with arguments:

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

This form is equivalent to appending the arguments to **command**.
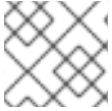
> **NOTE**
>
> Providing both **script** and **command** simultaneously creates an invalid build hook.

2.8.2.2. Using the CLI to set post commit build hooks

The **oc set build-hook** command can be used to set the build hook for a build configuration.

Procedure

1. To set a command as the post-commit build hook:

```
$ oc set build-hook bc/mybc \
    --post-commit \
    --command \
    -- bundle exec rake test --verbose
```

2. To set a script as the post-commit build hook:

```
$ oc set build-hook bc/mybc --post-commit --script="bundle exec rake test --verbose"
```

2.9. PERFORMING ADVANCED BUILDS

The following sections provide instructions for advanced build operations including setting build resources and maximum duration, assigning builds to nodes, chaining builds, build pruning, and build run policies.

2.9.1. Setting build resources

By default, builds are completed by pods using unbound resources, such as memory and CPU. These resources can be limited.

Procedure

You can limit resource use in two ways:

- Limit resource use by specifying resource limits in the default container limits of a project.

- 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"
   memory: "256Mi"
```
❶
❷

❶ **cpu** is in CPU units: **100m** represents 0.1 CPU units (100 * 1e-3).

❷ **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:
   cpu: "100m"
   memory: "256Mi"
```
❶

❶ 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.

2.9.2. Setting maximum duration

When defining a **BuildConfig** object, 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.

Procedure

- To set maximum duration, specify **completionDeadlineSeconds** in your **BuildConfig**. The following example shows the part of a **BuildConfig** specifying **completionDeadlineSeconds** field for 30 minutes:

```
spec:
  completionDeadlineSeconds: 1800
```

> **NOTE**
>
> This setting is not supported with the Pipeline Strategy option.

2.9.3. 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.

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.

> **NOTE**
>
> If the specified **NodeSelector** cannot be matched to a node with those labels, the build still stay in the **Pending** state indefinitely.

Procedure

- Assign builds to run on specific nodes by assigning labels in the **nodeSelector** field of the **BuildConfig**, for example:

  ```
  apiVersion: "v1"
  kind: "BuildConfig"
  metadata:
    name: "sample-build"
  spec:
    nodeSelector:❶
      key1: value1
      key2: value2
  ```

  ❶ Builds associated with this build configuration will run only on nodes with the **key1=value2** and **key2=value2** labels.

2.9.4. Chained builds

For compiled languages such as Go, C, C++, and Java, including the dependencies necessary for compilation in the application image might increase the size of the image or introduce vulnerabilities that can be exploited.

To avoid these problems, two builds can be chained together. One build that produces the compiled artifact, and a second build that places that artifact in a separate image that runs the artifact.

In the following example, a source-to-image (S2I) build is combined with a docker build to compile an artifact that is then placed in a separate runtime image.
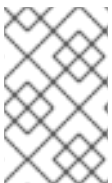
> **NOTE**
>
> Although this example chains a S2I build and a docker build, the first build can use any strategy that produces an image containing the desired artifacts, and the second build can use any strategy that can consume input content from an image.

The first build takes the application source and produces an image containing a **WAR** file. The image is pushed to the **artifact-image** image stream. The path of the output artifact depends on the **assemble** script of the S2I builder used. In this case, it is output to **/wildfly/standalone/deployments/ROOT.war**.

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: artifact-build
spec:
  output:
    to:
      kind: ImageStreamTag
      name: artifact-image:latest
  source:
    git:
      uri: https://github.com/openshift/openshift-jee-sample.git
      ref: "master"
  strategy:
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: wildfly:10.1
        namespace: openshift
```

The second build uses image source with a path to the WAR file inside the output image from the first build. An inline **dockerfile** copies that **WAR** file into a runtime image.

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: image-build
spec:
  output:
    to:
      kind: ImageStreamTag
      name: image-build:latest
  source:
    dockerfile: |-
      FROM jee-runtime:latest
      COPY ROOT.war /deployments/ROOT.war
    images:
    - from:                                            ❶
        kind: ImageStreamTag
        name: artifact-image:latest
      paths:                                           ❷
      - sourcePath: /wildfly/standalone/deployments/ROOT.war
        destinationDir: "."
  strategy:
    dockerStrategy:
      from:                                            ❸
        kind: ImageStreamTag
        name: jee-runtime:latest
  triggers:
  - imageChange: {}
    type: ImageChange
```

❶   **from** specifies that the docker build should include the output of the image from the **artifact-image** image stream, which was the target of the previous build.

2   **paths** specifies which paths from the target image to include in the current docker build.

3   The runtime image is used as the source image for the docker build.

The result of this setup is that the output image of the second build does not have to contain any of the build tools that are needed to create the **WAR** file. Also, because the second build contains an image change trigger, whenever the first build is run and produces a new image with the binary artifact, the second build is automatically triggered to produce a runtime image that contains that artifact. Therefore, both builds behave as a single build with two stages.

2.9.5. Pruning builds

By default, builds that have completed their lifecycle are persisted indefinitely. You can limit the number of previous builds that are retained.

Procedure

1. Limit the number of previous builds that are retained by supplying a positive integer value for **successfulBuildsHistoryLimit** or **failedBuildsHistoryLimit** in your **BuildConfig**, for example:

   ```
   apiVersion: "v1"
   kind: "BuildConfig"
   metadata:
     name: "sample-build"
   spec:
     successfulBuildsHistoryLimit: 2
     failedBuildsHistoryLimit: 2
   ```

   1   **successfulBuildsHistoryLimit** will retain up to two builds with a status of **completed**.

   2   **failedBuildsHistoryLimit** will retain up to two builds with a status of **failed**, **canceled**, or **error**.

2. Trigger build pruning by one of the following actions:

   - Updating a build configuration.

   - Waiting for a build to complete its lifecycle.

Builds are sorted by their creation timestamp with the oldest builds being pruned first.

> **NOTE**
>
> Administrators can manually prune builds using the 'oc adm' object pruning command.

2.9.6. 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, by:

- Changing **Parallel** to **Serial** or **SerialLatestOnly** and triggering a new build from this configuration causes 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 causes cancellation of all existing builds in queue, except the currently running build and the most recently created build. The newest build runs next.

2.10. USING RED HAT SUBSCRIPTIONS IN BUILDS

Use the following sections to run entitled builds on OpenShift Container Platform.

2.10.1. Creating an image stream tag for the Red Hat Universal Base Image

To use Red Hat subscriptions within a build, you create an image stream tag to reference the Universal Base Image (UBI).

To make the UBI available **in every project** in the cluster, you add the image stream tag to the **openshift** namespace. Otherwise, to make it available **in a specific project**, you add the image stream tag to that project.

The benefit of using image stream tags this way is that doing so grants access to the UBI based on the **registry.redhat.io** credentials in the install pull secret without exposing the pull secret to other users. This is more convenient than requiring each developer to install pull secrets with **registry.redhat.io** credentials in each project.

Procedure

- To create an **ImageStreamTag** in the **openshift** namespace, so it is available to developers in all projects, enter:

  ```
  $ oc tag --source=docker registry.redhat.io/ubi8/ubi:latest ubi:latest -n openshift
  ```

  TIP

  You can alternatively apply the following YAML to create an **ImageStreamTag** in the **openshift** namespace:

  ```
  apiVersion: image.openshift.io/v1
  kind: ImageStream
  metadata:
    name: ubi
    namespace: openshift
  spec:
    tags:
    - from:
        kind: DockerImage
        name: registry.redhat.io/ubi8/ubi:latest
      name: latest
      referencePolicy:
        type: Source
  ```

- To create an **ImageStreamTag** in a single project, enter:

```
$ oc tag --source=docker registry.redhat.io/ubi8/ubi:latest ubi:latest
```

TIP

You can alternatively apply the following YAML to create an **ImageStreamTag** in a single project:

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  name: ubi
spec:
  tags:
  - from:
      kind: DockerImage
      name: registry.redhat.io/ubi8/ubi:latest
    name: latest
    referencePolicy:
      type: Source
```

2.10.2. Adding subscription entitlements as a build secret

Builds that use Red Hat subscriptions to install content must include the entitlement keys as a build secret.

Prerequisites

You must have access to Red Hat entitlements through your subscription. The entitlement secret is automatically created by the Insights Operator.

TIP

When you perform an Entitlement Build using Red Hat Enterprise Linux (RHEL) 7, you must have the following instructions in your Dockerfile before you run any **yum** commands:

```
RUN rm /etc/rhsm-host
```

Procedure

1. Add the etc-pki-entitlement secret as a build volume in the build configuration's Docker strategy:

```
strategy:
  dockerStrategy:
    from:
      kind: ImageStreamTag
      name: ubi:latest
    volumes:
    - name: etc-pki-entitlement
      mounts:
      - destinationPath: /etc/pki/entitlement
      source:
```

```
        type: Secret
        secret:
          secretName: etc-pki-entitlement
```

2.10.3. Running builds with Subscription Manager

2.10.3.1. Docker builds using Subscription Manager

Docker strategy builds can use the Subscription Manager to install subscription content.

Prerequisites

The entitlement keys must be added as build strategy volumes.

Procedure

Use the following as an example Dockerfile to install content with the Subscription Manager:

```
FROM registry.redhat.io/ubi8/ubi:latest
RUN dnf search kernel-devel --showduplicates && \
    dnf install -y kernel-devel
```

2.10.4. Running builds with Red Hat Satellite subscriptions

2.10.4.1. Adding Red Hat Satellite configurations to builds

Builds that use Red Hat Satellite to install content must provide appropriate configurations to obtain content from Satellite repositories.

Prerequisites

- You must provide or create a **yum**-compatible repository configuration file that downloads content from your Satellite instance.

  Sample repository configuration

  ```
  [test-<name>]
  name=test-<number>
  baseurl = https://satellite.../content/dist/rhel/server/7/7Server/x86_64/os
  enabled=1
  gpgcheck=0
  sslverify=0
  sslclientkey = /etc/pki/entitlement/...-key.pem
  sslclientcert = /etc/pki/entitlement/....pem
  ```

Procedure

1. Create a **ConfigMap** containing the Satellite repository configuration file:

   ```
   $ oc create configmap yum-repos-d --from-file /path/to/satellite.repo
   ```

2. Add the Satellite repository configuration and entitlement key as a build volumes:

```
strategy:
  dockerStrategy:
    from:
      kind: ImageStreamTag
      name: ubi:latest
    volumes:
    - name: yum-repos-d
      mounts:
      - destinationPath: /etc/yum.repos.d
      source:
        type: ConfigMap
        configMap:
          name: yum-repos-d
    - name: etc-pki-entitlement
      mounts:
      - destinationPath: /etc/pki/entitlement
      source:
        type: Secret
        secret:
          secretName: etc-pki-entitlement
```

2.10.4.2. Docker builds using Red Hat Satellite subscriptions

Docker strategy builds can use Red Hat Satellite repositories to install subscription content.

Prerequisites

- You have added the entitlement keys and Satellite repository configurations as build volumes.

Procedure

Use the following as an example Dockerfile to install content with Satellite:

```
FROM registry.redhat.io/ubi8/ubi:latest
RUN dnf search kernel-devel --showduplicates && \
    dnf install -y kernel-devel
```

Additional resources

- [How to use builds with Red Hat Satellite subscriptions and which certificate to use](#)

2.10.5. Running entitled builds using SharedSecret objects

You can configure and perform a build in one namespace that securely uses RHEL entitlements from a **Secret** object in another namespace.

You can still access RHEL entitlements from OpenShift Builds by creating a **Secret** object with your subscription credentials in the same namespace as your **Build** object. However, now, in OpenShift Container Platform 4.10 and later, you can access your credentials and certificates from a **Secret** object in one of the OpenShift Container Platform system namespaces. You run entitled builds with a CSI volume mount of a **SharedSecret** custom resource (CR) instance that references the **Secret** object.

This procedure relies on the newly introduced Shared Resources CSI Driver feature, which you can use to declare CSI Volume mounts in OpenShift Container Platform Builds. It also relies on the OpenShift Container Platform Insights Operator.

> **IMPORTANT**
>
> The Shared Resources CSI Driver and The Build CSI Volumes are both Technology Preview features, which are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.
>
> For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .
>
> The Shared Resources CSI Driver and the Build CSI Volumes features also belong to the **TechPreviewNoUpgrade** feature set, which is a subset of the current Technology Preview features. You can enable the **TechPreviewNoUpgrade** feature set on test clusters, where you can fully test them while leaving the features disabled on production clusters. Enabling this feature set cannot be undone and prevents minor version updates. This feature set is not recommended on production clusters. See "Enabling Technology Preview features using feature gates" in the following "Additional resources" section.

Prerequisites

- You have enabled the **TechPreviewNoUpgrade** feature set by using the feature gates.

- You have a **SharedSecret** custom resource (CR) instance that references the **Secret** object where the Insights Operator stores the subscription credentials.

- You must have permission to perform the following actions:

  - Create build configs and start builds.

  - Discover which **SharedSecret** CR instances are available by entering the **oc get sharedsecrets** command and getting a non-empty list back.

  - Determine if the **builder** service account available to you in your namespace is allowed to use the given **SharedSecret** CR instance. In other words, you can run **oc adm policy who-can use <identifier of specific SharedSecret>** to see if the **builder** service account in your namespace is listed.

> **NOTE**
>
> If neither of the last two prerequisites in this list are met, establish, or ask someone to establish, the necessary role-based access control (RBAC) so that you can discover **SharedSecret** CR instances and enable service accounts to use **SharedSecret** CR instances.

Procedure

1. Grant the **builder** service account RBAC permissions to use the **SharedSecret** CR instance by using **oc apply** with YAML content:
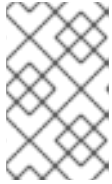
> **NOTE**
>
> Currently, **kubectl** and **oc** have hard-coded special case logic restricting the **use** verb to roles centered around pod security. Therefore, you cannot use **oc create role ...** to create the role needed for consuming **SharedSecret** CR instances.

Example oc apply -f command with YAML Role object definition

```
$ oc apply -f - <<EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: shared-resource-my-share
  namespace: my-namespace
rules:
  - apiGroups:
      - sharedresource.openshift.io
    resources:
      - sharedsecrets
    resourceNames:
      - my-share
    verbs:
      - use
EOF
```

2. Create the **RoleBinding** associated with the role by using the **oc** command:

   Example oc create rolebinding command

   ```
   $ oc create rolebinding shared-resource-my-share --role=shared-resource-my-share --serviceaccount=my-namespace:builder
   ```

3. Create a **BuildConfig** object that accesses the RHEL entitlements.

   Example YAML BuildConfig object definition

   ```
   apiVersion: build.openshift.io/v1
   kind: BuildConfig
   metadata:
     name: my-csi-bc
     namespace: my-csi-app-namespace
   spec:
     runPolicy: Serial
     source:
       dockerfile: |
         FROM registry.redhat.io/ubi8/ubi:latest
         RUN ls -la /etc/pki/entitlement
         RUN rm /etc/rhsm-host
         RUN yum repolist --disablerepo=*
         RUN subscription-manager repos --enable rhocp-4.9-for-rhel-8-x86_64-rpms
         RUN yum -y update
         RUN yum install -y openshift-clients.x86_64
     strategy:
       type: Docker
   ```

```
dockerStrategy:
 volumes:
   - mounts:
      - destinationPath: "/etc/pki/entitlement"
    name: my-csi-shared-secret
    source:
     csi:
       driver: csi.sharedresource.openshift.io
       readOnly: true
       volumeAttributes:
         sharedSecret: my-share-bc
     type: CSI
```

4. Start a build from the **BuildConfig** object and follow the logs with the **oc** command.

Example oc start-build command

```
$ oc start-build my-csi-bc -F
```

Example 2.1. Example output from the oc start-build command

> **NOTE**
>
> Some sections of the following output have been replaced with ...

```
build.build.openshift.io/my-csi-bc-1 started
Caching blobs under "/var/cache/blobs".

Pulling image registry.redhat.io/ubi8/ubi:latest ...
Trying to pull registry.redhat.io/ubi8/ubi:latest...
Getting image source signatures
Copying blob
sha256:5dcbdc60ea6b60326f98e2b49d6ebcb7771df4b70c6297ddf2d7dede6692df6e
Copying blob
sha256:8671113e1c57d3106acaef2383f9bbfe1c45a26eacb03ec82786a494e15956c3
Copying config
sha256:b81e86a2cb9a001916dc4697d7ed4777a60f757f0b8dcc2c4d8df42f2f7edb3a
Writing manifest to image destination
Storing signatures
Adding transient rw bind mount for /run/secrets/rhsm
STEP 1/9: FROM registry.redhat.io/ubi8/ubi:latest
STEP 2/9: RUN ls -la /etc/pki/entitlement
total 360
drwxrwxrwt. 2 root root  80 Feb  3 20:28 .
drwxr-xr-x. 10 root root 154 Jan 27 15:53 ..
-rw-r--r--. 1 root root   3243 Feb  3 20:28 entitlement-key.pem
-rw-r--r--. 1 root root 362540 Feb  3 20:28 entitlement.pem
time="2022-02-03T20:28:32Z" level=warning msg="Adding metacopy option, configured
globally"
--> 1ef7c6d8c1a
STEP 3/9: RUN rm /etc/rhsm-host
time="2022-02-03T20:28:33Z" level=warning msg="Adding metacopy option, configured
globally"
--> b1c61f88b39
```

```
STEP 4/9: RUN yum repolist --disablerepo=*
Updating Subscription Management repositories.


...


--> b067f1d63eb
STEP 5/9: RUN subscription-manager repos --enable rhocp-4.9-for-rhel-8-x86_64-rpms
Repository 'rhocp-4.9-for-rhel-8-x86_64-rpms' is enabled for this system.
time="2022-02-03T20:28:40Z" level=warning msg="Adding metacopy option, configured
globally"
--> 03927607ebd
STEP 6/9: RUN yum -y update
Updating Subscription Management repositories.


...


Upgraded:
  systemd-239-51.el8_5.3.x86_64      systemd-libs-239-51.el8_5.3.x86_64
  systemd-pam-239-51.el8_5.3.x86_64
Installed:
  diffutils-3.6-6.el8.x86_64          libxkbcommon-0.9.1-1.el8.x86_64
  xkeyboard-config-2.28-1.el8.noarch

Complete!
time="2022-02-03T20:29:05Z" level=warning msg="Adding metacopy option, configured
globally"
--> db57e92ff63
STEP 7/9: RUN yum install -y openshift-clients.x86_64
Updating Subscription Management repositories.


...


Installed:
  bash-completion-1:2.7-5.el8.noarch
  libpkgconf-1.4.2-1.el8.x86_64
  openshift-clients-4.9.0-202201211735.p0.g3f16530.assembly.stream.el8.x86_64
  pkgconf-1.4.2-1.el8.x86_64
  pkgconf-m4-1.4.2-1.el8.noarch
  pkgconf-pkg-config-1.4.2-1.el8.x86_64

Complete!
time="2022-02-03T20:29:19Z" level=warning msg="Adding metacopy option, configured
globally"
--> 609507b059e
STEP 8/9: ENV "OPENSHIFT_BUILD_NAME"="my-csi-bc-1"
"OPENSHIFT_BUILD_NAMESPACE"="my-csi-app-namespace"
--> cab2da3efc4
STEP 9/9: LABEL "io.openshift.build.name"="my-csi-bc-1"
"io.openshift.build.namespace"="my-csi-app-namespace"
COMMIT temp.builder.openshift.io/my-csi-app-namespace/my-csi-bc-1:edfe12ca
--> 821b582320b
Successfully tagged temp.builder.openshift.io/my-csi-app-namespace/my-csi-bc-
1:edfe12ca
821b582320b41f1d7bab4001395133f86fa9cc99cc0b2b64c5a53f2b6750db91
Build complete, no image push requested
```

2.10.6. Additional resources

- Importing simple content access certificates with Insights Operator

- Enabling features using feature gates

- Managing image streams

- build strategy

2.11. SECURING BUILDS BY STRATEGY

Builds in OpenShift Container Platform are run in privileged containers. Depending on the build strategy used, if you have privileges, you can run builds to escalate their permissions on the cluster and host nodes. And as a security measure, it limits who can run builds and the strategy that is used for those builds. Custom builds are inherently less safe than source builds, because they can execute any code within a privileged container, and are disabled by default. Grant docker build permissions with caution, because a vulnerability in the Dockerfile processing logic could result in a privileges being granted on the host node.

By default, all users that can create builds are granted permission to use the docker and Source-to-image (S2I) build strategies. Users with cluster administrator privileges can enable the custom build strategy, as referenced in the restricting build strategies to a user globally section.

You can control who can build and which build strategies they can use by using an authorization policy. Each build strategy has a corresponding build subresource. A user must have permission to create a build and permission to create on the build strategy subresource to create builds using that strategy. Default roles are provided that grant the create permission on the build strategy subresource.

Table 2.3. Build Strategy Subresources and Roles

Strategy	Subresource	Role
Docker	builds/docker	system:build-strategy-docker
Source-to-Image	builds/source	system:build-strategy-source
Custom	builds/custom	system:build-strategy-custom
JenkinsPipeline	builds/jenkinspipeline	system:build-strategy-jenkinspipeline

2.11.1. Disabling access to a build strategy globally

To prevent access to a particular build strategy globally, log in as a user with cluster administrator privileges, remove the corresponding role from the **system:authenticated** group, and apply the annotation **rbac.authorization.kubernetes.io/autoupdate: "false"** to protect them from changes between the API restarts. The following example shows disabling the docker build strategy.

Procedure

1. Apply the **rbac.authorization.kubernetes.io/autoupdate** annotation:

   ```
   $ oc edit clusterrolebinding system:build-strategy-docker-binding
   ```

   Example output

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: ClusterRoleBinding
   metadata:
     annotations:
       rbac.authorization.kubernetes.io/autoupdate: "false"  1
     creationTimestamp: 2018-08-10T01:24:14Z
     name: system:build-strategy-docker-binding
     resourceVersion: "225"
     selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindings/system%3Abuild-strategy-
   docker-binding
     uid: 17b1f3d4-9c3c-11e8-be62-0800277d20bf
   roleRef:
     apiGroup: rbac.authorization.k8s.io
     kind: ClusterRole
     name: system:build-strategy-docker
   subjects:
   - apiGroup: rbac.authorization.k8s.io
     kind: Group
     name: system:authenticated
   ```

