**Chapter 3. Pipeline Execution Flow**

In this chapter, we’ll explore the different constructs provided by the Jenkins pipeline DSL for controlling the execution flow in pipelines. We’ll start with specifying properties to trigger jobs and how to accept input.

Then we’ll look at how to keep things moving through constructs including timeouts, retries, and running tasks in parallel. We’ll also look at the constructs available to map the Conditional BuildStep functionality into pipelines.

Finally, we’ll see how to use pipeline methods to emulate the post-build processing functionality of traditional Jenkins jobs. Along the way, we’ll see how things differ for Scripted and Declarative Pipelines.

Let’s get started with defining the properties for triggering jobs.

# Triggering Jobs

To specify triggering events for pipeline code, there are three different approaches:

* If working in the Jenkins application itself in a pipeline job, the trigger(s) can be specified in the traditional way within the project’s General configuration section in the web interface.
* If creating a Scripted Pipeline, a properties block can be specified (usually before the start of the pipeline) that defines the triggers in code. (Note that this properties section will be merged with any properties defined in the web interface, with the web properties taking precedence.)
* If creating a Declarative Pipeline, there is a special triggers directive that can be used to define the types of things that should trigger the pipeline.

We’ll briefly look at each of the trigger options available in the traditional Jenkins interface, along with the corresponding scripted syntax and declarative syntax (if there is one).

## Build After Other Projects Are Built

As the name implies, selecting this option allows you to start your project building after one or more other projects. You can choose the ending status you want the builds of the other projects to have (stable, unstable, or failed).

For a Scripted Pipeline, the syntax for building your pipeline after another job, Job1, is successful would be like the following:

properties([

  pipelineTriggers([

    upstream(

      threshold: hudson.model.Result.SUCCESS,

      upstreamProjects: 'Job1'

    )

  ])

])

If you need to list multiple jobs, separate them with commas. If you need to specify a branch for a job (as for a multibranch job), add a slash after the job name and then the branch name (as in 'Job1/master').

## Build Periodically

Check the book as it is longer than expected

## GitHub Hook Trigger for GitSCM Polling

A GitHub project configured as the source location in a Jenkins project can have a push hook (on the GitHub side) to trigger a build for the Jenkins project. When this is in place, a push to the repository causes the hook to fire and trigger Jenkins, which then invokes the Jenkins SCM polling functionality. So the SCM polling functionality has to be configured for this to work as well.

Most of the initial work for this is in the setup for the hook side and in the source setup in the Jenkins project.More information is available [on the Jenkins wiki](http://bit.ly/2HM7a6z).

The syntax for setting the property in a Scripted Pipeline is as follows:

properties([pipelineTriggers([githubPush()])])

## Poll SCM

This is the standard polling functionality that periodically scans the source control system for updates. If any updates are found, then the job processes the changes. This can be a very expensive operation (in terms of system resources) depending on the SCM, how much content is scanned, and how often.

Specifying the values for this uses the same Jenkins cron syntax as is used for the “build periodically” option.

The syntax for Scripted Pipelines is as follows (polling every 30 minutes):

properties([pipelineTriggers([pollSCM('\*/30 \* \* \* \*')])])

The corresponding syntax for Declarative Pipelines would be this:

triggers { pollSCM(\*/30 \* \* \* \*) }

## Quiet Period

The value specified here serves as a “wait time” or offset between when the build is triggered (an update is detected) and when Jenkins acts on it. This can be useful for staggering jobs that frequently have changes at the same time, for example. If a value is not provided here, the value from the global configuration is used.

While the pipeline build step has a quietPeriod option, as of this writing, there isn’t a direct pipeline option or step to do this. You may be able to achieve a similar effect by using the throttle() step from the [Throttle Concurrent Builds plugin](http://bit.ly/2Hf0pJs).

## Trigger Builds Remotely

This allows for triggering builds by accessing a specific URL for the given job on the Jenkins system. This is useful for triggering builds via a hook or a script. An authorization token is required. For an example, see the note on “[URLs and Crumbs](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#sec_URLs_and_crumbs)”

In the pipeline-as-code semantics, Multibranch Pipelines can be triggered via changes in a Jenkinsfile.

After being triggered, certain stages of a pipeline may request or require input from a user for purposes such asverification, or to direct processing down one of multiple paths. We’ll look next at how to handle collecting that input in our pipelines.

# User Input

A key aspect of some Jenkins jobs is the ability to change their behavior based on user input. Jenkins offers a wide variety of parameters for gathering specific kinds of input. Jenkins pipelines provide constructs for this as well.

The DSL step input is the way we get user input through a pipeline. The step accepts the same kinds of parameters as a regular Jenkins job for a Scripted Pipeline. For a Declarative Pipeline, there is a special parameters directive that supports a subset of those parameters.

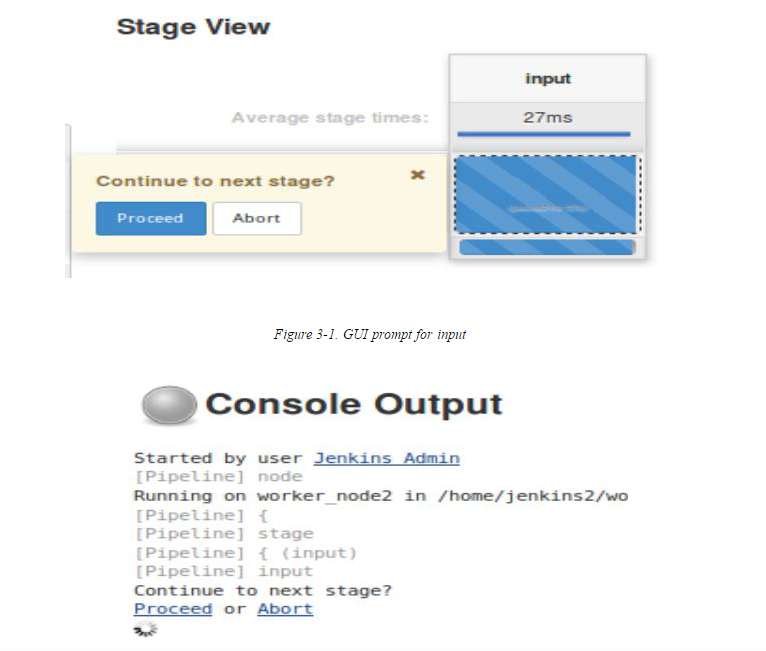
We describe this step and the parameters, as they can be used in the pipeline, next.

## input

As the name suggests, the input step allows your pipeline to stop and wait for a user response. Here’s a simple example:

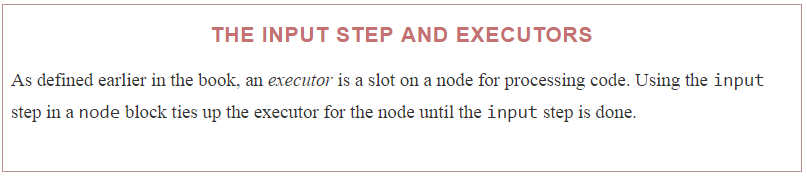
input 'Continue to next stage?'

This step can also optionally take parameters to gather additional information. Within the Jenkins application, the default form is to print a message and offer the user a choice of “Proceed” or “Abort.” In the GUI Stage View, this will be a dialog box that looks like [Figure 3-1](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#fig_gui_prompt_input). In the console output, this will be a line of output with links to click on to continue or stop ([Figure 3-2](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#fig_console_prompt_input)).



Choosing Proceed allows the pipeline to continue. Choosing Abort causes the pipeline to stop at that point with a status of “aborted.”

It is important to note that when the system executes an input step, the processing is paused on that node. This can lead to monopolizing system resources, as explained in the following warning.



The input step can have several parameters. These include:

Message (message)

The message to be displayed to the user, as demonstrated in the previous example. Can also be empty, as indicated by input ''.

Custom ID (id)

An ID that can be used to identify your input step to automated or external processing, such as when you want to respond via a REST API call. A unique identifier will be generated if you don’t supply one.

As an example, you could add the custom ID, ctns-prompt (for “Continue to next stage” prompt) to our input step definition. The input step would then look as follows:

input id: 'ctns-prompt', message: 'Continue to the next stage?'

Given this step, when you run the job, a POST to this URL could be used to respond. The URL format would be:

http://[jenkins-base-url]/job/[job\_name]/[build\_id]/input/Ctns-prompt/proceedEmpty

to tell Jenkins to proceed without any input, or

http://[jenkins-base-url]/job/[job\_name]/[build\_id]/input/Ctns-prompt/abort

to tell Jenkins to abort. (Notice that the parameter name is capitalized in the URL.)

# URLS AND CRUMBS

If your Jenkins is configured to prevent Cross-Site Request Forgery (CSRF) exploits via the Security settings (strongly recommended), then any URL used to POST will need to also include a CSRF protection token.

One way to do this is to first define an environment variable to get the token

CSRF\_TOKEN=

$(curl -s 'http://<username>:<password

or token>@<jenkins base

url>/crumbIssuer/api/xml?xpath=

concat(//crumbRequestField,":",//crumb)')

If you look at the environment variable with the token afterwards, you’ll see something like this:

$ echo $CSRF\_TOKEN

Jenkins-Crumb:0cd0babef95a70d0836c3f3e5bc4eea8

Then you can include the token in your POST call. Here’s an example using curl:

$ curl --user <userid>:<password or token>

-H "$CSRF\_TOKEN" -X POST

-s <jenkins base url>/job/<job name>/<build number>/input/

<input parameter with 1st letter capped>/proceedEmpty

If you don’t include the token, you’ll end up with a 403 error.

