OpenShift Container Platform 4.7

CI/CD

Contains information on builds, pipelines and GitOps for OpenShift Container Platform

Last Updated: 2021-05-21

OpenShift Container Platform 4.7 CI/CD

Contains information on builds, pipelines and GitOps for OpenShift Container Platform

Legal Notice

Copyright © 2021 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

CI/CD for the OpenShift Container Platform

Table of Contents

CHAPTER 1. BUILDS

1.1. UNDERSTANDING IMAGE BUILDS

1.1.1. 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

1.1.1.1. Docker build

The docker build strategy invokes the docker build command, and it expects a repository with a Dockerfile and all required artifacts in it to produce a runnable image.

1.1.1.2. 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.

1.1.1.3. 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.

1.1.1.4. 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 plug-in. 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.

1.2. UNDERSTANDING BUILD CONFIGURATIONS

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

1.2.1. 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. Refer to the documentation for each source type for details.
5. The **strategy** section describes the build strategy used to execute the build. You can specify 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.

1.3. 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.

1.3.1. 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)
  images:
  - from:
      kind: ImageStreamTag
      name: myinputimage:latest
      namespace: mynamespace
    paths:
```

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

(1) The repository to be cloned into the working directory for the build.

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

(3) The working directory for the build becomes **<original_workingdir>/app/dir**.

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

1.3.2. 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" (1)
```

(1) 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.

1.3.3. 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
  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.

Optionally, if an input image 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.

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

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

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

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 **master**).

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.

1.3.4.1. 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"
    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 plug-in for Jenkins, any Git operations through the Git plug-in do not leverage the HTTP or HTTPS proxy defined in the **BuildConfig**. The Git plug-in 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.

1.3.4.2. 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.

1.3.4.2.1. 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/*'
```

1.3.4.2.2. 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: "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
  ```

1.3.4.2.3. 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
```

1.3.4.2.4. 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.

1.3.4.2.5. 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
```

1.3.4.2.6. 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> \
    --type=kubernetes.io/ssh-auth
```

1.3.4.2.7. 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> 1
      ```

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

1.3.4.2.8. Source secret combinations

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

1.3.4.2.8.1. 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
  ```

1.3.4.2.8.2. 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>
```

1.3.4.2.8.3. 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
```

1.3.4.2.8.4. 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
```

1.3.4.2.8.5. 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
```

1.3.5. 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.

1.3.6. Input secrets and config maps

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.

1.3.6.1. Adding input secrets and config maps

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.

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.

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

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

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

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.

1.3.6.2. 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.

1.3.6.3. 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"]
```

NOTE

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.

1.3.6.4. 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.

1.3.7. 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)

1.3.8. 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.

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:
  https://index.docker.io/v1/: 1
    auth: "YWRfbGzhcGU6R2labnRib21ifTE=" 2
    email: "user@example.com" 3
```

1 URL of the registry.

2 Encrypted password.

3 Email address for the login.

You can define multiple container image registry entries in this file. Alternatively, you can also add authentication entries to this file by running the **docker login** command. The file will be created if it does not exist.

Kubernetes provides **Secret** objects, which 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.

1.3.9. 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.

1.3.9.1. 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
```

1.3.9.2. Using secrets as environment variables

You can make key values from secrets available as environment variables using the **valueFrom** syntax.

Procedure

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

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: MYVAL
        valueFrom:
          secretKeyRef:
            key: myval
            name: mysecret
```

1.3.10. 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 plug-in 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: dmFsdWUtMQ0K 3
  password: dmFsdWUtMg0KDQo=
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.

1.3.10.1. 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.

1.3.10.2. 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.

1.3.10.3. 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, then the version of the secret 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 a old **resourceVersion**. In the interim, do not update the data of existing secrets, but create new ones with distinct names.

1.3.10.4. 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: dXNlci1uYW1l
    password: cGFzc3dvcmQ=
  ```

  (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:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cg
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

1.3.10.4.1. 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: dmFsdWUtMQ0K     (1)
    password: dmFsdWUtMQ0KDQo= (2)
  stringData:
    hostname: myapp.mydomain.com (3)
    secret.properties: |-     (4)
      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: 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
```

1.3.11. 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.alpha.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.alpha.openshift.io/expiry** annotation on the secret, which is in RFC3339 format.

NOTE

In most cases, the service DNS name **<service.name>.<service.namespace>.svc** is not externally routable. The primary use of **<service.name>.<service.namespace>.svc** is for intracluster or intraservice communication, and with re-encrypt routes.

Other pods can trust cluster-created certificates, which are only signed for internal DNS names, by using the 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.

1.3.12. 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.

1.4. MANAGING BUILD OUTPUT

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

1.4.1. 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 Container Platform 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"
```

1.4.2. 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.

1.4.3. 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"
```

1.5. USING BUILD STRATEGIES

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

1.5.1. Docker build

The docker build strategy invokes the docker build command, and it expects a repository with a Dockerfile and all required artifacts in it to produce a runnable image.

1.5.1.1. 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"
```

1.5.1.2. 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
```

1.5.1.3. 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.

1.5.1.4. 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.

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"
```

1.5.1.5. Squash 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
  ```

1.5.2. 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.

1.5.2.1. 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.

1.5.2.2. 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.

1.5.2.3. 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.

1.5.2.3.1. 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.

1.5.2.3.2. 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.

1.5.2.4. 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.

1.5.2.5. 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.

1.5.2.5.1. 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.

1.5.2.5.2. 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 1.1. S2I scripts

Script	Description

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.
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:

```
#!/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:

```
#!/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

1.5.3. 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.

1.5.3.1. 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:

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

1.5.3.2. 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:

```
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.

1.5.3.3. 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:

   ```
   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>
   ```

1.5.3.4. 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.

1.5.3.4.1. Custom builder image

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

Table 1.2. 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.

Variable Name	Description
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.
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.

1.5.3.4.2. 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.

1.5.4. 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 plug-in. 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.

1.5.4.1. 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 Plug-in, 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.
- Injecting of environment variables.
- Pipeline visualization in the OpenShift Container Platform web console.
- Integration with the Jenkins git plugin, which passes commit information from
- Synchronizing secrets into Jenkins credential entries OpenShift Container Platform builds to the Jenkins git plugin.

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 Plug-in 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 plug-in 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**.

1.5.4.2. 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**.

1.5.4.3. 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.

1.5.4.3.1. 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.

1.5.4.4. 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 (1)
   ```

   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. Once 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' (1)
   def templateName = 'nodejs-mongodb-example' (2)
   pipeline {
     agent {
       node {
         label 'nodejs' (3)
   ```

```
 }
}
options {
 timeout(time: 20, unit: 'MINUTES') 4
}
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.

1.5.5. 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 the **Save** button.

1.5.6. 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.

1.6. CUSTOM IMAGE BUILDS WITH BUILDAH

With OpenShift Container Platform 4.7, 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.

1.6.1. Prerequisites

- Review how to grant custom build permissions.

1.6.2. 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}
```

1.6.3. 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**.

1.6.4. 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:

```
kind: BuildConfig
apiVersion: 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: 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**.

1.7. PERFORMING BASIC BUILDS

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

1.7.1. 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>
```

1.7.1.1. 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>
  ```

1.7.1.2. 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
  ```

1.7.1.3. 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>
  ```

1.7.1.4. 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.

Option	Description
--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
```

1.7.2. 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>
```

1.7.2.1. 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>
```

1.7.2.2. 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>
```

1.7.2.3. 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>
```

1.7.3. 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>
```

1.7.4. 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>
```

1.7.5. 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>
```

1.7.5.1. 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>
```

1.7.5.2. 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>
```

1.7.5.3. 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.

1.8. TRIGGERING AND MODIFYING BUILDS

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

1.8.1. 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

1.8.1.1. 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 master 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
```

1.8.1.1.1. 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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<s
ecret>/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/oapi/v1/namespaces/nsname/buildconfigs/bcname/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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/web
hooks/<secret>/github
```

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

Additional resources

- Gogs

1.8.1.1.2. 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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<s
ecret>/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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/web
   hooks/<secret>/gitlab
   ```
   The **-k** argument is only necessary if your API server does not have a properly signed certificate.

1.8.1.1.3. Using Bitbucket webhooks

Additional resources

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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<s
ecret>/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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/web
    hooks/<secret>/bitbucket
    ```

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

1.8.1.1.4. 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 1
```

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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/web
   hooks/<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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/web
   hooks/<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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/web
   hooks/<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.

1.8.1.1.5. 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>
```

1.8.1.2. Using image change triggers

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

> 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

Configuring an image change trigger requires the following actions:

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

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

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

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

1.8.1.3. Configuration change triggers

A configuration change trigger allows a build to be automatically invoked as soon as a new **BuildConfig** is created.

The following is an example trigger definition YAML within the **BuildConfig**:

```
type: "ConfigChange"
```

NOTE

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

1.8.1.3.1. 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
```

1.8.2. 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.

1.8.2.1. 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.

1.8.2.2. 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"
```

1.9. 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.

1.9.1. 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" 1
      memory: "256Mi" 2
```

1. **cpu** is in CPU units: **100m** represents 0.1 CPU units (100 * 1e-3).
2. **memory** is in bytes: **256Mi** represents 268435456 bytes (256 * 2 ^ 20).

However, if a quota has been defined for your project, one of the following two items is required:

- A **resources** section set with an explicit **requests**:

  ```
  resources:
    requests: 1
      cpu: "100m"
      memory: "256Mi"
  ```

  1. The **requests** object contains the list of resources that correspond to the list of resources in the quota.

- A limit range defined in your project, where the defaults from the **LimitRange** object apply to pods created during the build process.
  Otherwise, build pod creation will fail, citing a failure to satisfy quota.

1.9.2. 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.

1.9.3. 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: 1
    key1: value1
    key2: value2
```

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

1.9.4. 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: 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
  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: 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: 1
        kind: ImageStreamTag
        name: artifact-image:latest
      paths: 2
      - sourcePath: /wildfly/standalone/deployments/ROOT.war
        destinationDir: "."
  strategy:
    dockerStrategy:
      from: 3
        kind: ImageStreamTag
        name: jee-runtime:latest
  triggers:
  - imageChange: {}
    type: ImageChange
```

1 **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.

1.9.5. 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 1
  failedBuildsHistoryLimit: 2 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.

1.9.6. 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.

1.10. USING RED HAT SUBSCRIPTIONS IN BUILDS

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

1.10.1. 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/ubi7/ubi:latest ubi:latest -n openshift
  ```

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

  ```
  $ oc tag --source=docker registry.redhat.io/ubi7/ubi:latest ubi:latest
  ```

1.10.2. 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, and the entitlements must have separate public and private key files.

Procedure

1. Create a secret containing your entitlements, ensuring that there are separate files containing the public and private keys:

```
$ oc create secret generic etc-pki-entitlement --from-file /path/to/entitlement/{ID}.pem \
> --from-file /path/to/entitlement/{ID}-key.pem ...
```

2. Add the secret as a build input in the build configuration:

```
source:
  secrets:
  - secret:
      name: etc-pki-entitlement
    destinationDir: etc-pki-entitlement
```

1.10.3. Running builds with Subscription Manager

1.10.3.1. Docker builds using Subscription Manager

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

Prerequisites

The entitlement keys, subscription manager configuration, and subscription manager certificate authority must be added as build inputs.

Procedure

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

```
FROM registry.redhat.io/rhel7:latest
USER root
# Copy entitlements
COPY ./etc-pki-entitlement /etc/pki/entitlement
# Copy subscription manager configurations
COPY ./rhsm-conf /etc/rhsm
COPY ./rhsm-ca /etc/rhsm/ca
# Delete /etc/rhsm-host to use entitlements from the build container
RUN rm /etc/rhsm-host && \
    # Initialize /etc/yum.repos.d/redhat.repo
    # See https://access.redhat.com/solutions/1443553
    yum repolist --disablerepo=* && \
    subscription-manager repos --enable <enabled-repo> && \
    yum -y update && \
    yum -y install <rpms> && \
    # Remove entitlements and Subscription Manager configs
    rm -rf /etc/pki/entitlement && \
    rm -rf /etc/rhsm
# OpenShift requires images to run as non-root by default
USER 1001
ENTRYPOINT ["/bin/bash"]
```

1.10.4. Running builds with Red Hat Satellite subscriptions

1.10.4.1. 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 to the **BuildConfig**:

   ```
   source:
       configMaps:
       - configMap:
           name: yum-repos-d
         destinationDir: yum.repos.d
   ```

1.10.4.2. Docker builds using Red Hat Satellite subscriptions

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

Prerequisites

- The entitlement keys and Satellite repository configurations must be added as build inputs.

Procedure

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

```
FROM registry.redhat.io/rhel7:latest
USER root
# Copy entitlements
COPY ./etc-pki-entitlement /etc/pki/entitlement
# Copy repository configuration
COPY ./yum.repos.d /etc/yum.repos.d
# Delete /etc/rhsm-host to use entitlements from the build container
RUN sed -i".org" -e "s#^enabled=1#enabled=0#g" /etc/yum/pluginconf.d/subscription-manager.conf
1
#RUN cat /etc/yum/pluginconf.d/subscription-manager.conf
RUN yum clean all
#RUN yum-config-manager
```

```
RUN rm /etc/rhsm-host && \
    # yum repository info provided by Satellite
    yum -y update && \
    yum -y install <rpms> && \
    # Remove entitlements
    rm -rf /etc/pki/entitlement
# OpenShift requires images to run as non-root by default
USER 1001
ENTRYPOINT ["/bin/bash"]
```

1. If adding Satellite configurations to builds using **enabled=1** fails, add **RUN sed -i".org" -e "s#^enabled=1#enabled=0#g" /etc/yum/pluginconf.d/subscription-manager.conf** to the Dockerfile.

1.10.5. Additional resources

- Managing image streams
- build strategy

1.11. 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 1.3. 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

1.11.1. 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, remove the line that corresponds to the resource of the strategy to disable.

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

      ```
      kind: ClusterRole
      ```

```
metadata:
  name: admin
...
rules:
- resources:
  - builds/custom
  - builds/docker (1)
  - builds/source
  ...
...
```

(1) Delete this line to disable docker builds globally for users with the **admin** role.

1.11.2. 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.

1.11.3. 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
```

1.12. BUILD CONFIGURATION RESOURCES

Use the following procedure to configure build settings.

1.12.1. 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.
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.

Parameter	Description
BuildList	**items**: Standard object's metadata.

1.12.2. 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.

1.13. TROUBLESHOOTING BUILDS

Use the following to troubleshoot build issues.

1.13.1. 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
```

1.13.2. Service certificate generation failure

If your request for access to resources is denied:

Issue

If a service certificate generation fails with (service's **service.alpha.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.alpha.openshift.io/serving-cert-generation-error** and **service.alpha.openshift.io/serving-cert-generation-error-num**:

```
$ oc delete secret <secret_name>
```

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

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

NOTE

The command removing annotation has a **-** after the annotation name to be removed.

1.14. 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.

1.14.1. 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 cluster administrator privileges.
- 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
```

1.14.2. Additional resources

- Create a **ConfigMap**
- Secrets and **ConfigMaps**
- Configuring a custom PKI

CHAPTER 2. PIPELINES

2.1. RED HAT OPENSHIFT PIPELINES RELEASE NOTES

Red Hat OpenShift Pipelines is a cloud-native CI/CD experience based on the Tekton project which provides:

- Standard Kubernetes-native pipeline definitions (CRDs).
- Serverless pipelines with no CI server management overhead.
- Extensibility to build images using any Kubernetes tool, such as S2I, Buildah, JIB, and Kaniko.
- Portability across any Kubernetes distribution.
- Powerful CLI for interacting with pipelines.
- Integrated user experience with the **Developer** perspective of the OpenShift Container Platform web console.

For an overview of Red Hat OpenShift Pipelines, see Understanding OpenShift Pipelines.

2.1.1. Making open source more inclusive

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see Red Hat CTO Chris Wright's message .

2.1.2. Release notes for Red Hat OpenShift Pipelines General Availability 1.4

Red Hat OpenShift Pipelines General Availability (GA) 1.4 is now available on OpenShift Container Platform 4.7.

NOTE

In addition to the stable and preview Operator channels, the Red Hat OpenShift Pipelines Operator 1.4.0 comes with the ocp-4.6, ocp-4.5, and ocp-4.4 deprecated channels. These deprecated channels and support for them will be removed in the following release of Red Hat OpenShift Pipelines.

2.1.2.1. Compatibility and support matrix

Some features in this release are currently in Technology Preview. These experimental features are not intended for production use.

Technology Preview Features Support Scope

In the table below, features are marked with the following statuses:

- **TP**: *Technology Preview*
- **GA**: *General Availability*

Note the following scope of support on the Red Hat Customer Portal for these features:

Table 2.1. Compatibility and support matrix

Feature	Version	Support Status
Pipelines	0.22	GA
CLI	0.17	GA
Catalog	0.22	GA
Triggers	0.12	TP
Pipeline resources	-	TP

For questions and feedback, you can send an email to the product team at pipelines-interest@redhat.com.

2.1.2.2. New features

In addition to the fixes and stability improvements, the following sections highlight what is new in Red Hat OpenShift Pipelines 1.4.

- The custom tasks have the following enhancements:
  - Pipeline results can now refer to results produced by custom tasks.
  - Custom tasks can now use workspaces, service accounts, and pod templates to build more complex custom tasks.
- The **finally** task has the following enhancements:
  - The **when** expressions are supported in **finally** tasks, which provides efficient guarded execution and improved reusability of tasks.
  - A **finally** task can be configured to consume the results of any task within the same pipeline.

    NOTE

    Support for **when** expressions and **finally** tasks are unavailable in the OpenShift Container Platform 4.7 web console.
- Support for multiple secrets of the type **dockercfg** or **dockerconfigjson** is added for authentication at runtime.
- Functionality to support sparse-checkout with the **git-clone** task is added. This enables you to clone only a subset of the repository as your local copy, and helps you to restrict the size of the cloned repositories.
- You can create pipeline runs in a pending state without actually starting them. In clusters that are under heavy load, this allows Operators to have control over the start time of the pipeline runs.

- Ensure that you set the **SYSTEM_NAMESPACE** environment variable manually for the controller; this was previously set by default.
- A non-root user is now added to the build-base image of pipelines so that **git-init** can clone repositories as a non-root user.
- Support to validate dependencies between resolved resources before a pipeline run starts is added. All result variables in the pipeline must be valid, and optional workspaces from a pipeline can only be passed to tasks expecting it for the pipeline to start running.
- The controller and webhook runs as a non-root group, and their superfluous capabilities have been removed to make them more secure.
- You can use the **tkn pr logs** command to see the log streams for retried task runs.
- You can use the **--clustertask** option in the **tkn tr delete** command to delete all the task runs associated with a particular cluster task.
- Support for using Knative service with the **EventListener** resource is added by introducing a new **customResource** field.
- An error message is displayed when an event payload does not use the JSON format.
- The source control interceptors such as GitLab, BitBucket, and GitHub, now use the new **InterceptorRequest** or **InterceptorResponse** type interface.
- A new CEL function **marshalJSON** is implemented so that you can encode a JSON object or an array to a string.
- An HTTP handler for serving the CEL and the source control core interceptors is added. It packages four core interceptors into a single HTTP server that is deployed in the **tekton-pipelines** namespace. The **EventListener** object forwards events over the HTTP server to the interceptor. Each interceptor is available at a different path. For example, the CEL interceptor is available on the **/cel** path.
- The **pipelines-scc** Security Context Constraint (SCC) is used with the default **pipeline** service account for pipelines. This new service account is similar to **anyuid**, but with a minor difference as defined in the YAML for SCC of OpenShift Container Platform 4.7:

```
fsGroup:
  type: MustRunAs
```

2.1.2.3. Deprecated features

- The **build-gcs** sub-type in the pipeline resource storage, and the **gcs-fetcher** image, are not supported.
- In the **taskRun** field of cluster tasks, the label **tekton.dev/task** is removed.
- For webhooks, the value **v1beta1** corresponding to the field **admissionReviewVersions** is removed.
- The **creds-init** helper image for building and deploying is removed.
- In the triggers spec and binding, the deprecated field **template.Name** is removed in favor of **template.Ref**.

- For **EventListener** custom resources/objects, the fields **PodTemplate** and **ServiceType** are deprecated in favor of **Resource**.
- The deprecated spec style embedded bindings is removed.
- The **spec** field is removed from the **triggerSpecBinding**.
- The event ID representation is changed from a five-character random string to a UUID.