   1   Change the **rbac.authorization.kubernetes.io/autoupdate** annotation's value to **"false"**.

2. Remove the role:

   ```
   $ oc adm policy remove-cluster-role-from-group system:build-strategy-docker
   system:authenticated
   ```

3. Ensure the build strategy subresources are also removed from these roles:

   ```
   $ oc edit clusterrole admin
   ```

   ```
   $ oc edit clusterrole edit
   ```

4. For each role, specify the subresources that correspond to the resource of the strategy to disable.

   a. Disable the docker Build Strategy for **admin**:

      ```
      kind: ClusterRole
      metadata:
        name: admin
      ...
      - apiGroups:
        - ""
        - build.openshift.io
        resources:
        - buildconfigs
      ```

```
- buildconfigs/webhooks
- builds/custom 1
- builds/source
verbs:
- create
- delete
- deletecollection
- get
- list
- patch
- update
- watch
...
```

1   Add **builds/custom** and **builds/source** to disable docker builds globally for users with the **admin** role.

2.11.2. Restricting build strategies to users globally

You can allow a set of specific users to create builds with a particular strategy.

Prerequisites

- Disable global access to the build strategy.

Procedure

- Assign the role that corresponds to the build strategy to a specific user. For example, to add the **system:build-strategy-docker** cluster role to the user **devuser**:

  ```
  $ oc adm policy add-cluster-role-to-user system:build-strategy-docker devuser
  ```

> **WARNING**
>
> Granting a user access at the cluster level to the **builds/docker** subresource means that the user can create builds with the docker strategy in any project in which they can create builds.

2.11.3. Restricting build strategies to a user within a project

Similar to granting the build strategy role to a user globally, you can allow a set of specific users within a project to create builds with a particular strategy.

Prerequisites

- Disable global access to the build strategy.

Procedure

- Assign the role that corresponds to the build strategy to a specific user within a project. For example, to add the **system:build-strategy-docker** role within the project **devproject** to the user **devuser**:

```
$ oc adm policy add-role-to-user system:build-strategy-docker devuser -n devproject
```

2.12. BUILD CONFIGURATION RESOURCES

Use the following procedure to configure build settings.

2.12.1. Build controller configuration parameters

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

Parameter	Description
Build	Holds cluster-wide information on how to handle builds. The canonical, and only valid name is **cluster**. **spec**: Holds user-settable values for the build controller configuration.
buildDefaults	Controls the default information for builds. **defaultProxy**: Contains the default proxy settings for all build operations, including image pull or push and source download. You can override values by setting the **HTTP_PROXY**, **HTTPS_PROXY**, and **NO_PROXY** environment variables in the **BuildConfig** strategy. **gitProxy**: Contains the proxy settings for Git operations only. If set, this overrides any proxy settings for all Git commands, such as **git clone**. Values that are not set here are inherited from DefaultProxy. **env**: A set of default environment variables that are applied to the build if the specified variables do not exist on the build. **imageLabels**: A list of labels that are applied to the resulting image. You can override a default label by providing a label with the same name in the **BuildConfig**. **resources**: Defines resource requirements to execute the build.
ImageLabel	**name**: Defines the name of the label. It must have non-zero length.

Parameter	Description
buildOverrides	Controls override settings for builds. **imageLabels**: A list of labels that are applied to the resulting image. If you provided a label in the **BuildConfig** with the same name as one in this table, your label will be overwritten. **nodeSelector**: A selector which must be true for the build pod to fit on a node. **tolerations**: A list of tolerations that overrides any existing tolerations set on a build pod.
BuildList	**items**: Standard object's metadata.

2.12.2. Configuring build settings

You can configure build settings by editing the **build.config.openshift.io/cluster** resource.

Procedure

- Edit the **build.config.openshift.io/cluster** resource:

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

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

  ```
  apiVersion: config.openshift.io/v1
  kind: Build 1
  metadata:
    annotations:
      release.openshift.io/create-only: "true"
    creationTimestamp: "2019-05-17T13:44:26Z"
    generation: 2
    name: cluster
    resourceVersion: "107233"
    selfLink: /apis/config.openshift.io/v1/builds/cluster
    uid: e2e9cc14-78a9-11e9-b92b-06d6c7da38dc
  spec:
    buildDefaults: 2
      defaultProxy: 3
        httpProxy: http://proxy.com
        httpsProxy: https://proxy.com
        noProxy: internal.com
      env: 4
      - name: envkey
        value: envvalue
      gitProxy: 5
        httpProxy: http://gitproxy.com
        httpsProxy: https://gitproxy.com
  ```

```
      noProxy: internalgit.com
    imageLabels: 6
    - name: labelkey
      value: labelvalue
    resources: 7
      limits:
        cpu: 100m
        memory: 50Mi
      requests:
        cpu: 10m
        memory: 10Mi
  buildOverrides: 8
    imageLabels: 9
    - name: labelkey
      value: labelvalue
    nodeSelector: 10
      selectorkey: selectorvalue
    tolerations: 11
    - effect: NoSchedule
      key: node-role.kubernetes.io/builds
operator: Exists
```

[1] **Build**: Holds cluster-wide information on how to handle builds. The canonical, and only valid name is **cluster**.

[2] **buildDefaults**: Controls the default information for builds.

[3] **defaultProxy**: Contains the default proxy settings for all build operations, including image pull or push and source download.

[4] **env**: A set of default environment variables that are applied to the build if the specified variables do not exist on the build.

[5] **gitProxy**: Contains the proxy settings for Git operations only. If set, this overrides any Proxy settings for all Git commands, such as **git clone**.

[6] **imageLabels**: A list of labels that are applied to the resulting image. You can override a default label by providing a label with the same name in the **BuildConfig**.

[7] **resources**: Defines resource requirements to execute the build.

[8] **buildOverrides**: Controls override settings for builds.

[9] **imageLabels**: A list of labels that are applied to the resulting image. If you provided a label in the **BuildConfig** with the same name as one in this table, your label will be overwritten.

[10] **nodeSelector**: A selector which must be true for the build pod to fit on a node.

[11] **tolerations**: A list of tolerations that overrides any existing tolerations set on a build pod.

2.13. TROUBLESHOOTING BUILDS

Use the following to troubleshoot build issues.

2.13.1. Resolving denial for access to resources

If your request for access to resources is denied:

Issue

A build fails with:

> requested access to the resource is denied

Resolution

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

2.13.2. Service certificate generation failure

If your request for access to resources is denied:

Issue

If a service certificate generation fails with (service's **service.beta.openshift.io/serving-cert-generation-error** annotation contains):

Example output

> secret/ssl-key references serviceUID 62ad25ca-d703-11e6-9d6f-0e9c0057b608, which does not match 77b6dd80-d716-11e6-9d6f-0e9c0057b60

Resolution

The service that generated the certificate no longer exists, or has a different **serviceUID**. You must force certificates regeneration by removing the old secret, and clearing the following annotations on the service: **service.beta.openshift.io/serving-cert-generation-error** and **service.beta.openshift.io/serving-cert-generation-error-num**:

> $ oc delete secret <secret_name>

> $ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-

> $ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-num-
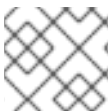
> **NOTE**
>
> The command removing annotation has a **-** after the annotation name to be removed.

2.14. SETTING UP ADDITIONAL TRUSTED CERTIFICATE AUTHORITIES FOR BUILDS

Use the following sections to set up additional certificate authorities (CA) to be trusted by builds when pulling images from an image registry.

The procedure requires a cluster administrator to create a **ConfigMap** and add additional CAs as keys in the **ConfigMap**.

- The **ConfigMap** must be created in the **openshift-config** namespace.

- **domain** is the key in the **ConfigMap** and **value** is the PEM-encoded certificate.

  - Each CA must be associated with a domain. The domain format is **hostname[..port]**.

- The **ConfigMap** name must be set in the **image.config.openshift.io/cluster** cluster scoped configuration resource's **spec.additionalTrustedCA** field.

2.14.1. Adding certificate authorities to the cluster

You can add certificate authorities (CA) to the cluster for use when pushing and pulling images with the following procedure.

Prerequisites

- You must have access to the public certificates of the registry, usually a **hostname/ca.crt** file located in the **/etc/docker/certs.d/** directory.

Procedure

1. Create a **ConfigMap** in the **openshift-config** namespace containing the trusted certificates for the registries that use self-signed certificates. For each CA file, ensure the key in the **ConfigMap** is the hostname of the registry in the **hostname[..port]** format:

   ```
   $ oc create configmap registry-cas -n openshift-config \
   --from-file=myregistry.corp.com..5000=/etc/docker/certs.d/myregistry.corp.com:5000/ca.crt \
   --from-file=otherregistry.com=/etc/docker/certs.d/otherregistry.com/ca.crt
   ```

2. Update the cluster image configuration:

   ```
   $ oc patch image.config.openshift.io/cluster --patch '{"spec":{"additionalTrustedCA":
   {"name":"registry-cas"}}}' --type=merge
   ```

2.14.2. Additional resources

- Create a **ConfigMap**

- Secrets and **ConfigMaps**

- Configuring a custom PKI

CHAPTER 3. PIPELINES

3.1. ABOUT RED HAT OPENSHIFT PIPELINES

Red Hat OpenShift Pipelines is a cloud-native, continuous integration and continuous delivery (CI/CD) solution based on Kubernetes resources. It uses Tekton building blocks to automate deployments across multiple platforms by abstracting away the underlying implementation details. Tekton introduces a number of standard custom resource definitions (CRDs) for defining CI/CD pipelines that are portable across Kubernetes distributions.
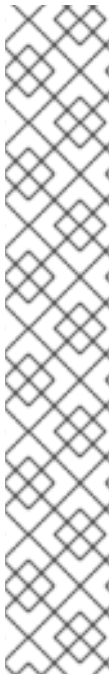
> **NOTE**
>
> Because Red Hat OpenShift Pipelines releases on a different cadence from OpenShift Container Platform, the Red Hat OpenShift Pipelines documentation is now available as separate documentation sets for each minor version of the product.
>
> The Red Hat OpenShift Pipelines documentation is available at https://docs.openshift.com/pipelines/.
>
> Documentation for specific versions is available using the version selector drop-down list, or directly by adding the version to the URL, for example, https://docs.openshift.com/pipelines/1.11.
>
> In addition, the Red Hat OpenShift Pipelines documentation is also available on the Red Hat Customer Portal at https://access.redhat.com/documentation/en-us/red_hat_openshift_pipelines/.
>
> For additional information about the Red Hat OpenShift Pipelines life cycle and supported platforms, refer to the Platform Life Cycle Policy.

