WHITE PAPER

Using the Pipeline Plugin to Accelerate Continuous Delivery

cloudbees

Contents

Contents (cont.)

INTRODUCTION: THE PIPELINE PLUGIN

Jenkins is a powerful, open source automation tool with an impressive plugin architecture that helps development teams automate their software lifecycle. Jenkins is used to power many industry-leading companies' software development pipelines.

Jenkins Pipeline is a powerful, first-class feature for managing complex, multi-step pipelines. Jenkins Pipeline, a set of open source plugins and integrations, brings the power of Jenkins and the plugin ecosystem into a scriptable Domain Specific Language (DSL). Best of all, like Jenkins core, Pipeline is extensible by third-party developers, supporting custom extensions to the Pipeline DSL and various options for plugin integration.

The Pipeline plugin was formerly known as the Workflow plugin prior to version 1.13 of the plugin.

Image 1: Pipeline Stage View UI

In this whitepaper, we provide an introduction and step-by-step guide on how to use the Pipeline plugin. We conclude with a full syntex reference card and a real-world delivery pipeline example, that is built on the more basic Pipeline snippets provided earlier in the whitepaper.

Installing the Pipeline Plugin

It is assumed that you have already installed Jenkins – either via the CloudBees Jenkins Platform™ or from the Jenkins open source project website. Jenkins Version 1.609.1+ is required.

- Open Jenkins in your web browser
- Navigate to Manage Jenkins > Manage Plugins
- Navigate to the Available tab, filter by Pipeline
- Select the Pipeline plugin and install
- Restart Jenkins

This whitepaper was written using Pipeline version 1.13. Installing the Pipeline plugin installs all necessary Pipeline dependencies and a new job type called Pipeline.

Creating a Pipeline

Now that you have Jenkins running and have installed the Pipeline plugin, you are ready to create your first pipeline. Create a new pipeline by selecting New Item from the Jenkins home page.

First, give your pipeline a name, e.g.: “hello-world-flow.” Pipelines are simple Groovy scripts, so let’s add the obligatory Hello World. Add a pipeline to the Pipeline script text area:

```
echo ‘Hello world
```

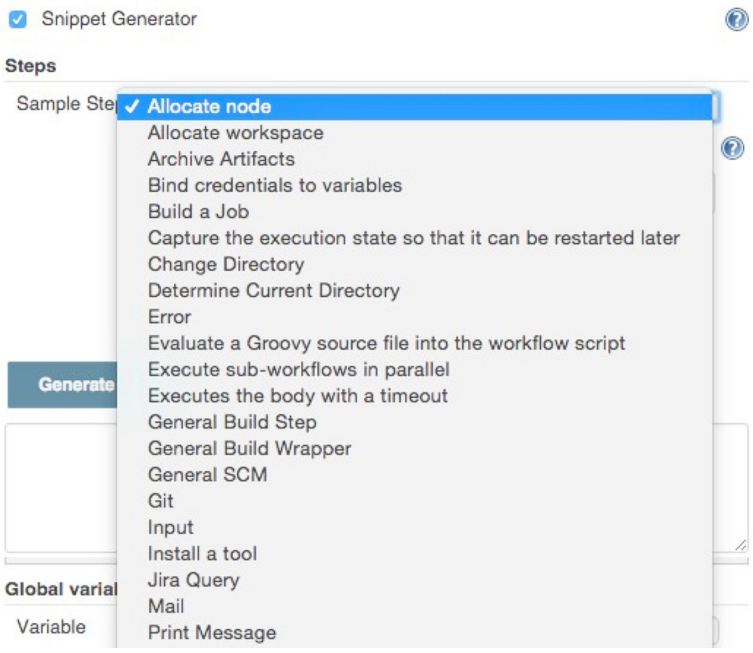
Now save your pipeline, ensuring the Use Groovy Sandbox option is checked (more details to follow on this setting). Click Build Now to run your pipeline.

EDITING YOUR PIPELINE

Because pipelines are simple text scripts, they are easy to edit. As you've seen, pipelines can be edited directly in the Jenkins UI when configuring your pipeline.