2.1.2.4. Known issues

- In the **Developer** perspective, the pipeline metrics and triggers features are available only on OpenShift Container Platform 4.7.6 or later versions.
- On IBM Power Systems, IBM Z, and LinuxONE, the **tkn hub** command is not supported.
- When you run Maven and Jib Maven cluster tasks on an IBM Power Systems (ppc64le), IBM Z, and LinuxONE (s390x) clusters, set the **MAVEN_IMAGE** parameter value to **maven:3.6.3-adoptopenjdk-11**.
- Triggers throw error resulting from bad handling of the JSON format, if you have the following configuration in the trigger binding:

  ```
  params:
    - name: github_json
      value: $(body)
  ```

  To resolve the issue:

  - If you are using triggers v0.11.0 and above, use the **marshalJSON** CEL function, which takes a JSON object or array and returns the JSON encoding of that object or array as a string.
  - If you are using older triggers version, add the following annotation in the trigger template:

    ```
    annotations:
      triggers.tekton.dev/old-escape-quotes: "true"
    ```

2.1.2.5. Fixed issues

- Previously, the **tekton.dev/task** label was removed from the task runs of cluster tasks, and the **tekton.dev/clusterTask** label was introduced. The problems resulting from that change is resolved by fixing the **clustertask describe** and **delete** commands. In addition, the **lastrun** function for tasks is modified, to fix the issue of the **tekton.dev/task** label being applied to the task runs of both tasks and cluster tasks in older versions of pipelines.
- When doing an interactive **tkn pipeline start pipelinename**, a **PipelineResource** is created interactively. The **tkn p start** command prints the resource status if the resource status is not **nil**.
- Previously, the **tekton.dev/task=name** label was removed from the task runs created from cluster tasks. This fix modifies the **tkn clustertask start** command with the **--last** flag to check for the **tekton.dev/task=name** label in the created task runs.
- When a task uses an inline task specification, the corresponding task run now gets embedded in the pipeline when you run the **tkn pipeline describe** command, and the task name is returned as embedded.

- The **tkn version** command is fixed to display the version of the installed Tekton CLI tool, without a configured **kubeConfiguration namespace** or access to a cluster.
- If an argument is unexpected or more than one arguments are used, the **tkn completion** command gives an error.
- Previously, pipeline runs with the **finally** tasks nested in a pipeline specification would lose those **finally** tasks, when converted to the **v1alpha1** version and restored back to the **v1beta1** version. This error occurring during conversion is fixed to avoid potential data loss. Pipeline runs with the **finally** tasks nested in a pipeline specification is now serialized and stored on the alpha version, only to be deserialized later.
- Previously, there was an error in the pod generation when a service account had the **secrets** field as **{}**. The task runs failed with **CouldntGetTask** because the GET request with an empty secret name returned an error, indicating that the resource name may not be empty. This issue is fixed by avoiding an empty secret name in the **kubeclient** GET request.
- Pipelines with the **v1beta1** API versions can now be requested along with the **v1alpha1** version, without losing the **finally** tasks. Applying the returned **v1alpha1** version will store the resource as **v1beta1**, with the **finally** section restored to its original state.
- Previously, an unset **selfLink** field in the controller caused an error in the Kubernetes v1.20 clusters. As a temporary fix, the **CloudEvent** source field is set to a value that matches the current source URI, without the value of the auto-populated **selfLink** field.
- Previously, a secret name with dots such as **gcr.io** led to a task run creation failure. This happened because of the secret name being used internally as part of a volume mount name. The volume mount name conforms to the RFC1123 DNS label and disallows dots as part of the name. This issue is fixed by replacing the dot with a dash that results in a readable name.
- Context variables are now validated in the **finally** tasks.
- Previously, when the task run reconciler was passed a task run that did not have a previous status update containing the name of the pod it created, the task run reconciler listed the pods associated with the task run. The task run reconciler used the labels of the task run, which were propagated to the pod, to find the pod. Changing these labels while the task run was running, caused the code to not find the existing pod. As a result, duplicate pods were created. This issue is fixed by changing the task run reconciler to only use the **tekton.dev/taskRun** Tekton-controlled label when finding the pod.
- Previously, when a pipeline accepted an optional workspace and passed it to a pipeline task, the pipeline run reconciler stopped with an error if the workspace was not provided, even if a missing workspace binding is a valid state for an optional workspace. This issue is fixed by ensuring that the pipeline run reconciler does not fail to create a task run, even if an optional workspace is not provided.
- The sorted order of step statuses matches the order of step containers.
- Previously, the task run status was set to **unknown** when a pod encountered the **CreateContainerConfigError** reason, which meant that the task and the pipeline ran until the pod timed out. This issue is fixed by setting the task run status to **false**, so that the task is set as failed when the pod encounters the **CreateContainerConfigError** reason.
- Previously, pipeline results were resolved on the first reconciliation, after a pipeline run was completed. This could fail the resolution resulting in the **Succeeded** condition of the pipeline run being overwritten. As a result, the final status information was lost, potentially confusing any

services watching the pipeline run conditions. This issue is fixed by moving the resolution of pipeline results to the end of a reconciliation, when the pipeline run is put into a **Succeeded** or **True** condition.

- Execution status variable is now validated. This avoids validating task results while validating context variables to access execution status.
- Previously, a pipeline result that contained an invalid variable would be added to the pipeline run with the literal expression of the variable intact. Therefore, it was difficult to assess whether the results were populated correctly. This issue is fixed by filtering out the pipeline run results that reference failed task runs. Now, a pipeline result that contains an invalid variable will not be emitted by the pipeline run at all.
- The **tkn eventlistener describe** command is fixed to avoid crashing without a template. It also displays the details about trigger references.

2.1.3. Release notes for Red Hat OpenShift Pipelines Technology Preview 1.3

2.1.3.1. New features

Red Hat OpenShift Pipelines Technology Preview (TP) 1.3 is now available on OpenShift Container Platform 4.7. Red Hat OpenShift Pipelines TP 1.3 is updated to support:

- Tekton Pipelines 0.19.0
- Tekton **tkn** CLI 0.15.0
- Tekton Triggers 0.10.2
- cluster tasks based on Tekton Catalog 0.19.0
- IBM Power Systems on OpenShift Container Platform 4.7
- IBM Z and LinuxONE on OpenShift Container Platform 4.7

In addition to the fixes and stability improvements, the following sections highlight what is new in Red Hat OpenShift Pipelines 1.3.

2.1.3.1.1. Pipelines

- Tasks that build images, such as S2I and Buildah tasks, now emit a URL of the image built that includes the image SHA.
- Conditions in pipeline tasks that reference custom tasks are disallowed because the **Condition** custom resource definition (CRD) has been deprecated.
- Variable expansion is now added in the **Task** CRD for the following fields: **spec.steps[].imagePullPolicy** and **spec.sidecar[].imagePullPolicy**.
- You can disable the built-in credential mechanism in Tekton by setting the **disable-creds-init** feature-flag to **true**.
- Resolved when expressions are now listed in the **Skipped Tasks** and the **Task Runs** sections in the **Status** field of the **PipelineRun** configuration.
- The **git init** command can now clone recursive submodules.

- A **Task** CR author can now specify a timeout for a step in the **Task** spec.
- You can now base the entry point image on the **distroless/static:nonroot** image and give it a mode to copy itself to the destination, without relying on the **cp** command being present in the base image.
- You can now use the configuration flag **require-git-ssh-secret-known-hosts** to disallow omitting known hosts in the Git SSH secret. When the flag value is set to **true**, you must include the **known_host** field in the Git SSH secret. The default value for the flag is **false**.
- The concept of optional workspaces is now introduced. A task or pipeline might declare a workspace optional and conditionally change their behavior based on its presence. A task run or pipeline run might also omit that workspace, thereby modifying the task or pipeline behavior. The default task run workspaces are not added in place of an omitted optional workspace.
- Credentials initialization in Tekton now detects an SSH credential that is used with a non-SSH URL, and vice versa in Git pipeline resources, and logs a warning in the step containers.
- The task run controller emits a warning event if the affinity specified by the pod template is overwritten by the affinity assistant.
- The task run reconciler now records metrics for cloud events that are emitted once a task run is completed. This includes retries.

2.1.3.1.2. Pipelines CLI

- Support for **--no-headers flag** is now added to the following commands: **tkn condition list**,**tkn triggerbinding list**,**tkn eventlistener list**,**tkn clustertask list**, **tkn clustertriggerbinding list**.
- When used together, the **--last** or **--use** options override the **--prefix-name** and **--timeout** options.
- The **tkn eventlistener logs** command is now added to view the **EventListener** logs.
- The **tekton hub** commands are now integrated into the **tkn** CLI.
- The **--nocolour** option is now changed to **--no-color**.
- The **--all-namespaces** flag is added to the following commands: **tkn triggertemplate list**, **tkn condition list**, **tkn triggerbinding list**, **tkn eventlistener list**.

2.1.3.1.3. Triggers

- You can now specify your resource information in the **EventListener** template.
- It is now mandatory for **EventListener** service accounts to have the **list** and **watch** verbs, in addition to the **get** verb for all the triggers resources. This enables you to use **Listers** to fetch data from **EventListener**, **Trigger**, **TriggerBinding**, **TriggerTemplate**, and **ClusterTriggerBinding** resources. You can use this feature to create a **Sink** object rather than specifying multiple informers, and directly make calls to the API server.
- A new **Interceptor** interface is added to support immutable input event bodies. Interceptors can now add data or fields to a new **extensions** field, and cannot modify the input bodies making them immutable. The CEL interceptor uses this new **Interceptor** interface.
- A **namespaceSelector** field is added to the **EventListener** resource. Use it to specify the namespaces from where the **EventListener** resource can fetch the **Trigger** object for

processing events. To use the **namespaceSelector** field, the service account for the **EventListener** resource must have a cluster role.

- The triggers **EventListener** resource now supports end-to-end secure connection to the **eventlistener** pod.
- The escaping parameters behavior in the **TriggerTemplates** resource by replacing **"** with **\"** is now removed.
- A new **resources** field, supporting Kubernetes resources, is introduced as part of the **EventListener** spec.
- A new functionality for the CEL interceptor, with support for upper and lower-casing of ASCII strings, is added.
- You can embed **TriggerBinding** resources by using the **name** and **value** fields in a trigger, or an event listener.
- The **PodSecurityPolicy** configuration is updated to run in restricted environments. It ensures that containers must run as non-root. In addition, the role-based access control for using the pod security policy is moved from cluster-scoped to namespace-scoped. This ensures that the triggers cannot use other pod security policies that are unrelated to a namespace.
- Support for embedded trigger templates is now added. You can either use the **name** field to refer to an embedded template or embed the template inside the **spec** field.

2.1.3.2. Deprecated features

- Pipeline templates that use **PipelineResources** CRDs are now deprecated and will be removed in a future release.
- The **template.name** field is deprecated in favor of the **template.ref** field and will be removed in a future release.
- The **-c** shorthand for the **--check** command has been removed. In addition, global **tkn** flags are added to the **version** command.

2.1.3.3. Known issues

- CEL overlays add fields to a new top-level **extensions** function, instead of modifying the incoming event body. **TriggerBinding** resources can access values within this new **extensions** function using the **$(extensions.<key>)** syntax. Update your binding to use the **$(extensions.<key>)** syntax instead of the **$(body.<overlay-key>)** syntax.
- The escaping parameters behavior by replacing **"** with **\"** is now removed. If you need to retain the old escaping parameters behavior add the **tekton.dev/old-escape-quotes: true"** annotation to your **TriggerTemplate** specification.
- You can embed **TriggerBinding** resources by using the **name** and **value** fields inside a trigger or an event listener. However, you cannot specify both **name** and **ref** fields for a single binding. Use the **ref** field to refer to a **TriggerBinding** resource and the **name** field for embedded bindings.
- An interceptor cannot attempt to reference a **secret** outside the namespace of an **EventListener** resource. You must include secrets in the namespace of the `EventListener`resource.

- In Triggers 0.9.0 and later, if a body or header based **TriggerBinding** parameter is missing or malformed in an event payload, the default values are used instead of displaying an error.
- Tasks and pipelines created with **WhenExpression** objects using Tekton Pipelines 0.16.x must be reapplied to fix their JSON annotations.
- When a pipeline accepts an optional workspace and gives it to a task, the pipeline run stalls if the workspace is not provided.
- To use the Buildah cluster task in a disconnected environment, ensure that the Dockerfile uses an internal image stream as the base image, and then use it in the same manner as any S2I cluster task.

2.1.3.4. Fixed issues

- Extensions added by a CEL Interceptor are passed on to webhook interceptors by adding the **Extensions** field within the event body.
- The activity timeout for log readers is now configurable using the **LogOptions** field. However, the default behavior of timeout in 10 seconds is retained.
- The **log** command ignores the **--follow** flag when a task run or pipeline run is complete, and reads available logs instead of live logs.
- References to the following Tekton resources: **EventListener**, **TriggerBinding**, **ClusterTriggerBinding**, **Condition**, and **TriggerTemplate** are now standardized and made consistent across all user-facing messages in **tkn** commands.
- Previously, if you started a canceled task run or pipeline run with the **--use-taskrun <canceled-task-run-name>**, **--use-pipelinerun <canceled-pipeline-run-name>** or **--last** flags, the new run would be canceled. This bug is now fixed.
- The **tkn pr desc** command is now enhanced to ensure that it does not fail in case of pipeline runs with conditions.
- When you delete a task run using the **tkn tr delete** command with the **--task** option, and a cluster task exists with the same name, the task runs for the cluster task also get deleted. As a workaround, filter the task runs by using the **TaskRefKind** field.
- The **tkn triggertemplate describe** command would display only part of the **apiVersion** value in the output. For example, only **triggers.tekton.dev** was displayed instead of **triggers.tekton.dev/v1alpha1**. This bug is now fixed.
- The webhook, under certain conditions, would fail to acquire a lease and not function correctly. This bug is now fixed.
- Pipelines with when expressions created in v0.16.3 can now be run in v0.17.1 and later. After an upgrade, you do not need to reapply pipeline definitions created in previous versions because both the uppercase and lowercase first letters for the annotations are now supported.
- By default, the **leader-election-ha** field is now enabled for high availability. When the **disable-ha** controller flag is set to **true**, it disables high availability support.
- Issues with duplicate cloud events are now fixed. Cloud events are now sent only when a condition changes the state, reason, or message.
- When a service account name is missing from a **PipelineRun** or **TaskRun** spec, the controller

uses the service account name from the **config-defaults** config map. If the service account name is also missing in the **config-defaults** config map, the controller now sets it to **default** in the spec.

- Validation for compatibility with the affinity assistant is now supported when the same persistent volume claim is used for multiple workspaces, but with different subpaths.

2.1.4. Release notes for Red Hat OpenShift Pipelines Technology Preview 1.2

2.1.4.1. New features

Red Hat OpenShift Pipelines Technology Preview (TP) 1.2 is now available on OpenShift Container Platform 4.6. Red Hat OpenShift Pipelines TP 1.2 is updated to support:

- Tekton Pipelines 0.16.3
- Tekton **tkn** CLI 0.13.1
- Tekton Triggers 0.8.1
- cluster tasks based on Tekton Catalog 0.16
- IBM Power Systems on OpenShift Container Platform 4.6
- IBM Z and LinuxONE on OpenShift Container Platform 4.6

In addition to the fixes and stability improvements, the following sections highlight what is new in Red Hat OpenShift Pipelines 1.2.

2.1.4.1.1. Pipelines

- This release of Red Hat OpenShift Pipelines adds support for a disconnected installation.

  NOTE

  Installations in restricted environments are currently not supported on IBM Power Systems, IBM Z, and LinuxONE.

- You can now use the **when** field, instead of **conditions** resource, to run a task only when certain criteria are met. The key components of **WhenExpression** resources are **Input**, **Operator**, and **Values**. If all the when expressions evaluate to **True**, then the task is run. If any of the when expressions evaluate to **False**, the task is skipped.
- Step statuses are now updated if a task run is canceled or times out.
- Support for Git Large File Storage (LFS) is now available to build the base image used by **git-init**.
- You can now use the **taskSpec** field to specify metadata, such as labels and annotations, when a task is embedded in a pipeline.
- Cloud events are now supported by pipeline runs. Retries with **backoff** are now enabled for cloud events sent by the cloud event pipeline resource.

- You can now set a default **Workspace** configuration for any workspace that a **Task** resource declares, but that a **TaskRun** resource does not explicitly provide.
- Support is available for namespace variable interpolation for the **PipelineRun** namespace and **TaskRun** namespace.
- Validation for **TaskRun** objects is now added to check that not more than one persistent volume claim workspace is used when a **TaskRun** resource is associated with an Affinity Assistant. If more than one persistent volume claim workspace is used, the task run fails with a **TaskRunValidationFailed** condition. Note that by default, the Affinity Assistant is disabled in Red Hat OpenShift Pipelines, so you will need to enable the assistant to use it.

2.1.4.1.2. Pipelines CLI

- The **tkn task describe**, **tkn taskrun describe**, **tkn clustertask describe**, **tkn pipeline describe**, and **tkn pipelinerun describe** commands now:
  - Automatically select the **Task**, **TaskRun**, **ClusterTask**, **Pipeline** and **PipelineRun** resource, respectively, if only one of them is present.
  - Display the results of the **Task**, **TaskRun**, **ClusterTask**, **Pipeline** and **PipelineRun** resource in their outputs, respectively.
  - Display workspaces declared in the **Task**, **TaskRun**, **ClusterTask**, **Pipeline** and **PipelineRun** resource in their outputs, respectively.
- You can now use the **--prefix-name** option with the **tkn clustertask start** command to specify a prefix for the name of a task run.
- Interactive mode support has now been provided to the **tkn clustertask start** command.
- You can now specify **PodTemplate** properties supported by pipelines using local or remote file definitions for **TaskRun** and **PipelineRun** objects.
- You can now use the **--use-params-defaults** option with the **tkn clustertask start** command to use the default values set in the **ClusterTask** configuration and create the task run.
- The **--use-param-defaults** flag for the **tkn pipeline start** command now prompts the interactive mode if the default values have not been specified for some of the parameters.

2.1.4.1.3. Triggers

- The Common Expression Language (CEL) function named **parseYAML** has been added to parse a YAML string into a map of strings.
- Error messages for parsing CEL expressions have been improved to make them more granular while evaluating expressions and when parsing the hook body for creating the evaluation environment.
- Support is now available for marshaling boolean values and maps if they are used as the values of expressions in a CEL overlay mechanism.
- The following fields have been added to the **EventListener** object:
  - The **replicas** field enables the event listener to run more than one pod by specifying the number of replicas in the YAML file.

    - The **NodeSelector** field enables the **EventListener** object to schedule the event listener pod to a specific node.

- Webhook interceptors can now parse the **EventListener-Request-URL** header to extract parameters from the original request URL being handled by the event listener.

- Annotations from the event listener can now be propagated to the deployment, services, and other pods. Note that custom annotations on services or deployment are overwritten, and hence, must be added to the event listener annotations so that they are propagated.

- Proper validation for replicas in the **EventListener** specification is now available for cases when a user specifies the **spec.replicas** values as **negative** or **zero**.

- You can now specify the **TriggerCRD** object inside the **EventListener** spec as a reference using the **TriggerRef** field to create the **TriggerCRD** object separately and then bind it inside the **EventListener** spec.

- Validation and defaults for the **TriggerCRD** object are now available.

2.1.4.2. Deprecated features

- **$(params)** parameters are now removed from the **triggertemplate** resource and replaced by **$(tt.params)** to avoid confusion between the **resourcetemplate** and **triggertemplate** resource parameters.

- The **ServiceAccount** reference of the optional **EventListenerTrigger**-based authentication level has changed from an object reference to a **ServiceAccountName** string. This ensures that the **ServiceAccount** reference is in the same namespace as the **EventListenerTrigger** object.

- The **Conditions** custom resource definition (CRD) is now deprecated; use the **WhenExpressions** CRD instead.

- The **PipelineRun.Spec.ServiceAccountNames** object is being deprecated and replaced by the **PipelineRun.Spec.TaskRunSpec[].ServiceAccountName** object.

2.1.4.3. Known issues

- This release of Red Hat OpenShift Pipelines adds support for a disconnected installation. However, some images used by the cluster tasks must be mirrored for them to work in disconnected clusters.

- Pipelines in the **openshift** namespace are not deleted after you uninstall the Red Hat OpenShift Pipelines Operator. Use the **oc delete pipelines -n openshift --all** command to delete the pipelines.

