**Pipeline Groovy**

1. Click “Pipeline” tab to bring that section up.

PROTIP: On Linux CentOS, the default folder Jenkins looks for the Jenkinsfile is (replacing “box2” with your job item name):

/var/lib/jenkins/workspace/box2@script/Jenkinsfile

Instead of manually clicking through the Jenkins UI, the [Pipeline plugin](https://jenkins.io/solutions/pipeline/) in Jenkins 2 reads a text-based **Jenkinsfile** [Groovy script code](https://wilsonmar.github.io/jenkins2-pipeline/#Groovy) checked into source control.

**Jenkinsfile format**

The Jenkins file used to build the Jenkins.io website is  
[https://github.com/jenkins-infra/jenkins.io/blob/master/Jenkinsfile](https://github.com/jenkins-infra/jenkins.io/blob/master/Jenkinsfile/)

The first line “shebang” defines the file as a Groovy language script:

#!/usr/bin/env groovy

In-line Pipeline files do not have a “shebang” because it is supplied internally.

1. Add Groovydoc comments
2. /\*\*
3. \* ReqA Class description
4. \*/

PROTIP: Use of GitHub reduces the need for this, but it’s helpful for special notes.

1. Select from the “try sample” pull down “Hello World” or, alternately, highlight the code below and paste it on the form:
2. #!/usr/bin/env groovy
4. /\*\*
5. \* Sample Jenkinsfile for Jenkins2 Pipeline
6. \* from https://github.com/hotwilson/jenkins2/edit/master/Jenkinsfile
7. \* by wilsonmar@gmail.com
8. \*/
10. import hudson.model.\*
11. import hudson.EnvVars
12. import groovy.json.JsonSlurperClassic
13. import groovy.json.JsonBuilder
14. import groovy.json.JsonOutput
15. import java.net.URL
17. try {
18. node {
19. stage '\u2776 Stage 1'
20. echo "\u2600 BUILD\_URL=${env.BUILD\_URL}"
22. def workspace = pwd()
23. echo "\u2600 workspace=${workspace}"
25. stage '\u2777 Stage 2'
26. } // node
27. } // try end
28. catch (exc) {
29. /\*
30. err = caughtError
31. currentBuild.result = "FAILURE"
32. String recipient = 'infra@lists.jenkins-ci.org'
33. mail subject: "${env.JOB\_NAME} (${env.BUILD\_NUMBER}) failed",
34. body: "It appears that ${env.BUILD\_URL} is failing, somebody should do something about that",
35. to: recipient,
36. replyTo: recipient,
37. from: 'noreply@ci.jenkins.io'
38. \*/
39. } finally {
41. (currentBuild.result != "ABORTED") && node("master") {
42. // Send e-mail notifications for failed or unstable builds.
43. // currentBuild.result must be non-null for this step to work.
44. step([$class: 'Mailer',
45. notifyEveryUnstableBuild: true,
46. recipients: "${email\_to}",
47. sendToIndividuals: true])
48. }
50. // Must re-throw exception to propagate error:
51. if (err) {
52. throw err
53. }
54. }

Each line in this sample is explained when the log is shown, below.

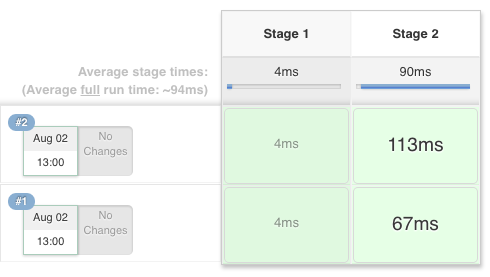
1. Click Save for the item screen.

**Build Now for Stage View**

1. Click on “Build Now”.

PROTIP: Only click once on Jenkins links or two executions will result from a double-click.

A sample response:



NOTE: Text in headings were specified in state keywords in the Groovy script above.

1. Cursor over one of the “ms” numbers (for milliseconds or thousands of a second) in the green area and click the Log button that appears.

A pop-up appears with the text specified by the echo command within the Groovy script.

1. Click the “X” at the upper-right of the dialog to dismiss it.
2. To remove the menu on the left, click “Full Stage View”.

PROTIP: The “full stage view” will be needed when there are more stages going across the screen.

**Build History**

1. To return to the menu with Build History,   
   click the item/job name in the breadcrumbs or press command+left arrow.
2. Click one of the jobs in the Build History section in http://…/job/box/5/

Notice the number in the URL corresponds to the number listed.

PROTIP: The time of the run is the server’s time, not your local time on your laptop.

