

Pipeline Defined

Pipelines are Jenkins jobs enabled by the Pipeline (formerly called “workflow”) plugin and built with simple text scripts that use a Pipeline DSL (domain-specific language) based on the Groovy programming language.

Pipelines leverage the power of multiple steps to execute both simple and complex tasks according to parameters that you establish. Once created, pipelines can build code and orchestrate the work required to drive applications from commit to delivery.

Pipeline Vocabulary

Pipeline terms such as “step,” “node,” and “stage” are a subset of the vocabulary used for Jenkins in general.

**Step**

A “step” (often called a “build step”) is a single task that is part of sequence. Steps tell Jenkins what to do.

**Node**

In pipeline coding contexts, as opposed to Jenkins generally, a “node” is a step that does two things, typically by enlisting help from available executors on agents:

* Schedules the steps contained within it to run by adding them to the Jenkins build queue (so that as soon as an executor slot is free on a node, the appropriate steps run).
* Creates a workspace, meaning a file directory specific to a particular job, where resource-intensive processing can occur without negatively impacting your pipeline performance. Workspaces last for the duration of the tasks assigned to them.

(In Jenkins generally, a “node” means any computer that is part of your Jenkins installation, whether that computer is used as a master or as an agent).

**Stage**

A “stage” is a step that calls supported APIs. Pipeline syntax is comprised of stages. Each stage can have one or more build steps within it.

Familiarity with Jenkins terms such as “master,” “agent,” and “executor” also helps with understanding how pipelines work. These terms are not specific to pipelines:

* master - A “master” is the basic installation of Jenkins on a computer; it handles tasks for your build system. Pipeline scripts are parsed on masters, and steps wrapped in node blocks are performed on available executors.
* agent - An “agent” (formerly "slave") is a computer set up to offload particular projects from the master. Your configuration determines the number and scope of operations that an agent can perform. Operations are performed by executors.
* executor - An “executor” is a computational resource for compiling code. It can run on master or agent machines, either by itself or in parallel with other executors. Jenkins assigns a *java.lang.Thread* to each executor.

Preparing Jenkins to Run Pipelines

To run pipelines, you need to have a Jenkins instance that is set up with the appropriate plugins. This requires:

* Jenkins 1.580.1 or later (Jenkins 2.0 is recommended)
* The core Pipeline plugin

## Creating a Simple Pipeline

Initial pipeline usage typically involves the following tasks:

1. Downloading and installing the Pipeline plugin (Unless it is already part of your Jenkins installation)
2. Creating a Pipeline of a specific type
3. Configuring your Pipeline
4. Controlling Flow through your Pipeline
5. Scaling your Pipeline

To create a simple pipeline from the Jenkins interface, perform the following steps:

1. Click **New Item** on your Jenkins home page, enter a name for your (pipeline) job, select **Pipeline**, and click **OK**.
2. In the Script text area of the configuration screen, enter your pipeline syntax. If you are new to pipeline creation, you might want to start by opening Snippet Generator and selecting the “Hello Word” snippet. **Note:** Pipelines are written as Groovy scripts that tell Jenkins what to do when they are run, but because relevant bits of syntax are introduced as needed, you do not need deep expertise in Groovy to create them, although basic understanding of Groovy is helpful.
3. Check the Use Groovy Sandbox option below the Script text area. **Note:** If you are a Jenkins administrator (in other words, authorized to approve your own scripts), sandboxing is optional but efficient, because it lets scripts run without approval as long as they limit themselves to operations that Jenkins considers inherently safe.
4. Click **Save**.
5. Click **Build Now** to create the pipeline.
6. Click ▾ and select **Console Output** to see the output.

The following example shows a successful build of a pipeline created with a one-line script that uses the “echo” step to output the phrase, “hello from pipeline:”

Started by user anonymous

[Pipeline] echo

hello from Pipeline

[Pipeline] End of Pipeline

Finished: SUCCESS

**Note:** You can also create complex and multi-branch pipelines in the script entry area of the Jenkins configuration page, but because they contain multiple stages and the configuration page UI provides limited scripting space, pipeline creation is more commonly done using an editor of your choice from which scripts can be loaded into Jenkins using the **Pipeline script from SCM** option.