CHAPTER 4. GITOPS

4.1. ABOUT RED HAT OPENSHIFT GITOPS

Red Hat OpenShift GitOps is an Operator that uses Argo CD as the declarative GitOps engine. It enables GitOps workflows across multicluster OpenShift and Kubernetes infrastructure. Using Red Hat OpenShift GitOps, administrators can consistently configure and deploy Kubernetes-based infrastructure and applications across clusters and development lifecycles. Red Hat OpenShift GitOps is based on the open source project Argo CD and provides a similar set of features to what the upstream offers, with additional automation, integration into Red Hat {OCP} and the benefits of Red Hat's enterprise support, quality assurance and focus on enterprise security.

> **NOTE**
>
> Because Red Hat OpenShift GitOps releases on a different cadence from {OCP}, the Red Hat OpenShift GitOps documentation is now available as separate documentation sets for each minor version of the product.
>
> The Red Hat OpenShift GitOps documentation is available at https://docs.openshift.com/gitops/.
>
> Documentation for specific versions is available using the version selector dropdown, or directly by adding the version to the URL, for example, https://docs.openshift.com/gitops/1.8.
>
> In addition, the Red Hat OpenShift GitOps documentation is also available on the Red Hat Portal at https://access.redhat.com/documentation/en-us/red_hat_openshift_gitops/.
>
> For additional information about the Red Hat OpenShift GitOps life cycle and supported platforms, refer to the Platform Life Cycle Policy.

Red Hat OpenShift GitOps ensures consistency in applications when you deploy them to different clusters in different environments, such as: development, staging, and production. Red Hat OpenShift GitOps organizes the deployment process around the configuration repositories and makes them the central element. It always has at least two repositories:

1. Application repository with the source code

2. Environment configuration repository that defines the desired state of the application

These repositories contain a declarative description of the infrastructure you need in your specified environment. They also contain an automated process to make your environment match the described state.

Red Hat OpenShift GitOps uses Argo CD to maintain cluster resources. Argo CD is an open-source declarative tool for the continuous integration and continuous deployment (CI/CD) of applications. Red Hat OpenShift GitOps implements Argo CD as a controller so that it continuously monitors application definitions and configurations defined in a Git repository. Then, Argo CD compares the specified state of these configurations with their live state on the cluster.

Argo CD reports any configurations that deviate from their specified state. These reports allow administrators to automatically or manually resync configurations to the defined state. Therefore, Argo CD enables you to deliver global custom resources, like the resources that are used to configure {OCP} clusters.

4.1.1. Key features

Red Hat OpenShift GitOps helps you automate the following tasks:

- Ensure that the clusters have similar states for configuration, monitoring, and storage

- Apply or revert configuration changes to multiple {OCP} clusters

- Associate templated configuration with different environments

- Promote applications across clusters, from staging to production

4.1.2. Additional resources

- Extending the Kubernetes API with custom resource definitions

- Managing resources from custom resource definitions

- What is GitOps?

CHAPTER 5. JENKINS

5.1. CONFIGURING JENKINS IMAGES

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

The image is based on the Red Hat Universal Base Images (UBI).

OpenShift Container Platform follows the LTS release of Jenkins. OpenShift Container Platform provides an image that contains Jenkins 2.x.

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

For example:

```
$ podman pull registry.redhat.io/ocp-tools-4/jenkins-rhel8:<image_tag>
```

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

But for convenience, OpenShift Container Platform provides image streams in the **openshift** namespace for the core Jenkins image as well as the example Agent images provided for OpenShift Container Platform integration with Jenkins.

5.1.1. Configuration and customization

You can manage Jenkins authentication in two ways:

- OpenShift Container Platform OAuth authentication provided by the OpenShift Container Platform Login plugin.

- Standard authentication provided by Jenkins.

5.1.1.1. OpenShift Container Platform OAuth authentication

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

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

Jenkins supports both browser and non-browser access.

Valid users are automatically added to the Jenkins authorization matrix at log in, where OpenShift Container Platform roles dictate the specific Jenkins permissions that users have. The roles used by default are the predefined **admin**, **edit**, and **view**. The login plugin executes self-SAR requests against those roles in the project or namespace that Jenkins is running in.

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

The default OpenShift Container Platform **admin**, **edit**, and **view** roles and the Jenkins permissions those roles are assigned in the Jenkins instance are configurable.

When running Jenkins in an OpenShift Container Platform pod, the login plugin looks for a config map named **openshift-jenkins-login-plugin-config** in the namespace that Jenkins is running in.

If this plugin finds and can read in that config map, you can define the role to Jenkins Permission mappings. Specifically:

- The login plugin treats the key and value pairs in the config map as Jenkins permission to OpenShift Container Platform role mappings.

- The key is the Jenkins permission group short ID and the Jenkins permission short ID, with those two separated by a hyphen character.

- If you want to add the **Overall Jenkins Administer** permission to an OpenShift Container Platform role, the key should be **Overall-Administer**.

- To get a sense of which permission groups and permissions IDs are available, go to the matrix authorization page in the Jenkins console and IDs for the groups and individual permissions in the table they provide.

- The value of the key and value pair is the list of OpenShift Container Platform roles the permission should apply to, with each role separated by a comma.

- If you want to add the **Overall Jenkins Administer** permission to both the default **admin** and **edit** roles, as well as a new Jenkins role you have created, the value for the key **Overall-Administer** would be **admin,edit,jenkins**.

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

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

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

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

5.1.1.2. Jenkins authentication

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

The first time Jenkins starts, the configuration is created along with the administrator user and

password. The default user credentials are **admin** and **password**. Configure the default password by setting the **JENKINS_PASSWORD** environment variable when using, and only when using, standard Jenkins authentication.

Procedure

- Create a Jenkins application that uses standard Jenkins authentication:

```
$ oc new-app -e \
    JENKINS_PASSWORD=<password> \
    ocp-tools-4/jenkins-rhel8
```

5.1.2. Jenkins environment variables

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

Variable	Definition	Example values and settings
OPENSHIFT_ENABLE_OAUTH	Determines whether the OpenShift Container Platform Login plugin manages authentication when logging in to Jenkins. To enable, set to **true**.	Default: **false**
JENKINS_PASSWORD	The password for the **admin** user when using standard Jenkins authentication. Not applicable when **OPENSHIFT_ENABLE_OAUTH** is set to **true**.	Default: **password**
JAVA_MAX_HEAP_PARAM, **CONTAINER_HEAP_PERCENT**, **JENKINS_MAX_HEAP_UPPER_BOUND_MB**	These values control the maximum heap size of the Jenkins JVM. If **JAVA_MAX_HEAP_PARAM** is set, its value takes precedence. Otherwise, the maximum heap size is dynamically calculated as **CONTAINER_HEAP_PERCENT** of the container memory limit, optionally capped at **JENKINS_MAX_HEAP_UPPER_BOUND_MB** MiB. By default, the maximum heap size of the Jenkins JVM is set to 50% of the container memory limit with no cap.	**JAVA_MAX_HEAP_PARAM** example setting: **-Xmx512m** **CONTAINER_HEAP_PERCENT** default: **0.5**, or 50% **JENKINS_MAX_HEAP_UPPER_BOUND_MB** example setting: **512 MiB**

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

Variable	Definition	Example values and settings
INSTALL_PLUGINS	Specifies additional Jenkins plugins to install when the container is first run or when **OVERRIDE_PV_PLUGINS_WITH_IMAGE_PLUGINS** is set to **true**. Plugins are specified as a comma-delimited list of name:version pairs.	Example setting: **git:3.7.0,subversion:2.10.2**.
OPENSHIFT_PERMISSIONS_POLL_INTERVAL	Specifies the interval in milliseconds that the OpenShift Container Platform Login plugin polls OpenShift Container Platform for the permissions that are associated with each user that is defined in Jenkins.	Default: **300000** - 5 minutes
OVERRIDE_PV_CONFIG_WITH_IMAGE_CONFIG	When running this image with an OpenShift Container Platform persistent volume (PV) for the Jenkins configuration directory, the transfer of configuration from the image to the PV is performed only the first time the image starts because the PV is assigned when the persistent volume claim (PVC) is created. If you create a custom image that extends this image and updates the configuration in the custom image after the initial startup, the configuration is not copied over unless you set this environment variable to **true**.	Default: **false**
OVERRIDE_PV_PLUGINS_WITH_IMAGE_PLUGINS	When running this image with an OpenShift Container Platform PV for the Jenkins configuration directory, the transfer of plugins from the image to the PV is performed only the first time the image starts because the PV is assigned when the PVC is created. If you create a custom image that extends this image and updates plugins in the custom image after the initial startup, the plugins are not copied over unless you set this environment variable to **true**.	Default: **false**

Variable	Definition	Example values and settings
ENABLE_FATAL_ERROR_LOG_FILE	When running this image with an OpenShift Container Platform PVC for the Jenkins configuration directory, this environment variable allows the fatal error log file to persist when a fatal error occurs. The fatal error file is saved at **/var/lib/jenkins/logs**.	Default: **false**
AGENT_BASE_IMAGE	Setting this value overrides the image used for the **jnlp** container in the sample Kubernetes plugin pod templates provided with this image. Otherwise, the image from the **jenkins-agent-base-rhel8:latest** image stream tag in the **openshift** namespace is used.	Default: **image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-base-rhel8:latest**
JAVA_BUILDER_IMAGE	Setting this value overrides the image used for the **java-builder** container in the **java-builder** sample Kubernetes plugin pod templates provided with this image. Otherwise, the image from the **java:latest** image stream tag in the **openshift** namespace is used.	Default: **image-registry.openshift-image-registry.svc:5000/openshift/java:latest**
NODEJS_BUILDER_IMAGE	Setting this value overrides the image used for the **nodejs-builder** container in the **nodejs-builder** sample Kubernetes plugin pod templates provided with this image. Otherwise, the image from the **nodejs:latest** image stream tag in the **openshift** namespace is used.	Default: **image-registry.openshift-image-registry.svc:5000/openshift/nodejs:latest**
JAVA_FIPS_OPTIONS	Setting this value controls how the JVM operates when running on a FIPS node. For more information, see [Configure OpenJDK 11 in FIPS mode](#).	Default: **-Dcom.redhat.fips=false**

5.1.3. Providing Jenkins cross project access

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

Procedure

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

```
$ oc describe serviceaccount jenkins
```

Example output

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

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

2. Retrieve the token from the secret:

```
$ oc describe secret <secret name from above>
```

Example output

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

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

5.1.4. Jenkins cross volume mount points

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

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

5.1.5. Customizing the Jenkins image through source-to-image

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

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

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

plugins

> This directory contains those binary Jenkins plugins you want to copy into Jenkins.

plugins.txt

> This file lists the plugins you want to install using the following syntax:

> pluginId:pluginVersion

configuration/jobs

> This directory contains the Jenkins job definitions.

configuration/config.xml

> This file contains your custom Jenkins configuration.