1. Click “Console Output” for log details created from that run.

We now try various other Groovy scripting techniques. But first:

**Vary Groovy scripting**

The analysis of the Console Log from running the sample Groovy script consists of these topics:

* [Jenkinsfile vs inline](https://wilsonmar.github.io/jenkins2-pipeline/#Jenkinsfile)
* [Stages](https://wilsonmar.github.io/jenkins2-pipeline/#Stages)
* [Imports](https://wilsonmar.github.io/jenkins2-pipeline/#Imports)
* [Try Catch to email](https://wilsonmar.github.io/jenkins2-pipeline/#TryCatch)
* [Environment Variables](https://wilsonmar.github.io/jenkins2-pipeline/#EnvVars)
* [Unicode icons](https://wilsonmar.github.io/jenkins2-pipeline/#UnicodeIcons)
* [Color wrapper](https://wilsonmar.github.io/jenkins2-pipeline/#ColorWarapper)
* [Specific Git URL](https://wilsonmar.github.io/jenkins2-pipeline/#GitURL)
* [Checkout SCM](https://wilsonmar.github.io/jenkins2-pipeline/#CheckoutSCM)

#!/usr/bin/env groovy is nickamed the “shebang” to announce that the file is in Groovy-language formatting. PROTIP: This is not needed for in-line scripts, but there in case in case this is copied to a [Jenkinsfile](https://wilsonmar.github.io/jenkins2-pipeline/" \l "Jenkinsfile).

\\*\* with two asterisk is the code for [code scanners](https://wilsonmar.github.io/jenkins2-pipeline/#CodeScanners) which extract metadata from comments in all related files to come up with an analysis of the codebase.

stage commands are used to separate timings reported in the log.

\u2776 and \u2777 are Unicode for a black dot with a 1 and 2 in it, to make “Entering stage” lines faster to find.

\u2600 is Unicode for a “star” icon to make frequently referenced information faster to find.

Notice that unlike Java code, there are **no semicolons**.

Code between the braces ({ and }) is the **body** of the node step.

We will be going to alter this code in the [next section](https://wilsonmar.github.io/jenkins2-pipeline/#VaryGroovy).

**Infrastructure as code**

Having infrastructure defined in text code enables experimentation (using experimental branches).

This improves quality by making use of **code review** features in GitHub.

Having infrastructure specifications in the same repository ensures that changes are automatically reflected.

https://github.com/jenkinsci/pipeline-examples

* Run several repos within an organization within GitHub.

[Jesse on GitHub](https://www.youtube.com/watch?v=emV60CcDVV0&t=1h6m7s)

**Jenkinsfile vs inline Groovy Scripts**

Here we want go beyond basic scripts and look at scripts used in **production** (productive) use, which are more complicated/complex than almost all the tutorials on the internet:

* The Jenkinsfile Groovy script used to build the website [Jenkins.io](https://github.com/jenkins-infra/jenkins.io/blob/master/Jenkinsfile)
* [The Jenkins file used to build FreeBSD](https://github.com/freebsd/freebsd-ci/blob/master/scripts/build/build-test.groovy) by Craig Rodrigues (rodrigc@FreeBSD.org)
* [Jenkinsfile with Maven](https://github.com/Jotschi/maven-release-workflow-test) by [Johannes Schüth](http://jotschi.de/)

Lessons from these are provided below.

**Imports**

The top set of lines in Jenkinsfile Groovy scripts need to be import statements, if any, to pull in libraries containing functions and methods referenced in the script. Examples:

import hudson.model.\*

import hudson.EnvVars

import groovy.json.JsonSlurperClassic

import groovy.json.JsonBuilder

import groovy.json.JsonOutput

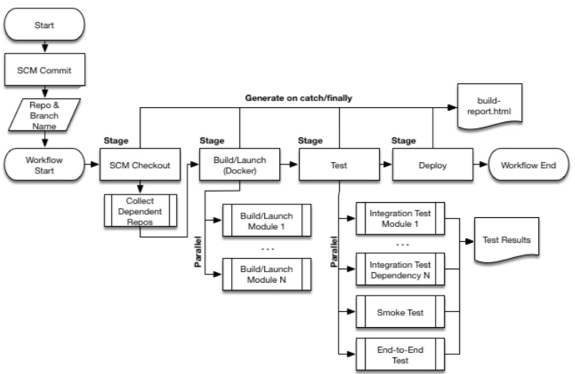
import java.net.URL

Hudson is the previous name of Jenkins before it forked.

The official documentation page for the Apache Groovy language is   
<http://www.groovy-lang.org/documentation.html#gettingstarted>

Groovy was used to build [Gradle](https://gradle.org/" \t "_blank) because it can handle larger projects than Maven, which Gradle replaces.