## Creating Multi-branch Pipelines

The **Multibranch Pipeline** project type enables you to configure different jobs for different branches of the same project. In a multi-branch pipeline configuration, Jenkins automatically discovers, manages, and executes jobs for multiple source repositories and branches. This eliminates the need for manual job creation and management, as would otherwise be necessary when, for example, a developer adds a new feature to an existing product. Multi-branch pipelines also enable you to stop or suspend jobs automatically if circumstances make that appropriate.

A multi-branch pipeline project always includes a Jenkinsfile in its repository root. Jenkins automatically creates a sub-project for each branch that it finds in a repository with a Jenkinsfile.

Multi-branch pipelines use the same version control as the rest of your software development process. This “pipeline as code” approach has the following advantages:

* You can modify pipeline code without special editing permissions.
* Finding out who changed what and why no longer depends on whether developers remember to comment their code changes in configuration files.
* Version control makes the history of changes to code readily apparent.

To create a Multi-branch Pipeline:

1. Click New Item on your Jenkins home page, enter a name for your job, select Multibranch Pipeline, and click OK.
2. Configure your SCM source (options include Git, GitHub, Mercurial, Subversion, and Bitbucket), supplying information about the owner, scan credentials, and repository in appropriate fields. For example, if you select Git as the branch source, you are prompted for the usual connection information, but then rather than enter a fixed refspec (Git’s name for a source/destination pair), you would enter a branch name pattern (Use default settings to look for any branch).
3. Configure the other multi-branch pipeline options:
   * API endpoint - an alternate API endpoint to use a self-hosted GitHub Enterprise
   * Checkout credentials - alternate credentials to use when checking out the code (cloning)
   * Include branches - a regular expression to specify branches to include
   * Exclude branches - a regular expression to specify branches to exclude; note that this will takes precedence over the contents of include expressions
   * Property strategy - where you can optionally define custom properties for each branch
4. Save your configuration.

Jenkins automatically scans the designated repository and creates appropriate branches.

For example (again in Git), if you started with a master branch, and then wanted to experiment with some changes, and so did git checkout -b newfeature and pushed some commits, Jenkins would automatically detect the new branch in your repository and create a new sub-project for it. That sub-project would have its own build history unrelated to the trunk (main line).

If you choose, you can ask for the sub-project to be automatically removed after its branch is merged with the main line and deleted. To change your Pipeline script—for example, to add a new Jenkins publisher step corresponding to new reports that your Makefile/pom.xml/etc. is creating—you edit the Jenkinsfile in your change. Your Pipeline script is always synchronized with the rest of the source code you are working on: the checkout scm command checks out the same revision as the script is loaded from.

## Writing Pipeline Scripts in the Jenkins UI

Because Pipelines are comprised of text scripts, they can be written (edited) in the same script creation area of the Jenkins user interface where you create them:



*Figure 2. Pipeline Editor*

**Note:** Pipeline script writing adds stages and steps to a pipeline; it does not convert one pipeline type into another. You determine which kind of pipeline you want to set up before writing it.

