Chapter 4. Implementing GitOps with Jenkins

[**Introducing Jenkins Declarative Pipelines**](https://rol.redhat.com/rol/app/courses/do380-4.5/pages/ch04)

[**Quiz: Introducing Jenkins Declarative Pipelines Quiz**](https://rol.redhat.com/rol/app/courses/do380-4.5/pages/ch04s02)

[**Deploying Jenkins on OpenShift**](https://rol.redhat.com/rol/app/courses/do380-4.5/pages/ch04s03)

[**Guided Exercise: Deploying Jenkins on OpenShift**](https://rol.redhat.com/rol/app/courses/do380-4.5/pages/ch04s04)

[**Configuring OpenShift Resources Using a Declarative GitOps Workflow**](https://rol.redhat.com/rol/app/courses/do380-4.5/pages/ch04s05)

[**Guided Exercise: Configuring OpenShift Resources Using a Declarative GitOps Workflow**](https://rol.redhat.com/rol/app/courses/do380-4.5/pages/ch04s06)

[**Configuring OpenShift using GitOps and Jenkins**](https://rol.redhat.com/rol/app/courses/do380-4.5/pages/ch04s07)

[**Guided Exercise: Configuring OpenShift using GitOps and Jenkins**](https://rol.redhat.com/rol/app/courses/do380-4.5/pages/ch04s08)

[**Lab: Implementing GitOps with Jenkins**](https://rol.redhat.com/rol/app/courses/do380-4.5/pages/ch04s09)

[**Summary**](https://rol.redhat.com/rol/app/courses/do380-4.5/pages/ch04s10)

**Abstract**

|  |  |
| --- | --- |
| **Goal** | Implement a GitOps workflow using containerized Jenkins to administer an OpenShift cluster. |
| **Objectives** | * Describe Jenkins concepts and Jenkins Pipeline concepts. * Deploy Jenkins on OpenShift using standard templates and image streams. * Configure cluster operators using resource files in a declarative fashion. * Automate the configuration of cluster operators from resources in Git using Jenkins Pipelines. |
| **Sections** | * Introducing Jenkins Declarative Pipelines (and Quiz) * Deploying Jenkins on OpenShift (and Guided Exercise) * Configuring OpenShift Resources Using a Declarative GitOps Workflow (and Guided Exercise) * Configuring OpenShift using GitOps and Jenkins (and Guided Exercise) |
| **Lab** | Implementing GitOps with Jenkins |

Introducing Jenkins Declarative Pipelines

Objectives

After completing this section, you should be able to describe Jenkins concepts and Jenkins Pipeline concepts.

Introducing Continuous Integration (CI) and Continuous Deployment (CD)

Jenkins is currently the most popular tool for automating software development processes. Created as a tool for managing software build processes, Jenkins evolved to deliver the Continuous Integration (CI) process, and is also flexible enough to support Continuous Deployment (CD) and GitOps processes.

The following figure shows a sample CI/CD workflow, illustrating how CI and CD processes are usually integrated into a larger process that allows a developer to make changes to an application source code and have quick access to a running instance of that application.

Continuous Integration (CI) Workflows

*Continuous Integration* (*CI*) is the process of automatically building a software piece from its source code and any required dependencies, such as programming libraries, whenever the source code changes.

To be successful and reliable, CI workflows requires automated testing, such as *unit testing*. A more sophisticated CI workflow might include many testing activities, such as security scanning and code coverage reports.

The final software artifact generated by a CI process can be anything that makes sense as a distributable piece of software, such as a native executable file, an RPM package, a Java library archive (JAR file), or a container image.

Continuous Delivery (CD) Workflows

Continuous Deployment (CD) is the process of automatically deploying a software piece to a target environment, such as development or production, whenever the source code changes.

To be successful and reliable, CI workflows also require automated testing, such as *integration testing*. It is also common to add manual approval gates to a CI workflow because other testing activities, such as *functional testing* and *user acceptance testing*, might not be fully automated.

CD workflows can depend on accessing a preconfigured target environment and be limited by the availability of that environment. More sophisticated CD workflows might create a target environment in a cloud environment, for example, by running Ansible Playbooks.

Container runtimes and Kubernetes clusters are popular ways to quickly provide temporary target environments for running tests before deploying in a real production environment.

GitOps Workflows

Compare and contrast the following with the diagram of the CI/CD worflow presented previously.

*GitOps* is a process that stores configurations of a target environment, such as a server or a cluster, in files managed by a version control system. It works under the assumption that system administrators do not change configurations on a live system directly. The best practice is to change the configurations under version control, and optionally have a fellow system administrator review them, before applying to a live system.

Typical GitOps workflows have two pieces: one that takes configurations from version control and applies those configurations to live systems, and another that verifies if the configurations from live systems have changed (drifted) compared to the current configuration under version control.