USING THE SNIPPET GENERATOR

To make editing your pipelines easier, use the Snippet Generator. The Snippet Generator is dynamically populated with the latest Pipeline steps. Depending on the plugins installed in your environment, you may see more available steps.



LOADING EXTERNAL PIPELINE SCRIPTS

Because pipelines are text assets, they are ideal to store in a source control system. Pipelines can be edited in your external IDE then loaded into Jenkins using the Pipeline Script from SCM option.

Building Your Pipeline

Now that you've created a pipeline, let's continue to build on it. For a complex flow, you should leverage Jenkins' job scheduling queue:

```
node{ sh 'uname' }
```

The concept of a node should be familiar to Jenkins users: node is a special step that schedules the contained steps to run by adding them to Jenkins' build queue. Even better, requesting a node leverages Jenkins' distributed build system. Of course, to select the right kind of node for your build, the node element takes a label expression:

```
node('unix && 64--bit'){ echo 'Hello world' }
```

The node step also creates a workspace: a directory specific to this job where you can check out source code, run commands and do other work. Resource-intensive work in your pipeline should occur on a node. You can also use the ws step to explicitly ask for another workspace on the current agent, without grabbing a new executor slot. Inside its body, all commands run in the second workspace.

CHECKING OUT CODE

Usually, your pipelines will retrieve source code from your source control server. Pipeline has a simple syntax for retrieving source code, leveraging the many existing SCM plugins for Jenkins.

```
checkout([$class: 'GitSCM', branches: [[name: '*/master']], userRemoteConfigs: [[url:
'http://github.com/cloudbees/todo--api.git']]])
```

If you happen to use a Git-based SCM – for example, GitHub - there's an even further simplified syntax:

```
git  'https://github.com/cloudbees/todo--api.git'
```

RUNNING YOUR PIPELINE

Because pipelines are built as Jenkins jobs, they can be built like other jobs. You can use the Build Now feature to manually trigger your build on-demand or set up triggers to execute your pipeline based on certain events.

ADDING STAGES AND STEPS

Stages are usually the top most element of Pipeline syntax. Stages allow you to group your build step into its component parts. By default, multiple builds of the same pipeline can run concurrently. The stage element also allows you to control this concurrency:

```
stage 'build'
node{ … }
stage name: 'test', concurrency: 3
node{ … }
stage name: 'deploy', concurrency: 1
node{ … }
```

In this example, we have set a limit of three concurrent executions of the test stage and only one execution of the deploy stage. You will likely want to control concurrency to prevent collisions (for example, deployments).

Newer builds are always given priority when entering a throttled stage; older builds will simply exit early if they are preempted.

GENERAL BUILD STEPS

Within your stages, you will add build steps. Just like with Freestyle Jenkins jobs, build steps make up the core logic of your pipeline. Jenkins Pipeline supports any compatible Build Step and populates the snippet generator with all available Build Steps in your Jenkins environment.

```
step([$class: 'JavadocArchiver', javadocDir: 'target/resources/javadoc', keepAll:
false])
```

```
step([$class: 'Fingerprinter', targets: 'target/api.war'])
```

SCRIPTING

Jenkins Pipeline supports executing shell (*nix) or batch scripts (Windows) just like freestyle jobs:

```
sh 'sleep 10'
```

```
bat 'timeout /t 10'
```

Scripts can integrate with various other tools and frameworks in your environment – more to come on tools in the next section.

Integrating Your Tools

For a real-life pipeline, Jenkins needs to integrate with other tools, jobs and the underlying environment.

SCRIPTING

Jenkins has a core capability to integrate with tools. Tools can be added and even automatically installed on your build nodes. From Pipeline, you can simply use the tool DSL syntax:

```
def mvnHome = tool 'M3'
```

```
sh "${mvnHome}/bin/mvn --B verify"
```

In addition to returning the path where the tool is installed, the tool command ensures the named tool is installed on the current node.

GLOBAL VARIABLES

The env global variable allows accessing environment variables available on your nodes:

```
echo env.PATH
```

Because the env variable is global, changing it directly is discouraged as it changes the environment globally, so the withEnv syntax is preferred (see example in Full Syntax Reference Card at the end of this whitepaper).