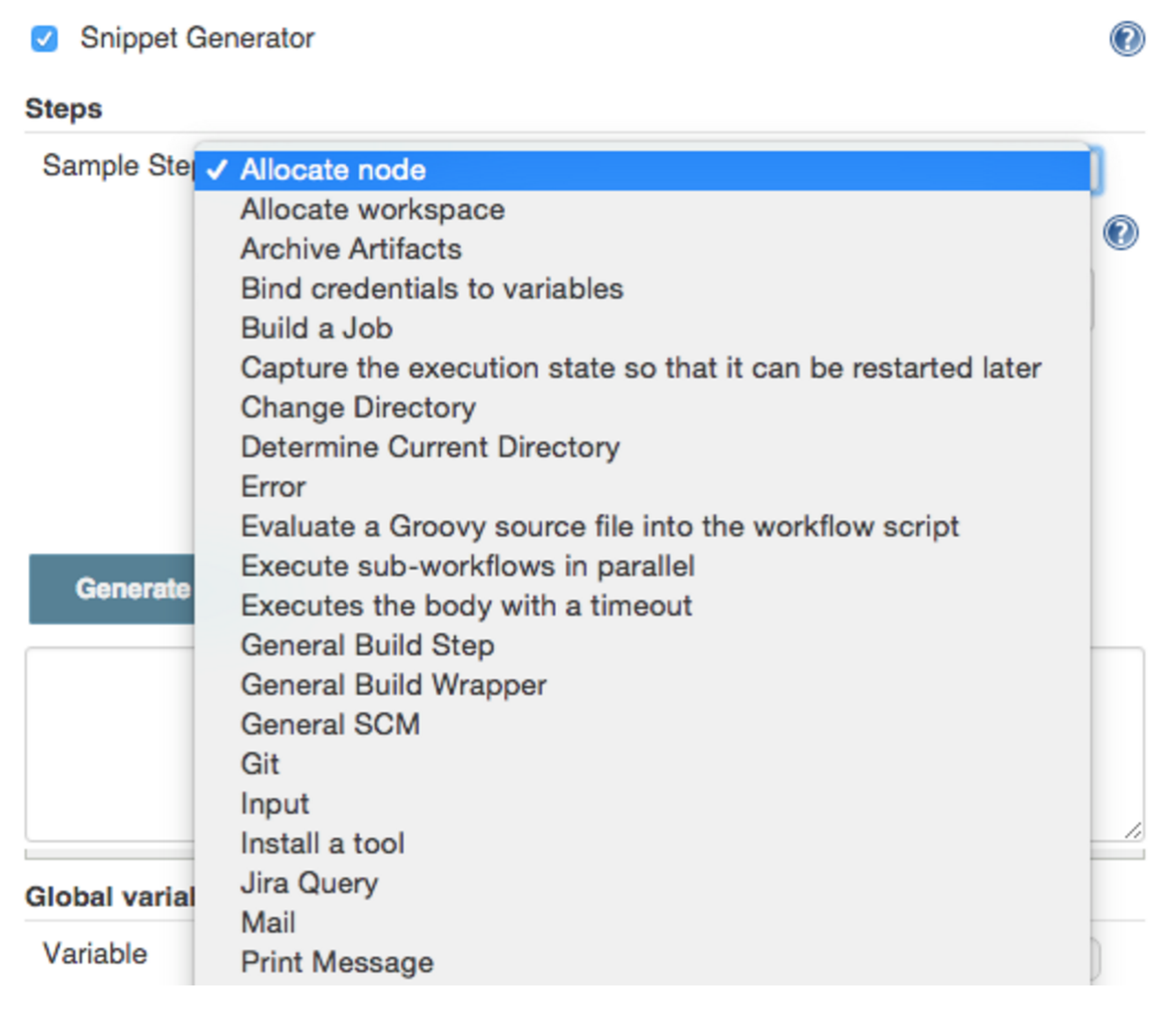
### Using Snippet Generator

You can automate much of the pipeline configuration process by using the Snippet Generator tool.

Snippet Generator is dynamically populated with a list of the steps available for pipeline configuration. Depending on the plugins installed to your Jenkins environment, you may see more or fewer items in the list exposed by Snippet Generator.

To add one or more steps from Snippet Generator to your pipeline code:

1. Open Snippet Generator
2. Scroll to the step you want
3. Click that step
4. Configure the selected step, if presented with configuration options
5. Click **Generate Groovy** to see a Groovy snippet that runs the step as configured
6. Optionally select and configure additional steps



*Figure 3. Snippet Generator*

When you click **Generate Groovy** after selecting a step, you see the function name used for that step, the names of any parameters it takes (if they are not default parameters), and the syntax used by Snippet Generator to create that step.

You can copy and paste the generated code right into your Pipeline, or use it as a starting point, perhaps deleting any optional parameters that you do not need.

To access information about steps marked with the help icon (question mark), click on that icon.

