**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.

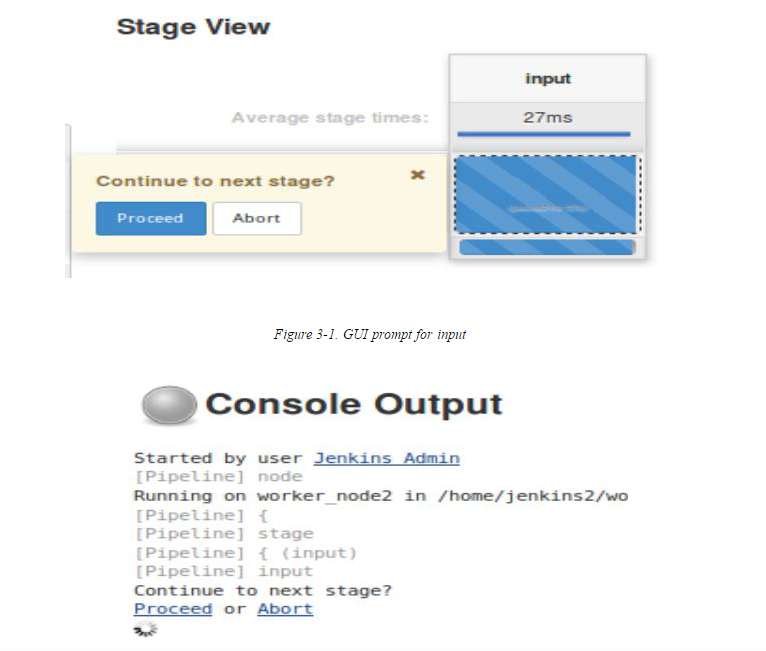
We describe this step and the parameters, as they can be used in the pipeline, next.

## input

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

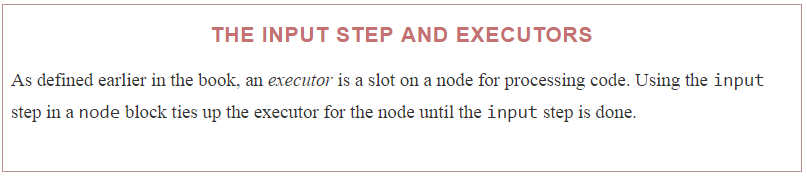
input 'Continue to next stage?'

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)

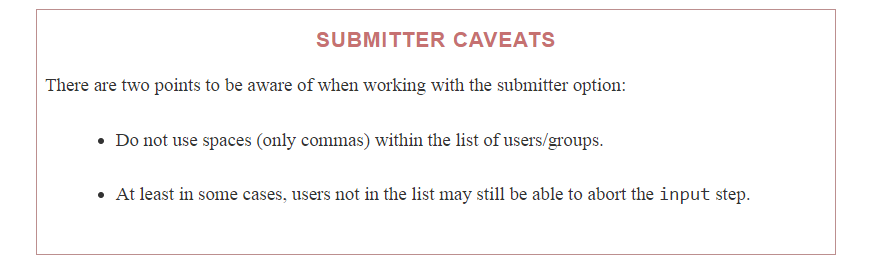
A different label you can use instead of “Proceed.” For example

