**Chapter 8. Understanding Project Types**

In the Jenkins 2 environment, several new project types have been added to provide extended functionality. Many of them leverage Jenkinsfiles, as markers, to automatically create jobs for the user. In this chapter, we’ll look at the most common project types in Jenkins, including these newer ones as well as traditional ones (like Freestyle and Maven projects).

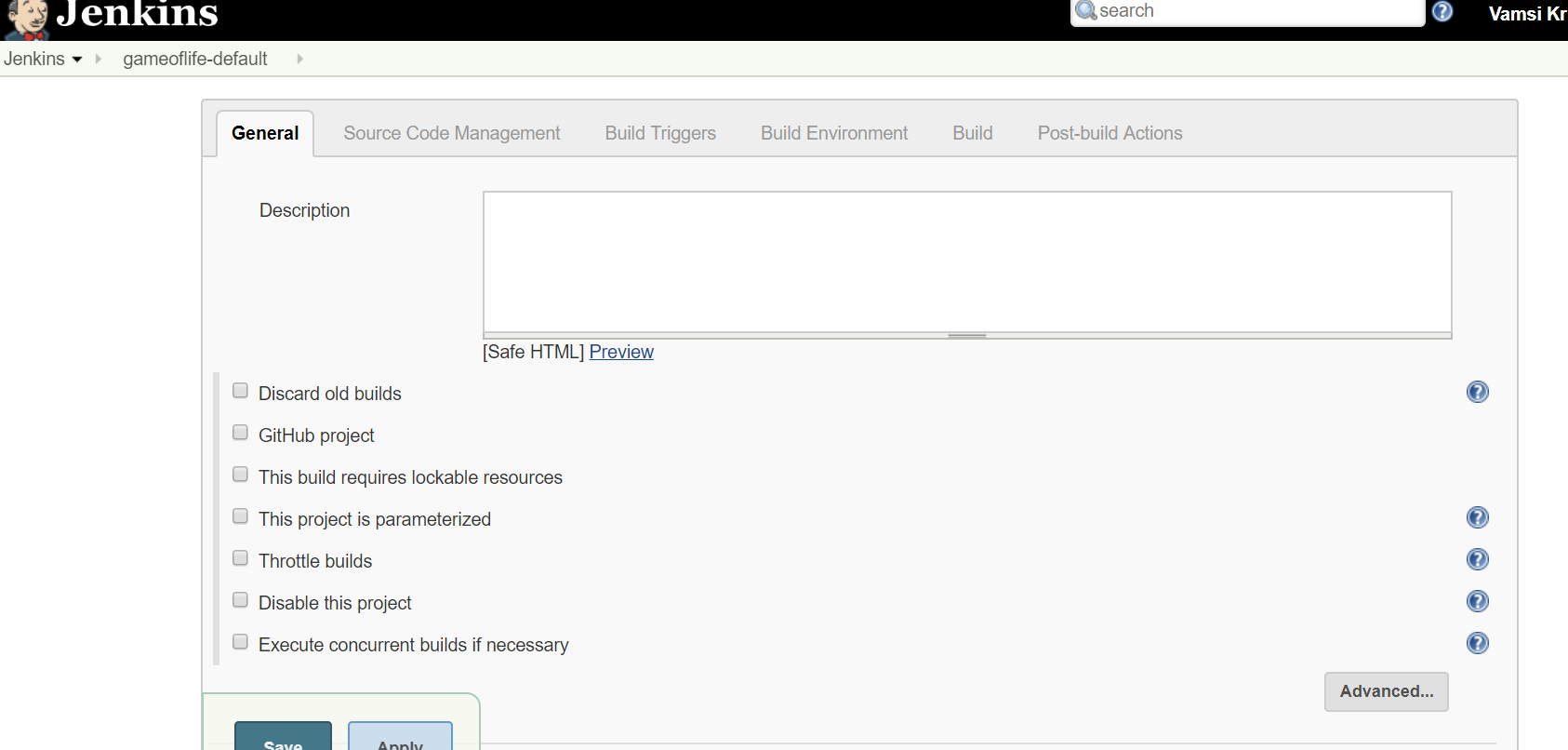
For most of the project types, there are certain common options present on the configuration page. These are in sections such as General, Build, Source Code Management, etc. In the first part of this chapter, we’ll cover those common options. Also, since we are focused on getting up and running with Jenkins 2, we’ll cover the corresponding pipeline functionality where we have an equivalent.

# Common Project Options

A number of the project types in Jenkins have configuration pages that are divided into specific sections. These sections can be scrolled to, or selected via tabs at the top of the page. We’ll look at each of the major sections, explain what the options mean, and also look at ways to implement corresponding functionality in pipelines. We’ll break these down based on the tab positions in a Freestyle project. Other types of projects may have some options on different tabs.

## **General**

The General section is where we configure the unique identifying information about the project, such as the description. (The project name will have already been set when the type of project was selected from the project selection dialog.)

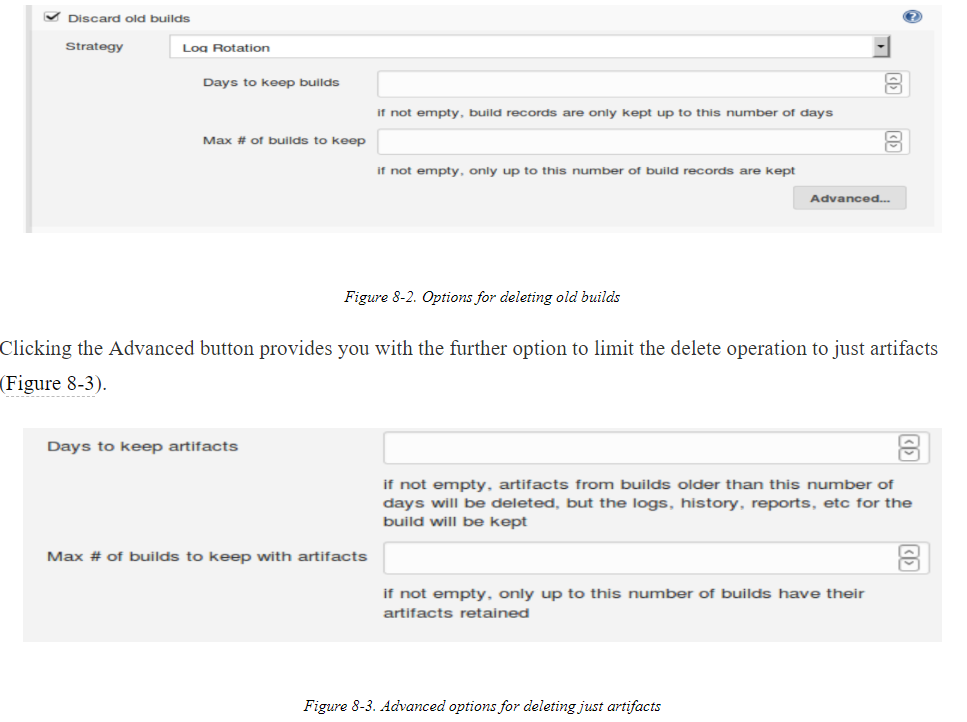


This is also where we can set some global options for the project, including ones that control job-level aspects. A survey of these follows.

### **DISCARD OLD BUILDS**

This option allows you to set up a strategy for Jenkins to follow in discarding previous builds of your project. While not required, it is helpful for aspects such as managing disk space (since each run of a project allocates a workspace area).