- Uninstalling the Red Hat OpenShift Pipelines Operator does not remove the event listeners. As a workaround, to remove the **EventListener** and **Pod** CRDs:

    1. Edit the **EventListener** object with the **foregroundDeletion** finalizers:

    ```
    $ oc patch el/<eventlistener_name> -p '{"metadata":{"finalizers":["foregroundDeletion"]}}'
    --type=merge
    ```

    For example:

```
$ oc patch el/github-listener-interceptor -p '{"metadata":{"finalizers":
["foregroundDeletion"]}}' --type=merge
```

2. Delete the **EventListener** CRD:

```
$ oc patch crd/eventlisteners.triggers.tekton.dev -p '{"metadata":{"finalizers":[]}}' --
type=merge
```

- When you run a multi-arch container image task without command specification on an IBM Power Systems (ppc64le) or IBM Z (s390x) cluster, the **TaskRun** resource fails with the following error:

  ```
  Error executing command: fork/exec /bin/bash: exec format error
  ```

  As a workaround, use an architecture specific container image or specify the sha256 digest to point to the correct architecture. To get the sha256 digest enter:

  ```
  $ skopeo inspect --raw <image_name>| jq '.manifests[] | select(.platform.architecture == "
  <architecture>") | .digest'
  ```

2.1.4.4. Fixed issues

- A simple syntax validation to check the CEL filter, overlays in the Webhook validator, and the expressions in the interceptor has now been added.
- Triggers no longer overwrite annotations set on the underlying deployment and service objects.
- Previously, an event listener would stop accepting events. This fix adds an idle timeout of 120 seconds for the **EventListener** sink to resolve this issue.
- Previously, canceling a pipeline run with a **Failed(Canceled)** state gave a success message. This has been fixed to display an error instead.
- The **tkn eventlistener list** command now provides the status of the listed event listeners, thus enabling you to easily identify the available ones.
- Consistent error messages are now displayed for the **triggers list** and **triggers describe** commands when triggers are not installed or when a resource cannot be found.
- Previously, a large number of idle connections would build up during cloud event delivery. The **DisableKeepAlives: true** parameter was added to the **cloudeventclient** config to fix this issue. Thus, a new connection is set up for every cloud event.
- Previously, the **creds-init** code would write empty files to the disk even if credentials of a given type were not provided. This fix modifies the **creds-init** code to write files for only those credentials that have actually been mounted from correctly annotated secrets.

2.1.5. Release notes for Red Hat OpenShift Pipelines Technology Preview 1.1

2.1.5.1. New features

Red Hat OpenShift Pipelines Technology Preview (TP) 1.1 is now available on OpenShift Container Platform 4.5. Red Hat OpenShift Pipelines TP 1.1 is updated to support:

- Tekton Pipelines 0.14.3
- Tekton **tkn** CLI 0.11.0
- Tekton Triggers 0.6.1
- cluster tasks based on Tekton Catalog 0.14

In addition to the fixes and stability improvements, the following sections highlight what is new in Red Hat OpenShift Pipelines 1.1.

2.1.5.1.1. Pipelines

- Workspaces can now be used instead of pipeline resources. It is recommended that you use workspaces in OpenShift Pipelines, as pipeline resources are difficult to debug, limited in scope, and make tasks less reusable. For more details on workspaces, see the Understanding OpenShift Pipelines section.
- Workspace support for volume claim templates has been added:
  - The volume claim template for a pipeline run and task run can now be added as a volume source for workspaces. The tekton-controller then creates a persistent volume claim (PVC) using the template that is seen as a PVC for all task runs in the pipeline. Thus you do not need to define the PVC configuration every time it binds a workspace that spans multiple tasks.
  - Support to find the name of the PVC when a volume claim template is used as a volume source is now available using variable substitution.
- Support for improving audits:
  - The **PipelineRun.Status** field now contains the status of every task run in the pipeline and the pipeline specification used to instantiate a pipeline run to monitor the progress of the pipeline run.
  - Pipeline results have been added to the pipeline specification and **PipelineRun** status.
  - The **TaskRun.Status** field now contains the exact task specification used to instantiate the **TaskRun** resource.
- Support to apply the default parameter to conditions.
- A task run created by referencing a cluster task now adds the **tekton.dev/clusterTask** label instead of the **tekton.dev/task** label.
- The kube config writer now adds the **ClientKeyData** and the **ClientCertificateData** configurations in the resource structure to enable replacement of the pipeline resource type cluster with the kubeconfig-creator task.
- The names of the **feature-flags** and the **config-defaults** config maps are now customizable.
- Support for the host network in the pod template used by the task run is now available.
- An Affinity Assistant is now available to support node affinity in task runs that share workspace volume. By default, this is disabled on OpenShift Pipelines.

- The pod template has been updated to specify **imagePullSecrets** to identify secrets that the container runtime should use to authorize container image pulls when starting a pod.
- Support for emitting warning events from the task run controller if the controller fails to update the task run.
- Standard or recommended k8s labels have been added to all resources to identify resources belonging to an application or component.
- The **Entrypoint** process is now notified for signals and these signals are then propagated using a dedicated PID Group of the **Entrypoint** process.
- The pod template can now be set on a task level at runtime using task run specs.
- Support for emitting Kubernetes events:
  - The controller now emits events for additional task run lifecycle events - **taskrun started** and **taskrun running**.
  - The pipeline run controller now emits an event every time a pipeline starts.
- In addition to the default Kubernetes events, support for cloud events for task runs is now available. The controller can be configured to send any task run events, such as create, started, and failed, as cloud events.
- Support for using the **$context.<task|taskRun|pipeline|pipelineRun>.name** variable to reference the appropriate name when in pipeline runs and task runs.
- Validation for pipeline run parameters is now available to ensure that all the parameters required by the pipeline are provided by the pipeline run. This also allows pipeline runs to provide extra parameters in addition to the required parameters.
- You can now specify tasks within a pipeline that will always execute before the pipeline exits, either after finishing all tasks successfully or after a task in the pipeline failed, using the **finally** field in the pipeline YAML file.
- The **git-clone** cluster task is now available.

2.1.5.1.2. Pipelines CLI

- Support for embedded trigger binding is now available to the **tkn evenlistener describe** command.
- Support to recommend subcommands and make suggestions if an incorrect subcommand is used.
- The **tkn task describe** command now auto selects the task if only one task is present in the pipeline.
- You can now start a task using default parameter values by specifying the **--use-param-defaults** flag in the **tkn task start** command.
- You can now specify a volume claim template for pipeline runs or task runs using the **--workspace** option with the **tkn pipeline start** or **tkn task start** commands.
- The **tkn pipelinerun logs** command now displays logs for the final tasks listed in the **finally** section.

- Interactive mode support has now been provided to the **tkn task start** command and the **describe** subcommand for the following **tkn** resources: **pipeline**, **pipelinerun**, **task**, **taskrun**, **clustertask**, and **pipelineresource**.
- The **tkn version** command now displays the version of the triggers installed in the cluster.
- The **tkn pipeline describe** command now displays parameter values and timeouts specified for tasks used in the pipeline.
- Support added for the **--last** option for the **tkn pipelinerun describe** and the **tkn taskrun describe** commands to describe the most recent pipeline run or task run, respectively.
- The **tkn pipeline describe** command now displays the conditions applicable to the tasks in the pipeline.
- You can now use the **--no-headers** and **--all-namespaces** flags with the **tkn resource list** command.

2.1.5.1.3. Triggers

- The following Common Expression Language (CEL) functions are now available:
  - **parseURL** to parse and extract portions of a URL
  - **parseJSON** to parse JSON value types embedded in a string in the **payload** field of the **deployment** webhook
- A new interceptor for webhooks from Bitbucket has been added.
- Event listeners now display the **Address URL** and the **Available status** as additional fields when listed with the **kubectl get** command.
- trigger template params now use the **$(tt.params.<paramName>)** syntax instead of **$(params.<paramName>)** to reduce the confusion between trigger template and resource templates params.
- You can now add **tolerations** in the **EventListener** CRD to ensure that event listeners are deployed with the same configuration even if all nodes are tainted due to security or management issues.
- You can now add a Readiness Probe for event listener Deployment at **URL/live**.
- Support for embedding **TriggerBinding** specifications in event listener triggers is now added.
- Trigger resources are now annotated with the recommended **app.kubernetes.io** labels.

2.1.5.2. Deprecated features

The following items are deprecated in this release:

- The **--namespace** or **-n** flags for all cluster-wide commands, including the **clustertask** and **clustertriggerbinding** commands, are deprecated. It will be removed in a future release.
- The **name** field in **triggers.bindings** within an event listener has been deprecated in favor of the **ref** field and will be removed in a future release.

[illegible]

- Variable interpolation in trigger templates using **$(params)** has been deprecated in favor of using **$(tt.params)** to reduce confusion with the pipeline variable interpolation syntax. The **$(params.<paramName>)** syntax will be removed in a future release.
- The **tekton.dev/task** label is deprecated on cluster tasks.
- The **TaskRun.Status.ResourceResults.ResourceRef** field is deprecated and will be removed.
- The **tkn pipeline create**, **tkn task create**, and **tkn resource create -f** subcommands have been removed.
- Namespace validation has been removed from **tkn** commands.
- The default timeout of **1h** and the **-t** flag for the **tkn ct start** command have been removed.
- The **s2i** cluster task has been deprecated.

2.1.5.3. Known issues

- Conditions do not support workspaces.
- The **--workspace** option and the interactive mode is not supported for the **tkn clustertask start** command.
- Support of backward compatibility for **$(params.<paramName>)** syntax forces you to use trigger templates with pipeline specific params as the trigger s webhook is unable to differentiate trigger params from pipelines params.
- Pipeline metrics report incorrect values when you run a promQL query for **tekton_taskrun_count** and **tekton_taskrun_duration_seconds_count**.
- pipeline runs and task runs continue to be in the **Running** and **Running(Pending)** states respectively even when a non existing PVC name is given to a workspace.

2.1.5.4. Fixed issues

- Previously, the **tkn task delete <name> --trs** command would delete both the task and cluster task if the name of the task and cluster task were the same. With this fix, the command deletes only the task runs that are created by the task **<name>**.
- Previously the **tkn pr delete -p <name> --keep 2** command would disregard the **-p** flag when used with the **--keep** flag and would delete all the pipeline runs except the latest two. With this fix, the command deletes only the pipeline runs that are created by the pipeline **<name>**, except for the latest two.
- The **tkn triggertemplate describe** output now displays resource templates in a table format instead of YAML format.
- Previously the **buildah** cluster task failed when a new user was added to a container. With this fix, the issue has been resolved.

2.1.6. Release notes for Red Hat OpenShift Pipelines Technology Preview 1.0

2.1.6.1. New features

Red Hat OpenShift Pipelines Technology Preview (TP) 1.0 is now available on OpenShift Container Platform 4.4. Red Hat OpenShift Pipelines TP 1.0 is updated to support:

- Tekton Pipelines 0.11.3
- Tekton **tkn** CLI 0.9.0
- Tekton Triggers 0.4.0
- cluster tasks based on Tekton Catalog 0.11

In addition to the fixes and stability improvements, the following sections highlight what is new in Red Hat OpenShift Pipelines 1.0.

2.1.6.1.1. Pipelines

- Support for v1beta1 API Version.
- Support for an improved limit range. Previously, limit range was specified exclusively for the task run and the pipeline run. Now there is no need to explicitly specify the limit range. The minimum limit range across the namespace is used.
- Support for sharing data between tasks using task results and task params.
- Pipelines can now be configured to not overwrite the **HOME** environment variable and the working directory of steps.
- Similar to task steps, **sidecars** now support script mode.
- You can now specify a different scheduler name in task run **podTemplate** resource.
- Support for variable substitution using Star Array Notation.
- Tekton controller can now be configured to monitor an individual namespace.
- A new description field is now added to the specification of pipelines, tasks, cluster tasks, resources, and conditions.
- Addition of proxy parameters to Git pipeline resources.

2.1.6.1.2. Pipelines CLI

- The **describe** subcommand is now added for the following **tkn** resources: **EventListener**, **Condition**, **TriggerTemplate**, **ClusterTask**, and **TriggerSBinding**.
- Support added for **v1beta1** to the following resources along with backward compatibility for **v1alpha1**: **ClusterTask**, **Task**, **Pipeline**, **PipelineRun**, and **TaskRun**.
- The following commands can now list output from all namespaces using the **--all-namespaces** flag option: **tkn task list**, **tkn pipeline list**, **tkn taskrun list**, **tkn pipelinerun list**
  The output of these commands is also enhanced to display information without headers using the **--no-headers** flag option.
- You can now start a pipeline using default parameter values by specifying **--use-param-defaults** flag in the **tkn pipelines start** command.
- Support for workspace is now added to **tkn pipeline start** and **tkn task start** commands.

- A new **clustertriggerbinding** command is now added with the following subcommands: **describe**, **delete**, and **list**.
- You can now directly start a pipeline run using a local or remote **yaml** file.
- The **describe** subcommand now displays an enhanced and detailed output. With the addition of new fields, such as **description**, **timeout**, **param description**, and **sidecar status**, the command output now provides more detailed information about a specific **tkn** resource.
- The **tkn task log** command now displays logs directly if only one task is present in the namespace.

2.1.6.1.3. Triggers

- Triggers can now create both **v1alpha1** and **v1beta1** pipeline resources.
- Support for new Common Expression Language (CEL) interceptor function - **compareSecret**. This function securely compares strings to secrets in CEL expressions.
- Support for authentication and authorization at the event listener trigger level.

2.1.6.2. Deprecated features

The following items are deprecated in this release:

- The environment variable **$HOME**, and variable **workingDir** in the **Steps** specification are deprecated and might be changed in a future release. Currently in a **Step** container, the **HOME** and **workingDir** variables are overwritten to **/tekton/home** and **/workspace** variables, respectively.
  In a later release, these two fields will not be modified, and will be set to values defined in the container image and the **Task** YAML. For this release, use the **disable-home-env-overwrite** and **disable-working-directory-overwrite** flags to disable overwriting of the **HOME** and **workingDir** variables.
- The following commands are deprecated and might be removed in the future release: **tkn pipeline create**, **tkn task create**.
- The **-f** flag with the **tkn resource create** command is now deprecated. It might be removed in the future release.
- The **-t** flag and the **--timeout** flag (with seconds format) for the **tkn clustertask create** command are now deprecated. Only duration timeout format is now supported, for example **1h30s**. These deprecated flags might be removed in the future release.

2.1.6.3. Known issues

- If you are upgrading from an older version of Red Hat OpenShift Pipelines, you must delete your existing deployments before upgrading to Red Hat OpenShift Pipelines version 1.0. To delete an existing deployment, you must first delete Custom Resources and then uninstall the Red Hat OpenShift Pipelines Operator. For more details, see the uninstalling Red Hat OpenShift Pipelines section.
- Submitting the same **v1alpha1** tasks more than once results in an error. Use the **oc replace** command instead of **oc apply** when re-submitting a **v1alpha1** task.
- The **buildah** cluster task does not work when a new user is added to a container.

When the Operator is installed, the **--storage-driver** flag for the **buildah** cluster task is not specified, therefore the flag is set to its default value. In some cases, this causes the storage driver to be set incorrectly. When a new user is added, the incorrect storage-driver results in the failure of the **buildah** cluster task with the following error:

```
useradd: /etc/passwd.8: lock file already used
useradd: cannot lock /etc/passwd; try again later.
```

As a workaround, manually set the **--storage-driver** flag value to **overlay** in the **buildah-task.yaml** file:

1. Login to your cluster as a **cluster-admin**:

   ```
   $ oc login -u <login> -p <password> https://openshift.example.com:6443
   ```

2. Use the **oc edit** command to edit **buildah** cluster task:

   ```
   $ oc edit clustertask buildah
   ```

   The current version of the **buildah** clustertask YAML file opens in the editor set by your **EDITOR** environment variable.

3. Under the **Steps** field, locate the following **command** field:

   ```
    command: ['buildah', 'bud', '--format=$(params.FORMAT)', '--tls-
   verify=$(params.TLSVERIFY)', '--layers', '-f', '$(params.DOCKERFILE)', '-t',
   '$(resources.outputs.image.url)', '$(params.CONTEXT)']
   ```

4. Replace the **command** field with the following:

   ```
    command: ['buildah', '--storage-driver=overlay', 'bud', '--format=$(params.FORMAT)', '--
   tls-verify=$(params.TLSVERIFY)', '--no-cache', '-f', '$(params.DOCKERFILE)', '-t',
   '$(params.IMAGE)', '$(params.CONTEXT)']
   ```

5. Save the file and exit.

Alternatively, you can also modify the **buildah** cluster task YAML file directly on the web console by navigating to **Pipelines** → **Cluster Tasks** → **buildah**. Select **Edit Cluster Task** from the **Actions** menu and replace the **command** field as shown in the previous procedure.

2.1.6.4. Fixed issues

- Previously, the **DeploymentConfig** task triggered a new deployment build even when an image build was already in progress. This caused the deployment of the pipeline to fail. With this fix, the **deploy task** command is now replaced with the **oc rollout status** command which waits for the in-progress deployment to finish.
- Support for **APP_NAME** parameter is now added in pipeline templates.
- Previously, the pipeline template for Java S2I failed to look up the image in the registry. With this fix, the image is looked up using the existing image pipeline resources instead of the user provided **IMAGE_NAME** parameter.
- All the OpenShift Pipelines images are now based on the Red Hat Universal Base Images (UBI).

- Previously, when the pipeline was installed in a namespace other than **tekton-pipelines**, the **tkn version** command displayed the pipeline version as **unknown**. With this fix, the **tkn version** command now displays the correct pipeline version in any namespace.
- The **-c** flag is no longer supported for the **tkn version** command.
- Non-admin users can now list the cluster trigger bindings.
- The event listener **CompareSecret** function is now fixed for the CEL Interceptor.
- The **list**, **describe**, and **start** subcommands for tasks and cluster tasks now correctly display the output in case a task and cluster task have the same name.
- Previously, the OpenShift Pipelines Operator modified the privileged security context constraints (SCCs), which caused an error during cluster upgrade. This error is now fixed.
- In the **tekton-pipelines** namespace, the timeouts of all task runs and pipeline runs are now set to the value of **default-timeout-minutes** field using the config map.
- Previously, the pipelines section in the web console was not displayed for non-admin users. This issue is now resolved.

2.2. UNDERSTANDING 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.

2.2.1. Key features

- Red Hat OpenShift Pipelines is a serverless CI/CD system that runs pipelines with all the required dependencies in isolated containers.
- Red Hat OpenShift Pipelines are designed for decentralized teams that work on microservice-based architecture.
- Red Hat OpenShift Pipelines use standard CI/CD pipeline definitions that are easy to extend and integrate with the existing Kubernetes tools, enabling you to scale on-demand.
- You can use Red Hat OpenShift Pipelines to build images with Kubernetes tools such as Source-to-Image (S2I), Buildah, Buildpacks, and Kaniko that are portable across any Kubernetes platform.
- You can use the OpenShift Container Platform Developer console to create Tekton resources, view logs of pipeline runs, and manage pipelines in your OpenShift Container Platform namespaces.

2.2.2. OpenShift Pipeline Concepts

This guide provides a detailed view of the various pipeline concepts.

2.2.2.1. Tasks

Tasks are the building blocks of a Pipeline and consists of sequentially executed steps. It is essentially a function of inputs and outputs. A Task can run individually or as a part of the pipeline. Tasks are reusable and can be used in multiple Pipelines.

Steps are a series of commands that are sequentially executed by the Task and achieve a specific goal, such as building an image. Every Task runs as a pod, and each Step runs as a container within that pod. Because Steps run within the same pod, they can access the same volumes for caching files, config maps, and secrets.

The following example shows the **apply-manifests** Task.

```
apiVersion: tekton.dev/v1beta1 1
kind: Task 2
metadata:
  name: apply-manifests 3
spec: 4
  workspaces:
  - name: source
  params:
    - name: manifest_dir
      description: The directory in source that contains yaml manifests
      type: string
      default: "k8s"
  steps:
    - name: apply
      image: image-registry.openshift-image-registry.svc:5000/openshift/cli:latest
      workingDir: /workspace/source
      command: ["/bin/bash", "-c"]
      args:
        - |-
          echo Applying manifests in $(params.manifest_dir) directory
          oc apply -f $(params.manifest_dir)
          echo -----------------------------------
```

1. The Task API version, **v1beta1**.
2. The type of Kubernetes object, **Task**.
3. The unique name of this Task.
4. The list of parameters and Steps in the Task and the workspace used by the Task.

This Task starts the pod and runs a container inside that pod using the specified image to run the specified commands.