The currentBuild global variable can retrieve and update the following properties:

```
currentBuild.result
currentBuild.displayName
currentBuild.description
```

EXISTING JOBS

Existing jobs can be triggered from your pipeline via the build command (e.g.: build 'existing- freestyle-job'). You can also pass parameters to your external jobs as follows:

```
def job = build job: 'say--hello', parameters: [[$class: 'StringParameterValue', name: 'who', value: 'Whitepaper Readers']]
```

Controlling Flow

Because the Pipeline plugin is based on the Groovy language, there are many powerful flow control mechanisms familiar to developers and operations teams, alike. In addition to standard Groovy flow control mechanisms like 'if statements', try/catch and closures there are several flow control elements specific to Pipeline.

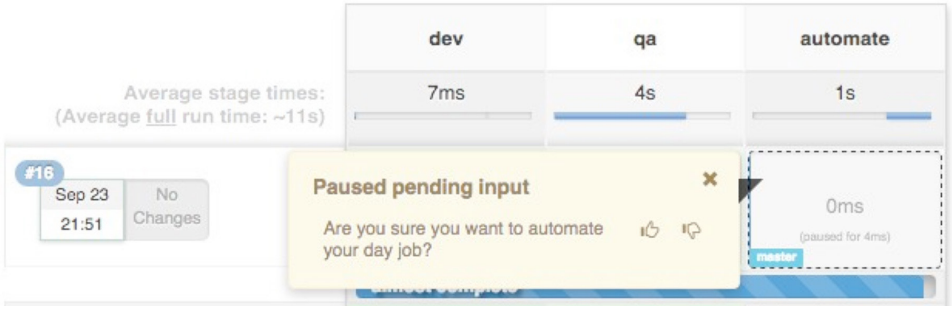
HANDLING APPROVALS

Pipeline supports approvals, manual or automated, through the input step:

```
input 'Are you sure?'
```

With the submitter parameter, the input step integrates the Jenkins security system to restrict the allowed approvers.

The input step in Jenkins Pipeline Stage View UI:



TIMING

Timeouts allow pipeline creators to set an amount of time to wait before aborting a build:

```
timeout(time: 30, unit: 'SECONDS') { … }
```

Parallel stages add a ton of horsepower to Pipeline, allowing simultaneous execution of build steps on the current node or across multiple nodes, thus increasing build speed:

```
parallel 'quality scan': {
   node {sh 'mvn sonar:sonar'}
}, 'integration test': {
   node {sh 'mvn verify'}
}
```

Jenkins can also wait for a specific condition to be true:

```
waitUntil { … }
```

HANDLING ERRORS

Jenkins Pipeline has several features for controlling flow by managing error conditions in your pipeline. Of course, because Pipeline is based on Groovy, standard try/catch semantics apply:

```
try {
} catch (e) {
}
```

Pipeline creators can also create error conditions if needed based on custom logic:

```
if(!sources) {
   error 'No sources'
}
```

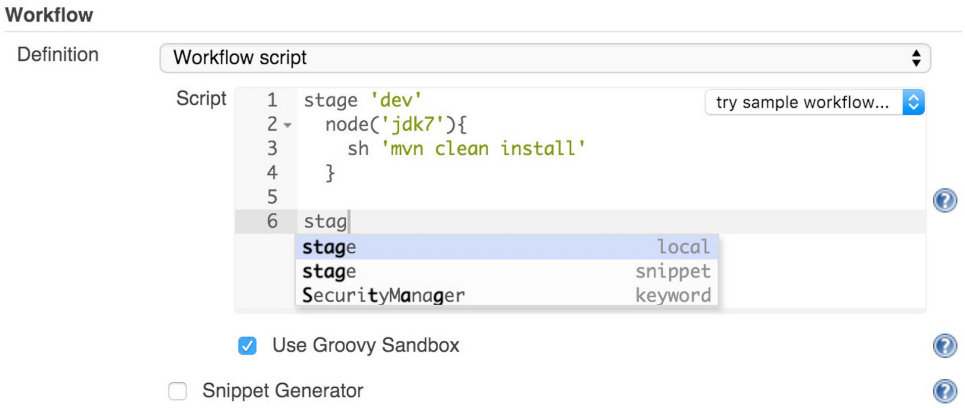
Jenkins can also retry specific Pipeline steps if there is variability in the steps for some reason:

```
retry(5) { … }
```

Script Security

As you've seen, Pipeline is quite powerful. Of course, with power comes risk, so Pipeline has a robust security and approval framework that integrates with Jenkins core security.

By default, when creating pipelines as a regular user (that is, without the Overall/RunScripts permission), the Groovy Sandbox is enabled. When the Sandbox is enabled, Pipeline creators will only be allowed to use pre-approved methods in their flow.



When adding pre-approved methods to a pipeline, script changes do not require approval. When adding a new method (such as a Java API), users will see a RejectedAccessException and an administrator will be prompted to approve usage of the specific new API or method.

Deselecting the Use Groovy Sandbox option changes this behavior. When the Sandbox is disabled, pipeline edits require administrator approval. Each change or update by a non-administrator user requires approval by an administrator. Users will see an

UnapprovedUsageException until their script is approved. Approving individual edits may not scale well, so the Groovy Sandbox is recommended for larger environments.

Accessing Files

During your pipeline development, you will very likely need to read and write files in your workspace.

STASHING FILES

Stashing files between stages is a convenient way to keep files from your workspace to share them between different nodes:

```
stage 'build'
   node{
       git  'https://github.com/cloudbees/todo--api.git'
       stash includes: 'pom.xml', name: 'pom'
   }
stage name: 'test', concurrency: 3
   node {
       unstash 'pom'
       sh 'cat pom.xml'
   }
```

Stash can be used to prevent cloning the same files from source control during different stages, while also ensuring the same exact files are used during compilation and tested in later pipeline stages.

ARCHIVING

Like other Jenkins job types, pipelines can archive their artifacts:

```
archive includes: '*.jar', excludes: '*--sources.jar'
```

Archives allow you to maintain binaries from your build in Jenkins for easy access later. Unlike stash, archive keeps artifacts around after a pipeline execution is complete (where stash is temporary).

Beyond stashing and archiving files, the following Pipeline elements also work with the file system (more details at the end of this whitepaper):

```
pwd()
dir(''){}
writeFile file: 'target/results.txt', text: ''
readFile 'target/results.txt'
fileExists 'target/results.txt'
```

Scaling Your Pipeline

As you build more of your DevOps pipelines with Pipeline, your needs will get more complex. The CloudBees Jenkins Platform helps scale Pipeline for more complex uses.

CHECKPOINTS

One powerful aspect of the CloudBees® extensions to Pipeline is the checkpoint syntax. Checkpoints allow capturing the workspace state so it can be reused as a starting point for subsequent runs:

```
checkpoint 'Functional Tests Complete'
```

Checkpoints are ideal to use after a longer portion of your pipeline has run, for example, a robust functional test suite.

PIPELINE TEMPLATES

The CloudBees Jenkins Platform has a robust Template feature. CloudBees Jenkins Platform users can create templated build steps, jobs, folders and publishers. Since Pipelines are a new job type, authors can create Pipeline templates so that similar pipelines can simply leverage the same Pipeline job template. More information on Templates is available on the CloudBees' website: https://www.cloudbees.com/products/cloudbees-jenkins-platform/enterprise- edition/features/templates-plugin

Tying It Together: Sample Pipeline

The following pipeline is an example tying together several of the Pipeline features we learned earlier. While not exhaustive, it provides a basic but complete pipeline that will help jump-start your pipeline development:

```
stage 'build'
node {
  git  'https://github.com/cloudbees/todo--api.git'
  withEnv(["PATH+MAVEN=${tool 'm3'}/bin"]) {
     sh "mvn --B –Dmaven.test.failure.ignore=true clean package"
  }
  stash excludes: 'target/', includes: '**', name: 'source'
}
stage 'test'
parallel 'integration': {
  node {
     unstash 'source'
     withEnv(["PATH+MAVEN=${tool 'm3'}/bin"]) {
        sh "mvn clean verify"
     }
  }
}, 'quality': {
  node {
     unstash 'source'
     withEnv(["PATH+MAVEN=${tool 'm3'}/bin"]) {
        sh "mvn sonar:sonar"
     }
  }
}
stage 'approve'
timeout(time: 7, unit: 'DAYS') {
     input message: 'Do you want to deploy?', submitter: 'ops'
}
stage name:'deploy', concurrency: 1
node {
     unstash 'source'
     withEnv(["PATH+MAVEN=${tool 'm3'}/bin"]) {
        sh "mvn cargo:deploy"
     }
}
```

Docker with Pipeline

The Docker Pipeline plugin exposes a Docker global variable that provides DSL for common Docker operations, only requiring a Docker client on the executor running the steps (use a label in your node step to target a Docker-enabled agent).

By default, the Docker global variable connects to the local Docker daemon. You may use the docker.withServer step to connect to a remote Docker host. The image step provides a handle to a specific Docker image and allows executing several other image related steps, including the image.inside step. The inside step will start up the specified container and run a block of steps in that container:

```
docker.image('maven:3.3.3--jdk8').inside('--v
  ~/.m2/repo:/m2repo') { sh 'mvn --Dmaven.repo.local=/m2repo
  clean package'
}
```

When the steps are complete, the container will be stopped and removed. There are many more features of the Docker Pipeline plugin; additional steps are outlined on page 20, in the Docker example at the end of this whitepaper.

Extending Pipeline

Like all Jenkins features, Pipeline relies on Jenkins' extensible architecture, allowing developers to extend Pipeline's features.

PLUGIN CAPABILITY

There are a large number of existing plugins for Jenkins. Many of these plugins integrate with Pipeline as build steps, wrappers and so on. Plugin maintainers must ensure their plugins are Pipeline-compatible. The community has documented the steps to ensure compatibility. More details on plugin development and Pipeline compatibility are on the jenkins-ci.org Wiki: https://github.com/jenkinsci/pipeline-plugin/blob/master/COMPATIBILITY.md

CUSTOM DSL

Beyond compatibility, plugin maintainers can also add specific Pipeline DSL for their plugins' behavior. The community has documented the steps to take to add plugin-specific DSL. One example is the Credentials Binding plugin, which contributes to the Credentials syntax.

Full Syntax Reference Card

Following is a full Jenkins Pipeline syntax reference card. Of course, as you add plugins or as plugins are updated new Pipeline script elements will become available in your environment. The Pipeline Snippet Generator and UI will automatically add these and any associated help text so you know how to use them!

BASICS

Pipeline Script	Examples
stage *Stage*	`stage 'build'` `stage concurrency: 3, name: 'test'`

node
Allocate a node
	`node('ubuntu') {`
	`  // some block`
	`}`

ws
Allocate a workspace
	`ws('sub--workspace') {`
	`  // some block`
	`}`

echo *Print a message*	`echo 'Hello Bees'`
batch *Windows batch script*	`bat 'dir'`
sh *Shell script*	`sh 'mvn --B verify'`
checkout *General SCM*	`checkout([$class: 'GitSCM', branches: [[name: '*/master']], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[url: 'http://github.com/cloudbees/todo--api.git']]])`
git *Git SCM*	`git 'http://github.com/cloudbees/todo--api.git'`
svn *Subversion SCM*	`svn 'svn://svn.cloudbees.com/repo/trunk/todo--api'`
step *General build step*	`step([$class: 'JUnitResultArchiver', testResults: 'target/test--reports/*.xml'])` `step([$class: 'Mailer', notifyEveryUnstableBuild: true, recipients: 'info@cloudbees.com', sendToIndividuals: false])`

wrap	`wrap([$class:'Xvnc', useXauthority: true]){`
	`  sh 'make selenium--tests'`
	`}`

tool
Install a tool
	`def mvnHome = tool name: 'M3'`
	`sh "${mvnHome}/bin/mvn --B verify"`
	`tool name: 'jgit', type: 'hudson.plugins.git.GitTool'`

mail *Send an e-mail*	`mail body: 'Uh oh.', charset: '', from: '', mimeType: '', replyTo: '', subject: 'Build Failed!', to: 'dev@cloudbees.com'`