OK button caption (ok)

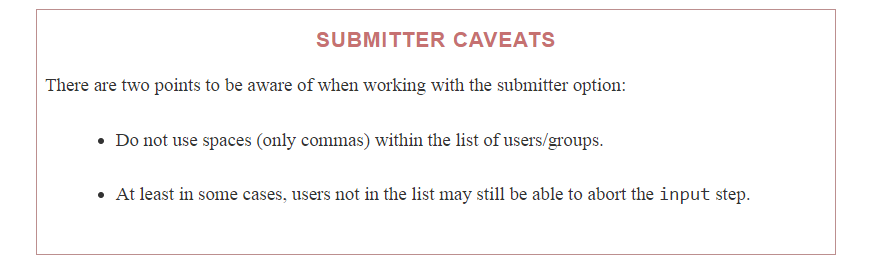
A different label you can use instead of “Proceed.” For example

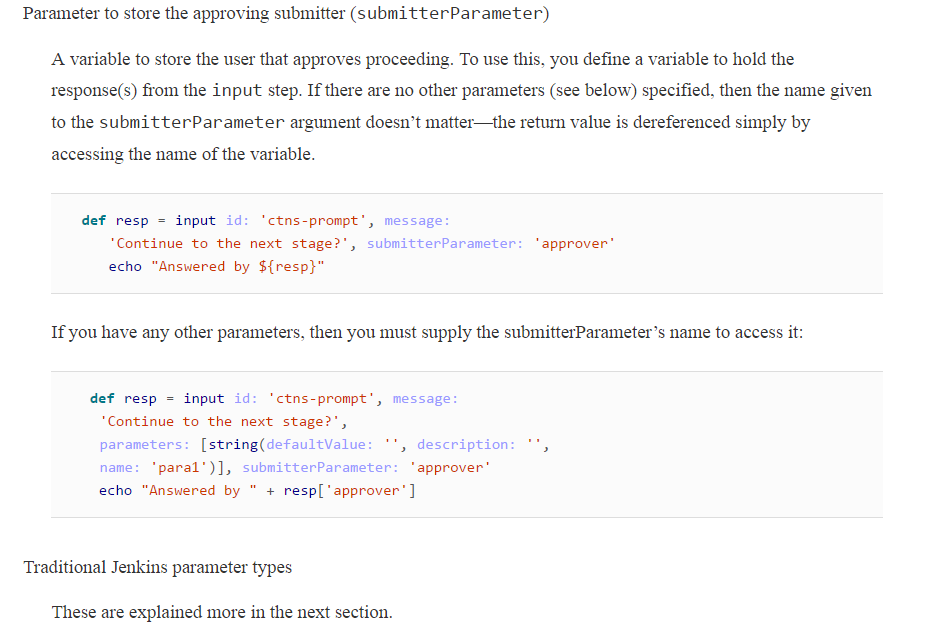
input message: '<message text>', ok: 'Yes'

Allowed submitter (submitter)

A comma-separated list of user IDs or group names for people authorized to respond. For example:

input message: '<message text>', submitter: 'user1,user2'





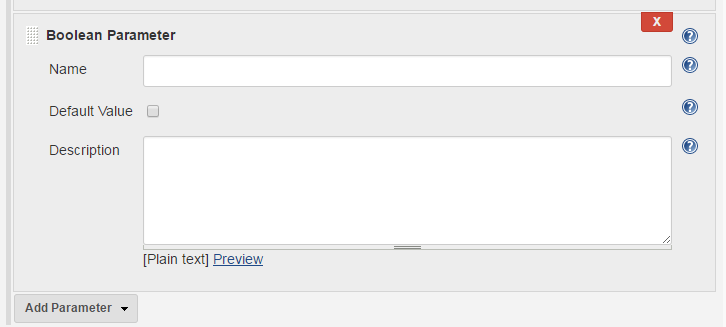
## Parameters

With the input statement, you have the option to add any of the standard Jenkins parameter types. If you’ve done any work with Jenkins before, you’re probably already familiar with most of these. The following sections briefly introduce each one and offer an example of what it looks like when used in a script.

For each parameter type, the different “subparameters” (arguments) that it can take are also listed. If the purpose of the subparameter is self-evident from its name (e.g., name, default value, description), the argument name will be listed without additional explanation.

### BOOLEAN

This is the basic true/false parameter. The subparameters for a Boolean are Name, Default Value, and Description.

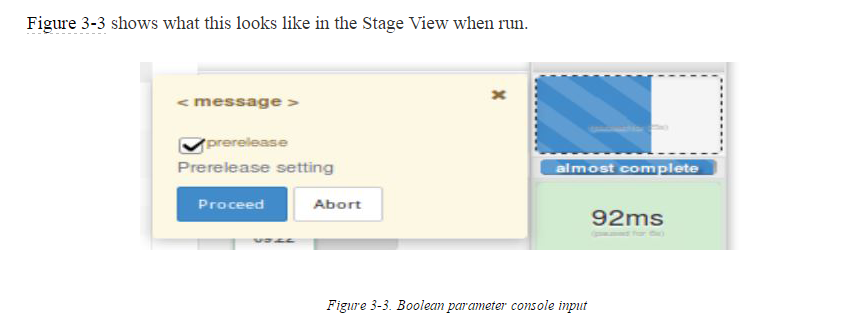


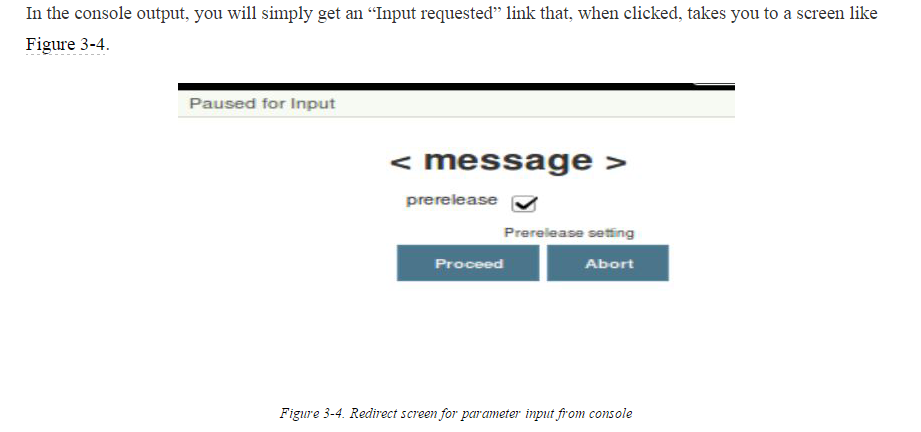
An example of the syntax would be:

**def** answer = input message: '<message>',

parameters: [booleanParam(defaultValue: **true**,

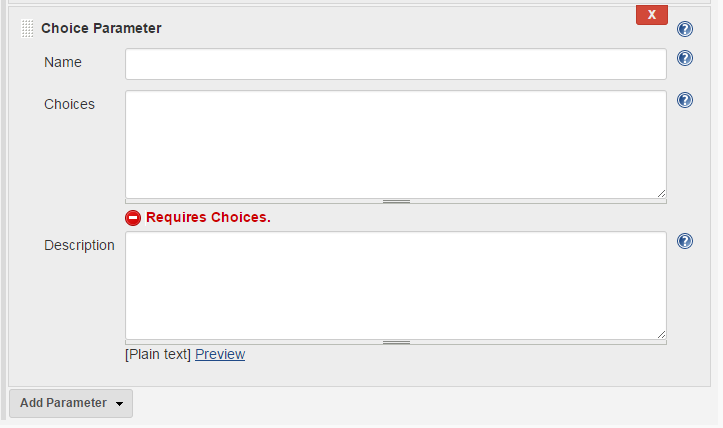
description: 'Prerelease setting', name: 'prerelease')]





### CHOICE

This parameter allows the user to select from a list of choices. The subparameters for a Choice are Name, Choices, and Description. Here, Choices refers to a list of choices you enter to present to the user. The first one in the list will be the default.



An example of the syntax would be:

**def** choice = input message: '<message>',

parameters: [choice(choices: "choice1\nchoice2\nchoice3\nchoice4\n",

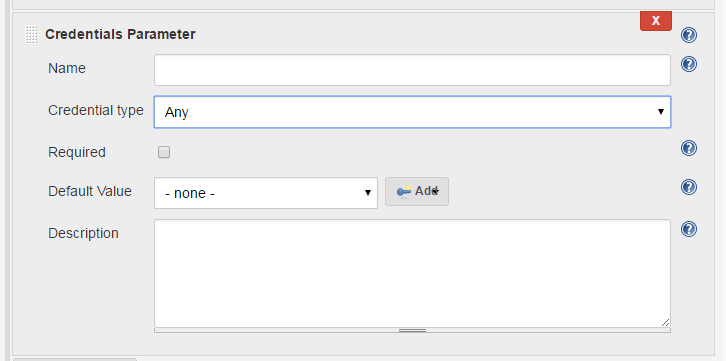
description: 'Choose an option', name: 'Options')]



### CREDENTIALS

This parameter allows the user to select a type and set of credentials to use.

The available subparameters include Name, Credential Type, Required, Default Value, and Description.



The options for Credential Type include Any, Username with password, Docker Host Certificate Authentication, SSH Username with private key, Secret file, Secret text, and Certificate.

If Required is specified, then a credential must be specified when the user is asked for this field. (It can’t be empty.) This doesn’t imply that a build will be able to use the credentials or that they will be valid, but just specifies that a selection is required.

The Default Value is the default credentials (selected from the set of ones already defined in Jenkins).

An example of the syntax follows for an SSH key:

**def** creds = input message: '<message>',

parameters: [[$class: 'CredentialsParameterDefinition', credentialType:

'com.cloudbees.jenkins.plugins.sshcredentials.impl.BasicSSHUserPrivateKey',

defaultValue: 'jenkins2-ssh', description: 'SSH key for access',

name: 'SSH', required: **true**]]

echo creds

This will print out the ID of the selected credentials.