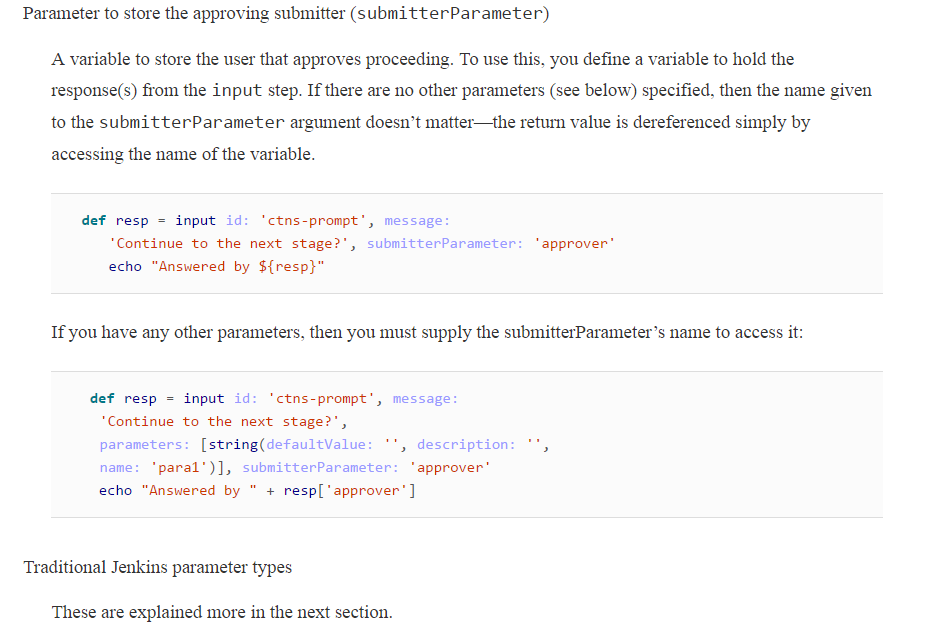
input message: '<message text>', ok: 'Yes'

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'





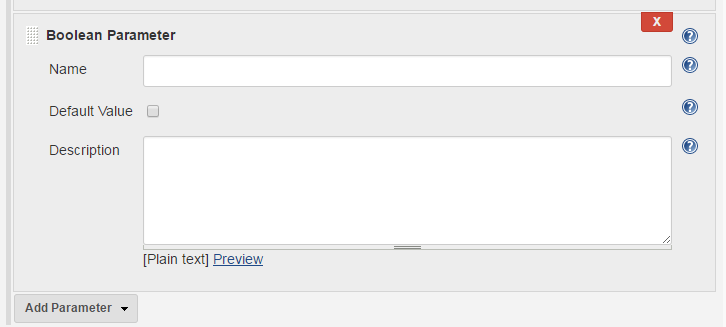
## Parameters

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.

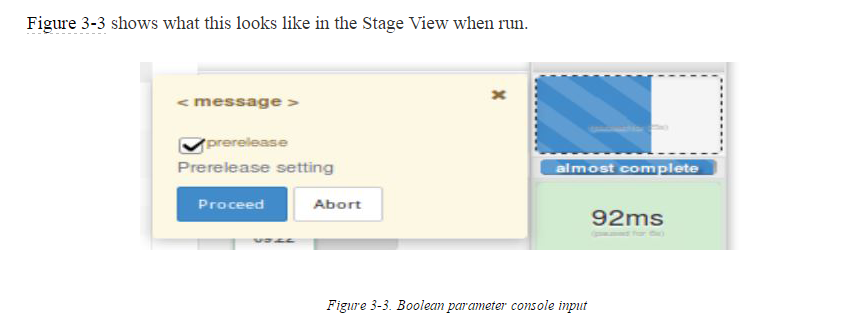


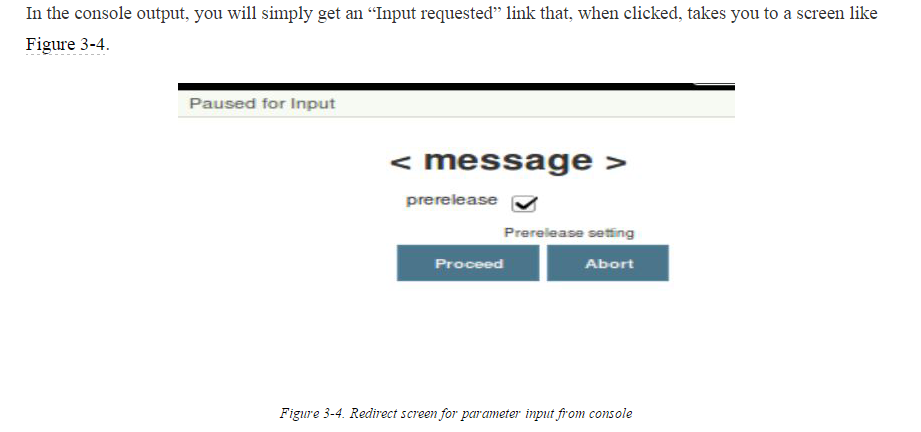
An example of the syntax would be:

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

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

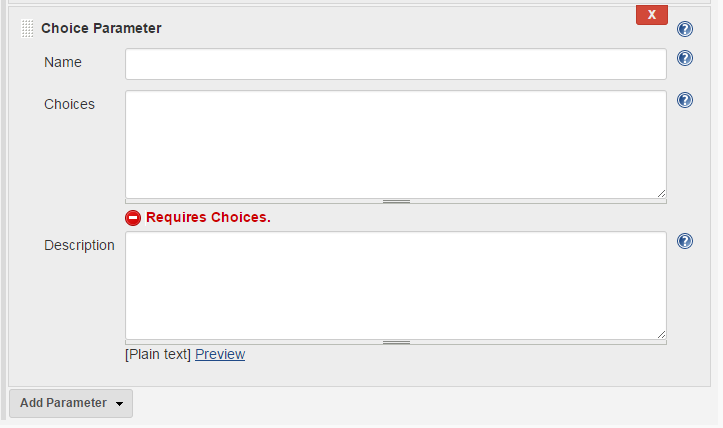
description: 'Prerelease setting', name: 'prerelease')]





### 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",

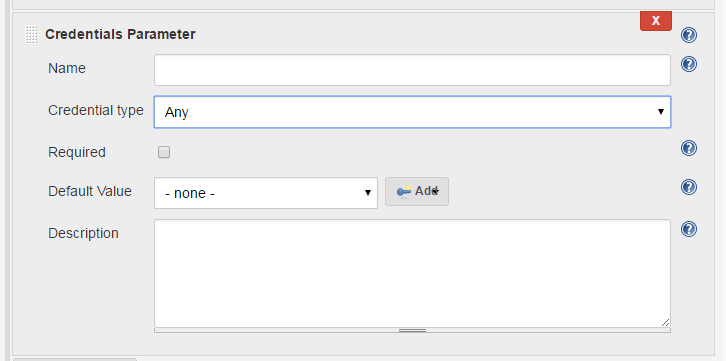
description: 'Choose an option', name: 'Options')]



### 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:

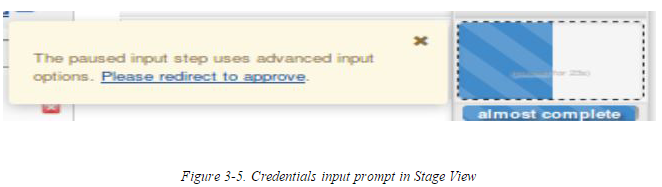
'CredentialsParameterDefinition', credentialType:

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

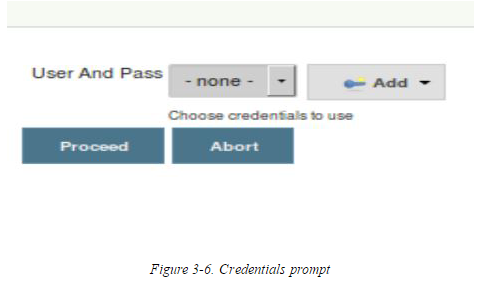
defaultValue: '', description: 'Enter username and password',

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)

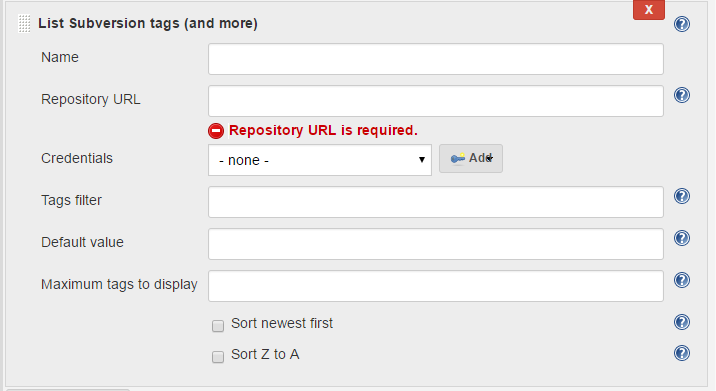


Once you click the “Please redirect to approve” link, you are taken to the prompts for selecting credentials ([Figure 3-6](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#fig_credenditals_prompt_a)). The prompt from the console is the same as in the previous cases.