2.2.2.2. TaskRun

A *TaskRun* instantiates a Task for execution with specific inputs, outputs, and execution parameters on a cluster. It can be invoked on its own or as part of a PipelineRun for each Task in a pipeline.

A Task consists of one or more Steps that execute container images, and each container image performs a specific piece of build work. A TaskRun executes the Steps in a Task in the specified order, until all Steps execute successfully or a failure occurs. A TaskRun is automatically created by a PipelineRun for each Task in a Pipeline.

The following example shows a TaskRun that runs the **apply-manifests** Task with the relevant input parameters:

```
apiVersion: tekton.dev/v1beta1 (1)
kind: TaskRun (2)
metadata:
  name: apply-manifests-taskrun (3)
spec: (4)
  serviceAccountName: pipeline
  taskRef: (5)
    kind: Task
    name: apply-manifests
  workspaces: (6)
  - name: source
    persistentVolumeClaim:
      claimName: source-pvc
```

1. TaskRun API version **v1beta1**.
2. Specifies the type of Kubernetes object. In this example, **TaskRun**.
3. Unique name to identify this TaskRun.
4. Definition of the TaskRun. For this TaskRun, the Task and the required workspace are specified.
5. Name of the Task reference used for this TaskRun. This TaskRun executes the **apply-manifests** Task.
6. Workspace used by the TaskRun.

2.2.2.3. Pipelines

A *Pipeline* is a collection of **Task** resources arranged in a specific order of execution. They are executed to construct complex workflows that automate the build, deployment and delivery of applications. You can define a CI/CD workflow for your application using pipelines containing one or more tasks.

A **Pipeline** resource definition consists of a number of fields or attributes, which together enable the pipeline to accomplish a specific goal. Each **Pipeline** resource definition must contain at least one **Task** resource, which ingests specific inputs and produces specific outputs. The pipeline definition can also optionally include *Conditions*, *Workspaces*, *Parameters*, or *Resources* depending on the application requirements.

The following example shows the **build-and-deploy** pipeline, which builds an application image from a Git repository using the **buildah ClusterTask** resource:

```
apiVersion: tekton.dev/v1beta1 (1)
kind: Pipeline (2)
metadata:
  name: build-and-deploy (3)
spec: (4)
  workspaces: (5)
  - name: shared-workspace
  params: (6)
```

```
- name: deployment-name
  type: string
  description: name of the deployment to be patched
- name: git-url
  type: string
  description: url of the git repo for the code of deployment
- name: git-revision
  type: string
  description: revision to be used from repo of the code for deployment
  default: "pipelines-1.4"
- name: IMAGE
  type: string
  description: image to be built from the code
tasks: 7
- name: fetch-repository
  taskRef:
    name: git-clone
    kind: ClusterTask
  workspaces:
  - name: output
    workspace: shared-workspace
  params:
  - name: url
    value: $(params.git-url)
  - name: subdirectory
    value: ""
  - name: deleteExisting
    value: "true"
  - name: revision
    value: $(params.git-revision)
- name: build-image 8
  taskRef:
    name: buildah
    kind: ClusterTask
  params:
  - name: TLSVERIFY
    value: "false"
  - name: IMAGE
    value: $(params.IMAGE)
  workspaces:
  - name: source
    workspace: shared-workspace
  runAfter:
  - fetch-repository
- name: apply-manifests 9
  taskRef:
    name: apply-manifests
  workspaces:
  - name: source
    workspace: shared-workspace
  runAfter: 10
  - build-image
- name: update-deployment
  taskRef:
    name: update-deployment
  workspaces:
    - name: source
      workspace: shared-workspace
    params:
    - name: deployment
      value: $(params.deployment-name)
    - name: IMAGE
      value: $(params.IMAGE)
    runAfter:
    - apply-manifests
```

1. Pipeline API version **v1beta1**.
2. Specifies the type of Kubernetes object. In this example, **Pipeline**.
3. Unique name of this Pipeline.
4. Specifies the definition and structure of the Pipeline.
5. Workspaces used across all the Tasks in the Pipeline.
6. Parameters used across all the Tasks in the Pipeline.
7. Specifies the list of Tasks used in the Pipeline.
8. Task **build-image**, which uses the **buildah** ClusterTask to build application images from a given Git repository.
9. Task **apply-manifests**, which uses a user-defined Task with the same name.
10. Specifies the sequence in which Tasks are run in a Pipeline. In this example, the **apply-manifests** Task is run only after the **build-image** Task is completed.

2.2.2.4. PipelineRun

A *PipelineRun* is the running instance of a Pipeline. It instantiates a Pipeline for execution with specific inputs, outputs, and execution parameters on a cluster. A corresponding TaskRun is created for each Task automatically in the PipelineRun.

All the Tasks in the Pipeline are executed in the defined sequence until all Tasks are successful or a Task fails. The **status** field tracks and stores the progress of each TaskRun in the PipelineRun for monitoring and auditing purpose.

The following example shows a PipelineRun to run the **build-and-deploy** Pipeline with relevant resources and parameters:

```
apiVersion: tekton.dev/v1beta1 1
kind: PipelineRun 2
metadata:
  name: build-deploy-api-pipelinerun 3
spec:
  pipelineRef:
    name: build-and-deploy 4
  params: 5
  - name: deployment-name
    value: vote-api
```

```
- name: git-url
  value: https://github.com/openshift-pipelines/vote-api.git
- name: IMAGE
  value: image-registry.openshift-image-registry.svc:5000/pipelines-tutorial/vote-api
workspaces: 6
- name: shared-workspace
  volumeClaimTemplate:
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 500Mi
```

1. PipelineRun API version **v1beta1**.
2. Specifies the type of Kubernetes object. In this example, **PipelineRun**.
3. Unique name to identify this PipelineRun.
4. Name of the Pipeline to be run. In this example, **build-and-deploy**.
5. Specifies the list of parameters required to run the Pipeline.
6. Workspace used by the PipelineRun.

2.2.2.5. Workspaces

NOTE

It is recommended that you use Workspaces instead of PipelineResources in OpenShift Pipelines, as PipelineResources are difficult to debug, limited in scope, and make Tasks less reusable.

Workspaces declare shared storage volumes that a Task in a Pipeline needs at runtime to receive input or provide output. Instead of specifying the actual location of the volumes, Workspaces enable you to declare the filesystem or parts of the filesystem that would be required at runtime. A Task or Pipeline declares the Workspace and you must provide the specific location details of the volume. It is then mounted into that Workspace in a TaskRun or a PipelineRun. This separation of volume declaration from runtime storage volumes makes the Tasks reusable, flexible, and independent of the user environment.

With Workspaces, you can:

- Store Task inputs and outputs
- Share data among Tasks
- Use it as a mount point for credentials held in Secrets
- Use it as a mount point for configurations held in ConfigMaps
- Use it as a mount point for common tools shared by an organization
- Create a cache of build artifacts that speed up jobs

You can specify Workspaces in the TaskRun or PipelineRun using:

- A read-only ConfigMaps or Secret
- An existing PersistentVolumeClaim shared with other Tasks
- A PersistentVolumeClaim from a provided VolumeClaimTemplate
- An emptyDir that is discarded when the TaskRun completes

The following example shows a code snippet of the **build-and-deploy** Pipeline, which declares a **shared-workspace** Workspace for the **build-image** and **apply-manifests** Tasks as defined in the Pipeline.

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-and-deploy
spec:
  workspaces: 1
  - name: shared-workspace
  params:
...
  tasks: 2
  - name: build-image
   taskRef:
     name: buildah
     kind: ClusterTask
   params:
   - name: TLSVERIFY
     value: "false"
   - name: IMAGE
     value: $(params.IMAGE)
   workspaces: 3
   - name: source 4
     workspace: shared-workspace 5
   runAfter:
   - fetch-repository
 - name: apply-manifests
   taskRef:
     name: apply-manifests
   workspaces: 6
   - name: source
     workspace: shared-workspace
   runAfter:
    - build-image
...
```

1 List of Workspaces shared between the Tasks defined in the Pipeline. A Pipeline can define as many Workspaces as required. In this example, only one Workspace named **shared-workspace** is declared.

2 Definition of Tasks used in the Pipeline. This snippet defines two Tasks, **build-image** and **apply-manifests**, which share a common Workspace.

3 List of Workspaces used in the **build-image** Task. A Task definition can include as many

4 Name that uniquely identifies the Workspace used in the Task. This Task uses one Workspace named **source**.

5 Name of the Pipeline Workspace used by the Task. Note that the Workspace **source** in turn uses the Pipeline Workspace named **shared-workspace**.

6 List of Workspaces used in the **apply-manifests** Task. Note that this Task shares the **source** Workspace with the **build-image** Task.

Workspaces help tasks share data, and allow you to specify one or more volumes that each task in the pipeline requires during execution. You can create a persistent volume claim or provide a volume claim template that creates a persistent volume claim for you.

The following code snippet of the **build-deploy-api-pipelinerun** PipelineRun uses a volume claim template to create a persistent volume claim for defining the storage volume for the **shared-workspace** Workspace used in the **build-and-deploy** Pipeline.

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: build-deploy-api-pipelinerun
spec:
  pipelineRef:
    name: build-and-deploy
  params:
...

  workspaces: 1
  - name: shared-workspace 2
    volumeClaimTemplate: 3
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 500Mi
```

1 Specifies the list of Pipeline Workspaces for which volume binding will be provided in the PipelineRun.

2 The name of the Workspace in the Pipeline for which the volume is being provided.

3 Specifies a volume claim template that creates a persistent volume claim to define the storage volume for the workspace.

2.2.2.6. Triggers

Use *Triggers* in conjunction with pipelines to create a full-fledged CI/CD system where Kubernetes resources define the entire CI/CD execution. Triggers capture the external events, such as a Git pull request, and process them to extract key pieces of information. Mapping this event data to a set of predefined parameters triggers a series of tasks that can then create and deploy Kubernetes resources and instantiate the pipeline.

For example, you define a CI/CD workflow using Red Hat OpenShift Pipelines for your application. The pipeline must start for any new changes to take effect in the application repository. Triggers automate this process by capturing and processing any change event and by triggering a pipeline run that deploys the new image with the latest changes.

Triggers consist of the following main resources that work together to form a reusable, decoupled, and self-sustaining CI/CD system:

- The **TriggerBinding** resource validates events, extracts the fields from an event payload, and stores them as parameters.
  The following example shows a code snippet of the **TriggerBinding** resource, which extracts the Git repository information from the received event payload:

  ```
  apiVersion: triggers.tekton.dev/v1alpha1 1
  kind: TriggerBinding 2
  metadata:
    name: vote-app 3
  spec:
    params: 4
    - name: git-repo-url
      value: $(body.repository.url)
    - name: git-repo-name
      value: $(body.repository.name)
    - name: git-revision
      value: $(body.head_commit.id)
  ```

  1 The API version of the **TriggerBinding** resource. In this example, **v1alpha1**.

  2 Specifies the type of Kubernetes object. In this example, **TriggerBinding**.

  3 Unique name to identify the **TriggerBinding** resource.

  4 List of parameters which will be extracted from the received event payload and passed to the **TriggerTemplate** resource. In this example, the Git repository URL, name, and revision are extracted from the body of the event payload.

- The **TriggerTemplate** resource acts as a standard for the way resources must be created. It specifies the way parameterized data from the **TriggerBinding** resource should be used. A trigger template receives input from the trigger binding, and then performs a series of actions that results in creation of new pipeline resources, and initiation of a new pipeline run.
  The following example shows a code snippet of a **TriggerTemplate** resource, which creates a pipeline run using the Git repository information received from the **TriggerBinding** resource you just created:

  ```
  apiVersion: triggers.tekton.dev/v1alpha1 1
  kind: TriggerTemplate 2
  metadata:
    name: vote-app 3
  spec:
    params: 4
    - name: git-repo-url
      description: The git repository url
    - name: git-revision
      description: The git revision
      default: pipelines-1.4
  ```

```
  - name: git-repo-name
    description: The name of the deployment to be created / patched

  resourcetemplates: (5)
  - apiVersion: tekton.dev/v1beta1
    kind: PipelineRun
    metadata:
      name: build-deploy-$(tt.params.git-repo-name)-$(uid)
    spec:
      serviceAccountName: pipeline
      pipelineRef:
        name: build-and-deploy
      params:
      - name: deployment-name
        value: $(tt.params.git-repo-name)
      - name: git-url
        value: $(tt.params.git-repo-url)
      - name: git-revision
        value: $(tt.params.git-revision)
      - name: IMAGE
        value: image-registry.openshift-image-registry.svc:5000/pipelines-
tutorial/$(tt.params.git-repo-name)
      workspaces:
      - name: shared-workspace
        volumeClaimTemplate:
         spec:
          accessModes:
           - ReadWriteOnce
          resources:
            requests:
              storage: 500Mi
```

1. The API version of the **TriggerTemplate** resource. In this example, **v1alpha1**.
2. Specifies the type of Kubernetes object. In this example, **TriggerTemplate**.
3. Unique name to identify the **TriggerTemplate** resource.
4. Parameters supplied by the **TriggerBinding** or **EventListerner** resources.
5. List of templates that specify the way resources must be created using the parameters received through the **TriggerBinding** or **EventListener** resources.

- The **Trigger** resource connects the **TriggerBinding** and **TriggerTemplate** resources, and this **Trigger** resource is referenced in the **EventListener** specification.
  The following example shows a code snippet of a **Trigger** resource, named **vote-trigger** that connects the **TriggerBinding** and **TriggerTemplate** resources.

```
apiVersion: triggers.tekton.dev/v1alpha1 (1)
kind: Trigger (2)
metadata:
  name: vote-trigger (3)
spec:
  serviceAccountName: pipeline (4)
```

```
bindings:
  - ref: vote-app (5)
template: (6)
   ref: vote-app
```

1. The API version of the **Trigger** resource. In this example, **v1alpha1**.
2. Specifies the type of Kubernetes object. In this example, **Trigger**.
3. Unique name to identify the **Trigger** resource.
4. Service account name to be used.
5. Name of the **TriggerBinding** resource to be connected to the **TriggerTemplate** resource.
6. Name of the **TriggerTemplate** resource to be connected to the **TriggerBinding** resource.

- The **EventListener** resource provides an endpoint, or an event sink, that listens for incoming HTTP-based events with a JSON payload. It extracts event parameters from each **TriggerBinding** resource, and then processes this data to create Kubernetes resources as specified by the corresponding **TriggerTemplate** resource. The **EventListener** resource also performs lightweight event processing or basic filtering on the payload using event **interceptors**, which identify the type of payload and optionally modify it. Currently, pipeline triggers support four types of interceptors: *Webhook Interceptors*, *GitHub Interceptors*, *GitLab Interceptors*, and *Common Expression Language (CEL) Interceptors* .
  The following example shows an **EventListener** resource, which references the **Trigger** resource named **vote-trigger**.

```
apiVersion: triggers.tekton.dev/v1alpha1 (1)
kind: EventListener (2)
metadata:
  name: vote-app (3)
spec:
  serviceAccountName: pipeline (4)
  triggers:
    - triggerRef: vote-trigger (5)
```

1. The API version of the **EventListener** resource. In this example, **v1alpha1**.
2. Specifies the type of Kubernetes object. In this example, **EventListener**.
3. Unique name to identify the **EventListener** resource.
4. Service account name to be used.
5. Name of the **Trigger** resource referenced by the **EventListener** resource.

Triggers in Red Hat OpenShift Pipelines support both HTTP (insecure) and HTTPS (secure HTTP) connections to the **Eventlistener** resource. With the secure HTTPS connection, you get end-to-end secure connection within and outside the cluster. After you create a namespace, you can enable this secure HTTPS connection for the **Eventlistener** resource by adding the **operator.tekton.dev/enable-annotation=enabled** label to the namespace, and then creating a **Trigger** resource and a secured route using re-encrypt TLS termination.

2.2.3. Additional resources

- For information on installing pipelines, see Installing OpenShift Pipelines.
- For more details on creating custom CI/CD solutions, see Creating applications with CI/CD Pipelines.
- For more details on re-encrypt TLS termination, see Re-encryption Termination.
- For more details on secured routes, see the Secured routes section.

2.3. INSTALLING OPENSHIFT PIPELINES

This guide walks cluster administrators through the process of installing the Red Hat OpenShift Pipelines Operator to an OpenShift Container Platform cluster.

Prerequisites

- You have access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- You have installed **oc** CLI.
- You have installed OpenShift Pipelines (**tkn**) CLI on your local system.

2.3.1. Installing the Red Hat OpenShift Pipelines Operator in web console

You can install Red Hat OpenShift Pipelines using the Operator listed in the OpenShift Container Platform OperatorHub. When you install the Red Hat OpenShift Pipelines Operator, the custom resources (CRs) required for the pipelines configuration are automatically installed along with the Operator.

The default Operator custom resource definition (CRD) **config.operator.tekton.dev** is now replaced by **tektonconfigs.operator.tekton.dev**. In addition, the Operator provides the following additional CRDs to individually manage OpenShift Pipelines components: **tektonpipelines.operator.tekton.dev**, **tektontriggers.operator.tekton.dev** and **tektonaddons.operator.tekton.dev**.

If you have OpenShift Pipelines already installed on your cluster, the existing installation is seamlessly upgraded. The Operator will replace the instance of **config.operator.tekton.dev** on your cluster with an instance of **tektonconfigs.operator.tekton.dev** and additional objects of the other CRDs as necessary.

WARNING

If you manually changed your existing installation, such as, changing the target namespace in the **config.operator.tekton.dev** CRD instance by making changes to the **resource name - cluster** field, then the upgrade path is not smooth. In such cases, the recommended workflow is to uninstall your installation and reinstall the Red Hat OpenShift Pipelines Operator.

The Red Hat OpenShift Pipelines Operator now provides the option to choose the components that you want to install by specifying profiles as part of the **TektonConfig** CR. The **TektonConfig** CR is automatically installed when the Operator is installed. The supported profiles are:

- Basic: This installs only Tekton pipelines.
- Default: This installs Tekton pipeline and Tekton triggers.
- All: This is the default profile used when the **TektonConfig** CR is installed. This profile installs all of the Tekton components: Tekton Pipelines, Tekton Triggers, Tekton Addons(which include **ClusterTasks**, **ClusterTriggerBindings**, **ConsoleCLIDownload**, **ConsoleQuickStart** and **ConsoleYAMLSample** resources).

Procedure

1. In the **Administrator** perspective of the web console, navigate to **Operators → OperatorHub**.
2. Use the **Filter by keyword** box to search for **Red Hat OpenShift Pipelines** Operator in the catalog. Click the **Red Hat OpenShift Pipelines** Operator tile.
3. Read the brief description about the Operator on the **Red Hat OpenShift Pipelines** Operator page. Click **Install**.
4. On the **Install Operator** page:
   a. Select **All namespaces on the cluster (default)** for the **Installation Mode**. This mode installs the Operator in the default **openshift-operators** namespace, which enables the Operator to watch and be made available to all namespaces in the cluster.
   b. Select **Automatic** for the **Approval Strategy**. This ensures that the future upgrades to the Operator are handled automatically by the Operator Lifecycle Manager (OLM). If you select the **Manual** approval strategy, OLM creates an update request. As a cluster administrator, you must then manually approve the OLM update request to update the Operator to the new version.
   c. Select an **Update Channel**.
      - The **Stable** channel enables installation of the latest stable and supported release of the Red Hat OpenShift Pipelines Operator.
      - The **preview** channel enables installation of the latest preview version of the Red Hat OpenShift Pipelines Operator, which may contain features that are not yet available from the **Stable** channel and is not supported.
5. Click **Install**. You will see the Operator listed on the **Installed Operators** page.

   NOTE

   The Operator is installed automatically into the **openshift-operators** namespace.

6. Verify that the **Status** is set to **Succeeded Up to date** to confirm successful installation of Red Hat OpenShift Pipelines Operator.

2.3.2. Installing the OpenShift Pipelines Operator using the CLI

You can install Red Hat OpenShift Pipelines Operator from the OperatorHub using the CLI.

Procedure

1. Create a Subscription object YAML file to subscribe a namespace to the Red Hat OpenShift Pipelines Operator, for example, **sub.yaml**:

   Example Subscription

   ```
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: openshift-pipelines-operator
     namespace: openshift-operators
   spec:
     channel:  <channel name> 1
     name: openshift-pipelines-operator-rh 2
     source: redhat-operators 3
     sourceNamespace: openshift-marketplace 4
   ```

   1 Specify the channel name from where you want to subscribe the Operator

   2 Name of the Operator to subscribe to.

   3 Name of the CatalogSource that provides the Operator.

   4 Namespace of the CatalogSource. Use **openshift-marketplace** for the default OperatorHub CatalogSources.

2. Create the Subscription object:

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

   The Red Hat OpenShift Pipelines Operator is now installed in the default target namespace **openshift-operators**.

2.3.3. Red Hat OpenShift Pipelines Operator in a restricted environment

The Red Hat OpenShift Pipelines Operator enables support for installation of pipelines in a restricted network environment.

The Operator installs a proxy webhook that sets the proxy environment variables in the containers of the pod created by tekton-controllers based on the **cluster** proxy object. It also sets the proxy environment variables in the **TektonPipelines**, **TektonTriggers**, **Controllers**, **Webhooks**, and **Operator Proxy Webhook** resources.

By default, the proxy webhook is disabled for the **openshift-pipelines** namespace. To disable it for any other namespace, you can add the **operator.tekton.dev/disable-proxy: true** label to the **namespace** object.

2.3.4. Additional Resources

- You can learn more about installing Operators on OpenShift Container Platform in the adding Operators to a cluster section.
- For more information on using pipelines in a restricted environment see:

- Mirroring images to run pipelines in a restricted environment
- Configuring Samples Operator for a restricted cluster
- Creating a cluster with a mirrored registry
- Mirroring a supported builder image

2.4. UNINSTALLING OPENSHIFT PIPELINES

Uninstalling the Red Hat OpenShift Pipelines Operator is a two-step process:

1. Delete the Custom Resources (CRs) that were added by default when you installed the Red Hat OpenShift Pipelines Operator.
2. Uninstall the Red Hat OpenShift Pipelines Operator.

Uninstalling only the Operator will not remove the Red Hat OpenShift Pipelines components created by default when the Operator is installed.

2.4.1. Deleting the Red Hat OpenShift Pipelines components and Custom Resources

Delete the Custom Resources (CRs) created by default during installation of the Red Hat OpenShift Pipelines Operator.

Procedure

1. In the **Administrator** perspective of the web console, navigate to **Administration → Custom Resource Definition**.
2. Type **config.operator.tekton.dev** in the **Filter by name** box to search for the Red Hat OpenShift Pipelines Operator CRs.
3. Click **CRD Config** to see the **Custom Resource Definition Details** page.
4. Click the **Actions** drop-down menu and select **Delete Custom Resource Definition**

NOTE

Deleting the CRs will delete the Red Hat OpenShift Pipelines components, and all the Tasks and Pipelines on the cluster will be lost.

5. Click **Delete** to confirm the deletion of the CRs.

2.4.2. Uninstalling the Red Hat OpenShift Pipelines Operator

Procedure

1. From the **Operators → OperatorHub** page, use the **Filter by keyword** box to search for **Red Hat OpenShift Pipelines Operator**.
2. Click the **OpenShift Pipelines Operator** tile. The Operator tile indicates it is installed.
3. In the **OpenShift Pipelines Operator** descriptor page, click **Uninstall**.

Additional Resources

- You can learn more about uninstalling Operators on OpenShift Container Platform in the deleting Operators from a cluster section.

2.5. CREATING CI/CD SOLUTIONS FOR APPLICATIONS USING OPENSHIFT PIPELINES

With Red Hat OpenShift Pipelines, you can create a customized CI/CD solution to build, test, and deploy your application.

To create a full-fledged, self-serving CI/CD pipeline for an application, perform the following tasks:

- Create custom tasks, or install existing reusable tasks.
- Create and define the delivery pipeline for your application.
- Provide a storage volume or filesystem that is attached to a workspace for the pipeline execution, using one of the following approaches:
  - Specify a volume claim template that creates a persistent volume claim
  - Specify a persistent volume claim
- Create a **PipelineRun** object to instantiate and invoke the pipeline.
- Add triggers to capture events in the source repository.

This section uses the **pipelines-tutorial** example to demonstrate the preceding tasks. The example uses a simple application which consists of:

- A front-end interface, **pipelines-vote-ui**, with the source code in the **pipelines-vote-ui** Git repository.
- A back-end interface, **pipelines-vote-api**, with the source code in the **pipelines-vote-api** Git repository.
- The **apply-manifests** and **update-deployment** tasks in the **pipelines-tutorial** Git repository.

2.5.1. Prerequisites

- You have access to an OpenShift Container Platform cluster.
- You have installed OpenShift Pipelines using the Red Hat OpenShift Pipelines Operator listed in the OpenShift OperatorHub. Once installed, it is applicable to the entire cluster.
- You have installed OpenShift Pipelines CLI.
- You have forked the front-end **pipelines-vote-ui** and back-end **pipelines-vote-api** Git repositories using your GitHub ID, and have administrator access to these repositories.
- Optional: You have cloned the **pipelines-tutorial** Git repository.

2.5.2. Creating a project and checking your pipeline service account

Procedure

1. Log in to your OpenShift Container Platform cluster:

```
$ oc login -u <login> -p <password> https://openshift.example.com:6443
```

2. Create a project for the sample application. For this example workflow, create the **pipelines-tutorial** project:

```
$ oc new-project pipelines-tutorial
```

NOTE

If you create a project with a different name, be sure to update the resource URLs used in the example with your project name.

3. View the **pipeline** service account:
Red Hat OpenShift Pipelines Operator adds and configures a service account named **pipeline** that has sufficient permissions to build and push an image. This service account is used by the **PipelineRun** object.

```
$ oc get serviceaccount pipeline
```

2.5.3. Creating pipeline tasks

Procedure

1. Install the **apply-manifests** and **update-deployment** task resources from the **pipelines-tutorial** repository, which contains a list of reusable tasks for pipelines:

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/01_pipeline/01_apply_manifest_task.yaml
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/01_pipeline/02_update_deployment_task.yaml
```

2. Use the **tkn task list** command to list the tasks you created:

```
$ tkn task list
```

The output verifies that the **apply-manifests** and **update-deployment** task resources were created:

```
NAME                DESCRIPTION   AGE
apply-manifests                   1 minute ago
update-deployment                 48 seconds ago
```

3. Use the **tkn clustertasks list** command to list the Operator-installed additional cluster tasks such as **buildah** and **s2i-python**:

NOTE

To use the **buildah** cluster task in a restricted environment, you must ensure that the Dockerfile uses an internal image stream as the base image.

```
$ tkn clustertasks list
```

The output lists the Operator-installed **ClusterTask** resources:

```
NAME                  DESCRIPTION   AGE
buildah                       1 day ago
git-clone                     1 day ago
s2i-python                     1 day ago
tkn                         1 day ago
```

2.5.4. Assembling a pipeline

A pipeline represents a CI/CD flow and is defined by the tasks to be executed. It is designed to be generic and reusable in multiple applications and environments.

A pipeline specifies how the tasks interact with each other and their order of execution using the **from** and **runAfter** parameters. It uses the **workspaces** field to specify one or more volumes that each task in the pipeline requires during execution.

In this section, you will create a pipeline that takes the source code of the application from GitHub, and then builds and deploys it on OpenShift Container Platform.

The pipeline performs the following tasks for the back-end application **pipelines-vote-api** and front-end application **pipelines-vote-ui**:

- Clones the source code of the application from the Git repository by referring to the **git-url** and **git-revision** parameters.
- Builds the container image using the **buildah** cluster task.
- Pushes the image to the internal image registry by referring to the **image** parameter.
- Deploys the new image on OpenShift Container Platform by using the **apply-manifests** and **update-deployment** tasks.

Procedure

1. Copy the contents of the following sample pipeline YAML file and save it:

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-and-deploy
spec:
  workspaces:
  - name: shared-workspace
  params:
  - name: deployment-name
    type: string
    description: name of the deployment to be patched
  - name: git-url
    type: string
    description: url of the git repo for the code of deployment
  - name: git-revision
    type: string
    description: revision to be used from repo of the code for deployment
    default: "pipelines-1.4"
  - name: IMAGE
    type: string
    description: image to be built from the code
  tasks:
  - name: fetch-repository
    taskRef:
      name: git-clone
      kind: ClusterTask
    workspaces:
    - name: output
      workspace: shared-workspace
    params:
    - name: url
      value: $(params.git-url)
    - name: subdirectory
      value: ""
    - name: deleteExisting
      value: "true"
    - name: revision
      value: $(params.git-revision)
  - name: build-image
    taskRef:
      name: buildah
      kind: ClusterTask
    params:
    - name: IMAGE
      value: $(params.IMAGE)
    workspaces:
    - name: source
      workspace: shared-workspace
    runAfter:
    - fetch-repository
  - name: apply-manifests
    taskRef:
      name: apply-manifests
    workspaces:
    - name: source
      workspace: shared-workspace
    runAfter:
    - build-image
  - name: update-deployment
    taskRef:
      name: update-deployment
    params:
    - name: deployment
      value: $(params.deployment-name)
    - name: IMAGE
```

```
      value: $(params.IMAGE)
    runAfter:
    - apply-manifests
```

The pipeline definition abstracts away the specifics of the Git source repository and image registries. These details are added as **params** when a pipeline is triggered and executed.

2. Create the pipeline:

```
$ oc create -f <pipeline-yaml-file-name.yaml>
```

Alternatively, you can also execute the YAML file directly from the Git repository:

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-
1.4/01_pipeline/04_pipeline.yaml
```

3. Use the **tkn pipeline list** command to verify that the pipeline is added to the application:

```
$ tkn pipeline list
```

The output verifies that the **build-and-deploy** pipeline was created:

```
NAME              AGE            LAST RUN   STARTED   DURATION   STATUS
build-and-deploy   1 minute ago   ---        ---       ---        ---
```

2.5.5. Mirroring images to run pipelines in a restricted environment

To run OpenShift Pipelines in a disconnected cluster or a cluster provisioned in a restricted environment, ensure that either the Samples Operator is configured for a restricted network, or a cluster administrator has created a cluster with a mirrored registry.

The following procedure uses the **pipelines-tutorial** example to create a pipeline for an application in a restricted environment using a cluster with a mirrored registry. To ensure that the **pipelines-tutorial** example works in a restricted environment, you must mirror the respective builder images from the mirror registry for the front-end interface, **pipelines-vote-ui**; back-end interface, **pipelines-vote-api**; and the **cli**.

Procedure

1. Mirror the builder image from the mirror registry for the front-end interface, **pipelines-vote-ui**.

   a. Verify that the required images tag is not imported:

   ```
   $ oc describe imagestream python -n openshift
   ```

   #### Example output

   ```
   Name:   python
   Namespace:  openshift
   [...]

   3.8-ubi8 (latest)
     tagged from registry.redhat.io/ubi8/python-38:latest
   ```

```
    prefer registry pullthrough when referencing this tag

  Build and run Python 3.8 applications on UBI 8. For more information about using this
builder image, including OpenShift considerations, see https://github.com/sclorg/s2i-
python-container/blob/master/3.8/README.md.
  Tags: builder, python
  Supports: python:3.8, python
  Example Repo: https://github.com/sclorg/django-ex.git

[...]
```

b. Mirror the supported image tag to the private registry:

```
$ oc image mirror registry.redhat.io/ubi8/python-38:latest <mirror-registry>:
<port>/ubi8/python-38
```

c. Import the image:

```
$ oc tag <mirror-registry>:<port>/ubi8/python-38 python:latest --scheduled -n openshift
```

You must periodically re-import the image. The **--scheduled** flag enables automatic re-import of the image.

d. Verify that the images with the given tag have been imported:

```
$ oc describe imagestream python -n openshift
```

Example output

```
Name:   python
Namespace:  openshift
[...]

latest
  updates automatically from registry <mirror-registry>:<port>/ubi8/python-38

  * <mirror-registry>:<port>/ubi8/python-
38@sha256:3ee3c2e70251e75bfeac25c0c33356add9cc4abcbc9c51d858f39e4dc29c5f58


[...]
```

2. Mirror the builder image from the mirror registry for the back-end interface, **pipelines-vote-api**.

a. Verify that the required images tag is not imported:

```
$ oc describe imagestream golang -n openshift
```

Example output

```
Name:   golang
Namespace:  openshift
[...]
```

```
1.14.7-ubi8 (latest)
  tagged from registry.redhat.io/ubi8/go-toolset:1.14.7
    prefer registry pullthrough when referencing this tag

  Build and run Go applications on UBI 8. For more information about using this builder
image, including OpenShift considerations, see https://github.com/sclorg/golang-
container/blob/master/README.md.
  Tags: builder, golang, go
  Supports: golang
  Example Repo: https://github.com/sclorg/golang-ex.git

[...]
```

b. Mirror the supported image tag to the private registry:

```
$ oc image mirror registry.redhat.io/ubi8/go-toolset:1.14.7 <mirror-registry>:
<port>/ubi8/go-toolset
```

c. Import the image:

```
$ oc tag <mirror-registry>:<port>/ubi8/go-toolset golang:latest --scheduled -n openshift
```

You must periodically re-import the image. The **--scheduled** flag enables automatic re-import of the image.

d. Verify that the images with the given tag have been imported:

```
$ oc describe imagestream golang -n openshift
```

Example output

```
Name:   golang
Namespace:  openshift
[...]

latest
  updates automatically from registry <mirror-registry>:<port>/ubi8/go-toolset

  * <mirror-registry>:<port>/ubi8/go-
toolset@sha256:59a74d581df3a2bd63ab55f7ac106677694bf612a1fe9e7e3e1487f55c421
b37

[...]
```

3. Mirror the builder image from the mirror registry for the **cli**.

a. Verify that the required images tag is not imported:

```
$ oc describe imagestream cli -n openshift
```

Example output

```
Name:                  cli
Namespace:                 openshift
```

```
[...]

latest
  updates automatically from registry quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551


  * quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551


[...]
```

b. Mirror the supported image tag to the private registry:

```
$ oc image mirror quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551
<mirror-registry>:<port>/openshift-release-dev/ocp-v4.0-art-dev:latest
```

c. Import the image:

```
$ oc tag <mirror-registry>:<port>/openshift-release-dev/ocp-v4.0-art-dev cli:latest --
scheduled -n openshift
```

You must periodically re-import the image. The **--scheduled** flag enables automatic re-import of the image.

d. Verify that the images with the given tag have been imported:

```
$ oc describe imagestream cli -n openshift
```

Example output

```
Name:              cli
Namespace:              openshift
[...]

latest
  updates automatically from registry <mirror-registry>:<port>/openshift-release-dev/ocp-
v4.0-art-dev

  * <mirror-registry>:<port>/openshift-release-dev/ocp-v4.0-art-
dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551


[...]
```

Additional resources

- Configuring Samples Operator for a restricted cluster
- Creating a cluster with a mirrored registry
- Mirroring a supported builder image

2.5.6. Running a pipeline

A **PipelineRun** resource starts a pipeline and ties it to the Git and image resources that should be used for the specific invocation. It automatically creates and starts the **TaskRun** resources for each task in the pipeline.

Procedure

1. Start the pipeline for the back-end application:

   ```
   $ tkn pipeline start build-and-deploy \
       -w name=shared-
   workspace,volumeClaimTemplateFile=https://raw.githubusercontent.com/openshift/pipelines-
   tutorial/pipelines-1.4/01_pipeline/03_persistent_volume_claim.yaml \
       -p deployment-name=pipelines-vote-api \
       -p git-url=https://github.com/openshift/pipelines-vote-api.git \
       -p IMAGE=image-registry.openshift-image-registry.svc:5000/pipelines-tutorial/pipelines-
   vote-api
   ```

   The previous command uses a volume claim template, which creates a persistent volume claim for the pipeline execution.

2. To track the progress of the pipeline run, enter the following command::

   ```
   $ tkn pipelinerun logs <pipelinerun_id> -f
   ```

   The <pipelinerun_id> in the above command is the ID for the **PipelineRun** that was returned in the output of the previous command.

3. Start the pipeline for the front-end application:

   ```
   $ tkn pipeline start build-and-deploy \
       -w name=shared-
   workspace,volumeClaimTemplateFile=https://raw.githubusercontent.com/openshift/pipelines-
   tutorial/pipelines-1.4/01_pipeline/03_persistent_volume_claim.yaml \
       -p deployment-name=pipelines-vote-ui \
       -p git-url=https://github.com/openshift/pipelines-vote-ui.git \
       -p IMAGE=image-registry.openshift-image-registry.svc:5000/pipelines-tutorial/pipelines-
   vote-ui
   ```

4. To track the progress of the pipeline run, enter the following command:

   ```
   $ tkn pipelinerun logs <pipelinerun_id> -f
   ```

   The <pipelinerun_id> in the above command is the ID for the **PipelineRun** that was returned in the output of the previous command.

5. After a few minutes, use **tkn pipelinerun list** command to verify that the pipeline ran successfully by listing all the pipeline runs:

   ```
   $ tkn pipelinerun list
   ```

   The output lists the pipeline runs:

```
NAME                        STARTED      DURATION     STATUS
build-and-deploy-run-xy7rw  1 hour ago   2 minutes    Succeeded
build-and-deploy-run-z2rz8  1 hour ago   19 minutes   Succeeded
```

6. Get the application route:

```
$ oc get route pipelines-vote-ui --template='http://{{.spec.host}}'
```

   Note the output of the previous command. You can access the application using this route.

7. To rerun the last pipeline run, using the pipeline resources and service account of the previous pipeline, run:

```
$ tkn pipeline start build-and-deploy --last
```

2.5.7. Adding triggers to a pipeline

Triggers enable pipelines to respond to external GitHub events, such as push events and pull requests. After you assemble and start a pipeline for the application, add the **TriggerBinding**, **TriggerTemplate**, **Trigger**, and **EventListener** resources to capture the GitHub events.

Procedure

1. Copy the content of the following sample **TriggerBinding** YAML file and save it:

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: TriggerBinding
metadata:
  name: vote-app
spec:
  params:
  - name: git-repo-url
    value: $(body.repository.url)
  - name: git-repo-name
    value: $(body.repository.name)
  - name: git-revision
    value: $(body.head_commit.id)
```

2. Create the **TriggerBinding** resource:

```
$ oc create -f <triggerbinding-yaml-file-name.yaml>
```

   Alternatively, you can create the **TriggerBinding** resource directly from the **pipelines-tutorial** Git repository:

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-
1.4/03_triggers/01_binding.yaml
```

3. Copy the content of the following sample **TriggerTemplate** YAML file and save it:

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: TriggerTemplate
metadata:
  name: vote-app
spec:
  params:
  - name: git-repo-url
    description: The git repository url
  - name: git-revision
    description: The git revision
    default: pipelines-1.4
  - name: git-repo-name
    description: The name of the deployment to be created / patched

  resourcetemplates:
  - apiVersion: tekton.dev/v1beta1
    kind: PipelineRun
    metadata:
      generateName: build-deploy-$(tt.params.git-repo-name)-
    spec:
      serviceAccountName: pipeline
      pipelineRef:
        name: build-and-deploy
      params:
      - name: deployment-name
        value: $(tt.params.git-repo-name)
      - name: git-url
        value: $(tt.params.git-repo-url)
      - name: git-revision
        value: $(tt.params.git-revision)
      - name: IMAGE
        value: image-registry.openshift-image-registry.svc:5000/pipelines-
tutorial/$(tt.params.git-repo-name)
      workspaces:
      - name: shared-workspace
        volumeClaimTemplate:
          spec:
            accessModes:
              - ReadWriteOnce
            resources:
              requests:
                storage: 500Mi
```

The template specifies a volume claim template to create a persistent volume claim for defining the storage volume for the workspace. Therefore, you do not need to create a persistent volume claim to provide data storage.