As shown in [Figure 8-2](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_opt_delet_oldbuilds), once you check the box, you can select the strategy to use for how many builds to keep. Although there is a Strategy drop-down, Log Rotation is currently the only choice; it is really the options underneath that dictate the strategy. Essentially, you can choose to keep each run’s work items and artifacts for a particular number of days or a particular number of builds.



#### Discarding builds in pipeline projects

For pipeline projects, there is a buildDiscarder option that can be configured. In a Scripted Pipeline, this is done via the properties step. Here’s an example constructed from the Snippet Generator:

properties([buildDiscarder(logRotator(artifactDaysToKeepStr: '',

artifactNumToKeepStr: '', daysToKeepStr: '3', numToKeepStr: '5')),

pipelineTriggers([])])

In a Declarative Pipeline, a similar entry can be made in the options section:

options {

    buildDiscarder(logRotator(numToKeepStr:'5'))

}

### **GITHUB PROJECT**

If you have the GitHub plugin installed, this option allows you to specify a GitHub URL for integration. With this integration, you can have links to your GitHub project in Jenkins (such as on the Changes page), and you can do integration builds based on changes to GitHub repositories. (Note that to be notified of changes from GitHub, there is additional setup required. See [“GitHub hook trigger for Git polling”](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#github_hook_trigger_for_git_polling).)



The project URL is the main parameter to set here. There is also an Advanced button, but it simply allows you to specify a simple name for information sent back to GitHub.

Note that using this functionality requires having a Jenkins URL reachable from the internet and some specific setup. See [the GitHub plugin page](https://plugins.jenkins.io/github) for more information.

#### Specifying the GitHub project property in pipeline projects

For Scripted Pipelines, you can set the GithubProjectProperty values in the properties step. For example:

properties([[$class: 'GithubProjectProperty',

displayName: '',

projectUrlStr: 'http://github.com/brentlaster/sampleproject/'],

pipelineTriggers([])]

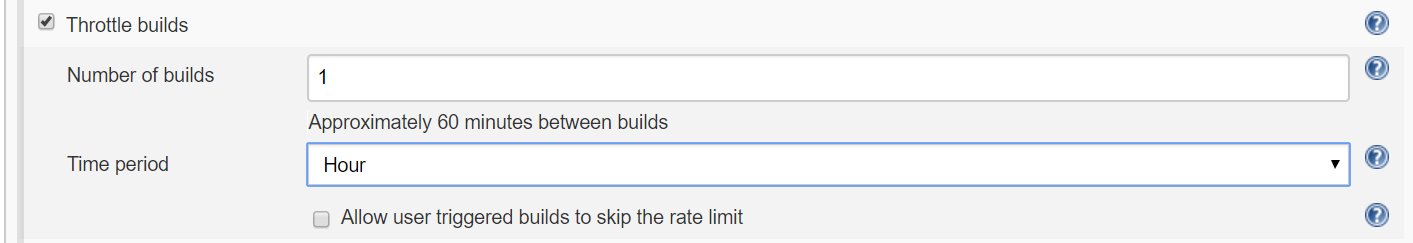
### THIS PROJECT IS PARAMETERIZED

This option allows you to add various kinds of input parameters to your job. Clicking the Add Parameter button brings up additional fields that you can fill in for the name of your parameter, default values, etc.

The different types of parameters and how to use them in pipeline projects are covered extensively in [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow).

### THROTTLE BUILDS

This option allows you to specify the number of builds to be allowed within a given time period. One field is for the number of builds and one is for the time period (hour, day, etc.).



#### Throttling builds in pipelines

The properties step does have a way to call the throttle builds functionality, but as of this writing, that functionality seems to be broken. It is also currently missing for the Declarative Pipeline options section. Here is an example of what the Snippet Generator creates for this:

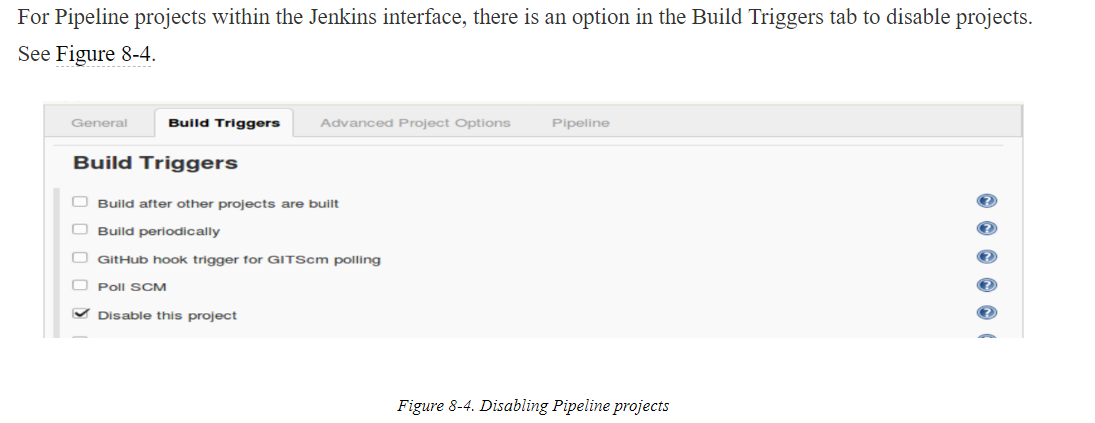
properties([[$class: 'JobPropertyImpl',

throttle: [count: 1, durationName: 'hour']], pipelineTriggers([])])

### DISABLE THIS PROJECT

As the name implies, clicking this box will disable the project (keep it from being executed). When this is unset, it will reenable the project.

#### Disabling Pipeline projects



### EXECUTE CONCURRENT BUILDS IF NECESSARY

By default, concurrent builds for the same project are not allowed. If this option is checked, and enough executors are available, then parallel builds are allowed. This can be useful for large or long-running project builds, and also for ones that are parameterized and can benefit from running with different parameters (such as for testing scenarios).

When concurrent builds happen, workspace names are appended with @# (where # is a number) to separate the workspaces. However, if a custom workspace is used, all of the concurrent builds run there.

#### Concurrent builds in pipelines

In the context of pipelines, the sense of this option is reversed. That is, we set an option to disable concurrentbuilds if desired. The [syntax](http://bit.ly/2HXNZH5) looks like this:

properties([disableConcurrentBuilds()])

or:

options { disableConcurrentBuilds() }

### RESTRICT WHERE THIS PROJECT CAN BE RUN

This option allows you to enter one or more “labels” identifying which nodes can be used to run the project. Labels are identifiers you put on nodes to make them selectable.

Selecting this option displays an additional entry box to enter the label(s).