### 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: '',

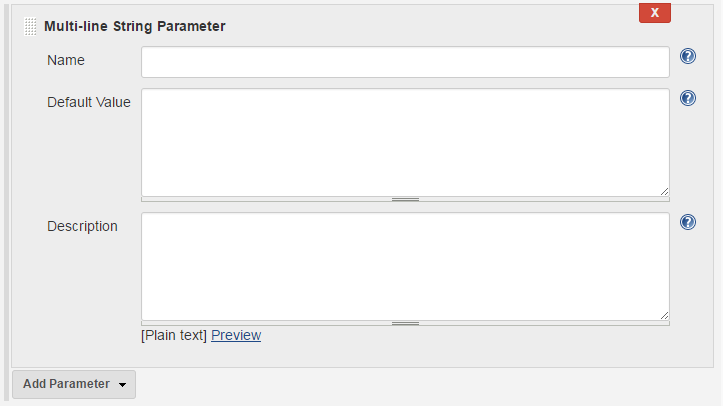
name: 'LocalSVN', reverseByDate: **false**, reverseByName: **false**,

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>',

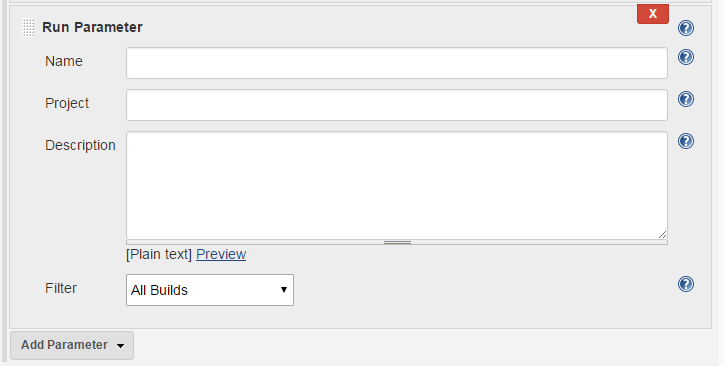
parameters: [password(defaultValue: '',

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') {

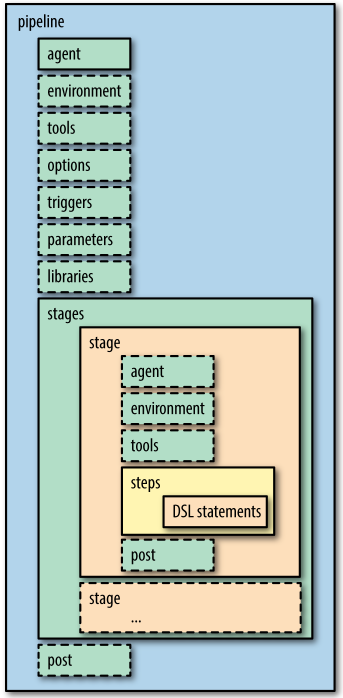
            steps {

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

            }

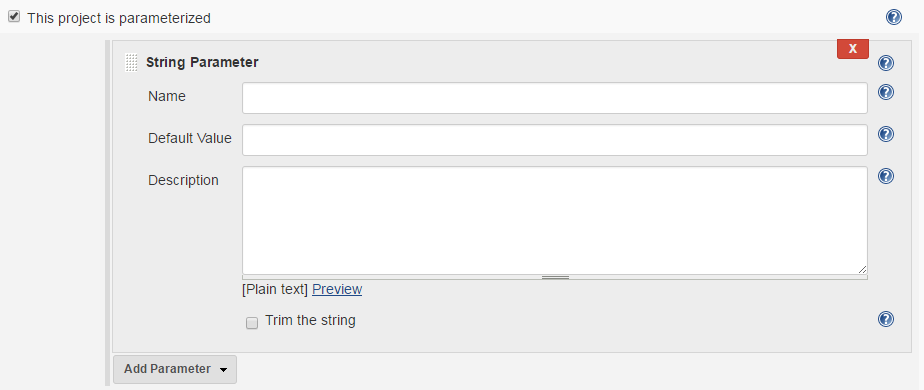
        }

    }



### 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:

