CONTENTS

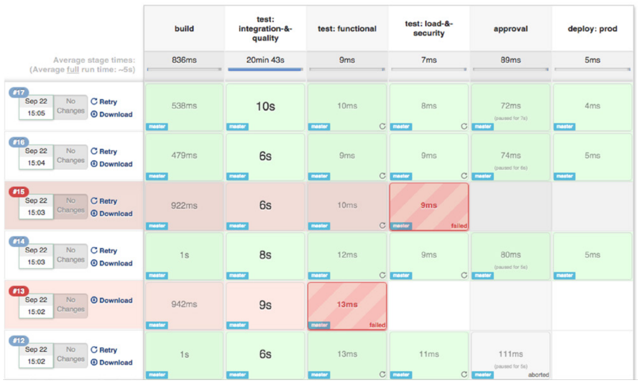
Continuous Delivery With Jenkins Workflow

By Andy Pemberton

JENKINS WORKFLOW

Jenkins is an open-source automation tool with a powerful plugin architecture that helps development teams automate their software lifecycle. Jenkins is used to drive many industry-leading companies' continuous integration and continuous delivery pipelines.

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



Average stage times: (Average full run time: ~5s)	build	test: integration-&-quality	test: functional	test: load-&-security	approval	deploy: prod
	836ms	20min 43s	9ms	7ms	89ms	5ms
#17 Sep 22 15:05 No Changes Retry Download	538ms	10s	10ms	8ms	72ms (paused for 7s)	4ms
#16 Sep 22 15:04 No Changes Retry Download	479ms	6s	9ms	9ms	74ms (paused for 6s)	5ms
#15 Sep 22 15:03 No Changes Retry Download	922ms	6s	10ms	9ms failed		
#14 Sep 22 15:03 No Changes Retry Download	1s	8s	12ms	9ms	80ms (paused for 5s)	5ms
#13 Sep 22 15:02 No Changes Download	942ms	9s	13ms failed			
#12 Sep 22 15:02 No Changes Retry Download	1s	6s	13ms	11ms	111ms (paused for 5s) aborted	

Workflow Stage View UI

This Refcard provides an overview and introduction to Jenkins Workflow, as well as a full Workflow syntax reference. It also provides a real-world delivery pipeline example, building on the more basic Workflow snippets from earlier examples.

INSTALLING JENKINS WORKFLOW

It is assumed you have already installed Jenkins—either via the CloudBees Jenkins Platform or jenkins-ci.org. For Jenkins Workflow, Jenkins version 1.580+ is required; version 1.609.1+ is recommended. To install Jenkins Workflow:

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

This Refcard was written using Workflow version 1.10. Installing the Workflow Aggregator installs all necessary Workflow dependencies and a new job type called Workflow.

CREATING A WORKFLOW

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

First, give your workflow a name (e.g., "hello-world-flow"). Workflows are simple Groovy scripts, so let's add the obligatory Hello World. Add a workflow to the Workflow script text area:

```
echo 'Hello world'
```

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

EDITING YOUR WORKFLOW

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

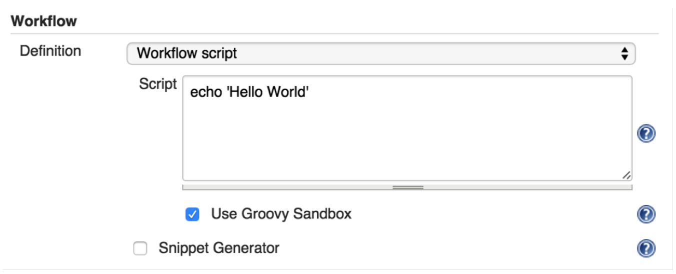
CloudBees The Enterprise Jenkins Company

