Continuous Delivery With Jenkins Pipeline

MARK WAITE
TECHNICAL EVANGELIST AT CLOUDBEES

CONTENTS

JENKINS PIPELINE

Pipeline adds a powerful set of automation tools to Jenkins. It supports software delivery from simple continuous integration tasks to comprehensive continuous delivery pipelines. The steps to build, test, and deliver each application become part of the application itself, stored in a Jenkinsfile.

The Jenkins Pipeline provides a domain-specific language to create, edit, view, and run software delivery pipelines. Users of all experience levels can quickly create Jenkins Pipelines using interactive tools included with Jenkins.

JENKINS PIPELINE VISUALIZATION

Jenkins Pipelines are best visualized through the Jenkins Blue Ocean user interface. The Jenkins Blue Ocean user interface is focused on continuous delivery pipelines and the expectations of modern development tools. With a simple click, users can switch between traditional Jenkins pages for administrative tasks and Blue Ocean to edit, monitor, and run their pipelines.

With the focus on new users and continuous delivery, Blue Ocean and Declarative Pipeline are designed to work together. They make the creation, review, and visualization of Pipelines accessible to all members of the DevOps team. Users can quickly see pipeline status, visually create and edit new Pipelines, and personalize their view of important Pipelines. They can easily collaborate on code changes with native integrations for branches and pull requests.

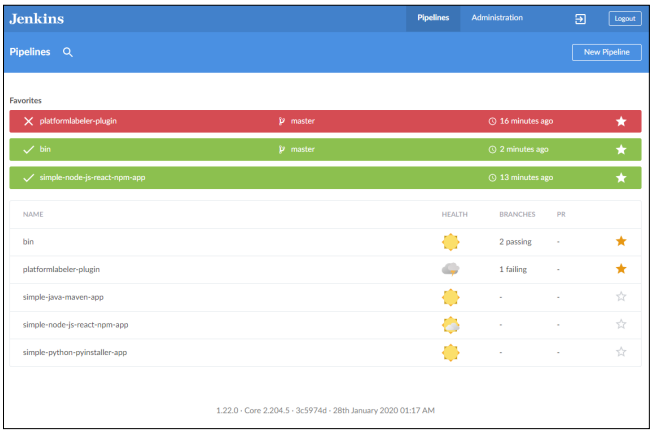

Figure 1: Introduction to Blue Ocean

CloudBees®

Build stuff that matters.

Just because you work in an enterprise, you shouldn't have to compromise on tools. Enterprise developers should be free to **embrace** diverse, best-in-class DevOps tools - not have to **replace** the tools they love. If you find yourself constrained, trying to work with a one-size-fits-all platform, let us show you what real freedom is.

With CloudBees, you keep the tools you love! You can focus on development and achieve the flow which drives innovation.

Our Software Delivery Automation solutions provide:

Continuous integration

Learn More

Continuous delivery

Learn More

Feature flag management

Learn More

Build and test acceleration

Learn More

Connect all of your DevOps tools and let us handle scaling, security and compliance while you **build stuff that matters.**

Learn more: **www.cloudbees.com**

Contact us: **info@cloudbees.com**

ABOUT THIS REFCARD

This Refcard will focus on Declarative Pipeline and Blue Ocean for all users, especially new and intermediate users. They allow users to easily create, view, and edit continuous delivery pipelines. The Refcard provides an overview of the essentials of Declarative Pipeline and Blue Ocean. It provides example snippets to illustrate specific points and a complete real-world continuous delivery example.

SYSTEM REQUIREMENTS

Declarative Pipeline and Blue Ocean are available with all Jenkins 2.x versions. This Refcard is known to work with:

- Jenkins 2.204.5 or higher
- Pipeline 2.6 or higher
- Blue Ocean 1.22 or higher

To start a new Jenkins with Pipeline and Blue Ocean preinstalled:

- Ensure Docker is installed
- Run `docker run -p 8888:8080 jenkinsci/blueocean:latest`
- Browse to http://localhost:8888/blue

CREATING A JENKINSFILE

The "Create Pipeline" experience in Blue Ocean helps users create a new Pipeline in clear, easy steps.