And here is an example for username and password:

**def** creds = input message: '', parameters: [[$class:

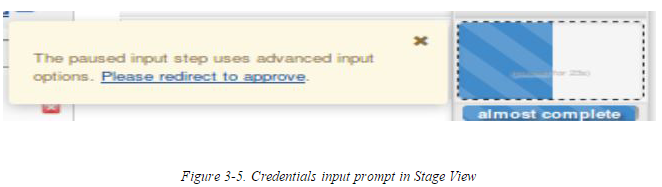
'CredentialsParameterDefinition', credentialType:

'com.cloudbees.plugins.credentials.impl.UsernamePasswordCredentialsImpl',

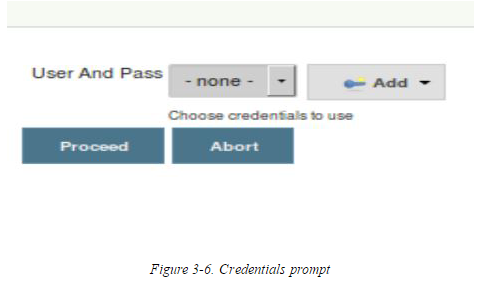
defaultValue: '', description: 'Enter username and password',

name: 'User And Pass', required: **true**]]

Note that this will not prompt with fields to enter a username and password. Rather, it presents the interface to select an existing credential or add a new one. In the Stage View, it looks like [Figure 3-5](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#fig_cred_inp_promp_SV)

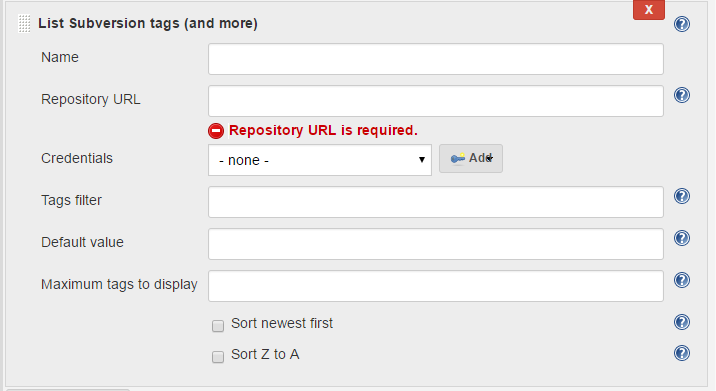


Once you click the “Please redirect to approve” link, you are taken to the prompts for selecting credentials ([Figure 3-6](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#fig_credenditals_prompt_a)). The prompt from the console is the same as in the previous cases.



### LIST SUBVERSION TAGS

This parameter allows you to specify a set of tags in Subversion to select from when running a build. The subparameters include Name, Repository URL, Credentials, Tag Filter, Default Value, the Maximum tags to display, and sorting options for newest first and/or alphabetical sorting.



For the Repository URL subparameter, Jenkins expects you to specify the URL of the Subversion repository that contains the tags you want to display. If this does not contain the tags and there are subfolders, then the subfolders will be displayed to enable drilling down.

Jenkins will check whether it can access this repository or not and prompt for credentials if needed.

The Credentials subparameter contains the credentials to access the repository, if required.

The Tag Filter refers to a regular expression to filter the list of tags presented.

The Default Value is used only if required for SVN polling or similar features.

**def** tag = input message: '<message>',

parameters: [[$class: 'ListSubversionTagsParameterDefinition',

credentialsId: 'jenkins2-ssh', defaultValue: '', maxTags: '',

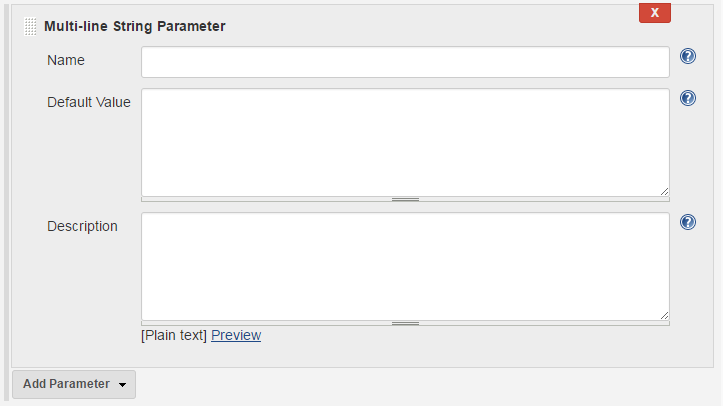
name: 'LocalSVN', reverseByDate: **false**, reverseByName: **false**,

tagsDir: 'file:///svnrepos/gradle-demo', tagsFilter: 'rel\_\*']]

The interfaces act like the ones for the File and Credentials parameters, except that there is a drop-down with the matching list of tags to choose from instead of a file or credential selection widget.

### MULTILINE STRING

This parameter allows the user to input multiple lines of text. The subparameters include Name, Default Value, and Description.



**def** lines = input message: '<message>',

parameters: [text(defaultValue: '''line 1

line 2

line 3''', description: '', name: 'Input Lines')]

Notice the entries in the commands are on different lines. This is because they have newlines entered with the default values. Also notice the triple quotes before and after the multiline message. The triple quotes are a standard notation used with Groovy for things that span multiple lines.

As you might expect, when executing, this will pop up (or link to) an entry box where you can type multiple lines of text.

### PASSWORD

This parameter allows the user to enter a password. For passwords, the text the user enters is hidden while they type it. The available subparameters are Name, Default Value, and Description.

Here’s an example:

**def** pw = input message: '<message>',

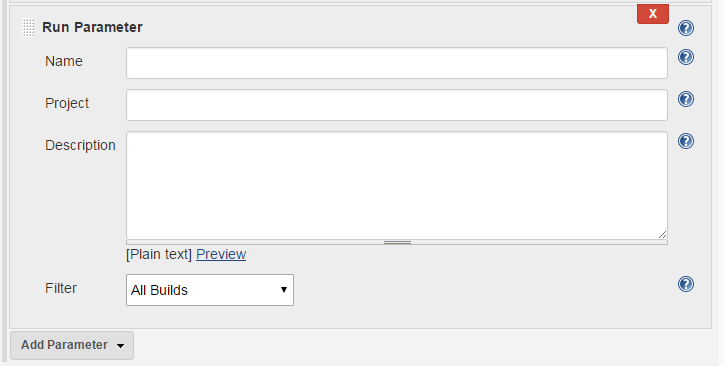
parameters: [password(defaultValue: '',

description: 'Enter your password.', name: 'passwd')]

When run, the user is presented with a field to enter the password, with the text being hidden as they type.

### RUN

This parameter allows the user to select a particular run (executed build) from a job. This might be used, for example, in a testing environment. The subparameters available include Name, Project, Description, and Filter.



The Project subparameter is the job that you want to allow the user to select a run from. The default run will be the most recent one.

The Filter subparameter allows you to filter the type of runs to offer based on the overall build status. Choices include:

* All Builds (including “in-progress” ones)
* Completed Builds
* Successful Builds (this includes stable and unstable ones)
* Stable Builds Only

Here’s an example of code for this one:

**def** selection = input message: '<message>',

parameters: [run(description: 'Choose a run of the project',

filter: 'ALL', name: 'RUN', projectName: 'pipe1')]

echo "selection is ${selection}"

This will output a response like:

selection is <project name> #<run number>

### STRING

This parameter allows the user to enter a string. (This value is not hidden, like with a Password parameter.) The subparameters include Name, Default Value, and Description.



Here’s an example:

**def** resp = input message: '<message>', parameters: [string(defaultValue: '',

description: 'Enter response', name: 'Response')]

When run, the user is presented with a field to enter in the desired string.

## Return Values from Multiple Input Parameters

In all of the examples just shown, we included only a single parameter. This syntax provides a simple return value that directly contains the value input by the user. If there were instead no parameters, such as having only a Proceed or Abort option, then the return value would be null. And when you have multiple parameters, a map is returned where you can extract each parameter’s return value via the parameter’s name. An example follows.

Suppose we wanted to add a traditional login screen to our pipeline. We would use two parameters—one String parameter for the login name and one Password parameter for the password. We can do that in the same inputstatement and then extract the return values for each from the returned map.

The following example code shows how to define the input statement along with some print statements that show different ways to access the individual return values (don’t forget that you can use the Snippet Generator for generating the input statement as well):

**def** loginInfo = input message: 'Login',

parameters: [string(defaultValue: '', description:

'Enter Userid:', name: 'userid'),

password(defaultValue: '',

description: 'Enter Password:', name: 'passwd')]

       echo "Username = " + loginInfo['userid']

       echo "Password = ${loginInfo['passwd']}"

       echo loginInfo.userid + " " + loginInfo.passwd

## Parameters and Declarative Pipelines

Since creating new local variables to hold the return values from input statements doesn’t fit the declarative model, you may be wondering how we can use the input statement in Declarative Pipelines. There are several approaches here, including one that leverages the declarative structure and one that works around it.

### USING THE PARAMETERS SECTION

Within the Declarative Pipeline structure, there is a section/directive for declaring parameters. This is within the agent block of the main pipeline closure. [Figure 3-7](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#fig_declar_pipl_struc) shows where this fits overall.