On-Demand Webinar

Orchestrating the Continuous Delivery Process in Jenkins with Workflow

WATCH NOW

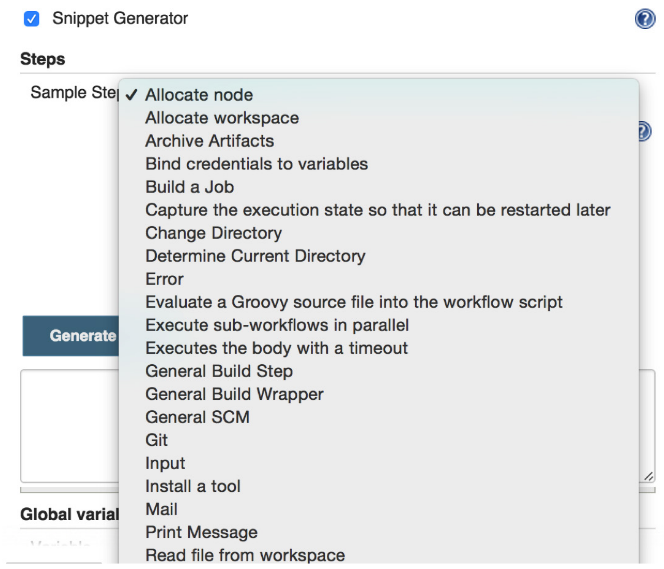
www.cloudbees.com/workflow-webinar



Workflow
Definition: Workflow script
Script:
```
echo 'Hello World'
```
[✓] Use Groovy Sandbox
[ ] Snippet Generator

USING THE SNIPPET GENERATOR

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



[✓] Snippet Generator

Steps
Sample Step: ✓ Allocate node

Allocate node
Allocate workspace
Archive Artifacts
Bind credentials to variables
Build a Job
Capture the execution state so that it can be restarted later
Change Directory
Determine Current Directory
Error
Evaluate a Groovy source file into the workflow script
Execute sub-workflows in parallel
Executes the body with a timeout
General Build Step
General Build Wrapper
General SCM
Git
Input
Install a tool
Mail
Print Message
Read file from workspace
Generate
Global varial

LOADING EXTERNAL WORKFLOW SCRIPTS

Because workflows are text assets, they are ideal to store in a source control system. Workflows can be edited in your external IDE then loaded into Jenkins using the Workflow script from SCM option.

BUILDING YOUR WORKFLOW

Now that you've created a workflow, 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 sources, 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 slave, without grabbing a new executor slot. Inside its body all commands run in the second workspace.

CHECKING OUT CODE

Usually, your workflows will retrieve source code from your source control server. Jenkins Workflow 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 like GitHub, there's an even further simplified syntax:

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

RUNNING YOUR WORKFLOW

Because workflows 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 topmost element of Workflow syntax. Stages allow you to group your build step into its component parts. By default, multiple builds of the same workflow 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 free-style Jenkins jobs, build steps make up the core logic of your pipeline. Jenkins Workflow 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 Workflow supports executing shell (*nix) or batch scripts (Windows) just like free-style 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 workflow, Jenkins needs to integrate with other tools, jobs, and the underlying environment.

TOOLS

Jenkins has a core capability to integrate with tools. Tools can be added and even automatically installed on your build nodes. From Workflow, 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 the Workflow Full Syntax Reference below).

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 workflow 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: 'DZone
Readers']]
```

CONTROLLING FLOW

Because Jenkins Workflow 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 Workflow.

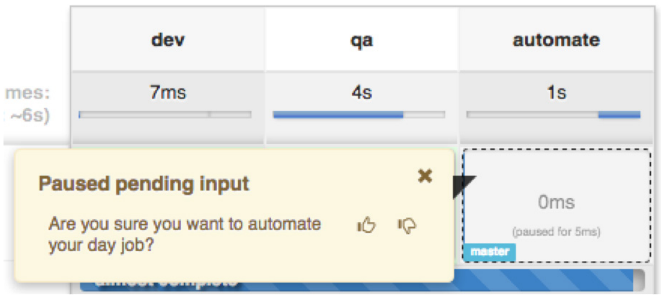
HANDLING APPROVALS

Jenkins Workflow supports approvals, manual or automated, through the input step:

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

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

The input step in Jenkins Workflow Stage View UI:



Paused pending input

Are you sure you want to automate your day job?

TIMING

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

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

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

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

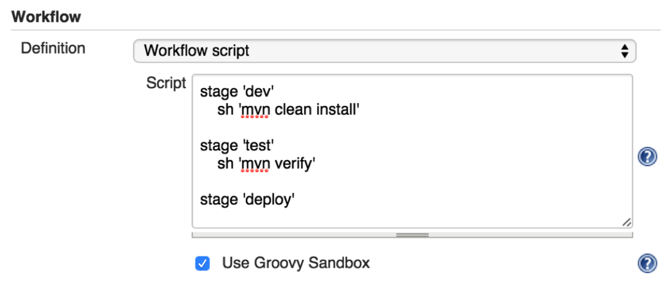
Jenkins can also retry specific Workflow steps if there is variability in the steps for some reason:

```
retry(5) { … }
```

SCRIPT SECURITY

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

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



Workflow

Definition: Workflow script

Script:

```
stage 'dev'
  sh 'mvn clean install'

stage 'test'
  sh 'mvn verify'

stage 'deploy'
```

[✓] Use Groovy Sandbox

When adding pre-approved methods to a workflow, 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, workflow edits require administrator approval. Each change or update by a non-administrator user requires approval by an administrator. Users will see an `UnnaprovedUsageException` 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 workflow 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 in order 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, workflows 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 workflow execution is complete (whereas stash is temporary).

Beyond stashing and archiving files, the following Workflow elements also work with the file system (more details in full syntax reference):

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

SCALING YOUR WORKFLOW

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

CHECKPOINTS

One powerful aspect of the CloudBees extensions to Jenkins Workflow 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 workflow has run (for example, a robust functional test suite).

WORKFLOW TEMPLATES

The CloudBees Jenkins Platform has a robust template feature. CloudBees Jenkins Platform users can create template build steps, jobs, folders, and publishers. Since Workflows are a new job type, authors can create Workflow templates so that similar pipelines can simply leverage the same Workflow 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: EXAMPLE PIPELINE

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

The Docker Workflow plugin exposes a `docker` global variable that provides a 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 slave).

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 Workflow plugin; additional steps are outlined in the Workflow Full Syntax Reference.

EXTENDING WORKFLOW

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

PLUGIN COMPATIBILITY

There are a large number of existing plugins for Jenkins. Many of these plugins integrate with Workflow as build steps, wrappers, and so on. Plugin maintainers must ensure their plugins are Workflow-compatible. The Jenkins community has documented the steps to ensure compatibility. More details on plugin development and Workflow compatibility are on the jenkins-ci.org Wiki:

https://wiki.jenkins-ci.org/display/JENKINS/Plugin+tutorial

CUSTOM DSL

Beyond compatibility, plugin maintainers can also add specific statements to the Workflow DSL for their plugins' behaviors. The Jenkins community has documented the steps to take to add plugin-specific statements to the Workflow DSL. Examples include the Credentials Binding Plugin, which contributes the `withCredentials` statement.

WORKFLOW FULL SYNTAX REFERENCE

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

BASICS

WORKFLOW SCRIPT	EXAMPLE(S)

stage
Stage
	`stage 'build'`
	`stage concurrency: 3, name: 'test'a`

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

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

WORKFLOW SCRIPT	EXAMPLE(S)
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

WORKFLOW SCRIPT	EXAMPLE(S)

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 workflow	`load 'deploymentMethods.groovy'`

FILE SYSTEM

WORKFLOW SCRIPT	EXAMPLE(S)

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

WORKFLOW SCRIPT	EXAMPLE(S)

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 without a timeout
	`timeout(time: 30, unit: 'SECONDS')`
	`{`
	`    // some block`
	`}`

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

DOCKER

WORKFLOW SCRIPT	EXAMPLE(S)

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()`

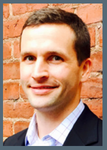
build
Builds Docker image
	`docker.build("cb/`
	`api:${tag}","target")`

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

withRegistry
Runs block using specified Docker registry
	`docker.withRegistry('https://`
	`registry.cloudbees.com/', 'docker-`
	`registry-login') {`
	`   // some block`
	`}`

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

ABOUT THE AUTHOR



Andy Pemberton leads CloudBees' Solution Architecture team and has many years' experience helping organizations ship higher quality software. Andy and his team work with CloudBees customers and internally with CloudBees product, sales, and engineering teams to help customers understand and start using the CloudBees Jenkins Platform. Based on his real-world DevOps and software delivery experience, Andy provides a realistic, practical approach to helping CloudBees customers implement Continuous Delivery with Jenkins. Andy speaks, blogs, and writes regularly in various outlets and industry events.

ADDITIONAL RESOURCES

Jenkins Workflow Plugin Wiki:
https://wiki.jenkins-ci.org/display/JENKINS/Workflow+Plugin

Jenkins Workflow Plugin on GitHub:
https://github.com/jenkinsci/workflow-plugin

Cloudbees CD With Jenkins Workflow:
https://www.cloudbees.com/products/jenkins-enterprise/workflow

Jenkins Workflow Additional Tutorial:
https://github.com/jenkinsci/workflow-plugin/blob/master/TUTORIAL.md

CREDITS:

Editor: **G. Ryan Spain** | Designer: **Yassee Mohebbi** | Production: **Chris Smith** | Sponsor Relations: **Chris Brumfield** | Marketing: **Chelsea Bosworth**

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0 $7.95

CloudBees®

The Enterprise Jenkins Company

Continuous Delivery Powered by Jenkins

Your DevOps Foundation

Build → Test → Stage → Deploy

The Enterprise Jenkins Company

Learn more: www.cloudbees.com/jenkins-workflow