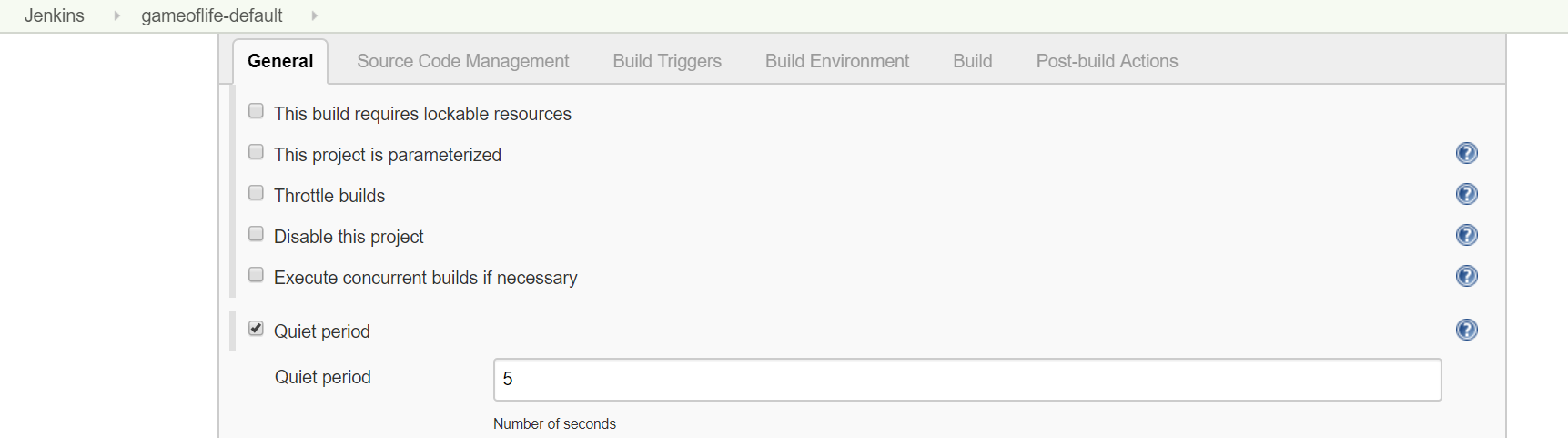
#### Pipelines and nodes

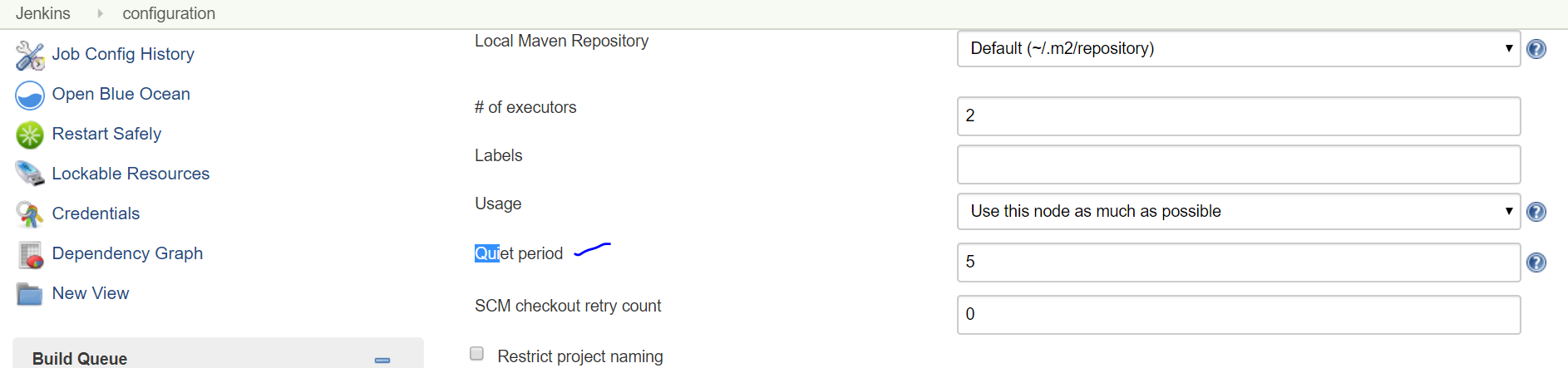
The node block and agent step (in Scripted and Declarative Pipelines, respectively) allow for deciding where all or part of a pipeline should run. This and related steps are covered in [Chapter 2](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch02.html#CH_The_Foundations) for Scripted Pipelines and [Chapter 7](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#CH_Declarative_Pipelines) for Declarative Pipelines.

A set of additional options can also be set on the General tab by clicking on the Advanced button below the “Restrict where this project can be run” option. We cover those next.

### QUIET PERIOD

Clicking on this option gives you a field where you can enter a number of seconds for Jenkins to wait before starting a build of this project. If builds are triggered, they will be added to the queue, waiting for this time period. If this is not set, the system will default to the global quiet period if one is configured in the Configure System settings (shown in [Figure 8-5](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_globop_incl_defquiet_per)).





This option is mainly a vestige from the early days of using systems like CVS where you might need to wait until all files were committed before initiating builds, rather than acting when the system saw the first one. It can still have similar applications today.

#### Pipelines and the quiet period

Pipelines have a build step where you can initiate the build of another project. From within that step, you can specify a quiet period for the intended job. The syntax is as follows:

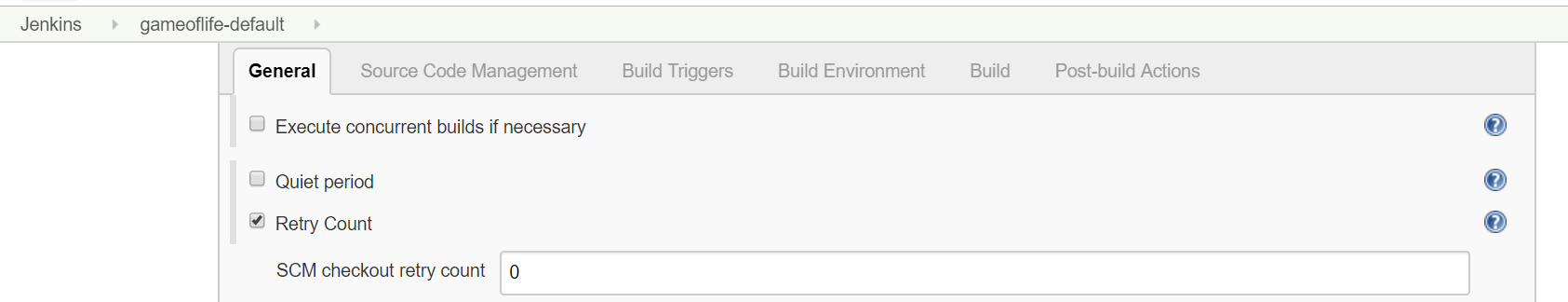
build job: 'myJob', quietPeriod: 5

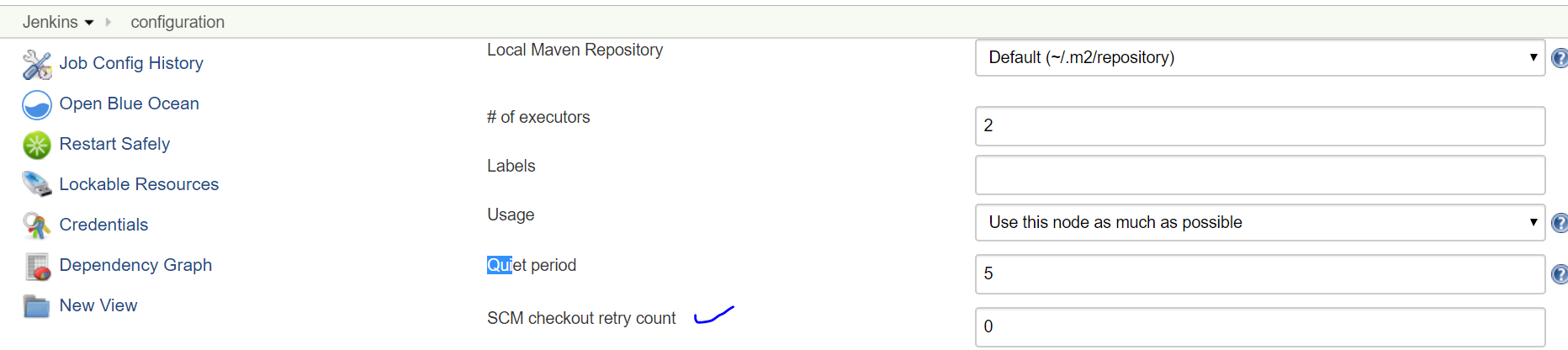
### RETRY COUNT

This setting is for retrying SCM checkouts. Clicking on the option pops up a field where you can enter the number of attempts to make to check code out. There is a 10-second delay between attempts. If this value is not set, then the system will default to the global retry value, if set, in the Configure System settings (the “SCM checkout retry count” setting in [Figure 8-5](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_globop_incl_defquiet_per)). Note that it is up to each SCM plugin provider to define what constitutes a failure that warrants a retry.