Use of the parameters directive is covered in detail with Declarative Pipelines in [Chapter 7](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#CH_Declarative_Pipelines), but here’s a simple example of the syntax (see [“parameters”](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#Ch7_sec_parameters) for more details):

pipeline {

    agent any

    parameters {

        string(name: 'USERID', defaultValue: '',

description: 'Enter your userid')

    }

    stages {

        stage('Login') {

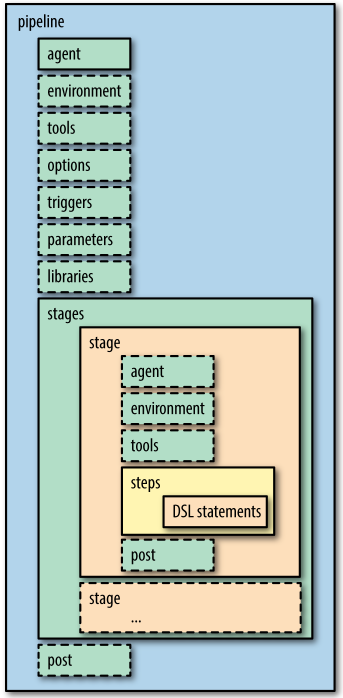
            steps {

                echo "Active user is now ${params.USERID}"

            }

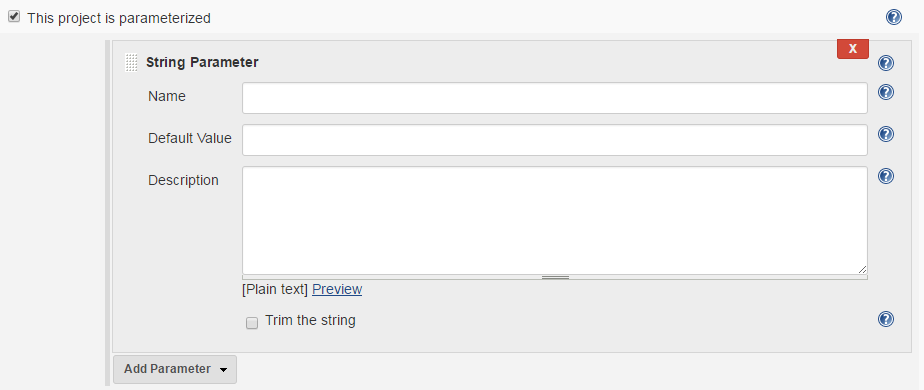
        }

    }



### USING THE JENKINS APPLICATION TO PARAMETERIZE THE BUILD

If you have created a job in the Jenkins application (rather than using a Jenkinsfile automatically), a second approach for adding parameters is to simply use the traditional method for parameterizing a job. That is, in the General configuration section, select the checkbox for “This project is parameterized” and then define your parameters as normal in the job’s web interface



You can then simply reference the job parameters via params.<name of parameter> without having the input line in the code, as shown here:

pipeline {

    agent any

    stages {

        stage('Login') {

            steps {

                echo "Active user is now ${params.USERID}"

            }

        }

    }

}

A variant of this approach is to define the parameters as properties before the pipeline. This can actually be done either for Scripted or Declarative Pipelines. Here’s how it might look in the code:

properties ([

    parameters ([

        string(defaultValue: '', description: '', name : 'USERID')

    ])

])

pipeline {

    agent any

    stages {

        stage('Login') {

            steps {

                echo "Active user is now ${params.USERID}"

            }

        }

    }

}

However, since this works only within the scope of the Jenkins application and the particular job within it, this is not recommended for production use. It also will overwrite any existing properties defined in Jenkins for the job.

With that said, it can be a useful way to prototype parameter usage in a pipeline for certain cases.

### USING A SCRIPT BLOCK

While Declarative Pipelines are continuing to evolve and add more functionality, there may still be instances where you need to do something in one that the declarative style doesn’t support or renders very difficult to implement. For those cases, the declarative syntax supports a script block.

A script block allows you to use nondeclarative syntax within the bounds of the block. This includes defining variables, which is not something you can do in a Declarative Pipeline outside of a script block. This also means that you cannot reference variables that are defined inside a script block outside of that block. Jenkins flags those with a “no such property” error.

As an example of all of this, consider the following section of code:



Here we have two parameters defined as part of an input step inside of a stage in a Declarative Pipeline. Since the first echo is in the script block where the variable resp is also defined, it will print out the response that is entered for that parameter as expected.

Notice, though, that the second echo is outside of the scope where the resp variable is defined. Groovy/Jenkins will throw an error when it gets to this one.

Because of this, it is advisable to try to limit accessing input to a small section of your code if you have to use a script block. However, there is one other workaround if you need to use the value outside the scope of the script block. You can put the return value into an environment variable and then access the environment variable wherever you need the value.

Updating our code to use this method could look like the following:

stage ('Input') {

       steps {

              script {

                 env.RESP1 = input message: '<message>', parameters: [

                 string(defaultValue: '', description: 'Enter response 1',

                 name: 'RESPONSE1')]

                 env.RESP2 = input message: '<message>', parameters: [

                 string(defaultValue: '', description: 'Enter response 2',

                 name: 'RESPONSE2')]

                 echo "${env.RESP1}"

              }

              echo "${env.RESP2}"

           }

        }

We are putting the results of the input steps into the environment variable namespace (env). Because these are environment variables, the values are set in the environment and therefore available for the pipeline to use wherever it needs.

Note that we’ve broken the single input statement down into two separate input statements. This results in the two environment variables RESP1 and RESP2 each having just the contents of their respective input lines. You can instead use multiple parameters in an input statement and set an environment variable with the results. The environment variable will have the form:

<parameter\_name>=<input\_value>, <parameter\_name>=<input\_value>, ...

# Flow Control Options

One of the benefits of writing your pipeline-as-code in Jenkins (versus using the traditional web forms) is that you have more options for controlling the flow through the pipeline. This includes handling cases that might otherwise cause your pipeline to stop or fail. The options available include ways to accomplish waiting, retries, etc. We’ll walk through each of them now.

## timeout

The timeout step allows you to limit the amount of time your script spends waiting for an action to happen. The syntax is fairly simple. Here’s an example:

timeout(time:60, unit: 'SECONDS') {

*// processing to be timed out inside this block*

}

The default unit for time is minutes. If you only specify a time value, it will be assumed to be in minutes. If the timeout is hit, then the step will throw an exception. This will cause the processing to abort if the exception isn’t handled some other way.

A best practice is to wrap any step that can pause the pipeline (such as an input step) with a timeout. This is so that your pipeline continues to execute (if desired) even if something goes wrong and the expected input doesn’t occur within the time limit. Here’s an example:

node {

**def** response

    stage('input') {

       timeout(time:10, unit:'SECONDS') {

          response = input message: 'User',

parameters: [string(defaultValue: 'user1',

description: 'Enter Userid:', name: 'userid')]

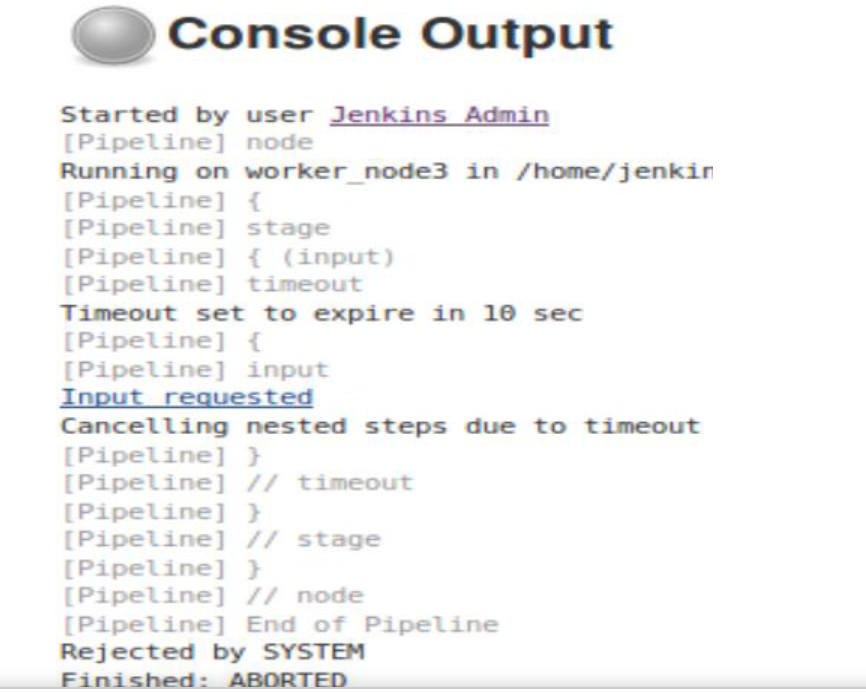
       }

       echo "Username = " + response

    }

}

In this case, Jenkins will wait for 10 seconds for the user to enter a response. If that time passes, Jenkins will throw an exception causing the pipeline to abort. You can see the sequence in the output captured in



As shown by the console output, the timeout does stop the pause in processing while waiting on input. However, when it does this, it throws an exception, causing our pipeline to abort. In order to not abort the pipeline, we can wrap the timeout in a traditional try-catch block, as shown in the following code. Notice that we set the response to the desired default when we handle the exception:

node {

**def** response

    stage('input') {

**try** {

         timeout(time:10, unit:'SECONDS') {

            response = input message: 'User',

parameters: [string(defaultValue: 'user1',

description: 'Enter Userid:', name: 'userid')]

         }

       }

**catch** (err) {

          response = 'user1'

       }

    }

}

## retry

The retry closure wraps code in a step that retries the process *n* times if an exception occurs in the code. *n* here refers to a value you pass in to the retry step. The syntax is just:

retry(<n>) { *// processing }*

If the retry limit is reached and an exception occurs, then the processing is aborted (unless that exception is handled, such as with a try-catch block).

## sleep

This is the basic delay step. It accepts a value and delays that amount of time before continuing processing. The default time unit is seconds, so sleep 5 waits for 5 seconds before continuing processing. If you want to specify a different unit, you just add the unit name parameter, as in:

sleep time: 5, unit: 'MINUTES'