**Stages**



This diagram from Jenkins.io illustrates the flow of work.

**Try Catch to email**

NOTE: Groovy is a derivative of Java, so it has Java’s capability to catch (handle) execution **exceptions** not anticipated by the programming code.

Notice that

err = caughtError

currentBuild.result = "FAILURE"

[This sample](https://github.com/jenkins-infra/jenkins.io/blob/master/Jenkinsfile/) catch block sends out an email:

catch (exc) {

String recipient = 'infra@lists.jenkins-ci.org'

mail subject: "${env.JOB\_NAME} (${env.BUILD\_NUMBER}) failed",

body: "It appears that ${env.BUILD\_URL} is failing, somebody should do something about that",

to: recipient,

replyTo: recipient,

from: 'noreply@ci.jenkins.io'

}

### Environment Variables #

To add text to actual build page, you can use [Groovy Postbuild plugin](https://wiki.jenkins-ci.org/display/JENKINS/Groovy+Postbuild+Plugin) to execute a groovy script in the Jenkins JVM to checs some conditions and changes accordingly the build result, puts badges next to the build in the build history and/or displays information on the build summary page.

def workspace = manager.build.getEnvVars()["WORKSPACE"]

String fileContents = new File('${workspace}/filename.txt').text

manager.createSummary("folder.gif").appendText("${fileContents }")

TODO: Verify the above works.

TODO: This doesn’t work:

def workspace = manager.build.getEnvVars()["WORKSPACE"]

env.WORKSPACE = pwd() // present working directory.

def version = readFile "${env.WORKSPACE}/version.txt"

EnvInject Plugin

Use the withEnv step to set a variable within a temporary scope:

node ('pull'){

git url: 'https://github.com/jglick/simple-maven-project-with-tests.git'

withEnv(["PATH+MAVEN=${tool 'M3'}/bin"]) {

sh 'mvn -B verify'

}

}

[Here](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/gitcommit/gitcommit.groovy) is an example of doing a Bash shell call to invok git commands on the Git client on Jenkins agent machines. This sends STDOUT output to a file (in workspace) named “GIT\_COMMIT”:

node {

// Get SHA1 token for the src folder:

sh('cd src && git rev-parse HEAD > GIT\_COMMIT')

git\_commit=readFile('src/GIT\_COMMIT')

short\_commit=git\_commit.take(6)

// Get first 6 char. of SHA1 token and use it to retrieve the image docker builds:

sh 'git rev-parse HEAD > GIT\_COMMIT'

def shortCommit = readFile('GIT\_COMMIT').take(6)

def image = docker.build(jenkinsciinfra/bind.build-${shortCommit})")

}

The [git rev-parse](https://git-scm.com/docs/git-rev-parse" \t "_blank) command is a internal Git utility to parse (pick out) revision/object names from a Git repo. This is done after Git has done a checkout to establish the branch and specific commit because the output is a SHA1 hash of the HEAD such as “2b9a2833bc3c6bc8e7b7344e8178ce98e29ebe4b”.

But not to a Pipeline job. Thus the need for shell commands.

git\_commit.take(6) extracts the first six characters to create a short SHA, much like what Git does because a smaller number of characters are enough to uniquely identify a specific commit.

https://docs.docker.com/engine/reference/commandline/build/

Such information was previously exposed to freestyle jobs by the Git plugin exposing[environment variables](https://wiki.jenkins-ci.org/display/JENKINS/Building+a+software+project#Buildingasoftwareproject-JenkinsSetEnvironmentVariables) set when a Jenkins job executes. The following table contains a list of [all these environment variables](https://github.com/jenkinsci/job-dsl-plugin/wiki/User-Power-Moves#access-the-jenkins-environment-variables):