#### Pipelines and retry count

Currently, pipelines should honor the global (Configure System) retry count, if set. Pipelines also include a general retry step that can be used to retry any operation. This is discussed in detail in [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow).





### BLOCK BUILD WHEN UPSTREAM PROJECT IS BUILDING

When this option is checked, the project won’t be allowed to build if one of its dependencies (direct or transitive) is building or in the queue.

### BLOCK BUILD WHEN DOWNSTREAM PROJECT IS BUILDING

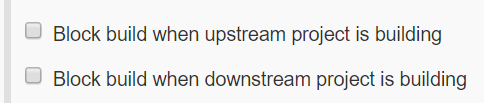
When this option is checked, the project won’t be allowed to build if one of its children (direct or transitive) is building or in the queue.

#### Waiting for downstream in pipelines

For pipelines, the build step has an option that defaults to true to wait for downstream builds. If you do not want to wait, then you need to explicitly set that value to false as shown here:

build job: 'declar2', wait: **false**

Note that if you use the default value of true, then the return value from the step is an object that you can examine for the build result and other attributes. More details can be found in the Snippet Generator help.



### USE CUSTOM WORKSPACE

As the name implies, selecting this option allows you to specify a particular directory as your workspace. (The location is entered in a separate field that opens up when this option is checked.) The location can be an absolute path or a relative path. If it is a relative path, it is relative to the node’s root directory.

Normally, it’s best (and easiest) to just let Jenkins manage the workspace. However, if the job requires builds or source downloads to be done in a specific location, this is a way you can accommodate that need.



### DISPLAY NAME

### DISPLAY NAME

The value put in this field is displayed in the Jenkins web interface as the name of the project. Duplicate names are allowed since this is just for display purposes. You could use this, for example, to display additional information about the project that is worth being easily seen.



#### Display name and pipelines

To set a display name and description in a pipeline, you can use code like the following:

currentBuild.displayName = <project name>

currentBuild.description = <project description>

### KEEP THE BUILD LOGS OF DEPENDENCIES

This option overrides log rotation policies for dependencies connected to your project. It is useful for ensuring those logs are still available to coincide with your project’s logs.



Next is the Source Code Management section.

## **Source Code Management**

Depending on which source code management plugins you have installed, you’ll have options here to select one and configure it appropriately. The specific options will vary depending on the system selected, but there are certain common features.

### REPOSITORY URL

This setting specifies the location of the repository you want to access for the project. Note that different protocols can be used, such as HTTPS or SSH.

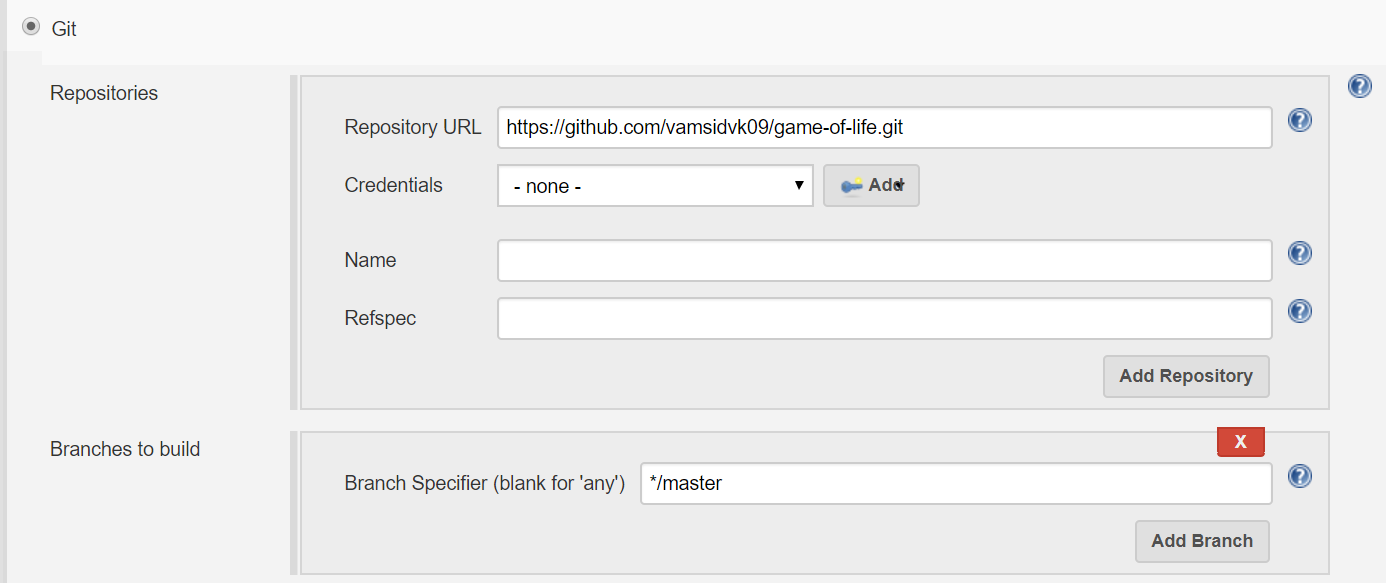
### CREDENTIALS

These are simply the credentials you’ve defined in Jenkins to access the SCM.

### REVISION

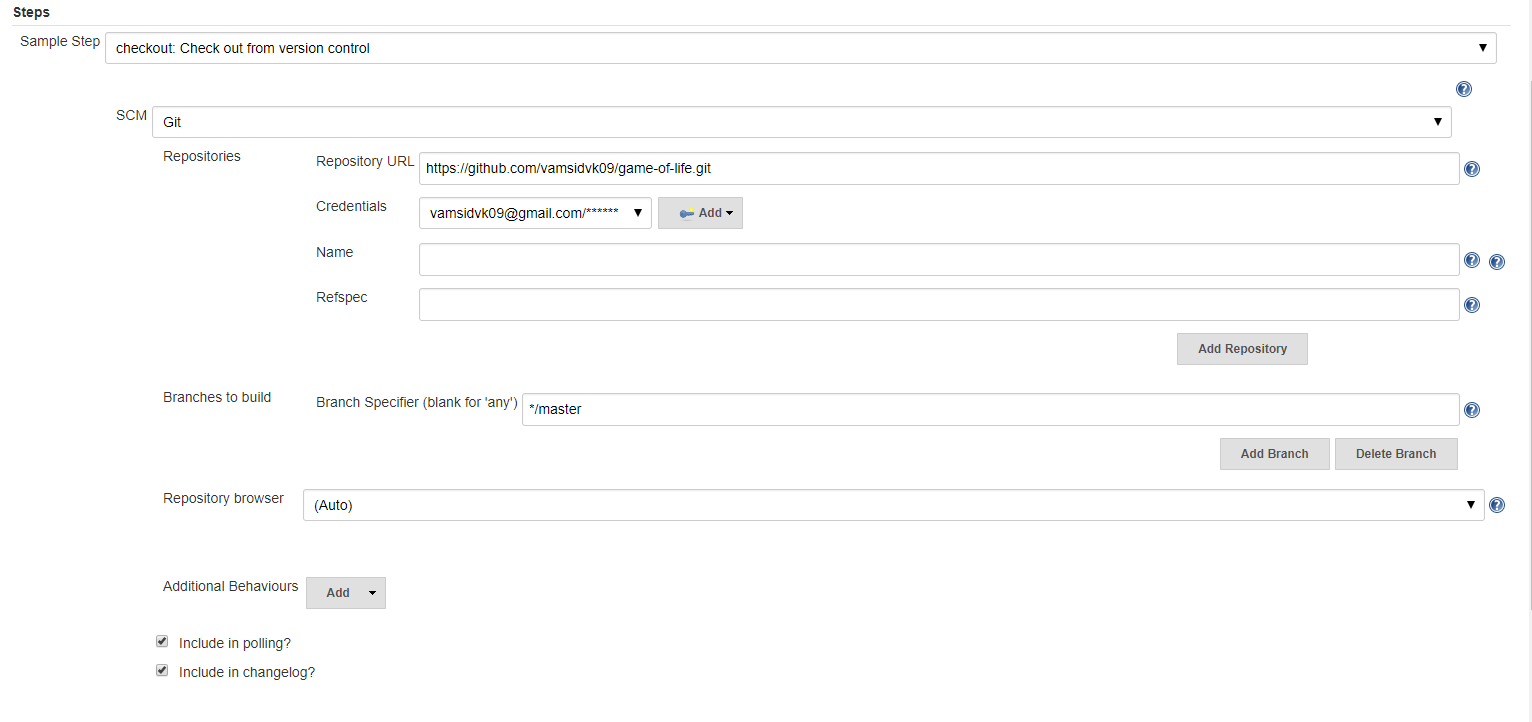
Specifying a revision is a way to specify a particular version of the code that you want to use (typically this might be a branch, but it could also be something like a tag or whatever the SCM uses to indicate a specific version)