## Basic Groovy Syntax for Pipeline Configuration

You typically add functionality to a new pipeline by performing the following tasks:

* Adding nodes
* Adding more complex logic (usually expressed as stages and steps)
* Using the “ws” step to create additional workspace on an agent without taking another executor slot

To configure a pipeline you have created through the Jenkins UI, select the pipeline and click **Configure**.

If you run Jenkins on Linux or another Unix-like operating system with a Git repository that you want to test, for example, you can do that with syntax like the following, substituting your own name for “joe-user”:

node {

git url: 'https://github.com/joe\_user/simple-maven-project-with-tests.git'

**def** mvnHome = tool 'M3'

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

}

Declare variable:

def varname =

Access variable

“${varname }” //variable access in double quote only

In Windows environments, use “bat” in place of “sh,” and use backslashes as the file separator where needed (backslashes need to be escaped inside strings).

For example, rather than:

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

you would use:

bat "${mvnHome}\\bin\\mvn -B verify"

Your Groovy pipeline script can include functions, conditional tests, loops, try/catch/finally blocks, and so on.

Sample syntax for one node in a Java environment that is using the open source Maven build automation tool (hence the definition for “mvnHome”) is shown below:



*Figure 4. Pipeline Sample*

Sample key:

* def is a keyword to define a function (you can also give a Java type in place of def to make it look more like a Java method)
* =~ is Groovy syntax to match text against a regular expression
* [0] looks up the first match
* [1] looks up the first (…) group within that match
* readFile step loads a text file from the workspace and returns its content (Note: Do not use java.io.File methods — these refer to files on the master where Jenkins is running, not files in the current workspace).
* The writeFile step saves content to a text file in the workspace
* The fileExists step checks whether a file exists without loading it.

The tool step makes sure a tool with the given name is installed on the current node. The script needs to know where it was installed, so the tool can be run later. For this, you need a variable.

The **def** keyword in Groovy is the quickest way to define a new variable (with no specific type).

In the sample syntax discussed above, a variable is defined by the following expression:

def mvnHome = tool 'M3'

This ensures that M3 is installed somewhere accessible to Jenkins and assigns the return value of the step (an installation path) to the mvnHome variable.

## Advanced Groovy Syntax for Pipeline Configuration

Groovy lets you omit parentheses around function arguments. The named-parameter syntax is also a shorthand for creating a map, which in Groovy uses the syntax [key1: value1, key2: value2], so you could write:

git([url: 'https://github.com/joe\_user/simple-maven-project-with-tests.git', branch: 'master'])

For convenience, when calling steps taking only one parameter (or only one mandatory parameter) you can omit the parameter name. For example:

sh 'echo hello'

is really shorthand for:

sh([script: 'echo hello'])

### Managing the Environment

One way to use tools by default is to add them to your executable path using the special variable env that is defined for all pipelines:

node {

git url: 'https://github.com/joe\_user/simple-maven-project-with-tests.git'

def mvnHome = tool 'M3'

env.PATH = "${mvnHome}/bin:${env.PATH}"

sh 'mvn -B verify'

}

* Properties of this variable are environment variables on the current node.
* You can override certain environment variables and the overrides are seen by subsequent sh steps (or anything else that pays attention to environment variables).
* You can run mvn without a fully-qualified path.

Setting a variable such as PATH in this way is only safe if you are using a single agent for this build. As an alternative, you can use the withEnv step to set a variable within a scope:

node {

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

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

sh 'mvn -B verify'

}

}

Jenkins defines some environment variables by default:

**Example:** env.BUILD\_TAG can be used to get a tag like jenkins-projname-1 from Groovy code, or $BUILD\_TAG can be used from a sh script. The Snippet Generator help for the withEnv step has additional detail on this topic.

### Build Parameters

If you configured your pipeline to accept parameters using the **Build with Parameters** option, those parameters are accessible as Groovy variables of the same name.

### Recording Test Results and Artifacts

If there are any test failures in a given build, you want Jenkins to record them, and then proceed, rather than stopping. If you want it saved, you must capture the JAR that you built. The following sample code for a node shows how (As previously seen in several examples from this guide, Maven is being used as a build tool):

node {

git url: 'https://github.com/joe\_user/simple-maven-project-with-tests.git'

def mvnHome = tool 'M3'

sh "${mvnHome}/bin/mvn -B -Dmaven.test.failure.ignore verify"

archiveArtifacts artifacts: '\*\*/target/\*.jar', fingerprint: true

junit \*\*/target/surefire-reports/TEST-\*.xml'

}

* If tests fail, the Pipeline is marked unstable (as denoted by a yellow ball in the Jenkins UI), and you can browse the Test Result Trend to see the involved history.
* You should see Last Successful Artifacts on the Pipeline index page.

## Loading Pipeline Scripts from SCM

Complex pipelines would be cumbersome to write and maintain if you could only do that in the text area provided by the Jenkins job configuration page.

Accordingly, you also have the option of writing pipeline scripts in in your IDE (integrated development environment) or SCM system, and then loading those scripts into Jenkins using the **Pipeline Script from SCM** option enabled by the workflow-scm-step plugin, which is one of the plugins that the Pipeline plugin depends on and automatically installs.

Loading pipeline scripts from another source leverages the idea of “pipeline as code,” and lets you maintain that source using version control and standalone Groovy editors.

To do this, select **Pipeline script from SCM** when defining the pipeline.

With the **Pipeline script from SCM** option selected, you do not enter any Groovy code in the Jenkins UI; you just indicate by specifying a path where in source code you want to retrieve the pipeline from. When you update the designated repository, a new build will be triggered, as long as your job is configured with an SCM polling trigger.

**Multibranch Pipeline** projects expose the name of the branch being built with the BRANCH\_NAME environment variable. They also provide a special **checkout scm** Pipeline command, which checks out the specific commit that the Jenkinsfile originated, so that branch integrity is automatically maintained.

## Declarative Pipeline

Declarative Pipeline is a relatively recent addition to Jenkins Pipeline [1] which presents a more simplified and opinionated syntax on top of the Pipeline sub-systems.

All valid Declarative Pipelines must be enclosed within a pipeline block, for example:

pipeline {

/\* insert Declarative Pipeline here \*/

}

##### **Example**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent { docker 'maven:3-alpine' }

stages {

stage('Example Build') {

steps {

sh 'mvn -B clean verify'

}

steps {

sh 'mvn -B compile'

}

steps {

sh 'mvn -B package'

}

steps {

sh 'mvn -B install'

}

}

}

}

|  |  |
| --- | --- |
|  | Execute all the steps defined in this Pipeline within a newly created container of the given name and tag (maven:3-alpine). |

###### **Stage-level agent section**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent none /\* run job on master server \*/

stages {

stage('Example Build') {

agent { docker 'maven:3-alpine' }

steps {

echo 'Hello, Maven'

sh 'mvn --version'

}

}

stage('Example Test') {

agent { docker 'openjdk:8-jre' }

steps {

echo 'Hello, JDK'

sh 'java -version'

}

}

}

}

|  |  |
| --- | --- |
|  | Defining agent none at the top-level of the Pipeline ensures that an Executor will not be assigned unnecessarily. Using agent none also forces each stage section to contain its own agent section. |
|  | Execute the steps in this stage in a newly created container using this image. |
|  | Execute the steps in this stage in a newly created container using a different image from the previous stage. |

#### post

The post section defines one or more additional steps that are run upon the completion of a Pipeline’s or stage’s run (depending on the location of the post section within the Pipeline). post can support any of of the following post-condition blocks: always, changed, fixed, regression, aborted, failure, success, unstable, unsuccessful, and cleanup. These condition blocks allow the execution of steps inside each condition depending on the completion status of the Pipeline or stage. The condition blocks are executed in the order shown below.

|  |  |
| --- | --- |
| **Required** | No |
| **Parameters** | None |
| **Allowed** | In the top-level pipeline block and each stage block. |

##### **Conditions**

**always**

Run the steps in the post section regardless of the completion status of the Pipeline’s or stage’s run.

**changed**

Only run the steps in post if the current Pipeline’s or stage’s run has a different completion status from its previous run.

**fixed**

Only run the steps in post if the current Pipeline’s or stage’s run is successful and the previous run failed or was unstable.

**regression**

Only run the steps in post if the current Pipeline’s or stage’s run’s status is failure, unstable, or aborted and the previous run was successful.

**aborted**

Only run the steps in post if the current Pipeline’s or stage’s run has an "aborted" status, usually due to the Pipeline being manually aborted. This is typically denoted by gray in the web UI.

**failure**

Only run the steps in post if the current Pipeline’s or stage’s run has a "failed" status, typically denoted by red in the web UI. Note that if you manually set currentBuild.result = 'FAILURE' in a stage and have a failure post condition on that stage, the failure will not fire for that stage.

**success**

Only run the steps in post if the current Pipeline’s or stage’s run has a "success" status, typically denoted by blue or green in the web UI.

**unstable**

Only run the steps in post if the current Pipeline’s or stage’s run has an "unstable" status, usually caused by test failures, code violations, etc. This is typically denoted by yellow in the web UI.

**unsuccessful**

Only run the steps in post if the current Pipeline’s or stage’s run has not a "success" status. This is typically denoted in the web UI depending on the status previously mentioned

**cleanup**

Run the steps in this post condition after every other post condition has been evaluated, regardless of the Pipeline or stage’s status.

##### **Example**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

post {

always {

echo 'I will always say Hello again!'

}

}

}

|  |  |
| --- | --- |
|  | Conventionally, the post section should be placed at the end of the Pipeline. |
|  | Post-condition blocks contain steps the same as the steps section. |

#### stages

Containing a sequence of one or more stage directives, the stages section is where the bulk of the "work" described by a Pipeline will be located. At a minimum it is recommended that stages contain at least one stage directive for each discrete part of the continuous delivery process, such as Build, Test, and Deploy.

|  |  |
| --- | --- |
| **Required** | Yes |
| **Parameters** | None |
| **Allowed** | Only once, inside the pipeline block. |

##### **Example**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

}

|  |  |
| --- | --- |
|  | The stages section will typically follow the directives such as agent, options, etc. |

#### steps

The steps section defines a series of one or more steps to be executed in a given stage directive.

|  |  |
| --- | --- |
| **Required** | Yes |
| **Parameters** | None |
| **Allowed** | Inside each stage block. |

##### **Example**

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

}

|  |  |
| --- | --- |
|  | The steps section must contain one or more steps. |

#### Example

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent none

stages {

stage('Non-Sequential Stage') {

agent {

label 'for-non-sequential'

}

steps {

echo "On Non-Sequential Stage"

}

}

stage('Sequential') {

agent {

label 'for-sequential'

}

environment {

FOR\_SEQUENTIAL = "some-value"

}

stages {

stage('In Sequential 1') {

steps {

echo "In Sequential 1"

}

}

stage('In Sequential 2') {

steps {

echo "In Sequential 2"

}

}

stage('Parallel In Sequential') {

parallel {

stage('In Parallel 1') {

steps {

echo "In Parallel 1"

}

}

stage('In Parallel 2') {

steps {

echo "In Parallel 2"

}

}

}

}

}

}

}

}

### Flow Control

Scripted Pipeline is serially executed from the top of a Jenkinsfile downwards, like most traditional scripts in Groovy or other languages. Providing flow control therefore rests on Groovy expressions, such as the if/else conditionals, for example:

*Jenkinsfile (Scripted Pipeline)*

node {

stage('Example') {

**if** (env.BRANCH\_NAME == 'master') {

echo 'I only execute on the master branch'

} **else** {

echo 'I execute elsewhere'

}

}

Stage(‘Example2’){

for(def i=1; i<10;i++){

Echo ‘ test’

}

}

}

Another way Scripted Pipeline flow control can be managed is with Groovy’s exception handling support. When Steps fail for whatever reason they throw an exception. Handling behaviors on-error must make use of the try/catch/finally blocks in Groovy, for example:

*Jenkinsfile (Scripted Pipeline)*

node {

stage('Example') {

**try** {

sh 'exit 1'

}

**catch** (exc) {

echo 'Something failed, I should sound the klaxons!'

**throw**

}

}

}