|  |  |
| --- | --- |
| **Environment Variable** | **Description** |
| BUILD\_CAUSE | BUILD\_CAUSE\_USERIDCAUSE |
| BUILD\_ID | The current build id, such as "2005-08-22\_23-59-59" (YYYY-MM-DD\_hh-mm-ss, [defunct](https://issues.jenkins-ci.org/browse/JENKINS-26520) since version 1.597) |
| BUILD\_NUMBER | The current build number, such as "153" |
| JOB\_NAME | Name of the project of this build. This is the name you gave your job when you first set it up. It's the third column of the Jenkins Dashboard main page. |
| BUILD\_TAG | String of jenkins-${JOB\_NAME}-${BUILD\_NUMBER}. Convenient to put into a resource file, a jar file, etc for easier identification. |
| BUILD\_URL | The URL where the results of this build can be found (e.g. http://buildserver/jenkins/job/MyJobName/666/) |
| EXECUTOR\_NUMBER | The unique number that identifies the current executor (among executors of the same machine) that's carrying out this build. This is the number you see in the "build executor status", except that the number starts from 0, not 1. |
| JENKINS\_URL | Set to the URL of the Jenkins master that's running the build. This value is used by [Jenkins CLI](https://wilsonmar.github.io/display/JENKINS/Jenkins+CLI) for example |
| HOME | - |
| HUDSON\_HOME | - |
| JAVA\_HOME | If your job is configured to use a specific JDK, this variable is set to the JAVA\_HOME of the specified JDK. When this variable is set, PATH is also updated to have $JAVA\_HOME/bin. |
| LANG | - |
| LOGNAME | - |
| NODE\_NAME | The name of the node the current build is running on. Equals 'master' for master node. |
| NODE\_LABELS | - |
| WORKSPACE | The absolute path of the workspace. |
| GIT\_COMMIT | For Git-based projects, this variable contains the Git hash of the commit checked out for the build (like ce9a3c1404e8c91be604088670e93434c4253f03) ﻿(all the GIT\_\* variables require git plugin) |
| GIT\_URL | For Git-based projects, this variable contains the Git url (like git@github.com:user/repo.git or [https://github.com/user/repo.git]) |
| GIT\_BRANCH | For Git-based projects, this variable contains the Git branch that was checked out for the build (normally origin/master) |
| CVS\_BRANCH | For CVS-based projects, this variable contains the branch of the module. If CVS is configured to check out the trunk, this environment variable will not be set. |
| SVN\_REVISION | For Subversion-based projects, this variable contains the revision number of the module. If you have more than one module specified, this won't be set. |
| PWD | - |
| OLDPWD | - |
| SHELL | - |
| TERM | - |
| USER | - |

**Tokenize environment variable**

[Here](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/github-org-plugin/access-repo-information.groovy) is an example of a Groovy script file “access-repo-information.groovy”.

Like other Groovy files, it has in the first line #!groovy.

#!groovy

tokens = "${env.JOB\_NAME}".tokenize('/')

org = tokens[0]

repo = tokens[1]

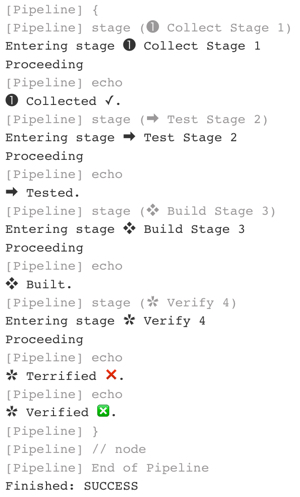
branch = tokens[2]

echo 'account-org/repo/branch=' + org +'/'+ repo +'/'+ branch

${env.JOB\_NAME} retrieves environment variable JOB\_NAME which contains the Git path “org/repo/branch” among github-organization-plugin jobs.

tokenize extracts out text between the slash character specified into an array named “tokens”. It is a [Groovy String method](http://docs.groovy-lang.org/latest/html/api/org/codehaus/groovy/runtime/StringGroovyMethods.html).

**Unicode icons**

PROTIP: Putting the same visual mark in both the stage name and echos related to the stage makes them visually easier to identify together.

stage '\u273F Verify 4'

* “\u2776” = ❶
* “\u27A1” = ➡
* “\u2756” = ❖
* “\u273F” = ✿
* “\u2795” = ➕
* “\u2713” = ✓
* “\u2705” = ✅
* “\u274E” = ❎
* “\u2717” = ✗
* “\u274C” = ❌
* “\u2600” = ☀
* “\u2601” = ☁
* “\u2622” = ☢
* “\u2623” = ☣
* “\u2639” = ☹
* “\u263A” = ☺
* [More icons in the \u2700 Unicode Digbats block](http://www.fileformat.info/info/unicode/block/dingbats/list.htm).
* [More icons in the \u2600 range](http://www.w3schools.com/charsets/ref_utf_symbols.asp)

**Color wrapper Stage View**

[Here](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/ansi-color-build-wrapper/AnsiColorBuildWrapper.groovy) is an example of adding color in a stage name:

wrap([$class: 'AnsiColorBuildWrapper']) {

stage "\u001B[31m I'm Red \u2717 \u001B[0m Now not"

}

This rather geeky technique uses Unicode “\u001B” ESCAPE codes followed by ANSI characters:

* “\u001B[31m” = RED
* “\u001B[30m” = BLACK
* “\u001B[32m” = GREEN
* “\u001B[33m” = YELLOW
* “\u001B[34m” = BLUE
* “\u001B[35m” = PURPLE
* “\u001B[36m” = CYAN
* “\u001B[37m” = WHITE
* “\u001B[0m” is for RESET