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

Sample build configuration customizes the Jenkins image in OpenShift Container Platform

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: custom-jenkins-build
spec:
 source:                          1
   git:
     uri: https://github.com/custom/repository
   type: Git
 strategy:                        2
   sourceStrategy:
     from:
       kind: ImageStreamTag
       name: jenkins:2
       namespace: openshift
   type: Source
 output:                          3
   to:
     kind: ImageStreamTag
     name: custom-jenkins:latest
```

1   The **source** parameter defines the source Git repository with the layout described above.

2   The **strategy** parameter defines the original Jenkins image to use as a source image for the build.

3   The **output** parameter defines the resulting, customized Jenkins image that you can use in deployment configurations instead of the official Jenkins image.

5.1.6. Configuring the Jenkins Kubernetes plugin

The OpenShift Jenkins image includes the pre-installed Kubernetes plugin so that Jenkins agents can be dynamically provisioned on multiple container hosts using Kubernetes and OpenShift Container Platform.

To use the Kubernetes plugin, OpenShift Container Platform provides images that are suitable for use as Jenkins agents, including the Base, Maven, and Node.js images.

> **IMPORTANT**
>
> OpenShift Container Platform 4.11 moves the OpenShift Jenkins and OpenShift Agent Base images to the **ocp-tools-4** repository at **registry.redhat.io** so that Red Hat can produce and update the images outside the OpenShift Container Platform lifecycle. Previously, these images were in the OpenShift Container Platform install payload and the **openshift4** repository at **registry.redhat.io**.
>
> OpenShift Container Platform 4.11 removes the OpenShift Jenkins Maven and NodeJS Agent images from its payload. Red Hat no longer produces these images, and they are not available from the **ocp-tools-4** repository at **registry.redhat.io**. Red Hat maintains the 4.10 and earlier versions of these images for any significant bug fixes or security CVEs, following the OpenShift Container Platform lifecycle policy.
>
> For more information, see the "Important changes to OpenShift Jenkins images" link in the following "Additional resources" section.

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

> **IMPORTANT**
>
> In OpenShift Container Platform 4.10 and later, the recommended pattern for running Jenkins agents using the Kubernetes plugin is to use pod templates with both **jnlp** and **sidecar** containers. The **jnlp** container uses the OpenShift Container Platform Jenkins Base agent image to facilitate launching a separate pod for your build. The **sidecar** container image has the tools needed to build in a particular language within the separate pod that was launched. Many container images from the Red Hat Container Catalog are referenced in the sample image streams present in the **openshift** namespace. The OpenShift Container Platform Jenkins image has two pod templates named **java-build** and **nodejs-builder** with sidecar containers that demonstrate this approach. These two pod templates use the latest Java and NodeJS versions provided by the **java** and **nodejs** image streams in the **openshift** namespace.
>
> With this update, in OpenShift Container Platform 4.10 and later, the non-sidecar **maven** and **nodejs** pod templates for Jenkins are deprecated. These pod templates are planned for removal in a future release. Bug fixes and support are provided through the end of that future life cycle, after which no new feature enhancements will be made.

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

With the OpenShift Container Platform sync plugin, the Jenkins image on Jenkins startup searches for the following within the project that it is running or the projects specifically listed in the plugin's configuration:

- Image streams that have the label **role** set to **jenkins-agent**.

- Image stream tags that have the annotation **role** set to **jenkins-agent**.

- Config maps that have the label **role** set to **jenkins-agent**.

When it finds an image stream with the appropriate label, or image stream tag with the appropriate annotation, it generates the corresponding Kubernetes plugin configuration so you can assign your Jenkins jobs to run in a pod that runs the container image that is provided by the image stream.

The name and image references of the image stream or image stream tag are mapped to the name and image fields in the Kubernetes plugin pod template. You can control the label field of the Kubernetes plugin pod template by setting an annotation on the image stream or image stream tag object with the key **agent-label**. Otherwise, the name is used as the label.

> **NOTE**
>
> Do not log in to the Jenkins console and change the pod template configuration. If you do so after the pod template is created, and the OpenShift Container Platform Sync plugin detects that the image associated with the image stream or image stream tag has changed, it replaces the pod template and overwrites those configuration changes. You cannot merge a new configuration with the existing configuration.
>
> Consider the config map approach if you have more complex configuration needs.

When it finds a config map with the appropriate label, it assumes that any values in the key-value data payload of the config map contain Extensible Markup Language (XML) that is consistent with the configuration format for Jenkins and the Kubernetes plugin pod templates. One key benefit of using config maps, rather than image streams or image stream tags, is that you can control all the parameters of the Kubernetes plugin pod template.

Sample config map for jenkins-agent

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: jenkins-agent
  labels:
    role: jenkins-agent
data:
  template1: |-
    <org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
      <inheritFrom></inheritFrom>
      <name>template1</name>
      <instanceCap>2147483647</instanceCap>
      <idleMinutes>0</idleMinutes>
      <label>template1</label>
      <serviceAccount>jenkins</serviceAccount>
      <nodeSelector></nodeSelector>
      <volumes/>
      <containers>
        <org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
          <name>jnlp</name>
          <image>openshift/jenkins-agent-maven-35-centos7:v3.10</image>
          // Writer, remove or update this in 4.12
```

```
        <privileged>false</privileged>
        <alwaysPullImage>true</alwaysPullImage>
        <workingDir>/tmp</workingDir>
        <command></command>
        <args>${computer.jnlpmac} ${computer.name}</args>
        <ttyEnabled>false</ttyEnabled>
        <resourceRequestCpu></resourceRequestCpu>
        <resourceRequestMemory></resourceRequestMemory>
        <resourceLimitCpu></resourceLimitCpu>
        <resourceLimitMemory></resourceLimitMemory>
        <envVars/>
      </org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
    </containers>
    <envVars/>
    <annotations/>
    <imagePullSecrets/>
    <nodeProperties/>
  </org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
```

The following example shows two containers that reference image streams that are present in the **openshift** namespace. One container handles the JNLP contract for launching Pods as Jenkins Agents. The other container uses an image with tools for building code in a particular coding language:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: jenkins-agent
  labels:
    role: jenkins-agent
data:
  template2: |-
      <org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
        <inheritFrom></inheritFrom>
        <name>template2</name>
        <instanceCap>2147483647</instanceCap>
        <idleMinutes>0</idleMinutes>
        <label>template2</label>
        <serviceAccount>jenkins</serviceAccount>
        <nodeSelector></nodeSelector>
        <volumes/>
        <containers>
          <org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
            <name>jnlp</name>
            <image>image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-base-rhel8:latest</image>
            <privileged>false</privileged>
            <alwaysPullImage>true</alwaysPullImage>
            <workingDir>/home/jenkins/agent</workingDir>
            <command></command>
            <args>\$(JENKINS_SECRET) \$(JENKINS_NAME)</args>
            <ttyEnabled>false</ttyEnabled>
            <resourceRequestCpu></resourceRequestCpu>
            <resourceRequestMemory></resourceRequestMemory>
            <resourceLimitCpu></resourceLimitCpu>
            <resourceLimitMemory></resourceLimitMemory>
            <envVars/>
```

```
        </org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
        <org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
          <name>java</name>
          <image>image-registry.openshift-image-registry.svc:5000/openshift/java:latest</image>
          <privileged>false</privileged>
          <alwaysPullImage>true</alwaysPullImage>
          <workingDir>/home/jenkins/agent</workingDir>
          <command>cat</command>
          <args></args>
          <ttyEnabled>true</ttyEnabled>
          <resourceRequestCpu></resourceRequestCpu>
          <resourceRequestMemory></resourceRequestMemory>
          <resourceLimitCpu></resourceLimitCpu>
          <resourceLimitMemory></resourceLimitMemory>
          <envVars/>
        </org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
      </containers>
      <envVars/>
      <annotations/>
      <imagePullSecrets/>
      <nodeProperties/>
    </org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
```

> **NOTE**
>
> If you log in to the Jenkins console and make further changes to the pod template configuration after the pod template is created, and the OpenShift Container Platform Sync plugin detects that the config map has changed, it will replace the pod template and overwrite those configuration changes. You cannot merge a new configuration with the existing configuration.
>
> Do not log in to the Jenkins console and change the pod template configuration. If you do so after the pod template is created, and the OpenShift Container Platform Sync plugin detects that the image associated with the image stream or image stream tag has changed, it replaces the pod template and overwrites those configuration changes. You cannot merge a new configuration with the existing configuration.
>
> Consider the config map approach if you have more complex configuration needs.

After it is installed, the OpenShift Container Platform Sync plugin monitors the API server of OpenShift Container Platform for updates to image streams, image stream tags, and config maps and adjusts the configuration of the Kubernetes plugin.

The following rules apply:

- Removing the label or annotation from the config map, image stream, or image stream tag results in the deletion of any existing **PodTemplate** from the configuration of the Kubernetes plugin.

- If those objects are removed, the corresponding configuration is removed from the Kubernetes plugin.

- Either creating appropriately labeled or annotated **ConfigMap**, **ImageStream**, or **ImageStreamTag** objects, or the adding of labels after their initial creation, leads to creating of a **PodTemplate** in the Kubernetes–plugin configuration.

- In the case of the **PodTemplate** by config map form, changes to the config map data for the **PodTemplate** are applied to the **PodTemplate** settings in the Kubernetes plugin configuration and overrides any changes that were made to the **PodTemplate** through the Jenkins UI between changes to the config map.

To use a container image as a Jenkins agent, the image must run the agent as an entry point. For more details, see the official Jenkins documentation.

Additional resources

- Important changes to OpenShift Jenkins images

5.1.7. Jenkins permissions

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

Consider the following scenario with service accounts used for the pod, which is launched by the Kubernetes Plugin that runs in the OpenShift Container Platform Jenkins image.

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

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

- Any pod templates that are automatically discovered by the OpenShift Container Platform sync plugin because their image streams or image stream tags have the required label or annotations are configured to use the Jenkins master service account as their service account.

- For the other ways you can provide a pod template definition into Jenkins and the Kubernetes plugin, you have to explicitly specify the service account to use. Those other ways include the Jenkins console, the **podTemplate** pipeline DSL that is provided by the Kubernetes plugin, or labeling a config map whose data is the XML configuration for a pod template.

- If you do not specify a value for the service account, the **default** service account is used.

- Ensure that whatever service account is used has the necessary permissions, roles, and so on defined within OpenShift Container Platform to manipulate whatever projects you choose to manipulate from the within the pod.

5.1.8. Creating a Jenkins service from a template

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

The two available templates both define deployment configuration and a service. The templates differ in their storage strategy, which affects whether the Jenkins content persists across a pod restart.
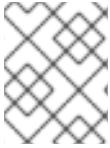
> **NOTE**
>
> A pod might be restarted when it is moved to another node or when an update of the deployment configuration triggers a redeployment.

- **jenkins-ephemeral** uses ephemeral storage. On pod restart, all data is lost. This template is only useful for development or testing.

- **jenkins-persistent** uses a Persistent Volume (PV) store. Data survives a pod restart.

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

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

Procedure

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

   - A PV:

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

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

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

With both templates, you can run **oc describe** on them to see all the parameters available for overriding.

For example:

```
$ oc describe jenkins-ephemeral
```

5.1.9. Using the Jenkins Kubernetes plugin

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

Sample BuildConfig that uses the Jenkins Kubernetes plugin

```
kind: List
apiVersion: v1
items:
- kind: ImageStream
  apiVersion: image.openshift.io/v1
  metadata:
    name: openshift-jee-sample
- kind: BuildConfig
  apiVersion: build.openshift.io/v1
  metadata:
    name: openshift-jee-sample-docker
  spec:
```

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

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

Sample BuildConfig that uses the Jenkins Kubernetes Plugin, specifying memory limit and environment variable

```
kind: BuildConfig
apiVersion: build.openshift.io/v1
metadata:
  name: openshift-jee-sample
spec:
  strategy:
    type: JenkinsPipeline
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        podTemplate(label: "mypod",                                      1
                    cloud: "openshift",                                  2
                    inheritFrom: "maven",                                3
                    containers: [
              containerTemplate(name: "jnlp",                            4
                        image: "openshift/jenkins-agent-maven-35-centos7:v3.10",   5
                        resourceRequestMemory: "512Mi",                  6
                        resourceLimitMemory: "512Mi",                    7
```

```
                envVars: [
            envVar(key: "CONTAINER_HEAP_PERCENT", value: "0.25") 8
          ])
        ]) {
        node("mypod") { 9
          sh "git clone https://github.com/openshift/openshift-jee-sample.git ."
          sh "mvn -B -Popenshift package"
          sh "oc start-build -F openshift-jee-sample-docker --from-file=target/ROOT.war"
        }
      }
  triggers:
  - type: ConfigChange
```

1 A new pod template called **mypod** is defined dynamically. The new pod template name is referenced in the node stanza.

2 The **cloud** value must be set to **openshift**.

3 The new pod template can inherit its configuration from an existing pod template. In this case, inherited from the Maven pod template that is pre-defined by OpenShift Container Platform.

4 This example overrides values in the pre-existing container, and must be specified by name. All Jenkins agent images shipped with OpenShift Container Platform use the Container name **jnlp**.

5 Specify the container image name again. This is a known issue.

6 A memory request of **512 Mi** is specified.

7 A memory limit of **512 Mi** is specified.

8 An environment variable **CONTAINER_HEAP_PERCENT**, with value **0.25**, is specified.

9 The node stanza references the name of the defined pod template.

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

Upstream Jenkins has more recently introduced a YAML declarative format for defining a **podTemplate** pipeline DSL in-line with your pipelines. An example of this format, using the sample **java-builder** pod template that is defined in the OpenShift Container Platform Jenkins image:

```
def nodeLabel = 'java-buidler'

pipeline {
  agent {
    kubernetes {
      cloud 'openshift'
      label nodeLabel
      yaml """
apiVersion: v1
kind: Pod
metadata:
  labels:
    worker: ${nodeLabel}
spec:
```

```
      containers:
      - name: jnlp
        image: image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-base-rhel8:latest
        args: ['\$(JENKINS_SECRET)', '\$(JENKINS_NAME)']
      - name: java
        image: image-registry.openshift-image-registry.svc:5000/openshift/java:latest
        command:
        - cat
        tty: true
"""
      }
    }

    options {
      timeout(time: 20, unit: 'MINUTES')
    }

    stages {
      stage('Build App') {
        steps {
          container("java") {
            sh "mvn --version"
          }
        }
      }
    }
  }
```

5.1.10. Jenkins memory requirements

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

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

And if **Project** quotas allow for it, see recommendations from the Jenkins documentation on what a Jenkins master should have from a memory perspective. Those recommendations proscribe to allocate even more memory for the Jenkins master.

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

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

5.1.11. Additional resources

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

5.2. JENKINS AGENT

OpenShift Container Platform provides Base, Maven, and Node.js images for use as Jenkins agents.

The Base image for Jenkins agents does the following:

- Pulls in both the required tools, headless Java, the Jenkins JNLP client, and the useful ones, including **git**, **tar**, **zip**, and **nss**, among others.

- Establishes the JNLP agent as the entry point.

- Includes the **oc** client tooling for invoking command line operations from within Jenkins jobs.

- Provides Dockerfiles for both Red Hat Enterprise Linux (RHEL) and **localdev** images.

The Maven v3.5, Node.js v10, and Node.js v12 images extend the Base image. They provide Dockerfiles for the Universal Base Image (UBI) that you can reference when building new agent images. Also note the **contrib** and **contrib/bin** subdirectories, which enable you to insert configuration files and executable scripts for your image.

> **IMPORTANT**
>
> Use a version of the agent image that is appropriate for your OpenShift Container Platform release version. Embedding an **oc** client version that is not compatible with the OpenShift Container Platform version can cause unexpected behavior.

The OpenShift Container Platform Jenkins image also defines the following sample pod templates to illustrate how you can use these agent images with the Jenkins Kubernetes plugin:

- The **maven** pod template, which uses a single container that uses the OpenShift Container Platform Maven Jenkins agent image.

- The **nodejs** pod template, which uses a single container that uses the OpenShift Container Platform Node.js Jenkins agent image.

- The **java-builder** pod template, which employs two containers. One is the **jnlp** container, which uses the OpenShift Container Platform Base agent image and handles the JNLP contract for starting and stopping Jenkins agents. The second is the **java** container which uses the **java** OpenShift Container Platform Sample ImageStream, which contains the various Java binaries, including the Maven binary **mvn**, for building code.

- The **nodejs-builder** pod template, which employs two containers. One is the **jnlp** container, which uses the OpenShift Container Platform Base agent image and handles the JNLP contract for starting and stopping Jenkins agents. The second is the **nodejs** container which uses the **nodejs** OpenShift Container Platform Sample ImageStream, which contains the various Node.js binaries, including the **npm** binary, for building code.

5.2.1. Jenkins agent images

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

Jenkins images are available through the Red Hat Registry:

```
$ docker pull registry.redhat.io/ocp-tools-4/jenkins-rhel8:<image_tag>
```

```
$ docker pull registry.redhat.io/ocp-tools-4/jenkins-agent-base-rhel8:<image_tag>
```

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

5.2.2. Jenkins agent environment variables

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

Variable	Definition	Example values and settings
JAVA_MAX_HEAP_PARAM, **CONTAINER_HEAP_PERCENT**, **JENKINS_MAX_HEAP_UPPER_BOUND_MB**	These values control the maximum heap size of the Jenkins JVM. If **JAVA_MAX_HEAP_PARAM** is set, its value takes precedence. Otherwise, the maximum heap size is dynamically calculated as **CONTAINER_HEAP_PERCENT** of the container memory limit, optionally capped at **JENKINS_MAX_HEAP_UPPER_BOUND_MB** MiB. By default, the maximum heap size of the Jenkins JVM is set to 50% of the container memory limit with no cap.	**JAVA_MAX_HEAP_PARAM** example setting: **-Xmx512m** **CONTAINER_HEAP_PERCENT** default: **0.5**, or 50% **JENKINS_MAX_HEAP_UPPER_BOUND_MB** example setting: **512 MiB**
JAVA_INITIAL_HEAP_PARAM, **CONTAINER_INITIAL_PERCENT**	These values control the initial heap size of the Jenkins JVM. If **JAVA_INITIAL_HEAP_PARAM** is set, its value takes precedence. Otherwise, the initial heap size is dynamically calculated as **CONTAINER_INITIAL_PERCENT** of the dynamically calculated maximum heap size. By default, the JVM sets the initial heap size.	**JAVA_INITIAL_HEAP_PARAM** example setting: **-Xms32m** **CONTAINER_INITIAL_PERCENT** example setting: **0.1**, or 10%
CONTAINER_CORE_LIMIT	If set, specifies an integer number of cores used for sizing numbers of internal JVM threads.	Example setting: **2**
JAVA_TOOL_OPTIONS	Specifies options to apply to all JVMs running in this container. It is not recommended to override this value.	Default: **-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap -Dsun.zip.disableMemoryMapping=true**

Variable	Definition	Example values and settings
JAVA_GC_OPTS	Specifies Jenkins JVM garbage collection parameters. It is not recommended to override this value.	Default: **-XX:+UseParallelGC -XX:MinHeapFreeRatio=5 -XX:MaxHeapFreeRatio=10 -XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyWeight=90**
JENKINS_JAVA_OVERRIDES	Specifies additional options for the Jenkins JVM. These options are appended to all other options, including the Java options above, and can be used to override any of them, if necessary. Separate each additional option with a space and if any option contains space characters, escape them with a backslash.	Example settings: **-Dfoo -Dbar**; **-Dfoo=first\ value -Dbar=second\ value**
USE_JAVA_VERSION	Specifies the version of Java version to use to run the agent in its container. The container base image has two versions of java installed: **java-11** and **java-1.8.0**. If you extend the container base image, you can specify any alternative version of java using its associated suffix.	The default value is **java-11**. Example setting: **java-1.8.0**

5.2.3. Jenkins agent memory requirements

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

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

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

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

5.2.4. Jenkins agent Gradle builds

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

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

- Ensure the long-lived Gradle daemon is disabled by adding **org.gradle.daemon=false** to the **gradle.properties** file.

- Disable parallel build execution by ensuring **org.gradle.parallel=true** is not set in the **gradle.properties** file and that **--parallel** is not set as a command line argument.

- To prevent Java compilations running out-of-process, set **java { options.fork = false }** in the **build.gradle** file.

- Disable multiple additional test processes by ensuring **test { maxParallelForks = 1 }** is set in the **build.gradle** file.

- Override the Gradle JVM memory parameters by the **GRADLE_OPTS**, **JAVA_OPTS** or **JAVA_TOOL_OPTIONS** environment variables.

- Set the maximum heap size and JVM arguments for any Gradle test JVM by defining the **maxHeapSize** and **jvmArgs** settings in **build.gradle**, or through the **-Dorg.gradle.jvmargs** command line argument.

5.2.5. Jenkins agent pod retention

Jenkins agent pods, are deleted by default after the build completes or is stopped. This behavior can be changed by the Kubernetes plugin pod retention setting. Pod retention can be set for all Jenkins builds, with overrides for each pod template. The following behaviors are supported:

- **Always** keeps the build pod regardless of build result.

- **Default** uses the plugin value, which is the pod template only.

- **Never** always deletes the pod.

- **On Failure** keeps the pod if it fails during the build.

You can override pod retention in the pipeline Jenkinsfile:

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

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

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

5.3. MIGRATING FROM JENKINS TO OPENSHIFT PIPELINES OR TEKTON

You can migrate your CI/CD workflows from Jenkins to Red Hat OpenShift Pipelines, a cloud-native CI/CD experience based on the Tekton project.

5.3.1. Comparison of Jenkins and OpenShift Pipelines concepts

You can review and compare the following equivalent terms used in Jenkins and OpenShift Pipelines.

5.3.1.1. Jenkins terminology

Jenkins offers declarative and scripted pipelines that are extensible using shared libraries and plugins. Some basic terms in Jenkins are as follows:

- **Pipeline**: Automates the entire process of building, testing, and deploying applications by using Groovy syntax.

- **Node**: A machine capable of either orchestrating or executing a scripted pipeline.

- **Stage**: A conceptually distinct subset of tasks performed in a pipeline. Plugins or user interfaces often use this block to display the status or progress of tasks.

- **Step**: A single task that specifies the exact action to be taken, either by using a command or a script.

5.3.1.2. OpenShift Pipelines terminology

OpenShift Pipelines uses YAML syntax for declarative pipelines and consists of tasks. Some basic terms in OpenShift Pipelines are as follows:

- **Pipeline**: A set of tasks in a series, in parallel, or both.

- **Task**: A sequence of steps as commands, binaries, or scripts.

- **PipelineRun**: Execution of a pipeline with one or more tasks.

- **TaskRun**: Execution of a task with one or more steps.

> **NOTE**
>
> You can initiate a PipelineRun or a TaskRun with a set of inputs such as parameters and workspaces, and the execution results in a set of outputs and artifacts.

- **Workspace**: In OpenShift Pipelines, workspaces are conceptual blocks that serve the following purposes:

  - Storage of inputs, outputs, and build artifacts.

  - Common space to share data among tasks.

  - Mount points for credentials held in secrets, configurations held in config maps, and common tools shared by an organization.

> **NOTE**
>
> In Jenkins, there is no direct equivalent of OpenShift Pipelines workspaces. You can think of the control node as a workspace, as it stores the cloned code repository, build history, and artifacts. When a job is assigned to a different node, the cloned code and the generated artifacts are stored in that node, but the control node maintains the build history.

5.3.1.3. Mapping of concepts

The building blocks of Jenkins and OpenShift Pipelines are not equivalent, and a specific comparison does not provide a technically accurate mapping. The following terms and concepts in Jenkins and OpenShift Pipelines correlate in general:

Table 5.1. Jenkins and OpenShift Pipelines – basic comparison

Jenkins	OpenShift Pipelines
Pipeline	Pipeline and PipelineRun
Stage	Task
Step	A step in a task

5.3.2. Migrating a sample pipeline from Jenkins to OpenShift Pipelines

You can use the following equivalent examples to help migrate your build, test, and deploy pipelines from Jenkins to OpenShift Pipelines.

5.3.2.1. Jenkins pipeline

Consider a Jenkins pipeline written in Groovy for building, testing, and deploying:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'make'
            }
        }
        stage('Test'){
            steps {
```

```
            sh 'make check'
            junit 'reports/**/*.xml'
        }
    }
    stage('Deploy') {
        steps {
            sh 'make publish'
        }
    }
}
}
```

5.3.2.2. OpenShift Pipelines pipeline

To create a pipeline in OpenShift Pipelines that is equivalent to the preceding Jenkins pipeline, you create the following three tasks:

Example build task YAML definition file

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: myproject-build
spec:
  workspaces:
  - name: source
  steps:
  - image: my-ci-image
    command: ["make"]
    workingDir: $(workspaces.source.path)
```

Example test task YAML definition file

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: myproject-test
spec:
  workspaces:
  - name: source
  steps:
  - image: my-ci-image
    command: ["make check"]
    workingDir: $(workspaces.source.path)
  - image: junit-report-image
    script: |
      #!/usr/bin/env bash
      junit-report reports/**/*.xml
    workingDir: $(workspaces.source.path)
```

Example deploy task YAML definition file

```
apiVersion: tekton.dev/v1beta1
kind: Task
```

```
metadata:
  name: myprojectd-deploy
spec:
  workspaces:
  - name: source
  steps:
  - image: my-deploy-image
    command: ["make deploy"]
    workingDir: $(workspaces.source.path)
```

You can combine the three tasks sequentially to form a pipeline in OpenShift Pipelines:

Example: OpenShift Pipelines pipeline for building, testing, and deployment

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: myproject-pipeline
spec:
  workspaces:
  - name: shared-dir
  tasks:
  - name: build
    taskRef:
      name: myproject-build
    workspaces:
    - name: source
      workspace: shared-dir
  - name: test
    taskRef:
      name: myproject-test
    workspaces:
    - name: source
      workspace: shared-dir
  - name: deploy
    taskRef:
      name: myproject-deploy
    workspaces:
    - name: source
      workspace: shared-dir
```

5.3.3. Migrating from Jenkins plugins to Tekton Hub tasks

You can extend the capability of Jenkins by using plugins. To achieve similar extensibility in OpenShift Pipelines, use any of the tasks available from Tekton Hub.

For example, consider the git-clone task in Tekton Hub, which corresponds to the git plugin for Jenkins.

Example: git-clone task from Tekton Hub

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
 name: demo-pipeline
spec:
```

```
  params:
    - name: repo_url
    - name: revision
  workspaces:
    - name: source
  tasks:
    - name: fetch-from-git
      taskRef:
        name: git-clone
      params:
        - name: url
          value: $(params.repo_url)
        - name: revision
          value: $(params.revision)
      workspaces:
      - name: output
        workspace: source
```

5.3.4. Extending OpenShift Pipelines capabilities using custom tasks and scripts

In OpenShift Pipelines, if you do not find the right task in Tekton Hub, or need greater control over tasks, you can create custom tasks and scripts to extend the capabilities of OpenShift Pipelines.

Example: A custom task for running the maven test command

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: maven-test
spec:
  workspaces:
  - name: source
  steps:
  - image: my-maven-image
    command: ["mvn test"]
    workingDir: $(workspaces.source.path)
```

Example: Run a custom shell script by providing its path

```
...
steps:
  image: ubuntu
  script: |
      #!/usr/bin/env bash
      /workspace/my-script.sh
...
```

Example: Run a custom Python script by writing it in the YAML file

```
...
steps:
  image: python
  script: |
```

```
#!/usr/bin/env python3
print("hello from python!")
...
```

5.3.5. Comparison of Jenkins and OpenShift Pipelines execution models

Jenkins and OpenShift Pipelines offer similar functions but are different in architecture and execution.

Table 5.2. Comparison of execution models in Jenkins and OpenShift Pipelines

Jenkins	OpenShift Pipelines
Jenkins has a controller node. Jenkins runs pipelines and steps centrally, or orchestrates jobs running in other nodes.	OpenShift Pipelines is serverless and distributed, and there is no central dependency for execution.
Containers are launched by the Jenkins controller node through the pipeline.	OpenShift Pipelines adopts a 'container-first' approach, where every step runs as a container in a pod (equivalent to nodes in Jenkins).
Extensibility is achieved by using plugins.	Extensibility is achieved by using tasks in Tekton Hub or by creating custom tasks and scripts.

5.3.6. Examples of common use cases

Both Jenkins and OpenShift Pipelines offer capabilities for common CI/CD use cases, such as:

- Compiling, building, and deploying images using Apache Maven

- Extending the core capabilities by using plugins

- Reusing shareable libraries and custom scripts

5.3.6.1. Running a Maven pipeline in Jenkins and OpenShift Pipelines

You can use Maven in both Jenkins and OpenShift Pipelines workflows for compiling, building, and deploying images. To map your existing Jenkins workflow to OpenShift Pipelines, consider the following examples:

Example: Compile and build an image and deploy it to OpenShift using Maven in Jenkins

```
#!/usr/bin/groovy
node('maven') {
    stage 'Checkout'
    checkout scm

    stage 'Build'
    sh 'cd helloworld && mvn clean'
    sh 'cd helloworld && mvn compile'

    stage 'Run Unit Tests'
    sh 'cd helloworld && mvn test'
```

```
stage 'Package'
sh 'cd helloworld && mvn package'

stage 'Archive artifact'
sh 'mkdir -p artifacts/deployments && cp helloworld/target/*.war artifacts/deployments'
archive 'helloworld/target/*.war'

stage 'Create Image'
sh 'oc login https://kubernetes.default -u admin -p admin --insecure-skip-tls-verify=true'
sh 'oc new-project helloworldproject'
sh 'oc project helloworldproject'
sh 'oc process -f helloworld/jboss-eap70-binary-build.json | oc create -f -'
sh 'oc start-build eap-helloworld-app --from-dir=artifacts/'

stage 'Deploy'
sh 'oc new-app helloworld/jboss-eap70-deploy.json' }
```

Example: Compile and build an image and deploy it to OpenShift using Maven in OpenShift Pipelines.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: maven-pipeline
spec:
 workspaces:
  - name: shared-workspace
  - name: maven-settings
  - name: kubeconfig-dir
    optional: true
 params:
  - name: repo-url
  - name: revision
  - name: context-path
 tasks:
  - name: fetch-repo
    taskRef:
      name: git-clone
    workspaces:
     - name: output
       workspace: shared-workspace
    params:
     - name: url
       value: "$(params.repo-url)"
     - name: subdirectory
       value: ""
     - name: deleteExisting
       value: "true"
     - name: revision
       value: $(params.revision)
  - name: mvn-build
    taskRef:
      name: maven
    runAfter:
     - fetch-repo
```

```
  workspaces:
    - name: source
      workspace: shared-workspace
    - name: maven-settings
      workspace: maven-settings
  params:
    - name: CONTEXT_DIR
      value: "$(params.context-path)"
    - name: GOALS
      value: ["-DskipTests", "clean", "compile"]
- name: mvn-tests
  taskRef:
    name: maven
  runAfter:
    - mvn-build
  workspaces:
    - name: source
      workspace: shared-workspace
    - name: maven-settings
      workspace: maven-settings
  params:
    - name: CONTEXT_DIR
      value: "$(params.context-path)"
    - name: GOALS
      value: ["test"]
- name: mvn-package
  taskRef:
    name: maven
  runAfter:
    - mvn-tests
  workspaces:
    - name: source
      workspace: shared-workspace
    - name: maven-settings
      workspace: maven-settings
  params:
    - name: CONTEXT_DIR
      value: "$(params.context-path)"
    - name: GOALS
      value: ["package"]
- name: create-image-and-deploy
  taskRef:
    name: openshift-client
  runAfter:
    - mvn-package
  workspaces:
    - name: manifest-dir
      workspace: shared-workspace
    - name: kubeconfig-dir
      workspace: kubeconfig-dir
  params:
    - name: SCRIPT
      value: |
        cd "$(params.context-path)"
        mkdir -p ./artifacts/deployments && cp ./target/*.war ./artifacts/deployments
        oc new-project helloworldproject
```

```
oc project helloworldproject
oc process -f jboss-eap70-binary-build.json | oc create -f -
oc start-build eap-helloworld-app --from-dir=artifacts/
oc new-app jboss-eap70-deploy.json
```

5.3.6.2. Extending the core capabilities of Jenkins and OpenShift Pipelines by using plugins

Jenkins has the advantage of a large ecosystem of numerous plugins developed over the years by its extensive user base. You can search and browse the plugins in the Jenkins Plugin Index .

OpenShift Pipelines also has many tasks developed and contributed by the community and enterprise users. A publicly available catalog of reusable OpenShift Pipelines tasks are available in the Tekton Hub.

In addition, OpenShift Pipelines incorporates many of the plugins of the Jenkins ecosystem within its core capabilities. For example, authorization is a critical function in both Jenkins and OpenShift Pipelines. While Jenkins ensures authorization using the Role-based Authorization Strategy plugin, OpenShift Pipelines uses OpenShift's built-in Role-based Access Control system.

5.3.6.3. Sharing reusable code in Jenkins and OpenShift Pipelines

Jenkins shared libraries provide reusable code for parts of Jenkins pipelines. The libraries are shared between Jenkinsfiles to create highly modular pipelines without code repetition.

Although there is no direct equivalent of Jenkins shared libraries in OpenShift Pipelines, you can achieve similar workflows by using tasks from the Tekton Hub in combination with custom tasks and scripts.

5.3.7. Additional resources

- Understanding OpenShift Pipelines

- Role-based Access Control

5.4. IMPORTANT CHANGES TO OPENSHIFT JENKINS IMAGES

OpenShift Container Platform 4.11 moves the OpenShift Jenkins and OpenShift Agent Base images to the **ocp-tools-4** repository at **registry.redhat.io**. It also removes the OpenShift Jenkins Maven and NodeJS Agent images from its payload:

- OpenShift Container Platform 4.11 moves the OpenShift Jenkins and OpenShift Agent Base images to the **ocp-tools-4** repository at **registry.redhat.io** so that Red Hat can produce and update the images outside the OpenShift Container Platform lifecycle. Previously, these images were in the OpenShift Container Platform install payload and the **openshift4** repository at **registry.redhat.io**.

- OpenShift Container Platform 4.10 deprecated the OpenShift Jenkins Maven and NodeJS Agent images. OpenShift Container Platform 4.11 removes these images from its payload. Red Hat no longer produces these images, and they are not available from the **ocp-tools-4** repository at **registry.redhat.io**. Red Hat maintains the 4.10 and earlier versions of these images for any significant bug fixes or security CVEs, following the OpenShift Container Platform lifecycle policy.

These changes support the OpenShift Container Platform 4.10 recommendation to use multiple container Pod Templates with the Jenkins Kubernetes Plugin.

5.4.1. Relocation of OpenShift Jenkins images

OpenShift Container Platform 4.11 makes significant changes to the location and availability of specific OpenShift Jenkins images. Additionally, you can configure when and how to update these images.

What stays the same with the OpenShift Jenkins images?

- The Cluster Samples Operator manages the **ImageStream** and **Template** objects for operating the OpenShift Jenkins images.

- By default, the Jenkins **DeploymentConfig** object from the Jenkins pod template triggers a redeployment when the Jenkins image changes. By default, this image is referenced by the **jenkins:2** image stream tag of Jenkins image stream in the **openshift** namespace in the **ImageStream** YAML file in the Samples Operator payload.

- If you upgrade from OpenShift Container Platform 4.10 and earlier to 4.11, the deprecated **maven** and **nodejs** pod templates are still in the default image configuration.

- If you upgrade from OpenShift Container Platform 4.10 and earlier to 4.11, the **jenkins-agent-maven** and **jenkins-agent-nodejs** image streams still exist in your cluster. To maintain these image streams, see the following section, "What happens with the **jenkins-agent-maven** and **jenkins-agent-nodejs** image streams in the **openshift** namespace?"

What changes in the support matrix of the OpenShift Jenkins image?

Each new image in the **ocp-tools-4** repository in the **registry.redhat.io** registry supports multiple versions of OpenShift Container Platform. When Red Hat updates one of these new images, it is simultaneously available for all versions. This availability is ideal when Red Hat updates an image in response to a security advisory. Initially, this change applies to OpenShift Container Platform 4.11 and later. It is planned that this change will eventually apply to OpenShift Container Platform 4.9 and later.

Previously, each Jenkins image supported only one version of OpenShift Container Platform and Red Hat might update those images sequentially over time.

What additions are there with the OpenShift Jenkins and Jenkins Agent Base ImageStream and ImageStreamTag objects?

By moving from an in-payload image stream to an image stream that references non-payload images, OpenShift Container Platform can define additional image stream tags. Red Hat has created a series of new image stream tags to go along with the existing **"value": "jenkins:2"** and **"value": "image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-base-rhel8:latest"** image stream tags present in OpenShift Container Platform 4.10 and earlier. These new image stream tags address some requests to improve how the Jenkins-related image streams are maintained.

About the new image stream tags:

ocp-upgrade-redeploy

To update your Jenkins image when you upgrade OpenShift Container Platform, use this image stream tag in your Jenkins deployment configuration. This image stream tag corresponds to the existing **2** image stream tag of the **jenkins** image stream and the **latest** image stream tag of the **jenkins-agent-base-rhel8** image stream. It employs an image tag specific to only one SHA or image digest. When the **ocp-tools-4** image changes, such as for Jenkins security advisories, Red Hat Engineering updates the Cluster Samples Operator payload.

user-maintained-upgrade-redeploy

To manually redeploy Jenkins after you upgrade OpenShift Container Platform, use this image stream tag in your Jenkins deployment configuration. This image stream tag uses the least specific

image version indicator available. When you redeploy Jenkins, run the following command: **$ oc import-image jenkins:user-maintained-upgrade-redeploy -n openshift**. When you issue this command, the OpenShift Container Platform **ImageStream** controller accesses the **registry.redhat.io** image registry and stores any updated images in the OpenShift image registry's slot for that Jenkins **ImageStreamTag** object. Otherwise, if you do not run this command, your Jenkins deployment configuration does not trigger a redeployment.

scheduled-upgrade-redeploy

To automatically redeploy the latest version of the Jenkins image when it is released, use this image stream tag in your Jenkins deployment configuration. This image stream tag uses the periodic importing of image stream tags feature of the OpenShift Container Platform image stream controller, which checks for changes in the backing image. If the image changes, for example, due to a recent Jenkins security advisory, OpenShift Container Platform triggers a redeployment of your Jenkins deployment configuration. See "Configuring periodic importing of image stream tags" in the following "Additional resources."

What happens with the jenkins-agent-maven and jenkins-agent-nodejs image streams in the openshift namespace?

The OpenShift Jenkins Maven and NodeJS Agent images for OpenShift Container Platform were deprecated in 4.10, and are removed from the OpenShift Container Platform install payload in 4.11. They do not have alternatives defined in the **ocp-tools-4** repository. However, you can work around this by using the sidecar pattern described in the "Jenkins agent" topic mentioned in the following "Additional resources" section.

However, the Cluster Samples Operator does not delete the **jenkins-agent-maven** and **jenkins-agent-nodejs** image streams created by prior releases, which point to the tags of the respective OpenShift Container Platform payload images on **registry.redhat.io**. Therefore, you can pull updates to these images by running the following commands:

```
$ oc import-image jenkins-agent-nodejs -n openshift
```

```
$ oc import-image jenkins-agent-maven -n openshift
```

5.4.2. Customizing the Jenkins image stream tag

To override the default upgrade behavior and control how the Jenkins image is upgraded, you set the image stream tag value that your Jenkins deployment configurations use.

The default upgrade behavior is the behavior that existed when the Jenkins image was part of the install payload. The image stream tag names, **2** and **ocp-upgrade-redeploy**, in the **jenkins-rhel.json** image stream file use SHA–specific image references. Therefore, when those tags are updated with a new SHA, the OpenShift Container Platform image change controller automatically redeploys the Jenkins deployment configuration from the associated templates, such as **jenkins-ephemeral.json** or **jenkins-persistent.json**.

For new deployments, to override that default value, you change the value of the **JENKINS_IMAGE_STREAM_TAG** in the **jenkins-ephemeral.json** Jenkins template. For example, replace the **2** in **"value": "jenkins:2"** with one of the following image stream tags:

- **ocp-upgrade-redeploy**, the default value, updates your Jenkins image when you upgrade OpenShift Container Platform.

- **user-maintained-upgrade-redeploy** requires you to manually redeploy Jenkins by running **$ oc import-image jenkins:user-maintained-upgrade-redeploy -n openshift** after upgrading OpenShift Container Platform.

- **scheduled-upgrade-redeploy** periodically checks the given **<image>:<tag>** combination for changes and upgrades the image when it changes. The image change controller pulls the changed image and redeploys the Jenkins deployment configuration provisioned by the templates. For more information about this scheduled import policy, see the "Adding tags to image streams" in the following "Additional resources."

> **NOTE**
>
> To override the current upgrade value for existing deployments, change the values of the environment variables that correspond to those template parameters.

Prerequisites

- You are running OpenShift Jenkins on OpenShift Container Platform 4.11.

- You know the namespace where OpenShift Jenkins is deployed.

Procedure

- Set the image stream tag value, replacing **<namespace>** with namespace where OpenShift Jenkins is deployed and **<image_stream_tag>** with an image stream tag:

  Example

  ```
  $ oc patch dc jenkins -p '{"spec":{"triggers":[{"type":"ImageChange","imageChangeParams":
  {"automatic":true,"containerNames":["jenkins"],"from":
  {"kind":"ImageStreamTag","namespace":"<namespace>","name":"jenkins:
  <image_stream_tag>"}}}]}}'
  ```

  TIP

  Alternatively, to edit the Jenkins deployment configuration YAML, enter **$ oc edit dc/jenkins -n <namespace>** and update the **value: 'jenkins:<image_stream_tag>'** line.

5.4.3. Additional resources

- [Adding tags to image streams](#)

- [Configuring periodic importing of image stream tags](#)

- [Jenkins agent](#)

- [Certified **jenkins** images](#)

- [Certified **jenkins-agent-base** images](#)

- [Certified **jenkins-agent-maven** images](#)

- [Certified **jenkins-agent-nodejs** images](#)