ADVANCED

Pipeline Script	Examples
build *Build an existing job*	`build job: 'hello--world'` `build job: 'hello--world', parameters: [[$class: 'StringParameterValue', name: 'who', value: 'World']]`
checkpoint *Capture the execution state so that it can be restarted later*	`checkpoint 'testing--complete'`

withEnv
Set environment variables in a scope
	`withEnv(["PATH+MAVEN=${tool 'M3'}/bin"]) {`
	`  sh 'mvn --B verify'`
	`}`

load *Evaluate a Groovy source file into the pipeline*	`load 'deploymentMethods.groovy'`

FILE SYSTEM

Pipeline Script	Examples

dir
Change directory
	`dir('src') {`
	`  // some block`
	`}`

pwd *Get current directory*	`def dir = pwd()` `echo dir`
stash *Stash files for use later in the build*	`stash excludes: 'target/*--sources.jar', includes: 'target/*', name: 'source'`
unstash *Restore files previously stashed*	`unstash 'source'`
archive *Archive artifacts*	`archive includes:'*.jar', excludes:'*--sources.jar'`
writeFile *Write file to workspace*	`writeFile file: 'target/result.txt', text: 'Fail Whale'`
readFile *Read file from the workspace*	`def file = readFile 'pom.xml'`

fileExists
Verify if file exists in workspace
	`if(fileExists 'src/main/java/Main.java') {`
	`  // some block`
	`}`

FLOW CONTROL

Pipeline Script	Examples
sleep *Sleep*	`sleep 60` `sleep time: 1000, unit: 'NANOSECONDS'`

waitUntil
Wait for condition
	`waitUntil {`
	`  // some block`
	`}`

retry
Retry body up to N times
	`retry(5) {`
	`  // some block`
	`}`

input *Pause for manual or automated intervention*	`input 'Are you sure?'` `input message: 'Are you sure?', ok: 'Deploy', submitter: 'qa-- team'`

parallel
Execute sub-flows in parallel
	`parallel "quality scan": {`
	`  // do something`
	`}, "integration test": {`
	`  // do something else`
	`},`
	`failFast: true`

timeout
Execute body with a timeout
	`timeout(time: 30, unit: 'SECONDS') {`
	`  // some block`
	`}`

error *Stop build with an error*	`error 'No sources'`

DOCKER

Pipeline Script	Examples
image *Provides a handle to image*	`def image = docker.image('maven:3.3.3--jdk8')`

image.inside
Runs steps inside image
	`image.inside('--v /repo:/repo') {`
	`  // some block`
	`}`

image.pull *Pulls image*	`image.pull()`
image.push *Push image to registry*	`image.push()` `image.push("latest")`
image.run *Runs Docker image and returns container*	`def container = image.run("----name my--api --p 8080:8080") container.stop()`

image.withRun
Runs image and auto stops container
	`image.withRun {api --> testImg.inside("----`
	`--link=${api.id}:api")`
	`  {`
	`  // some block`
	`  }`
	`}`

image.tag *Records tag of image*	`image.tag("${tag}", false)`
image.imageName() *Provides image name prefixed with registry info*	`sh "docker pull ${image.imageName()}"`
container.id *ID of running container*	`sh "docker logs ${container.id}"`
container.stop *Stops and removes container*	`container.stop()`
build *Builds Docker image*	`docker.build("cb/api:${tag}","target")`

withServer
Runs block on given Docker server
	`docker.withServer('tcp://swarm.cloudbees.com:2376',`
	`'swarm-- certs') {`
	`  // some block`
	`}`

withRegistry
Runs blog on given Docker server
	`docker.withRegistry('https://registry.cloudbees.com/',`
	`'docker--registry--login') {`
	`  // some block`
	`}`

withTool
Specifies name of Docker client to use
	`docker.withTool('toolName') {`
	`  // some block`
	`}`

CloudBees, Inc.
2001 Gateway Place, Suite 670W | San Jose, CA 95110 | United States
www.cloudbees.com | info@cloudbees.com

0816v01