Below shows a setup for accessing a Git repository.



### SOURCE CODE MANAGEMENT IN A PIPELINE

The pipeline includes a corresponding checkout step that you can use in place of the source code management forms in this section. The easiest way to fill this in is via the Pipeline Syntax/Snippet Generator form. [Figure 8-7](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_conf_checkout_pip_GSCM)shows an example of using the Snippet Generator to duplicate the setup in the previous section.



checkout([$class: 'GitSCM', branches: [[name: '\*/master']], doGenerateSubmoduleConfigurations: false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[credentialsId: '4c52307a-d746-4add-8a17-23e1daef437c', url: 'https://github.com/vamsidvk09/game-of-life.git']]]) 



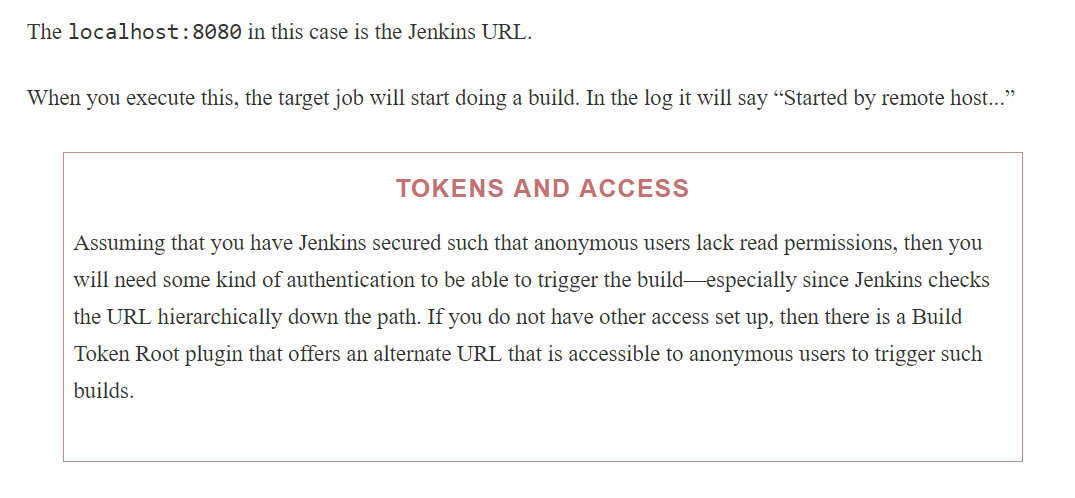
## Build Triggers

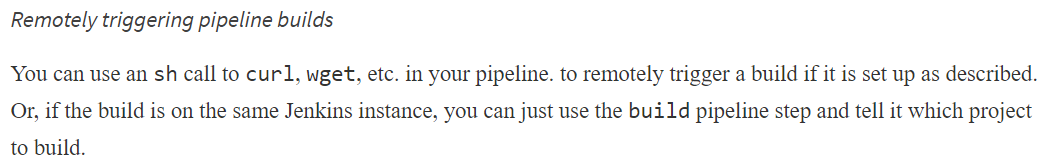
n this section of the project configuration, we define the events and/or processes that will start a build of our project running. The basic options are described in the following sections.