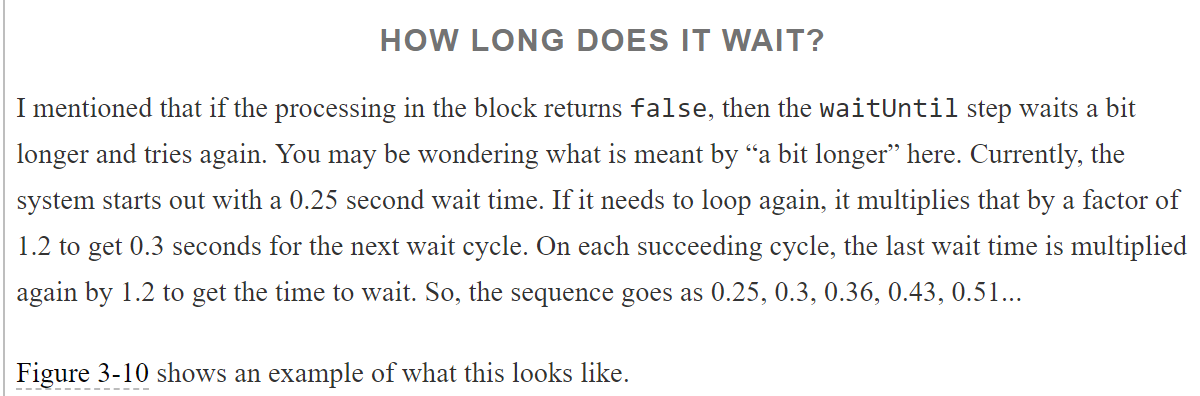
## waitUntil

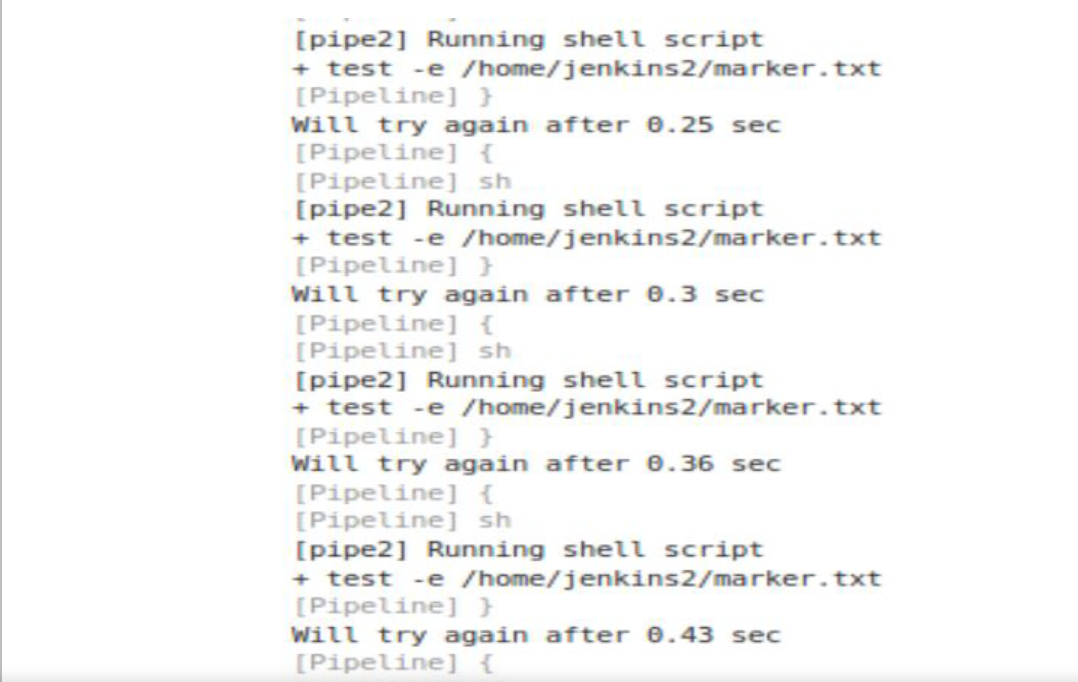
As you might guess, this step causes processing to wait until something happens. The “something” in this case is the closure returning true.

If the processing in the block returns false, then this step waits a bit longer and tries again. Any exceptions thrown in the processing cause the step to exit immediately and throw an error.

The syntax for waitUntil is simply:

waitUntil { *// processing that returns true or false }*





Because this step could end up waiting indefinitely if the processing never returns true (whether by intention or not), it is recommended to wrap this step with a timeout step so that eventually processing will.

Here is an example of using a waitUntil block to wait until we have a marker file in place. Notice that we have a timeout around the waitUntil to avoid staying in the waitUntil indefinitely. Also, we are setting the returnStatus parameter to true for the shell call, so that we get the return code back from the operation to check for success:

 timeout(time:15, unit:'SECONDS') {

    waitUntil {

**def** ret = sh returnStatus: **true**,

script: 'test -e /home/jenkins2/marker.txt'

**return** (ret == 0)

   }

 }

As another example, suppose we are waiting for a Docker container to start up so that we can get some data via a REST API call as part of our pipeline testing. In this case we get an exception if the URL isn’t available yet. To ensure that we don’t exit right away when the exception is thrown, we can use a try-catch structure to catch the exception and return false in that case. We also wrap it in a timeout as a guard against it not being available at all for some reason and holding up our pipeline

timeout(time: 120, unit: 'SECONDS') {

          waitUntil {

**try** {

                sh "docker exec ${webContainer.id} curl

--silent http://127.0.0.1:8080/roar/api/v1/registry

1>test/output/entries.txt"

**return** **true**

             }

**catch** (exception) {

**return** **false**

             }

          }

       }

Note that if we were doing this inside of a Declarative Pipeline, we would have to use a method such as a script block or shared library to handle this code.

# Dealing with Concurrency

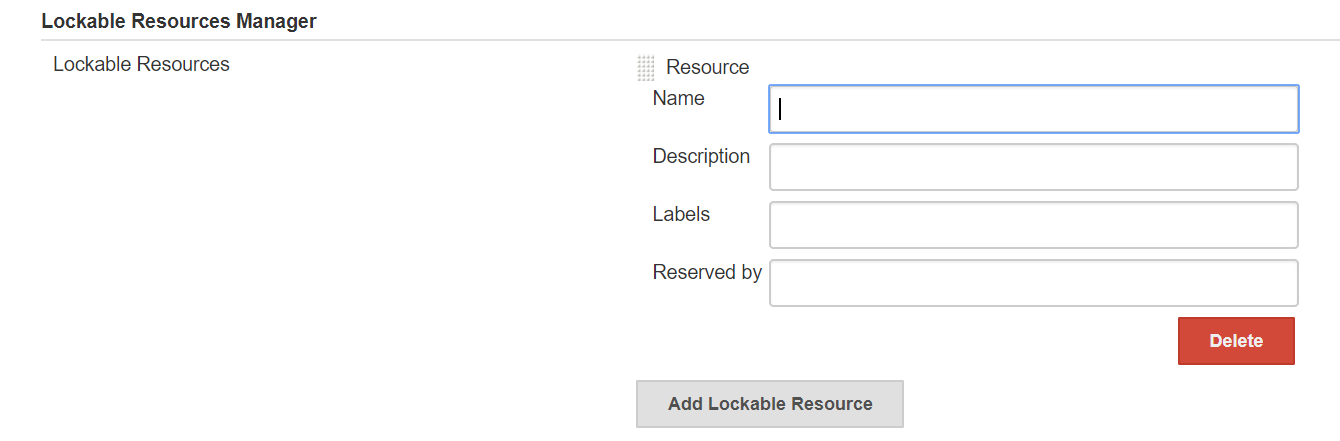
For the most part, having concurrency in your pipeline builds is a good thing. Typically, concurrency refers toparallelism—being able to run similar parts of your jobs concurrently on different nodes. This can be especially useful in cases such as running tests, as long as you limit duplicate access to resources appropriately.

Another form of concurrency in Jenkins is when multiple builds of the same job try to run at the same time or usethe same resources. In the case of very active repositories, branches, or pull requests, this may be an expected,common situation.

But there may also be cases where this is not expected and not desirable. Let’s look briefly at two mechanismsthat Jenkins pipelines have to address that situation.

## Locking Resources with the lock Step

If you have the [Lockable Resources plugin](http://bit.ly/2vtAOej) installed, there is a DSL lock step available to restrict multiple buildsfrom trying to use the same resource at the same time. (There will also be a Lockable Resources section on the Configure System page where you can globally define and reserve resources if necessary—for example, if you temporarily need to take a set of resources offline for a system.)



“Resource” here is a loose word. It could mean a node, an agent, a set of them, or just a name to use for thelocking. If the specified resource isn’t defined in the global configuration, it will be added automatically.

The DSL lock step is a blocking step. It locks the specified resource until the steps within its closure arecompleted. In its simplest case, you just supply the resource name as the default argument. For example:

lock('worker\_node1') {

*// steps to do on worker\_node1*

}

Alternatively, you can supply a label name to select a set of resources that have a certain label and a quantity to specify the number of resources that match that label to lock (reserve):

lock(label: 'docker-node', quantity: 3) {

*// steps*

}

You can think of this as, “How many of this resource do I have to have available to proceed?” If you specify a label but no quantity, then all resources with that label are locked.

As a quick example, consider a Declarative Pipeline where we want to use a certain agent to do the build on, no matter how many instances of the pipeline we are running. (Perhaps it is the only agent with the specific tools or setup we want at the moment.) Our code might look like this with the lock step:

stage('Build') {

*// Run the gradle build*

   steps {

      lock('worker\_node1') {

      sh 'gradle clean build -x test'

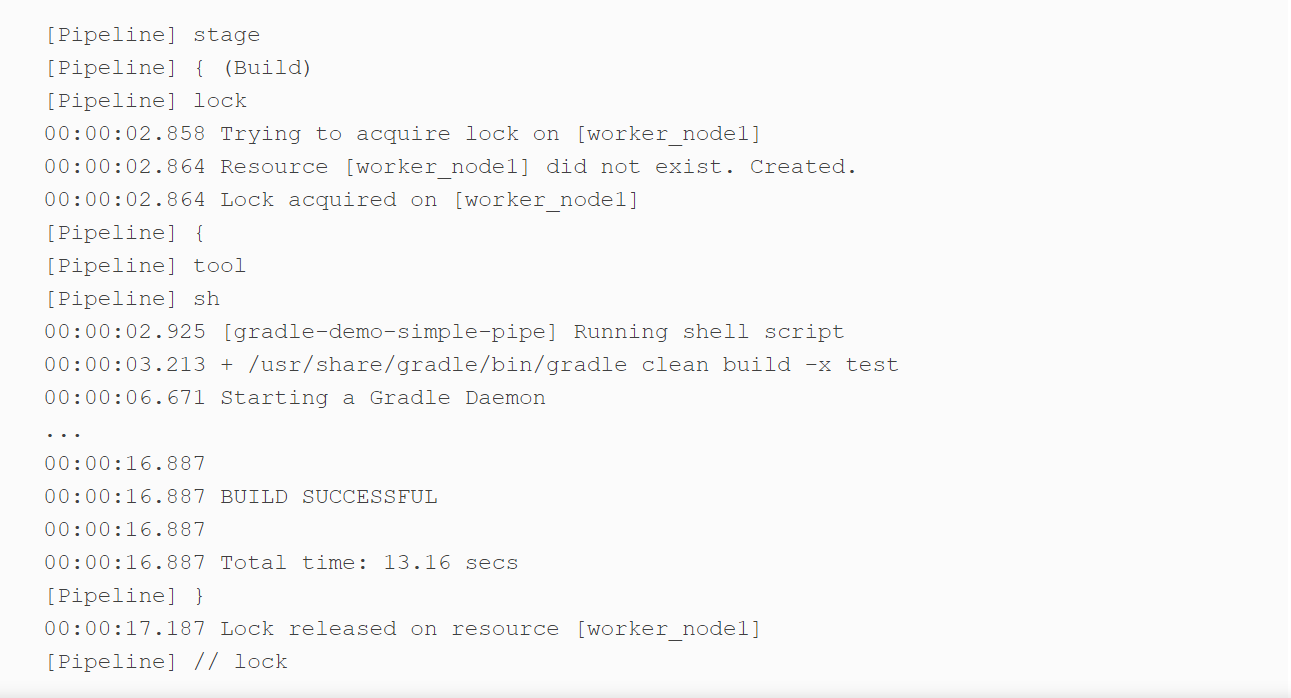
   }

   }

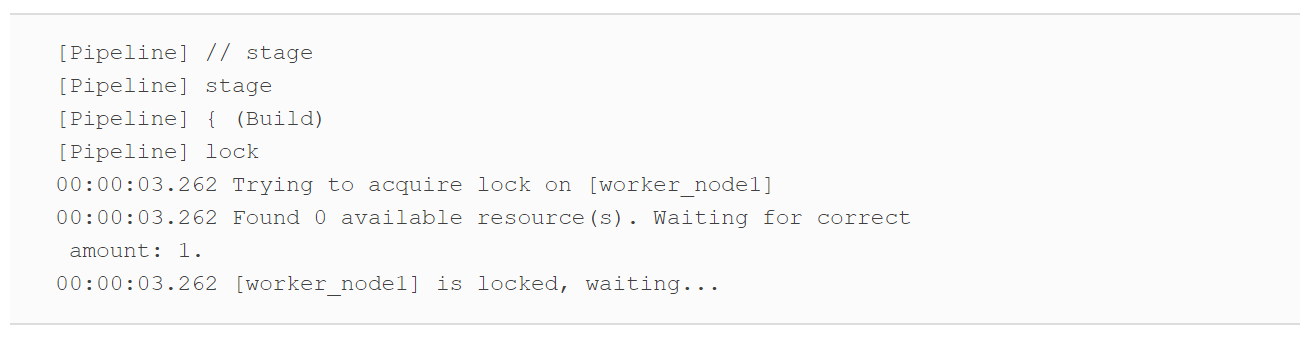
}

If we start multiple builds running of the same project or if we have multiple projects with this same lock code for the resource, then one build/project will get the resource first and other builds/projects will have to wait.

For the first build or project that gets the resource, the console log might show something like this:



And for the other builds/jobs trying to acquire the same lock, console output might look like this:



## Controlling Concurrent Builds with Milestones

One of the scenarios that you might have to deal with at some point in Jenkins is builds of the same pipeline running concurrently that can have contention for resources. The runs could be reaching key points at different times and stepping on each other, or one run could be modifying required resources that leave things in a bad state when the other run makes it to that point. In short, there’s no guarantee that after one run has modified a resource, another run won’t come along and modify it while the earlier run is still in progress.

To prevent the case where builds could run out of order (in terms of the order they were started) and step on each other, Jenkins pipelines can use the milestone step. When a milestone step is put in the pipeline, it prevents an older build from moving past the milestone, if the newer build has already gotten there.

The following example shows a milestone step placed in a script after a Gradle build:

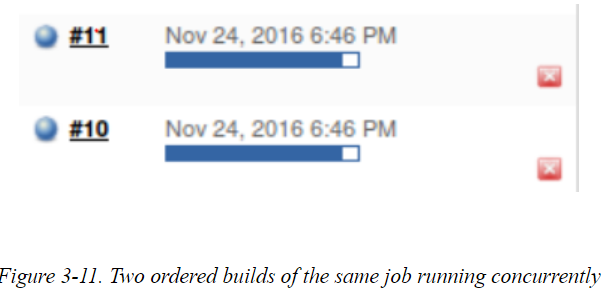
sh "'${gradleLoc}/bin/gradle' clean build"

}

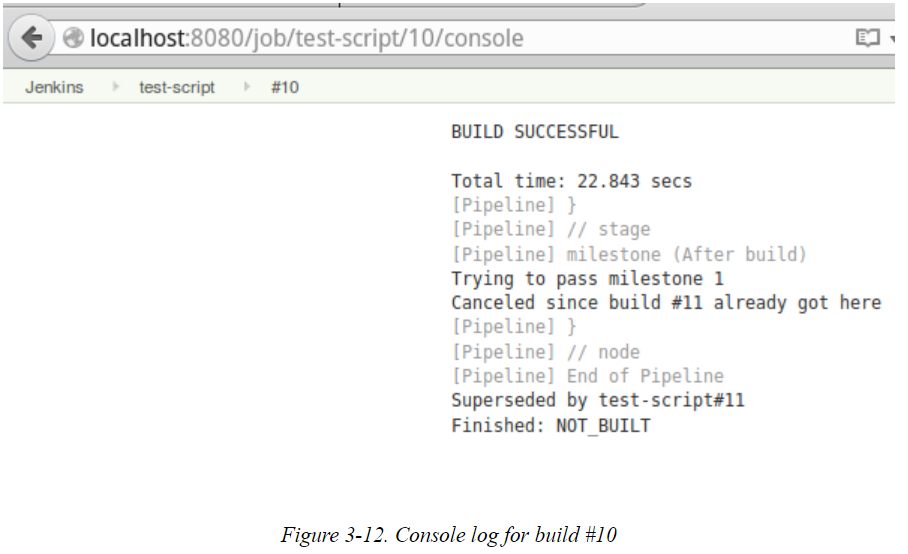
milestone label: 'After build', ordinal: 1

stage("NotifyOnFailure") {

Suppose we have two runs of this build happening concurrently, as shown



If build #11 gets to the milestone step first during its processing, then when build #10 arrives, it will becanceled. This prevents build #10 from overwriting or modifying any resources already in use or modified by build #11. The console log for build #10 for this part of the process is shown



.

The rules for milestone processing can be summed up as:

* Builds pass the milestones in order by build number.
* Older builds abort if a newer build has already passed the milestone.
* When a build passes a milestone, Jenkins aborts older builds that have passed the previous milestone, but not this milestone.
* If an older build passes a milestone, newer builds that haven’t passed the milestone won’t abort it.

To be clear, if concurrent builds reach the milestone in the order they were started, they can all pass the milestone.

The milestone step can take a couple of parameters. The first is a label, which is to identify the milestone. Itwill be shown in the build log. The second is an ordinal number. This is autogenerated if not set specifically. You only need to do this if you’re going to be adding/deleting milestones during the builds. There is also a way to restrict concurrent builds running for multiple branches in a Multibranch Pipeline project. That’s covered in the next section.

## Restricting Concurrency in Multibranch Pipelines

The pipeline DSL includes a way to restrict Multibranch Pipelines to only building one branch at a time. This is done with a property for either a Scripted or Declarative Pipeline. When this is in place (in the Jenkinsfiles of the branches), requested builds for branches other than the one currently building will be queued.

In a scripted syntax, the property can be set this way:

properties([disableConcurrentBuilds()])

In declarative syntax, it would look like this:

options {

disableConcurrentBuilds()

}

## Running Tasks in Parallel

In addition to the other constructs for controlling the logic flow of a pipeline, steps can also be run in parallel. In fact, the pipeline DSL has special constructs for doing this—a traditional one that fits both Scripted and Declarative Pipelines, and a newer one just for Declarative Pipelines. To illustrate the main points, we’ll talk about the more general one first and then the newer declarative syntax.

Look at the book - <https://www.cloudbees.com/blog/parallelism-and-distributed-builds-jenkins>

It is also possible to just define the mapping directly in the invocation of the parallel step. The following is an implementation done this way. Notice again that we are passing in a mapping with closures and nodes. In this implementation, the first occurrences of master and worker2 are the*keys* to the maps. The sections after the colons are the closures that make up the *value* portions of the map. In each of the closures for the map values, we allocate a block of code to run on specific nodes. In this case, the code block is a shell step (sh) that invokes Gradle to run a single test—a different one on each node

stage ('Test') {

*// execute required unit tests in parallel*

parallel (

master: { node ('master'){

sh '/opt/gradle-2.7/bin/gradle -D test.single=TestExample1 test'

}},

worker2: { node ('worker\_node2'){

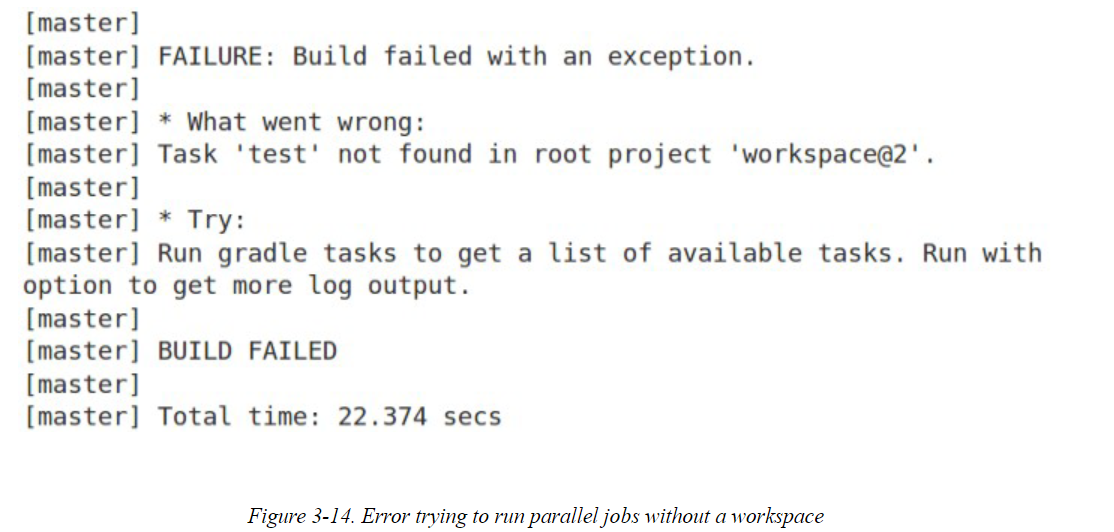
sh '/opt/gradle-2.7/bin/gradle -D test.single=TestExample2 test'

}},

)

}

However, trying to run this particular piece of code in most instances will run into a problem, as shown in



The challenge here is that the original build happened in a workspace on a different node and the new node (master, in this case) does not have access to that workspace.