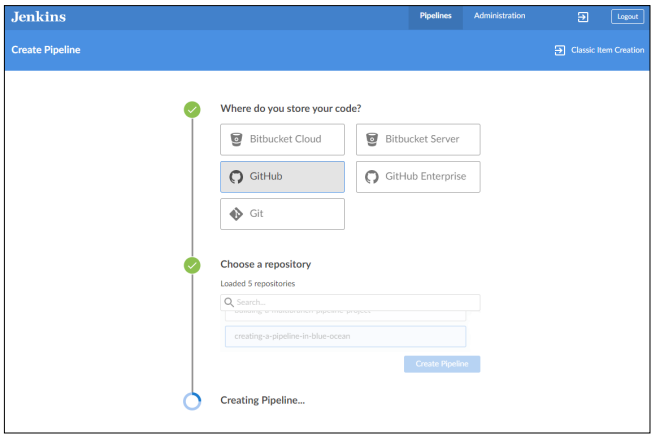

Figure 2: Creating a Pipeline

PIPELINE EDITOR

The Blue Ocean Pipeline Editor allows users to quickly design and save a new Jenkinsfile in the selected repository. If a Jenkinsfile already exists in the repository, it will create a new Pipeline based on the existing Jenkinsfile instead. You can always edit this Jenkinsfile in the editor by selecting a branch and clicking on the pencil icon.

BRANCHES AND PULL REQUESTS

When a new Pipeline is created in Blue Ocean, it also creates a new Multibranch Pipeline project for the repository. Jenkins monitors this repository and automatically creates a new Pipeline for each branch and pull request that contains a Jenkinsfile. Removing the branch, pull request, or Jenkinsfile will automatically remove the associated Pipeline.

Declarative Pipelines can adjust their execution based on a branch or pull request; adapt their execution based on user input; and publish test results and save artifacts.

PIPELINE FUNDAMENTALS

Pipelines use the definition in the Jenkinsfile to check out code on an agent and run the defined stages and steps. All Declarative Pipelines must start with pipeline and include these four directives to be syntactically correct: agent, stages, stage, and steps. These will be described in further detail in the following sections. Here is an example of a minimal Pipeline:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh 'npm -version'
      }
    }
  }
}
```

STEPS AND STAGES

STEPS

Pipelines are composed of multiple steps to build, test, and deploy applications. Think of a “step” as a single command that performs a single action. When a step succeeds it, it moves onto the next step.

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh 'echo "Hello World"'
        sh '''
           echo "Multiline shell steps"
           ls -lah
           '''
      }
    }
  }
}
```

SHELL STEPS

Jenkins is a distributed system that works across multiple nodes and executors. Jenkins can run many Pipelines being run simultaneously to orchestrate tasks. It can also run tasks on nodes with different operating systems, tools, environments, etc.

Most Pipelines work best by running native commands on different executors. This allows users to run the same commands on their local machines as Jenkins uses in the Pipeline.

Pipeline supports sh for Linux, macOS, FreeBSD, Solaris, and other Unix-like systems, and bat and PowerShell for Windows.

TIMEOUTS, RETRIES, AND OTHER WRAPPERS

Jenkins Pipeline provides powerful steps that “wrap” other steps. These "wrapper" steps help developers address common needs like retrying failing steps or exiting if a step takes too long.

Wrappers may contain multiple steps or may be recursive and contain other wrappers. We can compose these steps together. For example, if we wanted to retry our deployment five times, but never want to spend more than three minutes in total before failing the stage:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        timeout(time: 3, units: 'MINUTES') {
          retry(5) {
            powershell './flakey-deploy.ps1'
          }
        }
      }
    }
  }
}
```

STAGES

A stage in a Pipeline is a collection of related steps that share a common execution environment. Each stage must be named and must contain a steps section. Every stage is declared in the stages section of the Pipeline.

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo 'Building'
      }
    }
    stage('Test') {
      steps {
        echo 'Testing'
      }
    }
    stage('Deploy') {
      steps {
        echo 'Deploying'
      }
    }
  }
}
```

WHEN

A stage may be optionally skipped in a Pipeline based on criteria defined in the when section of the stage. The when section supports the branch and environment keywords so that stages may be skipped based on branch names or the values of environment variables.

```
pipeline {
  agent any
  stages {
    stage('Build & Test') {
      when {
        environment name: 'DEPLOY_IT', value: 'true'
      }
      steps {
        echo 'Building & Testing'
      }
    }
    stage('Deploy') {
      when {
        branch 'master'
      }
      steps {
        echo 'Deploying'
      }
    }
  }
}
```