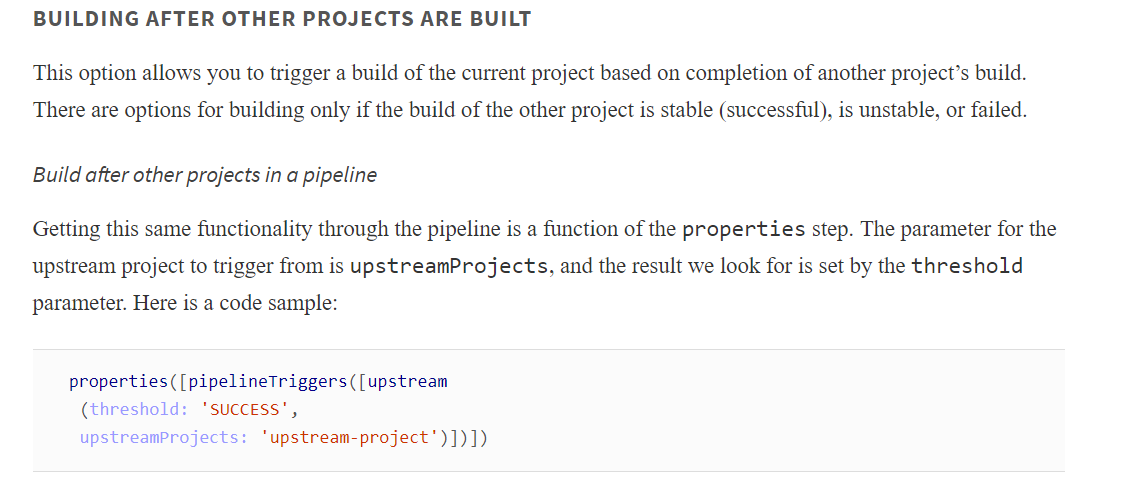
### TRIGGER BUILDS REMOTELY

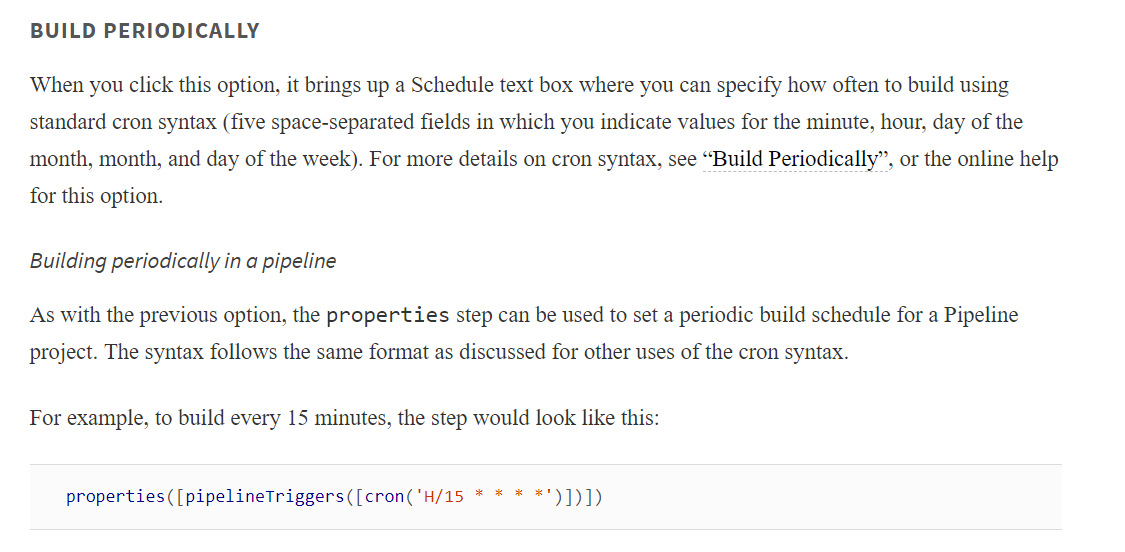
If you select this option, then Jenkins will provide you with a special URL that you can use to trigger a build (see [Figure 8-8](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_setup_remote_trigger)). Jenkins also asks you to provide a string that can be used as an authorization token in the URL. This is an additional security step since anyone or anything trying to trigger the build needs to also know the token.









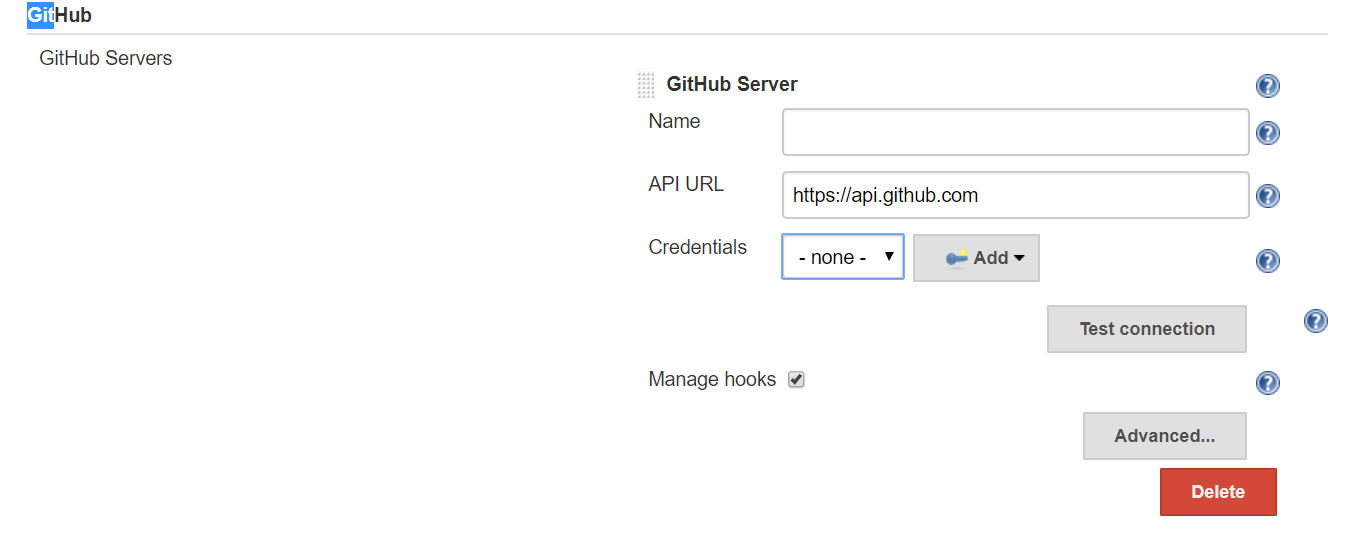


### GITHUB HOOK TRIGGER FOR GIT POLLING

This method of triggering builds allows you to set up a GitHub service to send notifications to Jenkins when an event happens in your repository on GitHub. So, rather than polling for changes in the repository, Jenkins is notified of updates by GitHub.

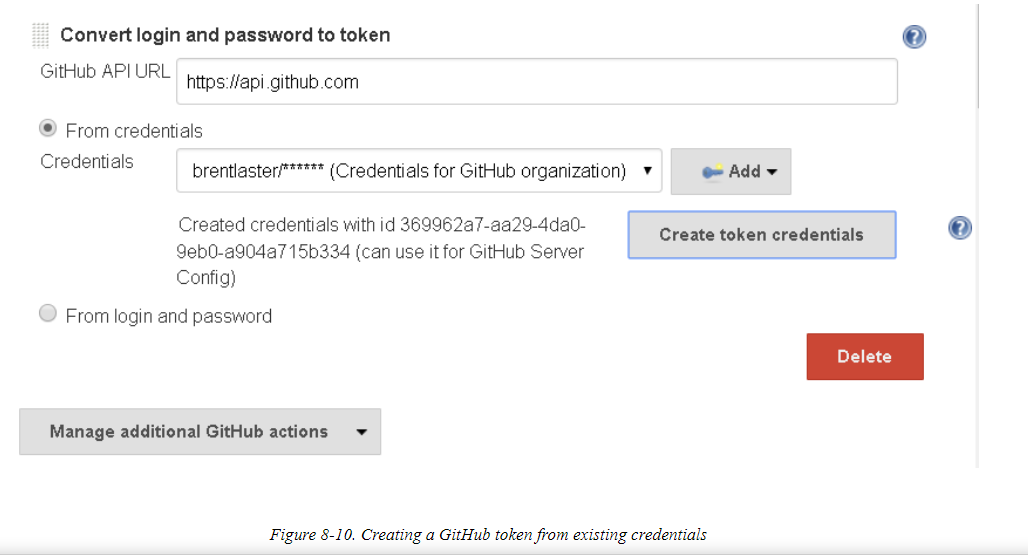
To use this, first you’ll need to have the GitHub Integration plugin installed. Then you need to do some global configuration for GitHub access.

On the Configure System screen, you will have a GitHub setup area ([Figure 8-9](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_basic_github_setconfig)). Initially there are two fields that need setup: the URL and credentials. If you just need to use the public GitHub, you can leave the API URL field as the default: https://api.github.com. If you had an enterprise GitHub system, you would put the URL for that in here instead. Likewise, the Name field can be left blank unless you have multiple GitHub enterprise systems, and need to easily differentiate one from the others.



For credentials, you need some kind of token to use with GitHub. One common choice is a personal access token. This can be set up in your personal settings area on GitHub, then added as a credential in Jenkins and selected here.

Alternatively, if you have a user ID and password that you use already with GitHub, you can let Jenkins automatically create a token for you. To do this, look for an the Advanced button further down in the GitHub section on this screen. Click that, then click Manage Additional GitHub Actions and “Convert login and password to token.” From here, you can convert an existing user ID and password credential to a token (if you already have that set up in Jenkins) or just a standard user ID and password.



This will then show up in your list of credentials in Jenkins and can be selected to use with GitHub



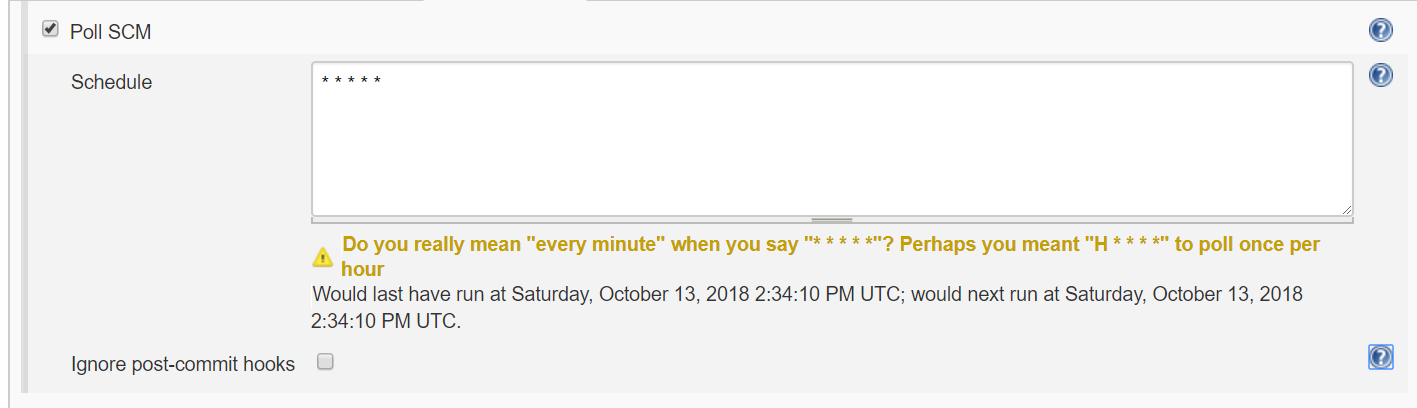
Now you want to tell Jenkins how to manage the notifications from GitHub. The notifications are sent as webhooks. There are two modes, referred to as “automatic” and “manual.”

Read the Book and in the internet for better understanding again

### POLL SCM

his option is like the “Build periodically” option discussed earlier. In fact, it uses the same cron-like syntax as that option. The difference is that instead of telling Jenkins when to start a build, we are telling it when to check the repository for changes. See [“Cron syntax”](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#SEC_cron_syntax) for details on the syntax.

This choice has an additional option to “Ignore post-commit hooks.” Basically, this tells Jenkins to not start activities based on signals from hooks after changes are made, but only to respond to changes in the SCM. This prevents double-triggering operations.



#### **Polling in the pipeline**

Once more, we use the properties step and cron specification for this option. Here’s the syntax for telling the pipeline to check every 15 minutes for changes in the repository, along with the option to ignore the post-commit hooks:

properties([pipelineTriggers

([pollSCM(ignorePostCommitHooks: **true**, scmpoll\_spec:

'H/15 \* \* \* \*')])])

Up next is the Build Environment section

## Build Environment

### This section allows you to specify certain global actions and integration settings for the project. There can be many of these, depending on which plugins you have installed. (For example, if you have the Artifactory plugin installed, you’ll have Artifactory integration items.) We’ll cover some common ones here. DELETE WORKSPACE BEFORE BUILD STARTS

This one is pretty self-explanatory. The workspace is removed before the build begins.

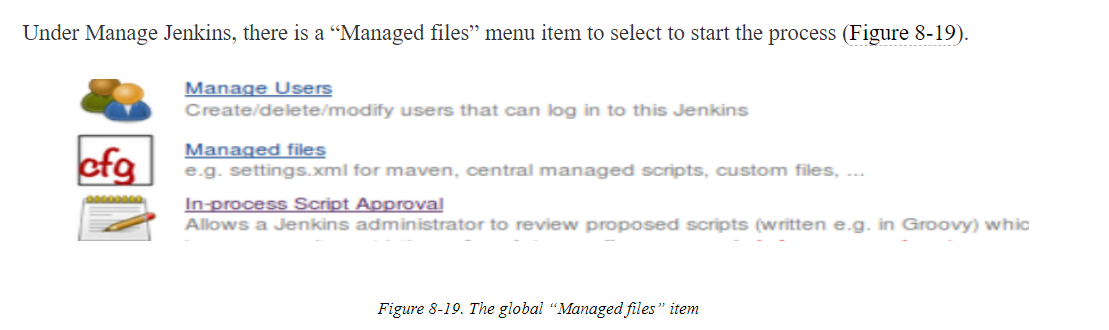
#### **Deleting workspaces in a pipeline**

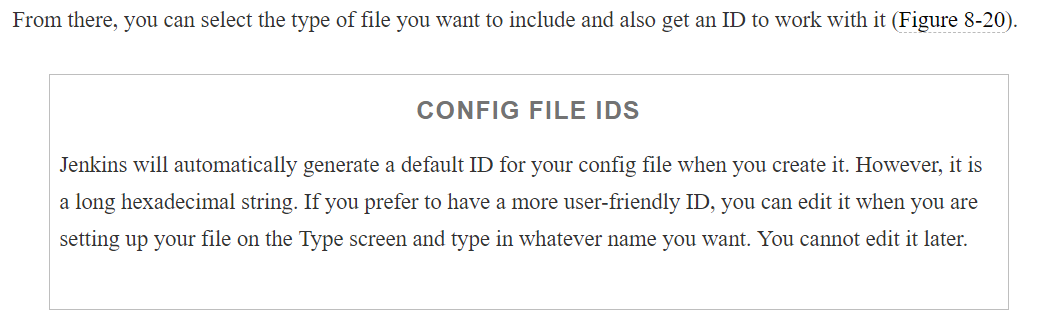
The Jenkins pipeline DSL provides the deleteDir step to clean a directory out of the workspace and also the cleanWs (clean workspace) step to delete a workspace. These steps are covered in more detail in [Chapter 11](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch11.html#CH_Integration_with_the_OS).