Comparing CI/CD and GitOps

CI/CD workflows are designed from a developer’s point of view and GitOps workflows are designed from a system administrator’s point of view.

Both GitOps and CI/CD assume that input comes from a version control system, and that some companion tool automatically reacts to changes committed to version control to produce some desired outcomes.

The outcomes from a CI/CD workflow is a running application and the side effect of a GitOps workflow is a live configuration.

It is a common pattern that a CI/CD process generates artifacts that serve as input to a GitOps process. For example, a CI/CD workflow generates a container image for an application and tests that application in a staging environment. A GitOps Workflow sets up the production environment for that same application.

Most tools that are capable of automating CI/CD processes are also capable of automating GitOps processes. For some organizations, tools designed for GitOps are replacing a CI/CD tools to perform the CD part of the workflow.

Introducing Tools for CI/CD

OpenShift features, such as Source-to-Image (S2I), build configurations, deployment configurations, and image streams, work together to implement simple CI/CD processes. If you deploy an application from source code using the oc new-app command, then OpenShift sets up these resources in a CI/CD workflow.

When you require more flexibility and power than provided by S2I, OpenShift integrates well with external CI/CD tools such as Jenkins. OpenShift includes container images and templates to deploy containerized Jenkins servers that are discussed elsewhere in this course.

Other popular CI/CD tools include OpenShift Pipelines, based on the Tekton open source project (Technical Preview as of Red Hat OpenShift Container Platform 4.4), and Jenkins X, an open source project that is creating a cloud-native variant of Jenkins.

There are also newer tools that specialize in GitOps processes, such as ArgoCD. These tools are usually not flexible enough to implement complete CI/CD workflows, but they compensate by making simpler to implement and manage GitOps workflows.

It becoming a recommended best practice withing the Kubernetes community to combine a Kubernetes-native CI/CD tool, such as OpenShift Pipelines, with a Kubernetes-native GitOps tool, such as GitOps, in larger integrated processes.

Introducing Jenkins

Jenkins is the leading open source automation server; it supports building, deploying, and automating any software development task.

Jenkins started as the Hudson build server, with a focus on building Java applications. In time, it evolved into a more general build automation server capable of automating generic CI and other software development workflows.

IT organizations that have already invested in Jenkins favor using it to implement GitOps workflows. Some organizations considering GitOps realize that they must also implement CI/CD to realize the benefits of GitOps and opt to automate it all with Jenkins. Other organizations consider it a major disadvantage that Jenkins is not Kubernetes-native, and look for alternatives.

Jenkins is a Java application that can be deployed to Servlet containers, such as Tomcat and Jetty, or application servers, such as JBoss EAP. Because Jenkins is coded in Java, Jenkins servers ran can on any operating system, which contributed to its popularity.

Despite being coded in Java, Jenkins is known for its wide ecosystem of plug-ins that target most other popular programming language run times.

Describing Essential Jenkins Concepts

Jenkins is a very sophisticated automation server. The following are essential concepts and terms for using and managing Jenkins.

**Project (or Job)**

A script that describes a workflow that Jenkins should perform, such as building an application.

**Pipeline**

A kind of Job that follows the pipeline concept and syntax and describes a workflow as a sequence of steps that are grouped in stages.

**Build**

A single execution of a project, including its runtime logs and output artifacts. This term comes from Jenkins' origins as a software build server.

**Node**

A server or container that runs builds.

**Worker**

A thread in a master node that either runs a build or orchestrates available agents.

**Workspace**

A file system folder, dedicated to a project and sometimes also to a node, where builds store data that is either temporary, or reused between multiple builds of the same project.

**Credential**

A Jenkins construct that provides projects and builds with access credentials to external resources. There are different credentials to store user name and password pairs, SSH keys, and other credential types.

**Plug-in**

Almost all of Jenkins functionality is extensible by plug-ins written in Java. There are many community plug-ins that support different kinds of nodes, credentials, and programming languages.

Types of Jenkins Nodes

There are two kinds of Jenkins nodes:

**Master**

Stores definitions of projects and their builds.

**Agent**

Run builds (or parts of a build) under the control of a master node.

A *Jenkins instance* contains one (and only one) master node and zero or more agent nodes. A master node is also an agent node, so a simple Jenkins deployment can have a single node. The following figure shows how the previous concepts of Jenkins projects and builds relate to the types of Jenkins nodes.

Figure 4.3: Jenkins Architecture

A master node provides a web UI and REST APIs to manage projects and builds.

Agents can be bare-metal servers, local containers, or pods in a Kubernetes cluster. Agent nodes also spread the load of running jobs, freeing up resources and capacity of a master node.