4. Create the **TriggerTemplate** resource:

```
$ oc create -f <triggertemplate-yaml-file-name.yaml>
```

Alternatively, you can create the **TriggerTemplate** resource directly from the **pipelines-tutorial** Git repository:

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-
1.4/03_triggers/02_template.yaml
```

5. Copy the contents of the following sample **Trigger** YAML file and save it:

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: Trigger
metadata:
  name: vote-trigger
spec:
  serviceAccountName: pipeline
  bindings:
    - ref: vote-app
  template:
    ref: vote-app
```

6. Create the **Trigger** resource:

```
$ oc create -f <trigger-yaml-file-name.yaml>
```

Alternatively, you can create the **Trigger** resource directly from the **pipelines-tutorial** Git repository:

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-
1.4/03_triggers/03_trigger.yaml
```

7. Copy the contents of the following sample **EventListener** YAML file and save it:

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: EventListener
metadata:
  name: vote-app
spec:
  serviceAccountName: pipeline
  triggers:
    - triggerRef: vote-trigger
```

Alternatively, if you have not defined a trigger custom resource, add the binding and template spec to the **EventListener** YAML file, instead of referring to the name of the trigger:

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: EventListener
metadata:
  name: vote-app
spec:
  serviceAccountName: pipeline
  triggers:
  - bindings:
    - ref: vote-app
    template:
      ref: vote-app
```

8. Create the **EventListener** resource by performing the following steps:

- To create an **EventListener** resource using a secure HTTPS connection:

  a. Add a label to enable the secure HTTPS connection to the **Eventlistener** resource:

```
$ oc label namespace <ns-name> operator.tekton.dev/enable-annotation=enabled
```

b. Create the **EventListener** resource:

```
$ oc create -f <eventlistener-yaml-file-name.yaml>
```

Alternatively, you can create the **EvenListener** resource directly from the **pipelines-tutorial** Git repository:

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/03_triggers/04_event_listener.yaml
```

c. Create a route with the re-encrypt TLS termination:

```
$ oc create route reencrypt --service=<svc-name> --cert=tls.crt --key=tls.key --ca-cert=ca.crt --hostname=<hostname>
```

Alternatively, you can create a re-encrypt TLS termination YAML file to create a secured route.

Example Re-encrypt TLS Termination YAML of the Secured Route

```
apiVersion: v1
kind: Route
metadata:
  name: route-passthrough-secured (1)
spec:
  host: <hostname>
  to:
    kind: Service
    name: frontend (2)
  tls:
    termination: reencrypt        (3)
    key: [as in edge termination]
    certificate: [as in edge termination]
    caCertificate: [as in edge termination]
    destinationCACertificate: |-  (4)
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

(1) (2) The name of the object, which is limited to 63 characters.

(3) The **termination** field is set to **reencrypt**. This is the only required **tls** field.

(4) Required for re-encryption. **destinationCACertificate** specifies a CA certificate to validate the endpoint certificate, securing the connection from the router to the destination pods. If the service is using a service signing certificate, or the administrator has specified a default CA certificate for the router and the service has a certificate signed by that CA, this field can be omitted.

See **oc create route reencrypt --help** for more options.

- To create an **EventListener** resource using an insecure HTTP connection:

a. Create the **EventListener** resource.

b. Expose the **EventListener** service as an OpenShift Container Platform route to make it publicly accessible:

```
$ oc expose svc el-vote-app
```

2.5.8. Creating webhooks

Webhooks are HTTP POST messages that are received by the event listeners whenever a configured event occurs in your repository. The event payload is then mapped to trigger bindings, and processed by trigger templates. The trigger templates eventually start one or more pipeline runs, leading to the creation and deployment of Kubernetes resources.

In this section, you will configure a webhook URL on your forked Git repositories **pipelines-vote-ui** and **pipelines-vote-api**. This URL points to the publicly accessible **EventListener** service route.

NOTE

Adding webhooks requires administrative privileges to the repository. If you do not have administrative access to your repository, contact your system administrator for adding webhooks.

Procedure

1. Get the webhook URL:

   - For a secure HTTPS connection:

     ```
     $ echo "URL: $(oc  get route el-vote-app --template='https://{{.spec.host}}')"
     ```

   - For an HTTP (insecure) connection:

     ```
     $ echo "URL: $(oc  get route el-vote-app --template='http://{{.spec.host}}')"
     ```

     Note the URL obtained in the output.

2. Configure webhooks manually on the front-end repository:

   a. Open the front-end Git repository **pipelines-vote-ui** in your browser.

   b. Click **Settings → Webhooks → Add Webhook**

   c. On the **Webhooks/Add Webhook** page:

      i. Enter the webhook URL from step 1 in **Payload URL** field

      ii. Select **application/json** for the **Content type**

      iii. Specify the secret in the **Secret** field

      iv. Ensure that the **Just the push event**is selected

      v. Select **Active**

      vi. Click **Add Webhook**

3. Repeat step 2 for the back-end repository **pipelines-vote-api**.

2.5.9. Triggering a pipeline run

Whenever a **push** event occurs in the Git repository, the configured webhook sends an event payload to the publicly exposed **EventListener** service route. The **EventListener** service of the application processes the payload, and passes it to the relevant **TriggerBinding** and **TriggerTemplate** resource pairs. The **TriggerBinding** resource extracts the parameters, and the **TriggerTemplate** resource uses these parameters and specifies the way the resources must be created. This may rebuild and redeploy the application.

In this section, you push an empty commit to the front-end **pipelines-vote-ui** repository, which then triggers the pipeline run.

Procedure

1. From the terminal, clone your forked Git repository **pipelines-vote-ui**:

```
$ git clone git@github.com:<your GitHub ID>/pipelines-vote-ui.git -b pipelines-1.4
```

2. Push an empty commit:

```
$ git commit -m "empty-commit" --allow-empty && git push origin pipelines-1.4
```

3. Check if the pipeline run was triggered:

```
$ tkn pipelinerun list
```

   Notice that a new pipeline run was initiated.

2.5.10. Additional resources

- For more details on pipelines in the **Developer** perspective, see the working with pipelines in the **Developer** perspective section.
- To learn more about Security Context Constraints (SCCs), see the Managing Security Context Constraints section.
- For more examples of reusable tasks, see the OpenShift Catalog repository. Additionally, you can also see the Tekton Catalog in the Tekton project.
- For more details on re-encrypt TLS termination, see Re-encryption Termination.
- For more details on secured routes, see the Secured routes section.

2.6. REDUCING RESOURCE CONSUMPTION OF PIPELINES

If you use clusters in multi-tenant environments you must control the consumption of CPU, memory, and storage resources for each project and Kubernetes object. This helps prevent any one application from consuming too many resources and affecting other applications.

To define the final resource limits that are set on the resulting pods, Red Hat OpenShift Pipelines use resource quota limits and limit ranges of the project in which they are executed.

To restrict resource consumption in your project, you can:

- Set and manage resource quotas to limit the aggregate resource consumption.
- Use limit ranges to restrict resource consumption for specific objects, such as pods, images, image streams, and persistent volume claims.

2.6.1. Understanding resource consumption in pipelines

Each task consists of a number of required steps to be executed in a particular order defined in the **steps** field of the **Task** resource. Every task runs as a pod, and each step runs as a container within that pod.

Steps are executed one at a time. The pod that executes the task only requests enough resources to run a single container image (step) in the task at a time, and thus does not store resources for all the steps in the task.

The **Resources** field in the **steps** spec specifies the limits for resource consumption. By default, the resource requests for the CPU, memory, and ephemeral storage are set to **BestEffort** (zero) values or to the minimums set through limit ranges in that project.

Example configuration of resource requests and limits for a step

```
spec:
  steps:
  - name: <step_name>
    resources:
      requests:
        memory: 2Gi
        cpu: 600m
      limits:
        memory: 4Gi
        cpu: 900m
```

When the **LimitRange** parameter and the minimum values for container resource requests are specified in the project in which the pipeline and task runs are executed, Red Hat OpenShift Pipelines looks at all the **LimitRange** values in the project and uses the minimum values instead of zero.

Example configuration of limit range parameters at a project level

```
apiVersion: v1
kind: LimitRange
metadata:
  name: <limit_container_resource>
spec:
  limits:
  - max:
      cpu: "600m"
      memory: "2Gi"
    min:
      cpu: "200m"
      memory: "100Mi"
    default:
      cpu: "500m"
      memory: "800Mi"
    defaultRequest:
      cpu: "100m"
      memory: "100Mi"
    type: Container
...
```

2.6.2. Mitigating extra resource consumption in pipelines

When you have resource limits set on the containers in your pod, OpenShift Container Platform sums up the resource limits requested as all containers run simultaneously.

To consume the minimum amount of resources needed to execute one step at a time in the invoked task, Red Hat OpenShift Pipelines requests the maximum CPU, memory, and ephemeral storage as specified in the step that requires the most amount of resources. This ensures that the resource requirements of all the steps are met. Requests other than the maximum values are set to zero.

However, this behavior can lead to higher resource usage than required. If you use resource quotas, this could also lead to unschedulable pods.

For example, consider a task with two steps that uses scripts, and that does not define any resource limits and requests. The resulting pod has two init containers (one for entrypoint copy, the other for writing scripts) and two containers, one for each step.

OpenShift Container Platform uses the limit range set up for the project to compute required resource requests and limits. For this example, set the following limit range in the project:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
spec:
  limits:
  - max:
      memory: 1Gi
    min:
      memory: 500Mi
    type: Container
```

In this scenario, each init container uses a request memory of 1Gi (the max limit of the limit range), and each container uses a request memory of 500Mi. Thus, the total memory request for the pod is 2Gi.

If the same limit range is used with a task of ten steps, the final memory request is 5Gi, which is higher than what each step actually needs, that is 500Mi (since each step runs after the other).

Thus, to reduce resource consumption of resources, you can:

- Reduce the number of steps in a given task by grouping different steps into one bigger step, using the script feature, and the same image. This reduces the minimum requested resource.

- Distribute steps that are relatively independent of each other and can run on their own to multiple tasks instead of a single task. This lowers the number of steps in each task, making the request for each task smaller, and the scheduler can then run them when the resources are available.

2.6.3. Additional Resources

- Resource Quotas
- Restricting resource consumption using limit ranges
- Resource requests and limits in Kubernetes

2.7. WORKING WITH RED HAT OPENSHIFT PIPELINES USING THE DEVELOPER PERSPECTIVE

You can use the **Developer** perspective of the OpenShift Container Platform web console to create CI/CD pipelines for your software delivery process.

In the **Developer** perspective:

- Use the **Add** → **Pipeline** → **Pipeline Builder** option to create customized pipelines for your application.
- Use the **Add** → **From Git** option to create pipelines using operator-installed pipeline templates and resources while creating an application on OpenShift Container Platform.

After you create the pipelines for your application, you can view and visually interact with the deployed pipelines in the **Pipelines** view. You can also use the **Topology** view to interact with the pipelines created using the **From Git** option. You need to apply custom labels to a pipeline created using the **Pipeline Builder** to see it in the **Topology** view.

Prerequisites

- You have access to an OpenShift Container Platform cluster and have switched to the **Developer** perspective in the web console.
- You have the OpenShift Pipelines Operator installed in your cluster.
- You are a cluster administrator or a user with create and edit permissions.
- You have created a project.

2.7.1. Constructing Pipelines using the Pipeline Builder

In the **Developer** perspective of the console, you can use the **+Add** → **Pipeline** → **Pipeline builder** option to:

- Configure pipelines using either the **Pipeline builder** or the **YAML view**.
- Construct a pipeline flow using existing tasks and cluster tasks. When you install the OpenShift Pipelines Operator, it adds reusable pipeline cluster tasks to your cluster.
- Specify the type of resources required for the pipeline run, and if required, add additional parameters to the pipeline.
- Reference these pipeline resources in each of the tasks in the pipeline as input and output resources.
- If required, reference any additional parameters added to the pipeline in the task. The parameters for a task are prepopulated based on the specifications of the task.
- Use the Operator-installed, reusable snippets and samples to create detailed pipelines.

Procedure

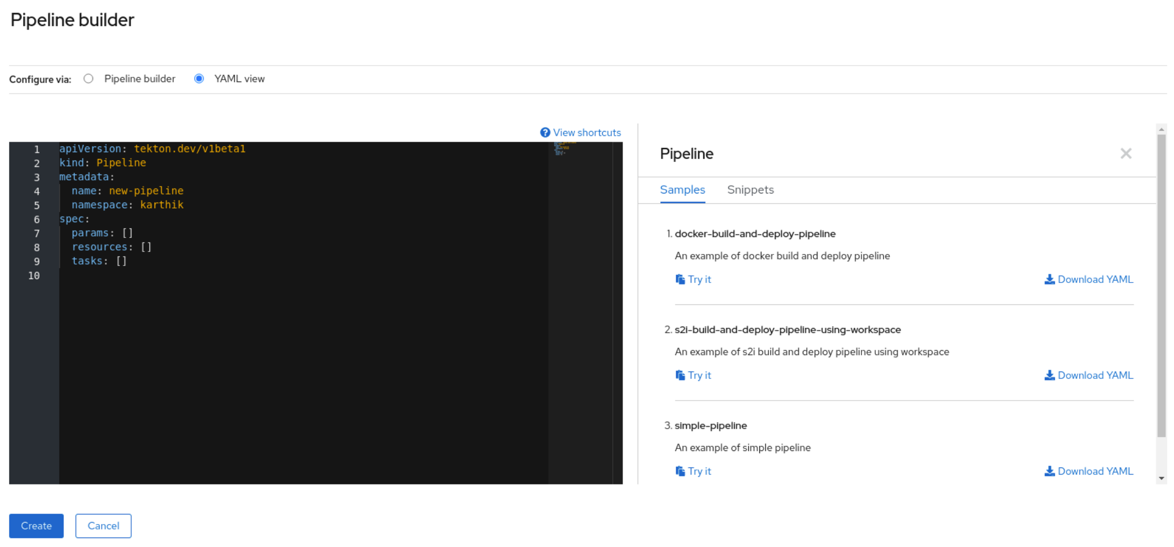
1. In the **+Add** view of the **Developer** perspective, click the **Pipeline** tile to see the **Pipeline builder** page.

2. Configure the pipeline using either the **Pipeline builder** view or the **YAML view**.

   NOTE

   The **Pipeline builder** view supports a limited number of fields whereas the **YAML view** supports all available fields. Optionally, you can also use the Operator-installed, reusable snippets and samples to create detailed Pipelines.

   Figure 2.1. YAML view

   Pipeline builder

   Configure via: ○ Pipeline builder [✓ YAML view]

   ```
   apiVersion: tekton.dev/v1beta1
   kind: Pipeline
   metadata:
     name: new-pipeline
     namespace: karthik
   spec:
     params: []
     resources: []
     tasks: []
   ```

   Pipeline

   Samples | Snippets

   1. docker-build-and-deploy-pipeline
      An example of docker build and deploy pipeline
      Try it — Download YAML
   2. s2i-build-and-deploy-pipeline-using-workspace
      An example of s2i build and deploy pipeline using workspace
      Try it — Download YAML
   3. simple-pipeline
      An example of simple pipeline
      Try it — Download YAML

   To configure your pipeline using the **Pipeline Builder**:

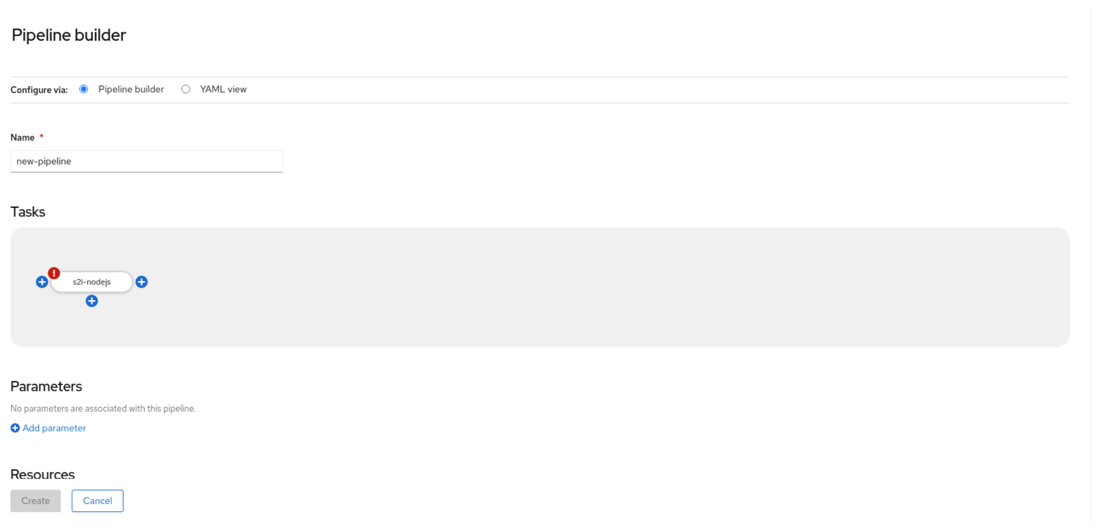
   a. Enter a unique name for the pipeline.

   b. Select a task from the **Select Task** list to add a task to the pipeline. This example uses the **s2i-nodejs** task.

      - To add sequential tasks to the pipeline, click the plus icon to the right or left of the task, and from the **Select Task** list, select the task you want to add to the pipeline. For this example, use the plus icon to the right of the **s2i-nodejs** task to add an **openshift-client** task.

      - To add a parallel task to the existing task, click the plus icon displayed next to the task, and from the **Select Task** list, select the parallel task you want to add to the pipeline.

Figure 2.2. Pipeline Builder



Pipeline builder

Configure via: (●) Pipeline builder ( ) YAML view

Name *
new-pipeline

Tasks

Parameters
No parameters are associated with this pipeline.
Add parameter

Resources

c. Click **Add Resources** to specify the name and type of resources that the pipeline run will use. These resources are then used by the tasks in the pipeline as inputs and outputs. For this example:

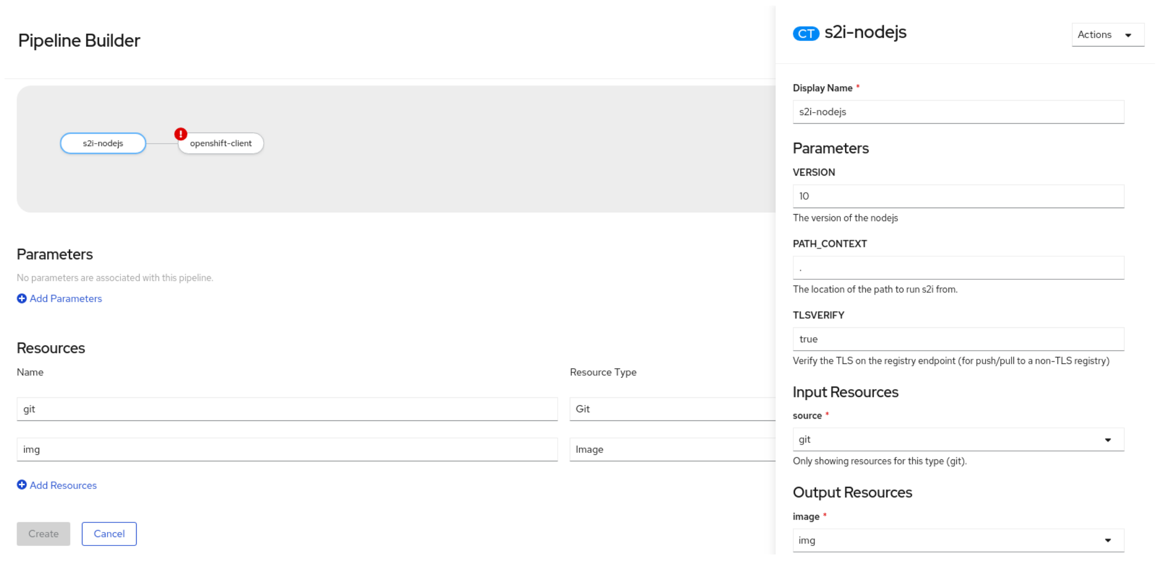
   i. Add an input resource. In the **Name** field, enter **Source**, and then from the **Resource Type** drop-down list, select **Git**.

   ii. Add an output resource. In the **Name** field, enter **Img**, and then from the **Resource Type** drop-down list, select **Image**.

d. Optional: The **Parameters** for a task are prepopulated based on the specifications of the task. If required, use the **Add Parameters** link to add additional parameters.

e. A **Missing Resources** warning is displayed on a task if the resources for the task are not specified. Click the **s2i-nodejs** task to see the side panel with details for the task.

Figure 2.3. Tasks details in Pipelines Builder



Pipeline Builder

Parameters
No parameters are associated with this pipeline.
Add Parameters

Resources

Name	Resource Type
git	Git
img	Image

Add Resources

s2i-nodejs

Display Name *
s2i-nodejs

Parameters

VERSION
10
The version of the nodejs

PATH_CONTEXT
.
The location of the path to run s2i from.

TLSVERIFY
true
Verify the TLS on the registry endpoint (for push/pull to a non-TLS registry)

Input Resources
source *
git
Only showing resources for this type (git).

Output Resources
image *
img

f. In the task side panel, specify the resources and parameters for the **s2i-nodejs** task:

   i. In the **Input Resources → Source** section, the **Select Resources** drop-down list displays the resources that you added to the pipeline. For this example, select **Source**.

ii. In the **Output Resources → Image** section, click the **Select Resources** list, and select **Img**.

iii. If required, in the **Parameters** section, add more parameters to the default ones, by using the **$(params.<param-name>)** syntax.

iv. Similarly, add an input resource for the **openshift-client** task.

3. Click **Create** to create and view the pipeline in the **Pipeline Details** page.

4. Click the **Actions** drop-down menu, and then click **Start** to start the Pipeline.

2.7.2. Creating applications with OpenShift Pipelines

To create pipelines along with applications, use the **From Git** option in the **Add** view of the **Developer** perspective. For more information, see Creating applications using the Developer perspective .

2.7.3. Interacting with pipelines using the Developer perspective

The **Pipelines** view in the **Developer** perspective lists all the pipelines in a project, along with the following details:

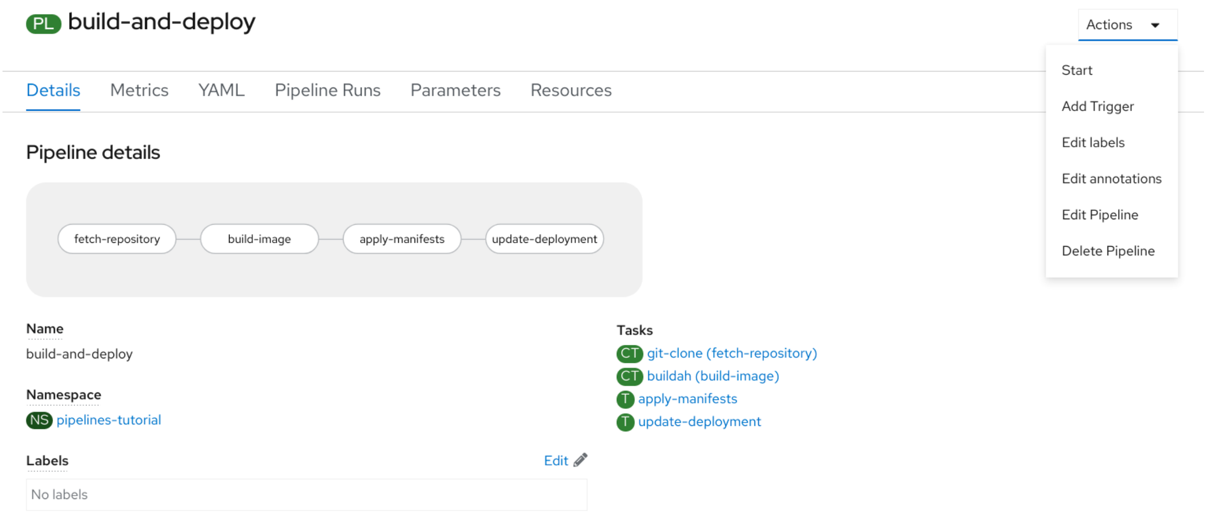
- The namespace in which the pipeline was created
- The last pipeline run
- The status of the tasks in the pipeline run
- The status of the pipeline run
- The creation time of the last pipeline run

Procedure

1. In the **Pipelines** view of the **Developer** perspective, select a project from the **Project** drop-down list to see the pipelines in that project.

2. Click the required pipeline to see the **Pipeline details** page. By default, the **Details** tab opens and provides a visual representation of all the serial and parallel tasks in the pipeline. The tasks are also listed in the lower right portion of the page. You can click the listed **Tasks** to view the task details.

Figure 2.4. Pipeline details



3. Optionally, in the **Pipeline details** page:

   - Click the **Metrics** tab to see the following information about pipelines:
     - **Pipeline Success Ratio**
     - **Number of Pipeline Runs**
     - **Pipeline Run Duration**
     - **Task Run Duration**
       You can use this information to improve the pipeline workflow and eliminate issues early in the pipeline lifecycle.
   - Click the **YAML** tab to edit the YAML file for the pipeline.
   - Click the **Pipeline Runs** tab to see the completed, running, or failed runs for the pipeline.

     NOTE

     The **Details** section of the **Pipeline Run Details** page displays a **Log Snippet** of the failed pipeline run. **Log Snippet** provides a general error message and a snippet of the log. A link to the **Logs** section provides quick access to the details about the failed run. The **Log Snippet** is also displayed in the the **Details** section of the **Task Run Details** page.

     You can use the Options menu ⋮ to stop a running pipeline, to rerun a pipeline using the same parameters and resources as that of the previous pipeline execution, or to delete a pipeline run.
   - Click the **Parameters** tab to see the parameters defined in the pipeline. You can also add or edit additional parameters, as required.
   - Click the **Resources** tab to see the resources defined in the pipeline. You can also add or edit additional resources, as required.

2.7.4. Starting pipelines

After you create a pipeline, you need to start it to execute the included tasks in the defined sequence. You can start a pipeline from the **Pipelines** view, the **Pipeline Details** page, or the **Topology** view.

Procedure

To start a pipeline using the **Pipelines** view:

1. In the **Pipelines** view of the **Developer** perspective, click the **Options** ⋮ menu adjoining a pipeline, and select **Start**.
2. The **Start Pipeline** dialog box displays the **Git Resources** and the **Image Resources** based on the pipeline definition.

   NOTE

   For pipelines created using the **From Git** option, the **Start Pipeline** dialog box also displays an **APP_NAME** field in the **Parameters** section, and all the fields in the dialog box are prepopulated by the pipeline template.

   a. If you have resources in your namespace, the **Git Resources** and the **Image Resources** fields are prepopulated with those resources. If required, use the drop-downs to select or create the required resources and customize the pipeline run instance.
3. Optional: Modify the **Advanced Options** to add the credentials that authenticate the specified private Git server or the image registry.

   a. Under **Advanced Options**, click **Show Credentials Options** and select **Add Secret**.

   b. In the **Create Source Secret** section, specify the following:

      i. A unique **Secret Name** for the secret.

      ii. In the **Designated provider to be authenticated** section, specify the provider to be authenticated in the **Access to** field, and the base **Server URL**.

      iii. Select the **Authentication Type** and provide the credentials:

      - For the **Authentication Type Image Registry Credentials**, specify the **Registry Server Address** that you want to authenticate, and provide your credentials in the **Username**, **Password**, and **Email** fields.
        Select **Add Credentials** if you want to specify an additional **Registry Server Address**.
      - For the **Authentication Type Basic Authentication**, specify the values for the **UserName** and **Password or Token** fields.
      - For the **Authentication Type SSH Keys**, specify the value of the **SSH Private Key** field.

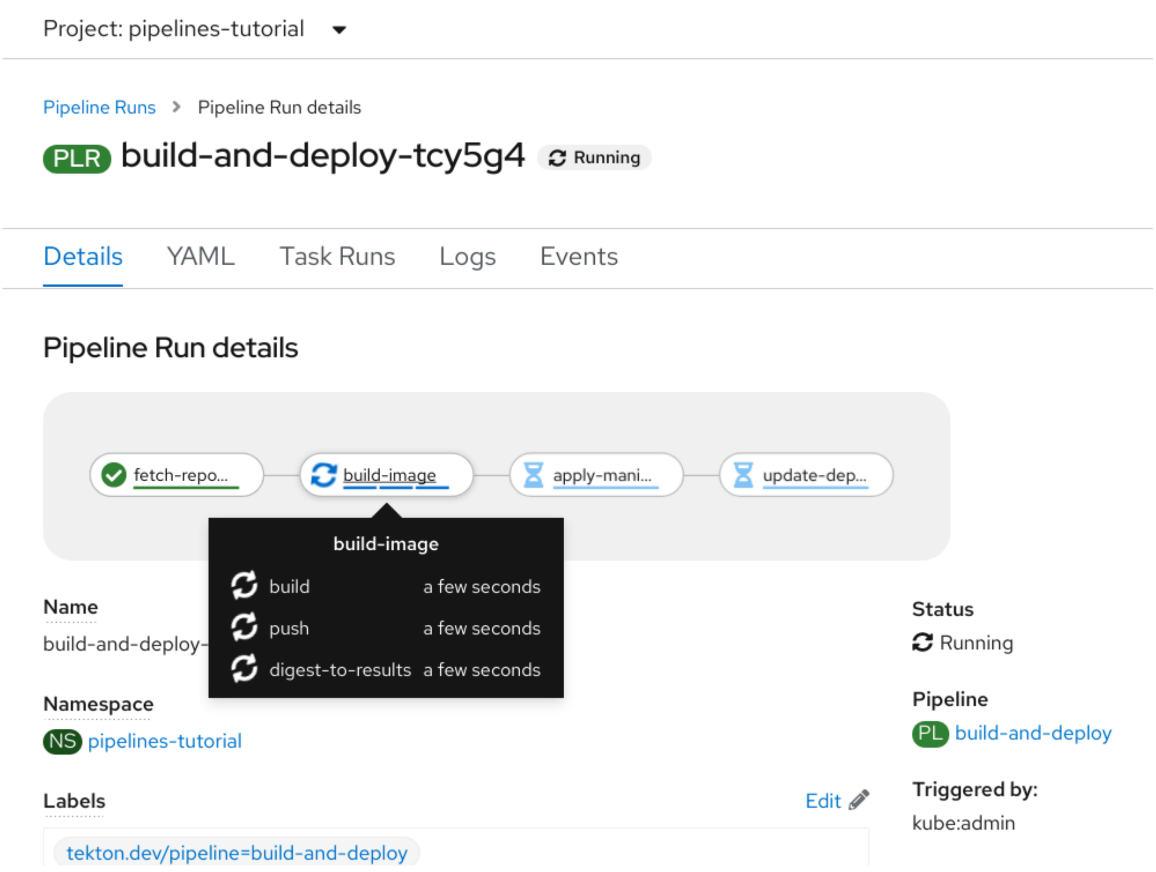
      iv. Select the check mark to add the secret.

   You can add multiple secrets based upon the number of resources in your pipeline.
4. Click **Start** to start the pipeline.

5. The **Pipeline Run Details** page displays the pipeline being executed. After the pipeline starts, the tasks and steps within each task are executed. You can:

   - Hover over the tasks to see the time taken to execute each step.
   - Click on a task to see the logs for each step in the task.
   - Click the **Logs** tab to see the logs relating to the execution sequence of the tasks. You can also expand the pane and download the logs individually or in bulk, by using the relevant button.
   - Click the **Events** tab to see the stream of events generated by a pipeline run.
     You can use the **Task Runs**, **Logs**, and **Events** tabs to assist in debugging a failed pipeline run or a failed task run.

   Figure 2.5. Pipeline run details



Project: pipelines-tutorial

Pipeline Runs > Pipeline Run details

PLR build-and-deploy-tcy5g4 Running

Details YAML Task Runs Logs Events

Pipeline Run details

fetch-repo... build-image apply-mani... update-dep...

build-image

build	a few seconds
push	a few seconds
digest-to-results	a few seconds

Name
build-and-deploy-

Namespace
NS pipelines-tutorial

Labels Edit
tekton.dev/pipeline=build-and-deploy

Status
Running

Pipeline
PL build-and-deploy

Triggered by:
kube:admin

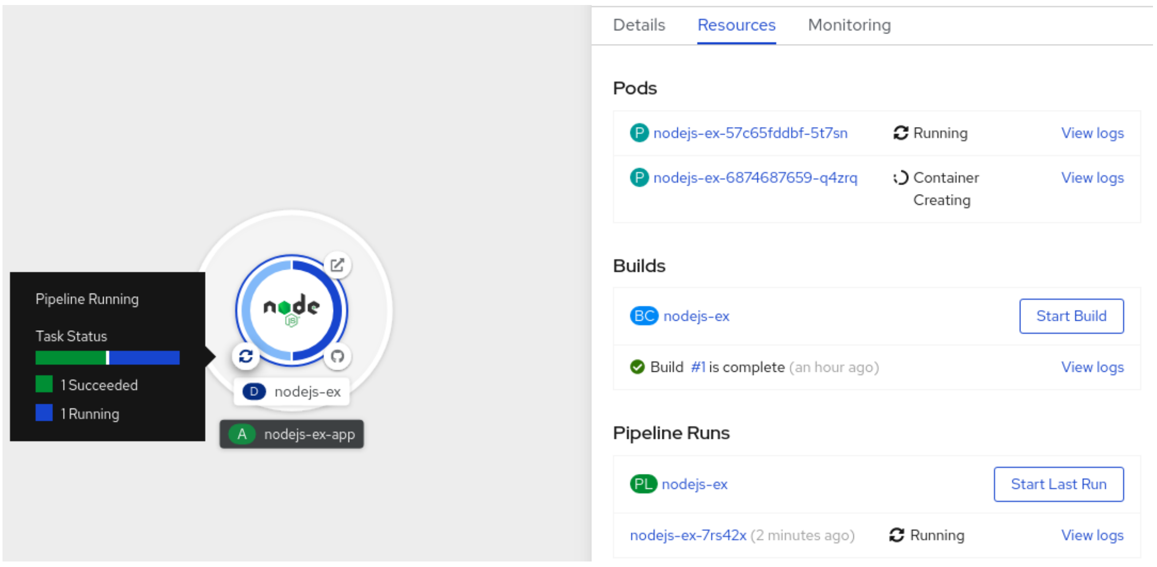
6. For pipelines created using the **From Git** option, you can use the **Topology** view to interact with pipelines after you start them:

   > **NOTE**
   >
   > To see the pipelines created using the **Pipeline Builder** in the **Topology** view, customize the pipeline labels to link the pipeline with the application workload.

   a. On the left navigation panel, click **Topology**, and click on the application to see the pipeline runs listed in the side panel.

   b. In the **Pipeline Runs** section, click **Start Last Run** to start a new pipeline run with the same parameters and resources as the previous one. This option is disabled if a pipeline run has not been initiated.

Figure 2.6. Pipelines in Topology view



Pods

P nodejs-ex-57c65fddbf-5t7sn	Running	View logs
P nodejs-ex-6874687659-q4zrq	Container Creating	View logs

Builds

BC nodejs-ex	Start Build
Build #1 is complete (an hour ago)	View logs

Pipeline Runs

PL nodejs-ex		Start Last Run
nodejs-ex-7rs42x (2 minutes ago)	Running	View logs

c. In the **Topology** page, hover to the left of the application to see the status of the pipeline run for the application.

> **NOTE**
>
> The side panel of the application node in the **Topology** page displays a **Log Snippet** when a pipeline run fails on a specific task run. You can view the **Log Snippet** in the **Pipeline Runs** section, under the **Resources** tab. **Log Snippet** provides a general error message and a snippet of the log. A link to the **Logs** section provides quick access to the details about the failed run.

2.7.5. Editing Pipelines

You can edit the Pipelines in your cluster using the **Developer** perspective of the web console:

Procedure

1. In the **Pipelines** view of the **Developer** perspective, select the Pipeline you want to edit to see the details of the Pipeline. In the **Pipeline Details** page, click **Actions** and select **Edit Pipeline**.
2. In the **Pipeline Builder** page:
   - You can add additional Tasks, parameters, or resources to the Pipeline.
   - You can click the Task you want to modify to see the Task details in the side panel and modify the required Task details, such as the display name, parameters and resources.
   - Alternatively, to delete the Task, click the Task, and in the side panel, click **Actions** and select **Remove Task**.
3. Click **Save** to save the modified Pipeline.

2.7.6. Deleting Pipelines

You can delete the Pipelines in your cluster using the **Developer** perspective of the web console.

Procedure

1. In the **Pipelines** view of the **Developer** perspective, click the **Options** ⋮ menu adjoining a Pipeline, and select **Delete Pipeline**.
2. In the **Delete Pipeline** confirmation prompt, click **Delete** to confirm the deletion.

CHAPTER 3. GITOPS

3.1. RED HAT OPENSHIFT GITOPS RELEASE NOTES

Red Hat OpenShift GitOps is a declarative way to implement continuous deployment for cloud native applications. 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 helps you automate the following tasks:

- Ensure that the clusters have similar states for configuration, monitoring, and storage
- Recover or recreate clusters from a known state
- Apply or revert configuration changes to multiple OpenShift Container Platform clusters
- Associate templated configuration with different environments
- Promote applications across clusters, from staging to production

For an overview of Red Hat OpenShift GitOps, see Understanding OpenShift GitOps .

3.1.1. Making open source more inclusive

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see Red Hat CTO Chris Wright's message .

3.1.2. Release notes for Red Hat OpenShift GitOps 1.1

Red Hat OpenShift GitOps 1.1 is now available on OpenShift Container Platform 4.7.

3.1.2.1. Support matrix

Some features in this release are currently in Technology Preview. These experimental features are not intended for production use.

Technology Preview Features Support Scope

In the table below, features are marked with the following statuses:

- **TP**: *Technology Preview*
- **GA**: *General Availability*

Note the following scope of support on the Red Hat Customer Portal for these features:

Table 3.1. Support matrix

Feature	Red Hat OpenShift GitOps 1.1
Argo CD	GA

Feature	Red Hat OpenShift GitOps 1.1
Argo CD ApplicationSet	TP
Red Hat OpenShift GitOps Application Manager (kam)	TP
Red Hat OpenShift GitOps Service	TP

3.1.2.2. New features

In addition to the fixes and stability improvements, the following sections highlight what is new in Red Hat OpenShift GitOps 1.1:

- The **ApplicationSet** feature is now added (Technology Preview). The **ApplicationSet** feature enables both automation and greater flexibility when managing Argo CD applications across a large number of clusters and within monorepos. It also makes self-service usage possible on multitenant Kubernetes clusters.
- Argo CD is now integrated with cluster logging stack and with the OpenShift Container Platform Monitoring and Alerting features.
- Argo CD auth is now integrated with OpenShift Container Platform.
- Argo CD applications controller now supports horizontal scaling.
- Argo CD Redis servers now support high availability (HA).

3.1.2.3. Fixed issues

The following issues were resolved in the current release:

- Previously, Red Hat OpenShift GitOps did not work as expected in a proxy server setup with active global proxy settings. This issue is fixed and now Argo CD is configured by the Red Hat OpenShift GitOps Operator using fully qualified domain names (FQDN) for the pods to enable communication between components. GITOPS-703
- The Red Hat OpenShift GitOps backend relies on the **?ref=** query parameter in the Red Hat OpenShift GitOps URL to make API calls. Previously, this parameter was not read from the URL, causing the backend to always consider the default reference. This issue is fixed and the Red Hat OpenShift GitOps backend now extracts the reference query parameter from the Red Hat OpenShift GitOps URL and only uses the default reference when there is no input reference provided. GITOPS-817
- Previously, the Red Hat OpenShift GitOps backend failed to find the valid GitLab repository. This was because the Red Hat OpenShift GitOps backend checked for **main** as the branch reference, instead of **master** in the GitLab repository. This issue is fixed now. GITOPS-768
- The **Environments** page in the **Developer** perspective of the OpenShift Container Platform web console now shows the list of applications and the number of environments. This page also displays an Argo CD link that directs you to the Argo CD **Applications** page that lists all the

applications. The Argo CD **Applications** page has **LABELS** (for example, **app.kubernetes.io/name=appName**) that help you filter only the applications of your choice. GITOPS-544

3.1.2.4. Known issues

These are the known issues in Red Hat OpenShift GitOps 1.1:

- Red Hat OpenShift GitOps does not support Helm v2 and ksonnet.
- The Red Hat SSO (RH SSO) Operator is not supported in disconnected clusters. As a result, the Red Hat OpenShift GitOps Operator and RH SSO integration is not supported in disconnected clusters.
- When you delete an Argo CD application from the OpenShift Container Platform web console, the Argo CD application gets deleted in the user interface, but the deployments are still present in the cluster. As a workaround, delete the Argo CD application from the Argo CD console. GITOPS-830

3.1.2.5. Breaking Change

3.1.2.5.1. Upgrading from Red Hat OpenShift GitOps v1.0.1

When you upgrade from Red Hat OpenShift GitOps **v1.0.1** to **v1.1**, the Red Hat OpenShift GitOps Operator renames the default Argo CD instance created in the **openshift-gitops** namespace from **argocd-cluster** to **openshift-gitops**.