We could archive artifacts here or try to copy them over ourselves, but Jenkins includes special steps to help withsuch a case. This is a good place to cover those.

### STASH AND UNSTASH

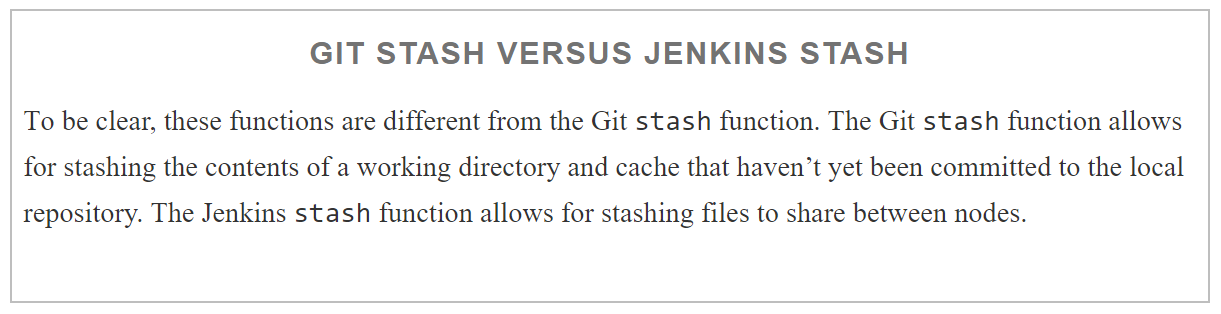
n the Jenkins DSL, the stash and unstash functions allow for saving and retrieving (respectively) filesbetween nodes and/or stages in a pipeline. Their format is:

stash name: "<name>" [includes: "<pattern>" excludes: "<pattern>"]

unstash "<name>"

The basic idea here is that we designate a set of included or excluded files via names and/or patterns. This stash of files is given a name to refer to it by.

Then, when we need to retrieve the set of files, we can simply pass the name of the stash to the unstashcommand. This can be done on a different stage or node.



The stash and unstash functions are not intended for formal management of large groups of files such as where you need to keep track of version numbers. For that type of requirement, it is better to use an artifactrepository designed for managing binary artifacts such as Artifactory or Nexus. (Artifactory and integration with Jenkins is discussed in [Chapter 13](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch13.html#CH_Integrating_Artifact_Management).)

An example of the use of the stash and unstash commands across nodes is shown next. In this case, after we get the source, we are stashing the *build.gradle* file and the entire *src/test* tree. This stash is given the name test-sources. Then, in the parallel section that runs on the other node (worker\_node2 in this case), the unstash command creates a copy of the stashed files and tree on that node. This allows the files to be present sothe testing can take place on that node as well and we can achieve the parallelism:

stages {

stage('Source') {

git branch: 'test', url: 'git@diyv:repos/gradle-greetings'

stash name: 'test-source', includes: 'build.gradle,src/test/'

}

...

stage ('Test') {

*// execute required unit tests in parallel*

parallel (

master: { node ('master') {

unstash 'test-sources'

sh '/opt/gradle-2.7/bin/gradle -D test.single=TestExample1 test'

}},

worker2: { node ('worker\_node2') {

unstash 'test-sources'

sh '/opt/gradle-2.7/bin/gradle -D test.single=TestExample2 test'

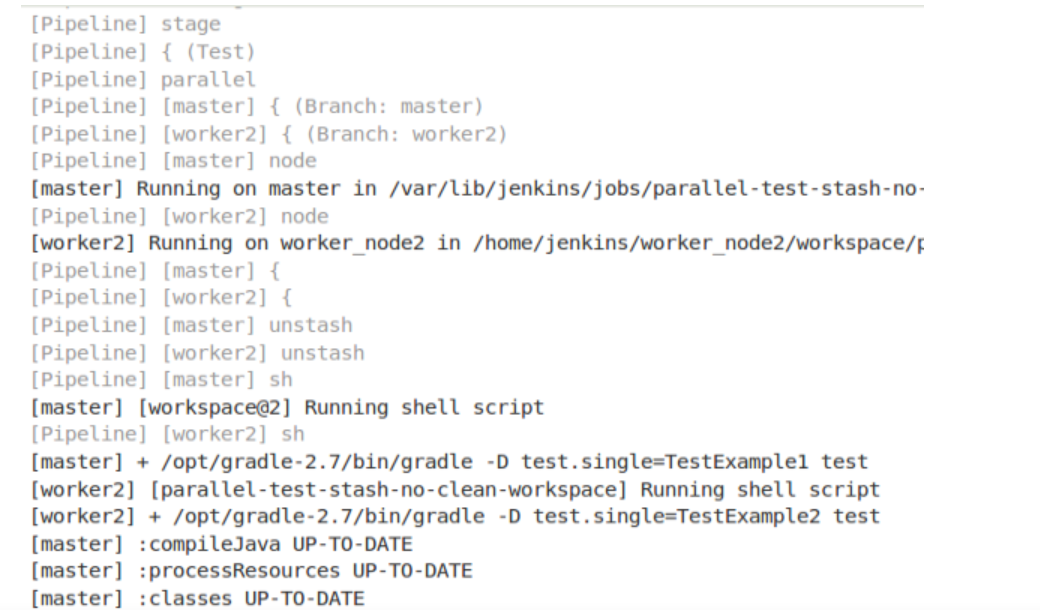
}},

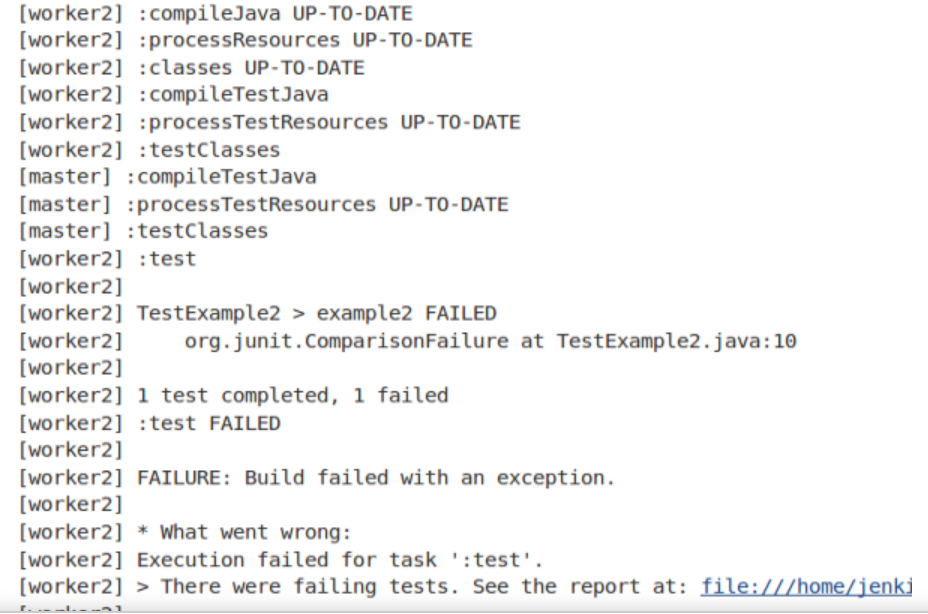
)

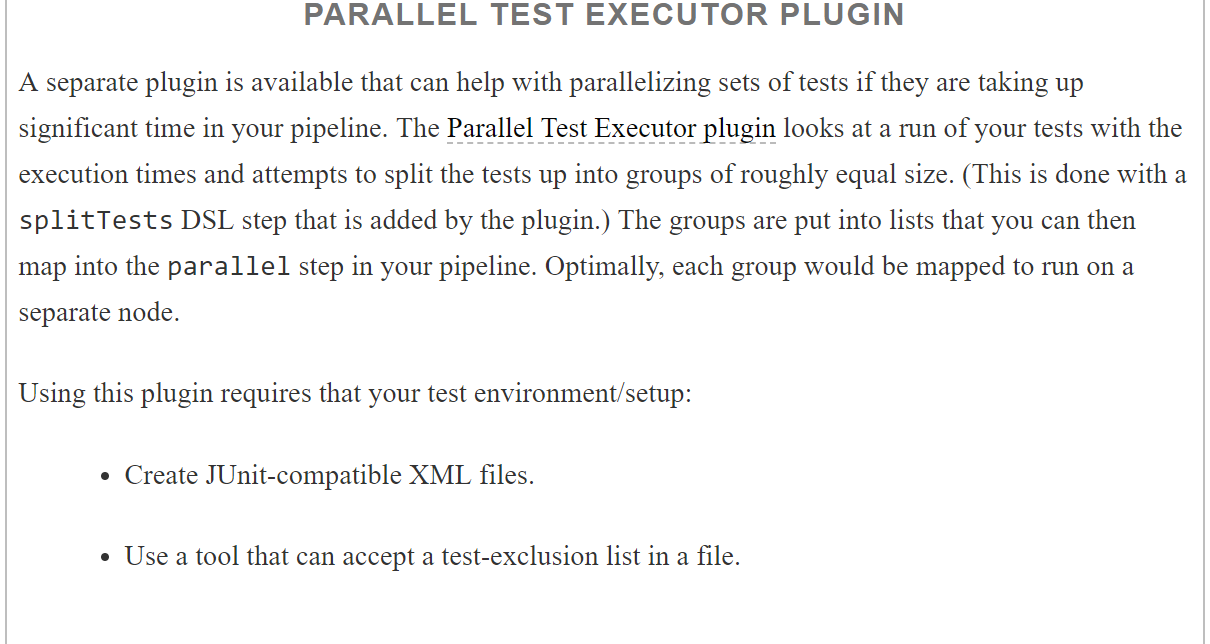
}

}

The log of running this sequence can be seen in [Figure 3-15](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#fig_para_run_w_stash_acroz). Again, note the interleaving of execution betweenthe nodes. (In this case there is a test that is supposed to fail, so this run is successful.)







### ALTERNATIVE PARALLEL SYNTAX FOR DECLARATIVE PIPELINES

The new syntax elevates the parallel step to a separate construct within a stage. It can have stages definedwithin itself for each branch to run in parallel. Within each branch you can define an agent to run on and steps toexecute just as you can for other Declarative Pipeline sections.

An excerpt of a stage from a Declarative Pipeline that uses this syntax is shown here:

stage('Unit Test') {

           parallel{

               stage ('Util unit tests') {

                  agent { label 'worker\_node2' }

                  steps {

                     cleanWs()

                     unstash 'ws-src'

                     gbuild4 ':util:test'

                  }

               }

               stage ('API unit tests set 1') {

                  agent { label 'worker\_node3'}

                  steps {

*// always run with a new workspace*

                     cleanWs()

                     unstash 'ws-src'

                     gbuild4 '-D test.single=TestExample1\* :api:test'

                  }

               }

               stage ('API unit tests set 2') {

                   agent { label 'worker\_node2' }

                   steps {

*// always run with a new workspace*

                      cleanWs()

                      unstash 'ws-src'

                      gbuild4 '-D test.single=TestExample2\* :api:test'

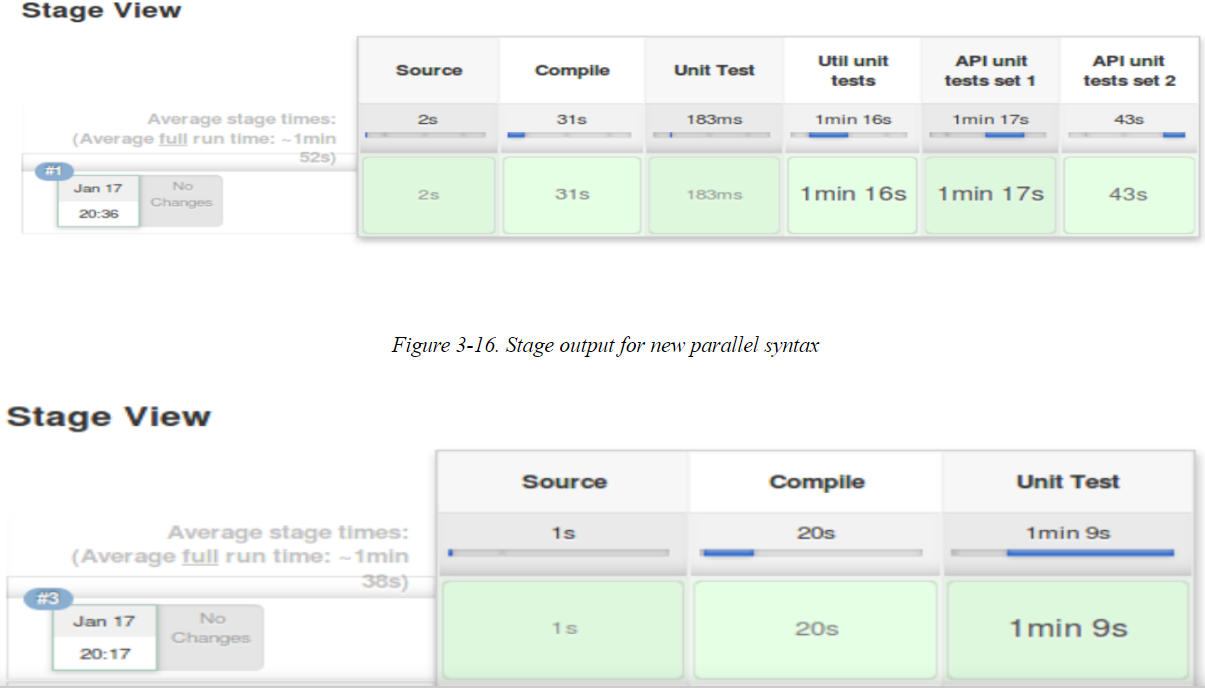
                   }

               }

           }

       }

As you can see, this syntax is somewhat “cleaner” than the map syntax and more consistent with the declarativesyntax. When run, due to the individual stage definitions, it will also produce stage output for each “substage” ([Figure 3-16](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#fig_stage_output_new_para_syntax)) as opposed to the single set of output of the traditional parallel syntax ([Figure 3-17](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#fig_stage_output_trad_PS)).



# Conditional Execution

Historically, the Conditional BuildStep plugin let users add conditional execution functionality to Freestyle jobs in Jenkins. It allowed a way to test certain conditions, and, based on the outcome, execute single or multiple buildsteps.

Jenkins pipelines can provide similar functionality. In the case of a Scripted Pipeline, it’s as simple as using the Groovy/Java language conditionals in your pipeline code. An example is included here using an if statementwith conditions that must be true for multiple parameters:

node ('worker\_node1') {

**def** responses = **null**

    stage('selection') {

        responses = input message: 'Enter branch and select build type',

parameters:[string(defaultValue: '', description: '',

name: 'BRANCH\_NAME'),choice(choices: 'DEBUG\nRELEASE\nTEST',

description: '', name: 'BUILD\_TYPE')]

    }

    stage('process') {

**if** ((responses.BRANCH\_NAME == 'master') &&

(responses.BUILD\_TYPE == 'RELEASE')) {

            echo "Kicking off production build\n"

        }

    }

}

Since these kinds of Groovy/Java-specific language features don’t fit in a declarative model, Declarative Pipelines in Jenkins provide their own implementation for executing code based on conditionals. In general, it takes the form of a when that tests one or more expression blocks to see whether they are true. If so, then the remaining code in a stage is executed. If not, then the code is not executed.

Here’s an example of a Declarative Pipeline that corresponds to the Scripted Pipeline just shown:

pipeline {

   agent any

      parameters {

         string(defaultValue: '',

               description: '',

               name : 'BRANCH\_NAME')

        choice (

            choices: 'DEBUG\nRELEASE\nTEST',

            description: '',

            name : 'BUILD\_TYPE')

      }

   stages {

      stage('process') {

         when {

            allOf {

               expression {params.BRANCH\_NAME == "master"}

               expression {params.BUILD\_TYPE == 'RELEASE'}

            }

         }

         steps {

            echo "Kicking off production build\n"

         }

      }

   }

}

Notice the use of the parameters section to formally define the parameters in use in the Declarative Pipeline.Also, you can see how the when and allOf blocks combine like the if-&& construct in the Scripted Pipeline.

Using these kinds of conditional constructs in Declarative Pipelines is covered in more detail in [“Conditional execution of a stage”](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#sec_condit_exec_stage).

# Post-Processing

Traditional (web-based) Jenkins Freestyle jobs include a Post-build Actions section where users can add actionsthat always occur after a build is finished, regardless of whether it completed successfully, failed, or was aborted.

We can replicate this functionality in both Scripted and Declarative Pipelines. The Scripted Pipeline relies on programming constructs to emulate this, while Declarative Pipelines have built-in functionality for it. We’ll look at both of these implementations next.

### TRY-CATCH-FINALLY

What we want to have is a way to always do certain actions regardless of the final state of the build. We can accomplish that by catching any exceptions with a try-catch and using the finally clause to then do our processing based on the build’s state. Most commonly, the processing we do in the finally clause would be sending mail or other notifications about the build’s state. Here’s an example of the structure with try-catch-finally:

**def** err = **null**

**try** {

*// pipeline code*

node ('node-name') {

stage ('first stage') {

...

} *// end of last stage*

  }

}

**catch** (err) {

   currentBuild.result = "FAILURE"

}

**finally** {

(currentBuild.result != "ABORTED"){

*// Send email notifications for builds that failed*

*// or are unstable*

   }

}

Notice that we are setting the value of currentBuild.result if there is an error, to ensure the build status is consistent with what we expect from Jenkins. Also, we don’t send mail if the build was aborted. (For examples of how to send mail and other notifications, see [Chapter 4](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch04.html#CH_Notifications_and_Reports).)

The try-catch could also be within the node block if we preferred. That would, however, not catch issues thrown while trying to get the node, which might also not be able to send the notification. Finally, if we wanted to propagate the error, we could throw it again in our finally block.

### CATCHERROR

The Jenkins pipeline syntax also provides a more advanced way of handling exceptions. The catchError block provides a way to detect the exception and change the overall build status, but still continue the processing.

With the catchError construct, if an exception is thrown by a block of code, the build is marked as a failure.But the code in the pipeline continues to be executed from the statement following the catchError block.

The advantage of this processing is that you can still do things like send notifications after processing has failed. This simulates the post-build processing that we’re accustomed to in the more traditional Jenkins model and also provides a shortcut over the try-catch block.

An example of using this is shown here:

node ('node-name') {

catchError {

      stage ('first stage') {

         ...

      } *// end of last stage*

   }

*// step to send email notifications*

}

This is essentially equivalent to the following code:

node ('node-name') {

**try** {

  stage ('first stage') {

         ...

       } *// end of last stage*

    } **catch** (err) {

       echo "Caught: ${err}"

       currentBuild.result = 'FAILURE'

    }

*// step to send email notifications*

}

The advantages are the simpler syntax and the build result automatically being marked as failed if an exception occurs.

## Declarative Pipelines and Post-Processing

Declarative Pipelines have a dedicated section for post-build processing. Not surprisingly, the section is calledpost. A post section can be at the end of a stage or at the end of a pipeline—or both.

The most common use for this is to mimic the post-build operations, especially notifications, that are availablefor Freestyle jobs. The declarative syntax provides several predefined “build conditions” that can be checked and, if true, then initiate further action. Their names and uses are explained in below