You can tune agent nodes according to the specific requirements of an application, such as the version of an operating system, independent of the masters. For example, a Jenkins master running Linux could use a Jenkins agent running Windows in a multiplatform application.

Managing Jenkins

The Jenkins web UI, also known as the *Jenkins Dashboard* because of its welcome page, is a web application that enables managing all aspects of a Jenkins server. The following list illustrates the main things that you can manage through the Jenkins Dashboard:

* Projects and builds for each project.
* Folders that group projects related to a business unit.
* Security settings that allow a Jenkins user to manage the server, or only those projects the user owns.
* Installed and available plug-ins.
* Online and offline agent nodes.

Jenkins also provides both a Java-based CLI and a REST API that allow limited management of a Jenkins server, including creating projects and starting builds.

A standard Jenkins setup provides few automation capabilities and is very GUI-oriented. Several plug-ins, for example the Configuration as Code (CasC) plug-in, add more extensive automation capabilities to Jenkins. These plug-ins are beyond the scope for this course.

Introducing Jenkins Declarative Pipelines

You can define a pipeline from the Jenkins Dashboard using an interactive graphical builder, and store it in a Jenkins instance. The preferred approach is writing a Jenkinsfile using a text editor and storing it on a version control system, such as Git.

A Jenkinsfile is a text file using Groovy syntax, which is very similar to Java and JavaScript. There are two possible styles for writing a Jenkinsfile:

**Scripted pipelines**

Pipelines that start with a node directive and define imperative scripts using the full Groovy programming language.

**Declarative pipelines**

Pipelines that start with a pipeline directive and define declarative scripts using a special-purpose domain-specific language (DSL) that is a subset of Groovy.

Many Jenkins plug-ins provide DSL statements and directives to perform tasks, such as running Java Maven builds, running NUnit integration tests, resolving NPM dependencies, and building container images. Note that these plug-ins may depend on software tools, such as the Java Development Kit (JDK) and the Node Package Manager (NPM) on agent nodes.

The declarative style is preferred because it is simpler. If you find a plug-in whose DSL statements require the scripted style, then you can add script blocks to a declarative pipeline and use these statements.

Describing the Structure of a Jenkinsfile

The following example illustrates a minimal Jenkinsfile, using the declarative style, that includes comments and variable substitution.

pipeline {

agent any

stages {

stage('Example') {

// use interactive input judiciously

input {

message 'Should we continue?'

ok 'Yes, we should.'

parameters {

string(name: 'PERSON', defaultValue: 'Mr Jenkins',

description: 'Who should I say hello to?')

}

}

steps {

echo "Hello, ${PERSON}, nice to meet you."

}

}

}

}

|  |  |
| --- | --- |
|  | Most directives require brackets. |
|  | This entire pipeline runs in any agent available, including the master node. |
|  | The Jenkins Dashboard displays the name of each stage and the overall status for each build. |
|  | Example of a single-line comment. |
|  | Sample input directive that displays a text entry box and assigns the value to the PERSON variable. |
|  | Line breaks and indentation are allowed inside a statement or directive. |
|  | Pipelines (and Groovy) support string quoting, escaping, and variable substitution rules similar to Bash commands. |

Most directives inside a declarative pipeline are optional.

* The pipeline directive requires one agent and one stages directive.
* The stages directive requires one or more stage directives.
* A stage directive requires one or more steps directives.

The following directives are allowed inside pipelines directive:

* triggers: defines conditions that fire automatic execution of builds from that pipeline.
* options: defines general configuration settings for the pipeline, overriding most properties from the web UI, for example: timeouts for running builds and retention of build logs.
* parameters: defines parameters that a user, or an upstream pipeline, can provide for running a build.
* environment: defines environment variables available inside a pipeline or a stage.
* agent: defines which agent nodes should execute, either all stages or a single stage of the pipeline.

Some directives, for example agent, can occur at different levels inside a pipeline, for example, at either pipeline or stage.

Most directives can be defined in any order inside a pipeline or stage.

The stage directive defines a logical piece of a workflow, for example, building, testing, or deploying an application.

Besides its mandatory steps directive, and the optional directives that were already introduced, a stage can also include:

* when: defines conditions under which the stage is executed.
* input: allows a stage to display interactive prompts and wait for an answer.

Inside a steps directive you include statements such as echo, input, and sh that are provided by Jenkins and its plug-ins.

An optional post directive can exist inside either the pipeline or stages directives. It defines steps that execute after a stage (or all stages) are completed to allow tasks, such as clean up and recovery from an error.

**REFERENCES**

CI/CD from the Wikipedia at <https://en.wikipedia.org/wiki/CI/CD>

Jenkins Glossary at <https://www.jenkins.io/doc/book/glossary/>

Jenkins Pipeline Syntax at <https://www.jenkins.io/doc/book/pipeline/syntax/>