More complex logic can use the expression keyword to evaluate a Groovy statement for its boolean return value.

```
pipeline {
  agent any
  stages {
    stage('Build & Test') {
      when {
        expression {
          'something' != 'other thing'
        }
      }
      steps {
        echo 'Building & Testing'
      }
    }
  }
}
```

CODE CONTINUED IN NEXT COLUMN

Multiple conditions may be combined in when by using the keywords `anyOf`, `allOf`, and `not`.

```
pipeline {
  agent any
  stages {
    stage('Build & Discard') {
      when {
        allOf {
          not { branch 'master' }
          expression {
            'something' != 'other thing'
          }
        }
      }
      steps {
        echo 'Building & Discarding'
      }
    }
  }
}
```

PARALLEL STAGES

Pipelines can finish sooner by running stages in parallel. For example, executables for multiple architectures might be compiled in parallel, or unit tests might run at the same time as integration tests. Parallel stages can be added with a single click from the Blue Ocean Pipeline editor.

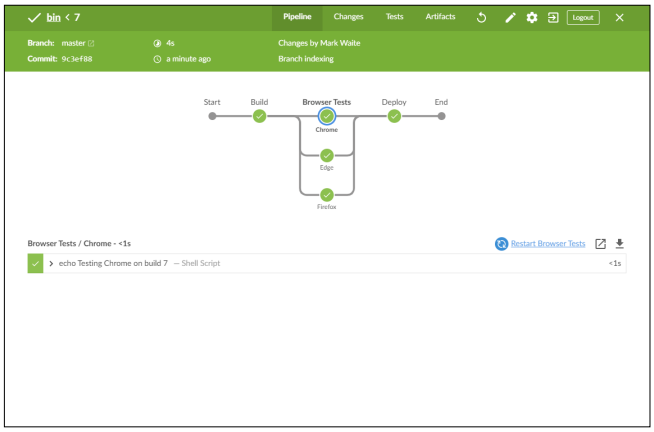

Figure 3: Parallel Pipelines

AGENTS

The agent directive tells Jenkins where to execute the Pipeline stage. Most Pipeline steps require an appropriately configured agent.

So far, all examples have used agent any, which allows the Pipeline stage to execute on any available agent. Underneath the hood, there are a few things the agent keyword causes to happen:

- All the steps contained within the block are queued for execution by Jenkins. As soon as an executor is available, the steps will begin to execute.
- A workspace is allocated which will contain files checked out from source control, as well as any additional working files for the Pipeline.

Jenkins agents are grouped by labels. Specifying an agent label restricts the potential agents that can be used in the Pipeline.

```
pipeline {
  agent {
    node {
      label 'my-agent'
    }
  }
}
```

It is also possible to use a custom workspace directory on each agent using a relative or absolute path reference to maintain a consistent file location.

```
pipeline {
  agent {
    node {
      label 'another-label'
      customWorkspace '/some/other/path'
    }
  }
}
```

DOCKER AGENTS

Pipeline is designed to easily use Docker images and containers. The Pipeline defines the environment and tools required without having to configure various system tools and dependencies on the agents. With a Docker agent, the Pipeline can use practically any tool that can be packaged in a Docker container.

```
pipeline {
  agent {
    docker {
      label 'docker-maven'
      image 'maven:3.6.3-jdk-8'
    }
  }
}
```

Containers create an immutable environment that defines the specific tools required for a build. Rather than creating one large image with every tool needed by the Pipeline, each stage can use the specific container that it needs. By reusing the workspace from one stage to the next, files remain in a single location while each stage uses the tools from its container on the files.

```
pipeline {
  agent {
    node { label 'my-docker' }
  }
  stages {
    stage('Build') {
      agent {
        docker {
          reuseNode true
          image 'maven:3.6.3-jdk-8'
        }
      }
      steps {
        sh 'mvn clean verify'
      }
    }
  }
}
```

This will check out the source code onto an agent with the label `my-docker`. It will re-use the workspace while running the build steps inside the Apache Maven 3.6.3 Docker container.