This is a breaking change and needs the following steps to be performed manually, before the upgrade:

1. Go to the OpenShift Container Platform web console and copy the content of the **argocd-cm.yml** config map file in the **openshift-gitops** namespace to a local file. The content may look like the following example:

   Example argocd config map YAML

```
kind: ConfigMap
apiVersion: v1
metadata:
 selfLink: /api/v1/namespaces/openshift-gitops/configmaps/argocd-cm
 resourceVersion: '112532'
 name: argocd-cm
 uid: f5226fbc-883d-47db-8b53-b5e363f007af
 creationTimestamp: '2021-04-16T19:24:08Z'
 managedFields:
...
 namespace: openshift-gitops
 labels:
   app.kubernetes.io/managed-by: argocd-cluster
   app.kubernetes.io/name: argocd-cm
   app.kubernetes.io/part-of: argocd
data:                                ""  (1)
 admin.enabled: 'true'
 statusbadge.enabled: 'false'
 resource.exclusions: |
   - apiGroups:
        - tekton.dev
        clusters:
        - '*'
        kinds:
        - TaskRun
        - PipelineRun
    ga.trackingid: ''
    repositories: |
      - type: git
        url: https://github.com/user-name/argocd-example-apps
    ga.anonymizeusers: 'false'
    help.chatUrl: ''
    url: >-  https://argocd-cluster-server-openshift-gitops.apps.dev-svc-4.7-
041614.devcluster.openshift.com    "" (2)
    help.chatText: ''
    kustomize.buildOptions: ''
    resource.inclusions: ''
    repository.credentials: ''
    users.anonymous.enabled: 'false'
    configManagementPlugins: ''
    application.instanceLabelKey: ''
```

1. Restore only the **data** section of the content in the **argocd-cm.yml** config map file manually.
2. Replace the URL value in the config map entry with the new instance name **openshift-gitops**.

2. Delete the default **argocd-cluster** instance.
3. Edit the new **argocd-cm.yml** config map file to restore the entire **data** section manually.
4. Replace the URL value in the config map entry with the new instance name **openshift-gitops**. For example, in the preceding example, replace the URL value with the following URL value:

```
url: >-  https://openshift-gitops-server-openshift-gitops.apps.dev-svc-4.7-
041614.devcluster.openshift.com
```

5. Login to the Argo CD cluster and verify that the previous configurations are present.

3.2. UNDERSTANDING OPENSHIFT GITOPS

3.2.1. About GitOps

GitOps is a declarative way to implement continuous deployment for cloud native applications. You can use GitOps to create repeatable processes for managing OpenShift Container Platform clusters and applications across multi-cluster Kubernetes environments. GitOps handles and automates complex deployments at a fast pace, saving time during deployment and release cycles.

The GitOps workflow pushes an application through development, testing, staging, and production. GitOps either deploys a new application or updates an existing one, so you only need to update the repository; GitOps automates everything else.

GitOps is a set of practices that use Git pull requests to manage infrastructure and application configurations. In GitOps, the Git repository is the only source of truth for system and application configuration. This Git repository contains a declarative description of the infrastructure you need in your specified environment and contains an automated process to make your environment match the described state. Also, it contains the entire state of the system so that the trail of changes to the system state are visible and auditable. By using GitOps, you resolve the issues of infrastructure and application configuration sprawl.

GitOps defines infrastructure and application definitions as code. Then, it uses this code to manage multiple workspaces and clusters to simplify the creation of infrastructure and application configurations. By following the principles of the code, you can store the configuration of clusters and applications in Git repositories, and then follow the Git workflow to apply these repositories to your chosen clusters. You can apply the core principles of developing and maintaining software in a Git repository to the creation and management of your cluster and application configuration files.

3.2.2. About Red Hat OpenShift GitOps

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 OpenShift Container Platform clusters.

3.2.2.1. Key features

Red Hat OpenShift GitOps helps you automate the following tasks:

- Ensure that the clusters have similar states for configuration, monitoring, and storage
- Recover or recreate clusters from a known state
- Apply or revert configuration changes to multiple OpenShift Container Platform clusters
- Associate templated configuration with different environments
- Promote applications across clusters, from staging to production

3.3. GETTING STARTED WITH OPENSHIFT GITOPS

Red Hat OpenShift GitOps uses Argo CD to manage specific cluster-scoped resources, including platform operators, optional Operator Lifecycle Manager (OLM) operators, and user management.

This guide explains how to install the Red Hat OpenShift GitOps Operator to an OpenShift Container Platform cluster and logging in to the Argo CD instance.

3.3.1. Installing GitOps Operator in web console

Prerequisites

- You are logged in to the OpenShift cluster as an administrator.

Procedure

1. Open the **Administrator** perspective of the web console and navigate to **Operators → OperatorHub** in the menu on the left.
2. Search for **OpenShift GitOps**, click the **Red Hat OpenShift GitOps** tile and then click the **Install** button.
   Red Hat OpenShift GitOps will be installed in all namespaces of the cluster.

Once the Red Hat OpenShift GitOps Operator is installed, it automatically sets up a ready-to-use Argo CD instance that is available in the **openshift-gitops** namespace, and an Argo CD icon is displayed in the console toolbar.

3.4. CONFIGURING ARGO CD TO RECURSIVELY SYNC A GIT REPOSITORY WITH YOUR APPLICATION

3.4.1. Configuring an OpenShift cluster by deploying an application with cluster configurations

With Red Hat OpenShift GitOps, you can configure Argo CD to recursively sync the content of a Git directory with an application that contains custom configurations for your cluster.

Prerequisites

- Red Hat OpenShift GitOps is installed in your cluster.

3.4.1.1. Logging in to the Argo CD instance by using your OpenShift credentials

Red Hat OpenShift GitOps Operator automatically creates a ready-to-use Argo CD instance that is available in the **openshift-gitops** namespace.

Prerequisites

- You have installed the Red Hat OpenShift GitOps Operator in your cluster.

Procedure

1. In the **Administrator** perspective of the web console, navigate to **Operators → Installed Operators** to verify that the Red Hat OpenShift GitOps Operator is installed.

2. Navigate to the ▦ menu → **Application Stages** → **Argo CD**. The login page of the Argo CD UI is displayed in a new window.

3. Obtain the password for the Argo CD instance:

   a. Navigate to the **Developer** perspective of the web console. A list of available projects is displayed.

   b. Navigate to the **openshift-gitops** project.

   c. Use the left navigation panel to navigate to the **Secrets** page.

   d. Select the **openshift-gitops-cluster** instance to display the password.

   e. Copy the password.

4. Use this password and **admin** as the username to log in to the Argo CD UI in the new window.

3.4.1.2. Creating an application by using the Argo CD dashboard

Argo CD provides a dashboard which allows you to create applications.

This sample workflow walks you through the process of configuring Argo CD to recursively sync the content of the **cluster** directory to the **cluster-configs** application. The directory defines OpenShift web console cluster configurations that add a link to the **Red Hat Developer Blog** under the ▦ menu in the web console, and defines a namespace **spring-petclinic** on the cluster.

Procedure

1. In the Argo CD dashboard, click the **New App** button to add a new Argo CD application.

2. For this workflow, create a **cluster-configs** application with the following configurations:

   Application Name

   cluster-configs

   Project

   default

   Sync Policy

   Manual

   Repository URL

   https://github.com/redhat-developer/openshift-gitops-getting-started

   Revision

   HEAD

   Path

   cluster

   Destination

   https://kubernetes.default.svc

   Namespace

default

Directory Recurse

checked

3. Click **Create** to create your application.

3.4.1.3. Creating an application by using the oc tool

You can create Argo CD applications in your terminal by using the **oc** tool.

Procedure

1. Download the sample application:

```
$ git clone git@github.com:redhat-developer/openshift-gitops-getting-started.git
```

2. Create the application:

```
$ oc create -f openshift-gitops-getting-started/argo/cluster.yaml
```

3. Run the **oc get** command to review the created application:

```
$ oc get application -n openshift-gitops
```

3.4.1.4. Synchronizing your application with your Git repository

Procedure

1. In the Argo CD dashboard, notice that the **cluster-configs** Argo CD application has the statuses **Missing** and **OutOfSync**. Because the application was configured with a manual sync policy, Argo CD does not sync it automatically.

2. Click the **Sync** button on the **cluster-configs** tile, review the changes, and then click **Synchronize**. Argo CD will detect any changes in the Git repository automatically. If the configurations are changed, Argo CD will change the status of the **cluster-configs** to **OutOfSync**. You can modify the synchronization policy for Argo CD to automatically apply changes from your Git repository to the cluster.

3. Notice that the **cluster-configs** Argo CD application now has the statuses **Healthy** and **Synced**. Click the **cluster-configs** tile to check the details of the synchronized resources and their status on the cluster.

4. Navigate to the OpenShift web console and click ▦ to verify that a link to the **Red Hat Developer Blog** is now present there.

5. Navigate to the **Project** page and search for the **spring-petclinic** namespace to verify that it has been added to the cluster.
   Your cluster configurations have been successfully synchronized to the cluster.

3.4.2. Deploying a Spring Boot application with Argo CD

With Argo CD, you can deploy your applications to the OpenShift cluster either by using the Argo CD dashboard or by using the **oc** tool.

Prerequisites

- Red Hat OpenShift GitOps is installed in your cluster.

3.4.2.1. Logging in to the Argo CD instance by using your OpenShift credentials

Red Hat OpenShift GitOps Operator automatically creates a ready-to-use Argo CD instance that is available in the **openshift-gitops** namespace.

Prerequisites

- You have installed the Red Hat OpenShift GitOps Operator in your cluster.

Procedure

1. In the **Administrator** perspective of the web console, navigate to **Operators → Installed Operators** to verify that the Red Hat OpenShift GitOps Operator is installed.
2. Navigate to the menu → **Application Stages → Argo CD**. The login page of the Argo CD UI is displayed in a new window.
3. Obtain the password for the Argo CD instance:
   a. Navigate to the **Developer** perspective of the web console. A list of available projects is displayed.
   b. Navigate to the **openshift-gitops** project.
   c. Use the left navigation panel to navigate to the **Secrets** page.
   d. Select the **openshift-gitops-cluster** instance to display the password.
   e. Copy the password.
4. Use this password and **admin** as the username to log in to the Argo CD UI in the new window.

3.4.2.2. Creating an application by using the Argo CD dashboard

Argo CD provides a dashboard which allows you to create applications.

This sample workflow walks you through the process of configuring Argo CD to recursively sync the content of the **cluster** directory to the **cluster-configs** application. The directory defines OpenShift web console cluster configurations that add a link to the **Red Hat Developer Blog** under the menu in the web console, and defines a namespace **spring-petclinic** on the cluster.

Procedure

1. In the Argo CD dashboard, click the **New App** button to add a new Argo CD application.
2. For this workflow, create a **cluster-configs** application with the following configurations:

Application Name
: **cluster-configs**

Project
: **default**

Sync Policy
: **Manual**

Repository URL
: **https://github.com/redhat-developer/openshift-gitops-getting-started**

Revision
: **HEAD**

Path
: **cluster**

Destination
: **https://kubernetes.default.svc**

Namespace
: **default**

Directory Recurse
: **checked**

3. For this workflow, create a **spring-petclinic** application with the following configurations:

Application Name
: **spring-petclinic**

Project
: **default**

Sync Policy
: **Automatic**

Repository URL
: **https://github.com/redhat-developer/openshift-gitops-getting-started**

Revision
: **HEAD**

Path
: **app**

Destination
: **https://kubernetes.default.svc**

Namespace
: **spring-petclinic**

Directory Recurse
: **checked**

4. Click **Create** to create your application.

3.4.2.3. Creating an application by using the oc tool

You can create Argo CD applications in your terminal by using the **oc** tool.

Procedure

1. Download the sample application:

   ```
   $ git clone git@github.com:redhat-developer/openshift-gitops-getting-started.git
   ```

2. Create the application:

   ```
   $ oc create -f openshift-gitops-getting-started/argo/app.yaml
   ```

   ```
   $ oc create -f openshift-gitops-getting-started/argo/cluster.yaml
   ```

3. Run the **oc get** command to review the created application:

   ```
   $ oc get application -n openshift-gitops
   ```

3.4.2.4. Verifying Argo CD self-healing behavior

Argo CD constantly monitors the state of deployed applications, detects differences between the specified manifests in Git and live changes in the cluster, and then automatically corrects them. This behavior is referred to as self-healing.

You can test and observe the self-healing behavior in Argo CD.

Prerequisites

- The sample **app-spring-petclinic** application is deployed and configured.

Procedure

1. In the Argo CD dashboard, verify that your application has the **Synced** status.
2. Click the **app-spring-petclinic** tile in the Argo CD dashboard to view the application resources that are deployed to the cluster.
3. In the OpenShift web console, navigate to the **Developer** perspective.
4. Modify the Spring PetClinic deployment and commit the changes to the **app/** directory of the Git repository. Argo CD will automatically deploy the changes to the cluster.
5. Test the self-healing behavior by modifying the deployment on the cluster and scaling it up to two pods while watching the application in the OpenShift web console.
   a. Run the following command to modify the deployment:

      ```
      $ oc scale deployment spring-petclinic --replicas 2  -n spring-petclinic
      ```

   b. In the OpenShift web console, notice that the deployment scales up to two pods and immediately scales down again to one pod. Argo CD detected a difference from the Git repository and auto-healed the application on the OpenShift cluster.

6. In the Argo CD dashboard, click the **app-spring-petclinic** tile → **App Details** → **Events**. The **Events** tab displays the following events: Argo CD detecting out of sync deployment resources on the cluster and then resyncing the Git repository to correct it.

3.5. CONFIGURING SSO FOR ARGO CD ON OPENSHIFT

Once the Red Hat OpenShift GitOps Operator is installed, Argo CD automatically creates a user with **admin** permissions. To manage multiple users, Argo CD allows cluster administrators to configure SSO. You can use Keycloak or a bundled Dex OIDC provider.

Prerequisites

- Red Hat SSO is installed on the cluster.

3.5.1. Creating a new client in Keycloak

Procedure

1. Log in to your Keycloak server, select the realm you want to use, navigate to the **Clients** page, and then click the **Create** button in the upper-right section of the screen.

2. Specify the following values:

   Client ID

      argocd

   Client Protocol

      openid-connect

   Route URL

      <your-argo-cd-route-url>

   Access Type

      confidential

   Valid Redirect URIs

      <your-argo-cd-route-url>/auth/callback

   Base URL

      /applications

3. Click **Save** to see the **Credentials** tab added to the **Client** page.

4. Copy the secret from the **Credentials** tab for further configuration.

3.5.2. Configuring the groups claim

To manage users in Argo CD, you must configure a groups claim that can be included in the authentication token.

Procedure

1. In the Keycloak dashboard, navigate to **Client Scope** and add a new client with the following values:

   Name

groups

Protocol

openid-connect

Display On Content Scope

On

Include to Token Scope

On

2. Click the **Save** button and navigate to **groups** → **Mappers**.
3. Add a new token mapper with the following values:

   Name

   groups

   Mapper Type

   Group Membership

   Token Claim Name

   groups
   The token mapper adds the **groups** claim to the token when the client requests **groups**.

4. Navigate to **Clients** → **Client Scopes** and configure the client to provide the groups scope. Select **groups** in the **Assigned Default Client Scopes** table and click **Add selected**. The **groups** scope must be in the **Available Client Scopes** table.
5. Navigate to **Users** → **Admin** → **Groups** and create a group **ArgoCDAdmins**.

3.5.3. Configuring Argo CD OIDC

To configure Argo CD OpenID Connect (OIDC), you must generate your client secret, encode it, and add it to your custom resource.

Prerequisites

- You have obtained your client secret.

Procedure

1. Store the client secret you generated.
   a. Encode the client secret in base64:

   ```
   $ echo -n '83083958-8ec6-47b0-a411-a8c55381fbd2' | base64
   ```

   b. Edit the secret and add the base64 value to an **oidc.keycloak.clientSecret** key:

   ```
   $ oc edit secret openshift-gitops-secret -n <namespace>
   ```

   Example YAML of the secret

   ```
   yaml apiVersion: v1
   ```

```
kind: Secret
metadata: name: argocd-secret
data:
  oidc.keycloak.clientSecret:
ODMwODM5NTgtOGVjNi00N2IwLWE0MTEtYThjNTUzODFmYmQy …
```

2. Edit the **argocd** custom resource and add the OIDC configuration to enable the Keycloak authentication:

```
$ oc edit argocd -n <your_namespace>
```

Example of argocd custom resource

```
apiVersion: argoproj.io/v1alpha1
kind: ArgoCD
metadata:
  creationTimestamp: null
  name: argocd
  namespace: argocd
spec:
  resourceExclusions: |
    - apiGroups:
      - tekton.dev
      clusters:
      - '*'
      kinds:
      - TaskRun
      - PipelineRun
  oidcConfig: |
    name: OpenShift Single Sign-On
    issuer: https://keycloak.example.com/auth/realms/myrealm (1)
    clientID: argocd (2)
    clientSecret: $oidc.keycloak.clientSecret (3)
    requestedScopes: ["openid", "profile", "email", "groups"] (4)
  server:
    route:
      enabled: true
```

(1) **issuer** must end with the correct realm name (in this example **myrealm**).

(2) **clientID** is the Client ID you configured in your Keycloak account.

(3) **clientSecret** points to the right key you created in the argocd-secret secret.

(4) **requestedScopes** contains the groups claim if you did not add it to the Default scope.

3.5.4. Keycloak Identity Brokering with OpenShift

You can configure a Keycloak instance to use OpenShift for authentication through Identity Brokering. This allows for Single Sign-On (SSO) between the OpenShift cluster and the Keycloak instance.

Prerequisites

- **jq** CLI tool is installed.

Procedure

1. Obtain the OpenShift Container Platform API URL:

```
$ curl -s -k -H "Authorization: Bearer $(oc whoami -t)" https://<openshift-user-facing-api-
url>/apis/config.openshift.io/v1/infrastructures/cluster | jq ".status.apiServerURL".
```

NOTE

The address of the OpenShift Container Platform API is often protected by HTTPS. Therefore, you must configure X509_CA_BUNDLE in the container and set it to **/var/run/secrets/kubernetes.io/serviceaccount/ca.crt**. Otherwise, Keycloak cannot communicate with the API Server.

2. In the Keycloak server dashboard, navigate to **Identity Providers** and select **Openshift v4**. Specify the following values:

Base Url
: OpenShift 4 API URL

Client ID
: **keycloak-broker**

Client Secret
: A secret that you want define
Now you can log in to Argo CD with your OpenShift credentials through Keycloak as an Identity Broker.

3.5.5. Registering an additional an OAuth client

If you need an additional OAuth client to manage authentication for your OpenShift Container Platform cluster, you can register one.

Procedure

- To register your client:

```
$ oc create -f <(echo '
kind: OAuthClient
apiVersion: oauth.openshift.io/v1
metadata:
 name: keycloak-broker (1)
secret: "..." (2)
redirectURIs:
- "https://keycloak-keycloak.apps.dev-svc-4.7-
020201.devcluster.openshift.com/auth/realms/myrealm/broker/openshift-v4/endpoint" (3)
grantMethod: prompt (4)
')
```

1 The name of the OAuth client is used as the **client_id** parameter when making requests to **<namespace_route>/oauth/authorize** and `**<namespace_route>/oauth/token**.

2 The **secret** is used as the client_secret parameter when making requests to **<namespace_route>/oauth/token**.

3 The **redirect_uri** parameter specified in requests to **<namespace_route>/oauth/authorize` and** <namespace_route>/oauth/token` must be equal to or prefixed by one of the URIs listed in the **redirectURIs** parameter value.

4 If the user has not granted access to this client, the **grantMethod** determines which action to take when this client requests tokens. Specify **auto** to automatically approve the grant and retry the request, or **prompt** to prompt the user to approve or deny the grant.

3.5.6. Configure groups and Argo CD RBAC

Role-based access control (RBAC) allows you to provide relevant permissions to users.

Prerequisites

- You have created the **ArgoCDAdmins** group in Keycloak.
- The user you want to give permissions to has logged in to Argo CD.

Procedure

1. In the Keycloak dashboard navigate to **Users → Groups**. Add the user to the Keycloak group **ArgoCDAdmins**.
2. Ensure that **ArgoCDAdmins** group has the required permissions in the **argocd-rbac** config map.
   - Edit the config map:

```
$ oc edit configmap argocd-rbac-cm -n <namespace>
```

Example of a config map that defines admin permissions.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-rbac-cm
data:
  policy.csv: |
    g, ArgoCDAdmins, role:admin
```

3.5.7. In-built permissions for Argo CD

This section lists the permissions that are granted to ArgoCD to manage specific cluster-scoped resources which include platform operators, optional OLM operators, and user management. Note that ArgoCD is not granted **cluster-admin** permissions.

Table 3.2. Permissions granted to Argo CD

Resource group	What it configures for a user or an administrator
operators.coreos.com	Optional operators managed by OLM
user.openshift.io, rbac.authorization.k8s.io	Groups, Users, and their permissions
config.openshift.io	Control plane operators managed by CVO used to configure cluster-wide build configuration, registry configuration, and scheduler policies
storage.k8s.io	Storage
console.openshift.io	Console customization