**Time stamp wrapper**

[Here](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/timestamper-wrapper/timestamperWrapper.groovy) is an example of invoking a build wrapper that adds a time stamp to echo output to the console log :

wrap([$class: 'TimestamperBuildWrapper']) {

echo "Done"

}

### Specific Git URL #

In single-branch contexts, one can download a specific repo from GitHub into Jenkins’s local workspace:

node {

git url: "https://github.com/hotwilson/jenkins2.git", branch: 'master'

sh 'make all'

}

CAUTION: The “.git” at the end is necessary in the URL and the repo needs to contain a **Jenkinsfile** (no file extension).

A sample Console response to git url:

[Pipeline] git

Cloning the remote Git repository

Cloning repository https://github.com/hotwilson/jenkins2.git

> git init /var/lib/jenkins/workspace/box2 # timeout=10

Fetching upstream changes from https://github.com/hotwilson/jenkins2.git

> git --version # timeout=10

> git -c core.askpass=true fetch --tags --progress https://github.com/hotwilson/jenkins2.git +refs/heads/\*:refs/remotes/origin/\*

> git config remote.origin.url https://github.com/hotwilson/jenkins2.git # timeout=10

> git config --add remote.origin.fetch +refs/heads/\*:refs/remotes/origin/\* # timeout=10

> git config remote.origin.url https://github.com/hotwilson/jenkins2.git # timeout=10

Fetching upstream changes from https://github.com/hotwilson/jenkins2.git

> git -c core.askpass=true fetch --tags --progress https://github.com/hotwilson/jenkins2.git +refs/heads/\*:refs/remotes/origin/\*

> git rev-parse refs/remotes/origin/master^{commit} # timeout=10

> git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10

Checking out Revision b5f1136a0e55363ff143d6ad5b311f7838d8ad82 (refs/remotes/origin/master)

> git config core.sparsecheckout # timeout=10

> git checkout -f b5f1136a0e55363ff143d6ad5b311f7838d8ad82 # timeout=10

> git branch -a -v --no-abbrev # timeout=10

> git checkout -b master b5f1136a0e55363ff143d6ad5b311f7838d8ad82

First time build. Skipping changelog.

Additional response for make not included here.

**Checkout SCM**

An example using it:

node {

checkout scm

sh 'git rev-parse HEAD > GIT\_COMMIT'

def shortCommit = readFile('GIT\_COMMIT').take(6)

def image = docker.build(jenkinsciinfra/bind.build-${shortCommit})")

stage 'Deploy'

image.push()

}

TODO: See https://github.com/jenkinsci/workflow-scm-step-plugin#generic-scm-step

**Multiple SCM**

As described in  
<https://github.com/jenkinsci/workflow-scm-step-plugin>:

While freestyle projects can use the Multiple SCMs plugin to check out more than one repository, or specify multiple locations in SCM plugins that support that (notably the Git plugin), this support is quite limited.

In a Pipeline type job, you can check out multiple SCMs, of the same or different kinds, in the same or different workspaces, wherever and whenever you like. For example, to check out and build several repositories in parallel, each on its own slave:

parallel repos.collectEntries {repo -> [/\* thread label \*/repo, {

node {

dir('sources') { // switch to subdir

git url: "https://github.com/user/${repo}"

sh 'make all -Dtarget=../build'

}

}

}]}

**NodeJS**

To work with NodeJS.

**Push changes to Git**

[Here](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/push-git-repo/pushGitRepo.Groovy) an example of pushing changes to Git from inside Pipeline.

withCredentials([[$class: 'UsernamePasswordMultiBinding', credentialsId: 'MyID', usernameVariable: 'GIT\_USERNAME', passwordVariable: 'GIT\_PASSWORD']]) {

sh("git tag -a some\_tag -m 'Jenkins'")

sh('git push https://${GIT\_USERNAME}:${GIT\_PASSWORD}@<REPO> --tags')

}

This approach requires the repository to be setup in Jenkins to be authenticated to the repo (GitHub) using username and password rather than other methods.

A [open JIRA](https://issues.jenkins-ci.org/browse/JENKINS-28335) requests getting the GitPublisher Jenkins functionality working with Pipeline.

TODO: For SSH private key authentication, try the sshagent step from the SSH Agent plugin.

**Git stash**

[This](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/unstash-different-dir/unstashDifferentDir.groovy) script uses the Git stash command to move text among Jenkins nodes.

// First we'll generate a text file in a subdirectory on one node and stash it.

stage "first step on first node"

// Run on a node with the "first-node" label.

node('first-node') {

// Make the output directory.

sh "mkdir -p output"

// Write a text file there.

writeFile file: "output/somefile", text: "Hey look, some text."

// Stash that directory and file.

// Note that the includes could be "output/", "output/\*" as below, or even

// "output/\*\*/\*" - it all works out basically the same.

stash name: "first-stash", includes: "output/\*"

}

// Next, we'll make a new directory on a second node, and unstash the original

// into that new directory, rather than into the root of the biuld.

stage "second step on second node"

// Run on a node with the "second-node" label.

node('second-node') {

// Run the unstash from within that directory!

dir("first-stash") {

unstash "first-stash"

}

// Look, no output directory under the root!

// pwd() outputs the current directory Pipeline is running in.

sh "ls -la ${pwd()}"

// And look, output directory is there under first-stash!

sh "ls -la ${pwd()}/first-stash"

}

[This says](http://getmesh.io/Blog/Jenkins+2+Pipeline+101) the name of the node is node('!master'), any node except the master is selected.

**Remote Loader Plugin**

[This video](https://www.youtube.com/watch?v=emV60CcDVV0&t=46m01s) introduces this feature.

A pipline Groovy file can be loaded from any repo within a pipeline, like what “include” statements do.

[This](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/load-from-file/pipeline.groovy) shows how to “include” a file instead of duplicating code.

node {

// Load file from the current directory:

def externalMethod = load("externalMethod.groovy")

externalMethod.lookAtThis("Steve")

def externalCall = load("externalCall.groovy")

externalCall("Steve")

}

The [externalMethod.groovy](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/load-from-file/externalMethod.groovy" \t "_blank) file contains:

def myVar = build.getEnvironment(listener).get('myVar')

// Methods in this file will end up as object methods on the object that load returns.

def lookAtThis(String whoAreYou) {

echo "Look at this, ${whoAreYou}! You loaded this from another file!"

}

The [externalCall.groovy](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/load-from-file/externalCall.groovy" \t "_blank) file contains:

def call(String whoAreYou) {

echo "Now we're being called more magically, ${whoAreYou}, thanks to the call(...) method."

}

[This](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/load-from-file/loadFromGithub.groovy) shows how to “include” a file instead of duplicating code.

#!groovy

apply from: 'https://raw.githubusercontent.com/org-name/repo-name/master/subfolder/Jenkinsfile?token=${env.GITHUB\_TOKEN}'

In this example, the “org-name”, “repo-name” are replaced with actuals.

The “GITHUB\_TOKEN” is a variable so its value is not exposed in code.

### GitHub Hooks #

Git Hooks are programs placed in a hooks directory to trigger actions at certain points in git’s execution. The list of hooks at:  
https://www.kernel.org/pub/software/scm/git/docs/githooks.html  
include pre-commit, post-commit, etc.

Hooks that don’t have the executable bit set are ignored.

* http://www.chilipepperdesign.com/2013/01/07/deploying-code-with-git/

**Item: Organization Folders**

Organization folders enable Jenkins to automatically detect and include as resources any new repository under an account/organization.

### Run Maven #

Alternately, if you want to run a Maven file:

node ('linux'){

stage 'Build and Test'

env.PATH = "${tool 'Maven 3'}/bin:${env.PATH}"

checkout scm

sh 'mvn clean package'

}

CAUTION: Watch for “Error while checking in file to scm repository”.

**Parallel jobs**

[Here](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/jobs-in-parallel/jobs_in_parallel.groovy) is an example of how to run several jobs in parallel. Define array “branches” for execution by in a “parallel” command which executes all items in the array at the same time.

def branches = [:]

for (int i = 0; i < 4; i++) {

def index = i //if we tried to use i below, it would equal 4 in each job execution.

branches["branch${i}"] = {

build job: 'test\_jobs', parameters: [[$class: 'StringParameterValue', name: 'param1', value:

'test\_param'], [$class: 'StringParameterValue', name:'dummy', value: "${index}"]]

}

}

parallel branches

The for loop interrates 4 times through the array (0, 1, 2, 3), each containing a “build job” command.

param1 : an example string parameter for the triggered job.

dummy: a parameter used to prevent triggering the job with the same parameters value. This parameter has to accept a different value each time the job is triggered. WQ//we have to assign it outside the closure or it will run the job multiple times with the same parameter “4” //and jenkins will unite them into a single run of the job

Remember it’s not permitted to have a stage step inside a parallel block.

**Cause**

[Here](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/get-build-cause/getBuildCause.groovy) is a definition pulling in [cause](http://javadoc.jenkins-ci.org/hudson/model/class-use/Cause.html) info available to Freestyle build jobs as a $CAUSE variable

// Get all Causes for the current build

def causes = currentBuild.rawBuild.getCauses()

// Get a specific Cause type (in this case the user who kicked off the build):

def specificCause = currentBuild.rawBuild.getCause(hudson.model.Cause$UserIdCause)

[Here](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/parallel-from-grep/parallelFromGrep.groovy) is an alternative.

**Multi-branch projects**

Instead of Jenkins types Freestyle and Pipeline, select **Multi-branch**.

* https://jenkins.io/blog/2015/12/03/pipeline-as-code-with-multibranch-workflows-in-jenkins/
* https://www.youtube.com/watch?v=emV60CcDVV0&t=1h20m20s

[In this video](https://www.youtube.com/watch?v=emV60CcDVV0&t=59m57s) Jesse

**Durable Task Plugin**

For long-running build runs at   
https://wiki.jenkins-ci.org/display/JENKINS/Durable+Task+Plugin

“Library offering an extension point for processes which can run outside of Jenkins yet be monitored.”

**Safe Restart Plugin**

Some install the [SafeRestart plug-in](https://wiki.jenkins-ci.org/display/JENKINS/SafeRestart+Plugin" \t "_blank) which adds the **Restart Safely** option to the [Jenkins left menu](https://cloud.githubusercontent.com/assets/300046/12584913/9681b1d2-c3fe-11e5-9359-e51fc5809734.png) to avoid needing to be at the server console at all.

To restart Jenkins server:

1. PROTIP: Restart Jenkins by changing the URL from:
2. http://.../pluginManager/installed

to

http://.../restart

1. Click Yes to “Are you sure”.
2. “Please wait while Jenkins is restarting”.

**Workspace Cleanup**

[Workspace](https://wiki.jenkins-ci.org/display/JENKINS/Workspace+Cleanup+Plugin)

**Discard Old Builds**

The [Jenkins.io](https://github.com/jenkins-infra/jenkins.io/blob/master/Jenkinsfile) Jenkinsfile Groovy script has this:

/\* Only keep the 10 most recent builds. \*/

properties([[$class: 'BuildDiscarderProperty',

strategy: [$class: 'LogRotator', numToKeepStr: '10']]])

This is needed because build jobs can fill up a lot of disk space, especially if you store the build artifacts (binary files, such as JARs, WARs, TARs, etc.). So specify a limit on how many builds to store.

There is a default ‘Discard Old Build’ function. But there is a plugin provides more choices to trigger deletion.

1. In Manage Plugins, filter for “Discard Old Build” plugin at  
   <https://wiki.jenkins-ci.org/display/JENKINS/Discard+Old+Build+plugin>

Alas, there are [bugs in it since Feb 2016](https://issues.jenkins-ci.org/browse/JENKINS-32858) which render it unusable. We didn’t find this bug until we wasted several hours trying to figure out what we did wrong.

1. Specify the “Max # of builds to keep” (10).
2. Check the “Status to keep” checkboxes (to the left of) “Unstable” and “Failure”.
3. Click “Keep this job forever” on a specific build.

**Monitoring**

1. Install the “Jenkins Monitoring” plugin.
2. From the Manage Jenkins screen, access the JavaMelody graphs in the “Monitoring of Jenkins/Jenkins master” or “Jenkins/Jenkins nodes” menu entries.

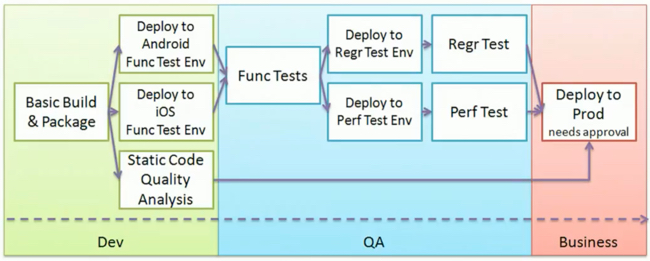
It reports about the state of your build server, including CPU and system load, average response time, and memory usage. JavaMelody.

BTW, @jenx\_monitor is a A Jenkins build server monitor for Mac OS X, powered by MacRuby. This app sits in your status bar and reports the status of all your Jenkins builds.

1. Install the **Disk Usage** plugin to show a trend graph over time the disk space used by each project.

**Chaining in the Pipeline**

Jenkins v1 consisted of many atomic jobs chained together by a mix of triggers and parameters.



### CI vs CD #

Continuous Integration (CI) **merges** development work in a developer’s branch with the team’s common code to ensure that changes still work in a testing environment. It’s call continuous to emphasize small changes being integrated frequently to stay in sync with an evolving team codebase. This requires individual changes to be scoped for completion in a short amount of time (hours at most).

Continuous Delivery (CD) delivers code for running in an UAT or Staging enviornment (of full production scale) used by end-users (QA or customers) to process business transactions in inspection mode. It’s called continuous for frequency to find more issues early, before each particular version has left the memory of developers.

Continuous Deployment moves code to Production. This is done by merging to the Mainline/Master branch which gets copied to the Production enviornment. It’s called continuous to make this happen as soon as code is ready.

BTW, [Assembla](https://blog.assembla.com/AssemblaBlog/tabid/12618/bid/92411/Continuous-Delivery-vs-Continuous-Deployment-vs-Continuous-Integration-Wait-huh.aspx" \t "_blank) has more ideas about this.

**Slack notification REST API**

[This](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/slacknotify/slackNotify.groovy) script imports Goovy’s built-in library to output Json using the “JsonOutput.toJson” class.method. The JSON output is the body (payload) of a REST API call to Slack.com made using the Linux curl command:

import groovy.json.JsonOutput

def notifySlack(text, channel) {

def slackURL = 'https://hooks.slack.com/services/xxxxxxx/yyyyyyyy/zzzzzzzzzz'

def payload = JsonOutput.toJson([text : text,

channel : channel,

username : "jenkins",

icon\_emoji: ":jenkins:"])

sh "curl -X POST --data-urlencode \'payload=${payload}\' ${slackURL}"

}

// Add whichever params you think you’d most want to have // replace the slackURL below with the hook url provided by // slack when you configure the webhook

**IRC notification**

[This](https://github.com/jenkinsci/pipeline-examples/blob/master/pipeline-examples/ircnotify-commandline/ircNotify.groovy) snippet for just the “notify” stage shows how to invoke a Bash shell command (“sh”) to use the “nc” (netcat) command (on the last line below).

stage "notify"

node {

sh '''

MSG='This is the message here'

SERVER=irc.freenode.net

CHANNEL=#mictest

USER=mic2234test

(

echo NICK $USER

echo USER $USER 8 \* : $USER

sleep 1

#echo PASS $USER:$MYPASSWORD

echo "JOIN $CHANNEL"

echo "PRIVMSG $CHANNEL" :$MSG

echo QUIT

) | nc $SERVER 6667

'''

}

The shell command above first defines environment variables, then between (parentheses) specifies the commands piped into the nc command.

The “nc” (netcat) command is built into most Linux servers (including Mac OSX). It’s used instead of Telnet here.

A [“raw” connection](http://phil.lavin.me.uk/tag/irc-with-netcat/) is established on default port 6667. Your enterprise network may not allow it.

You need to modify the “This is the message here” text to a variable updated internally in the script based on results of the previous stage.

Also modify values to variables CHANNEL and USER.

$MYPASSWORD comes from outside the script.

* Command “NICK” sets the user nickname.
* Command “PRIVMSG” sets the private message.

BTW, https://github.com/jenkins-infra/ircbot

**Code Static Scans**

Typical rules include “Constant If Expression,” “Empty Else Block,” “GString As Map Key,” and “Grails Stateless Service.”

For Groovy code, CodeNarc (similar to PMD for Java) checks for potential defects, poor coding practices and styles, overly complex code, and so on.

SonarQube runs on its own server a set of Maven code quality related plugins against your Maven project, and stores results into its relational database

1. Install Sonar on its own server.
2. On Jenkins install the “Sonar” plugin.
3. Configure access to the Sonar database via JDBC.
4. Sonar has Maven and Ant bootstrap and Gradle bootstraps.
5. Schedule Sonar build once a day for metrics.

PROTIP: For metrics, schedule Sonar to run only once a day because Sonar stores metrics in 24-hour slices.

Code Complexity.

Test Coverage.

The “Violations” plugin generates reports and graphs from code quality run metrics generated for individual builds and trends over time. It’s configured in the “Build Settings” section.

**Build using Maven**

pom.xml file.

**IntelliJ IDE Auto-complete**

[.gdsl file for IntelliJ IDE code completion](https://www.youtube.com/watch?v=emV60CcDVV0&t=46m01s&t=40m50s) begins like this:

//The global script scope

def ctx = context(scope: scriptScope())

contributor(ctx) {

method(name: 'build', type: 'Object', params: [job:'java.lang.String'], doc: 'Build a job')

**Parallel Runs**

Running in parallel is especially useful for testing multi-platform apps.

stage "test on supported OSes"

parallel (

windows: { node {

sh "echo building on windows now"

}},

mac: { node {

sh "echo building on mac now"

}}