ENVIRONMENT VARIABLES AND CREDENTIALS

Environment variables can be set globally, like the example below, or per stage. As you might expect, setting environment variables per stage means they will only apply to the stage in which they're defined.

```
pipeline {
  agent any
  environment {
    DISABLE_AUTH = 'true'
    DB_ENGINE = 'sqlite'
  }
  stages {
    stage('Build') {
      steps {
        sh 'printenv'
      }
    }
  }
}
```

Environment variables in the Jenkinsfile can be referenced in build scripts to alter the behavior of the script. A Makefile, an automated test, or a deployment script may use the value of the environment variable for its execution.

Another common use for environment variables is to set or override "dummy" credentials in build or test scripts. Pipeline allows users to quickly and safely access pre-defined credentials without knowing their values and without storing credentials in the Jenkinsfile.

CREDENTIALS IN THE ENVIRONMENT

Build secrets and API tokens can be inserted into environment variables for use in the Pipeline. The snippet below is for "Secret Text" type Credentials, for example:

```
environment {
  AWS_ACCESS = credentials('AWS_KEY')
  AWS_SECRET = credentials('AWS_SECRET')
}
```

Just as the first example, these variables will be available either globally or per-stage, depending on the location of the environment directive in the Jenkinsfile.

The second most common type of Credentials is "Username and Password," which can still be used in the environment directive but results in slightly different variables being set.

```
environment {
  SAUCE_ACCESS = credentials('sauce-dev')
}
```

This will set three environment variables:

```
SAUCE_ACCESS containing <username>:<password>
SAUCE_ACCESS_USR containing the username
SAUCE_ACCESS_PSW containing the password
```

POST ACTIONS

The entire Pipeline and each Stage may optionally define a post section. This post section of the Pipeline runs at the end of the enclosing Pipeline or Stage. It is useful for removing files, archiving results, or sending notifications. A number of additional conditions blocks are supported within the post section: always, changed, failure, success, and unstable. They are executed at the end of a stage depending upon the status of the stage.

```
pipeline {
  agent any
  stages {
    stage('No-op') {
      steps {
        sh 'ls'
      }
    }
  }
  post {
    always {
      echo 'Finished'
      deleteDir() // Clean the workspace
    }
    success {
      echo 'I succeeded!'
    }
    unstable {
```

CODE CONTINUED ON NEXT PAGE

```
      echo 'I am unstable :/'
    }
    failure {
      echo 'I failed :('
    }
    changed {
      echo 'Latest job status differs from previous
job status'
    }
  }
}
```

NOTIFICATIONS

Job status notification can be sent through e-mail or through chat systems like Slack.

E-mail

```
post {
  failure {
    mail to: 'team@example.com',
         subject: 'Failed Pipeline',
         body: 'Something is wrong'
  }
}
```

Slack

```
post {
  success {
    slackSend channel: '#ops-room',
         color: 'good',
         message: 'Completed successfully'
  }
}
```

INPUT

Often, when passing between stages, especially environment stages, you may want human input before continuing. For example, to judge if the application is in a good enough state to "promote" to the production environment, this can be accomplished with the input step. In the example below, the "Sanity check" stage blocks for input and won't proceed without a person confirming the progress. The input step does not require an agent. It should always be run from a stage using agent none so that it does not reserve an agent unnecessarily.

```
stage('Sanity Check') {
  agent none
  steps {
    input 'Does the staging environment look ok?'
  }
}
```

ADVANCED PIPELINE SETTINGS

Refer to the Tutorials and Jenkins Handbook at jenkins.io/doc for more information on advanced topics such as Snippet Generator, Declarative Directive Generator, Shared Libraries, Script Blocks, Pipeline Options, and Parameterized Pipelines.

ADDITIONAL RESOURCES

- Getting Started with Jenkins Pipeline
- Shared Libraries
- Blue Ocean Documentation
- Pipeline Syntax Reference
- Pipeline Steps Reference
- CloudBees University Training
  - https://standard.cbu.cloudbees.com/
  - https://www.cloudbees.com/jenkins/training

Written by Mark Waite,

Technical Evangelist at CloudBees

Mark Waite is the Jenkins Documentation Officer. He leads the Documentation Special Interest Group and is involved in several other Special Interest Groups. He is a Technical Evangelist at CloudBees and focuses on the development and growth of Jenkins and the Jenkins community. He has delivered software commercially and in open source communities for many years. He maintains the Jenkins git plugin, has presented Jenkins training, and is active in Jenkins community forums.

DZone
A DEVADA MEDIA PROPERTY

DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of tousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensu, and Sauce Labs.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC 27709

888.678.0399 919.678.0300