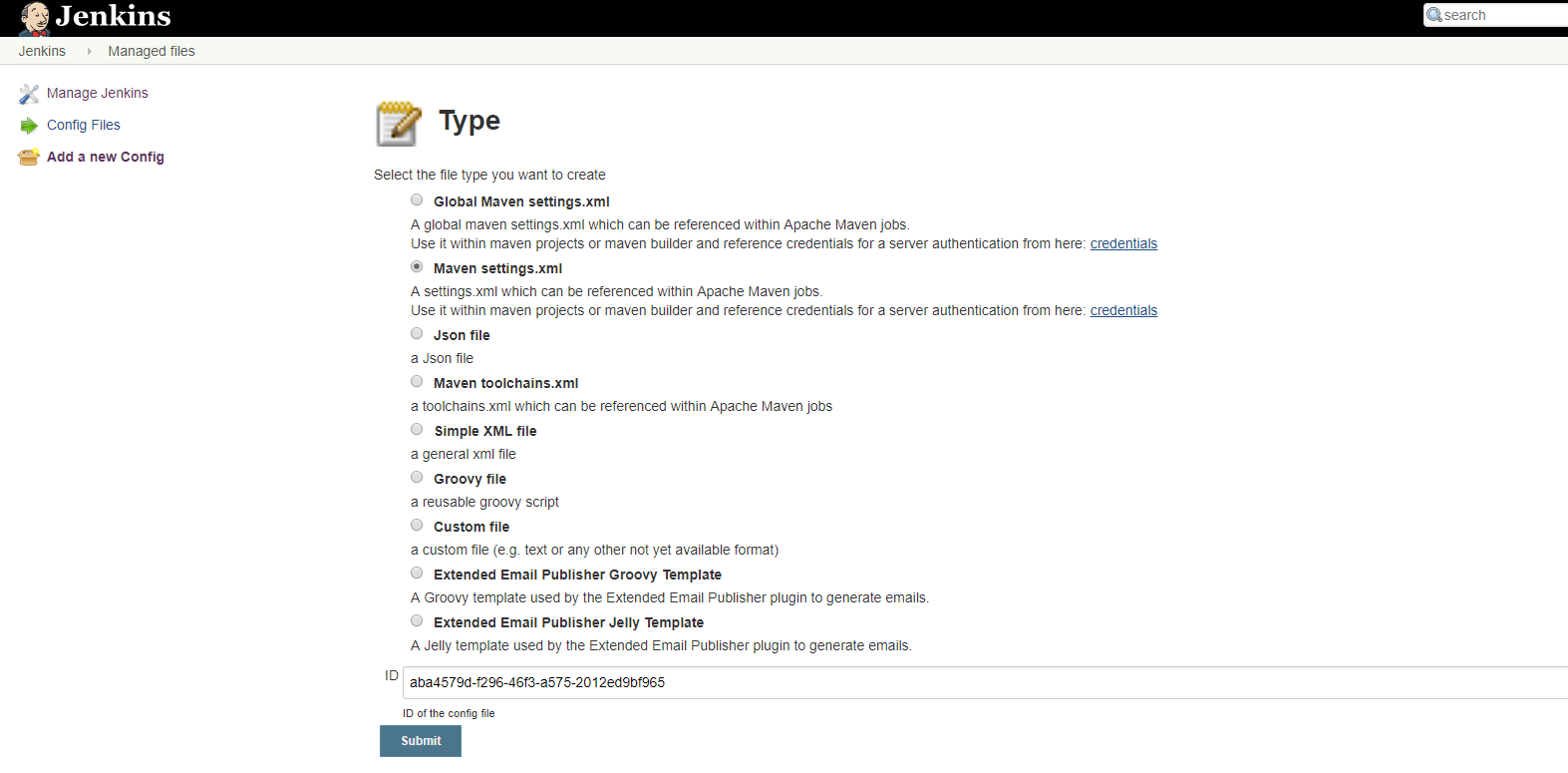
### PROVIDE CONFIGURATION FILES

This option allows you to select files of a certain type and copy them to all your nodes, as well as providing a way to edit them through the Jenkins UI. Some global setup is required first.

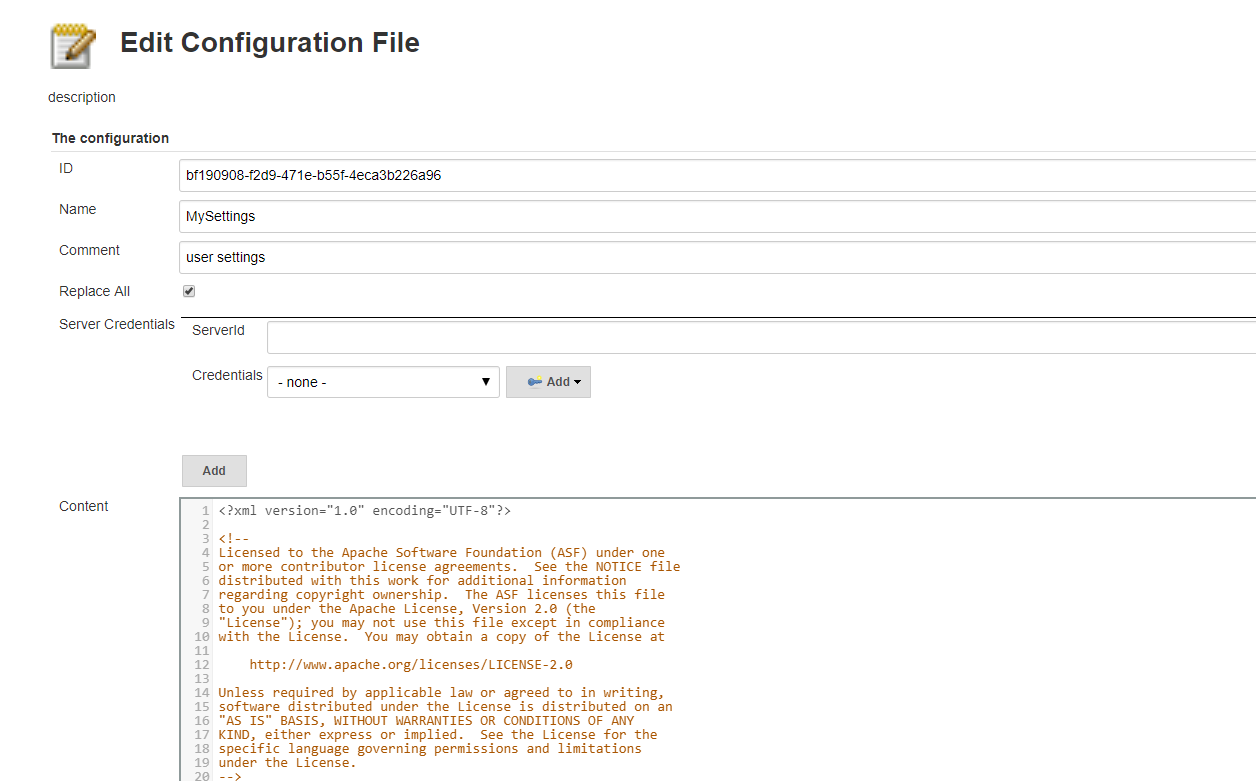




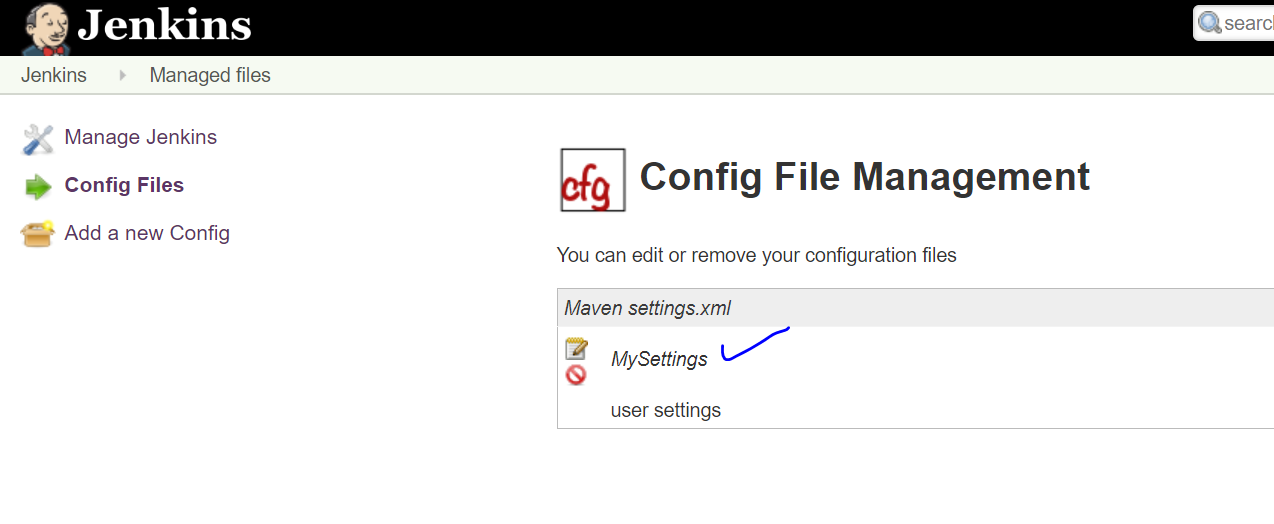




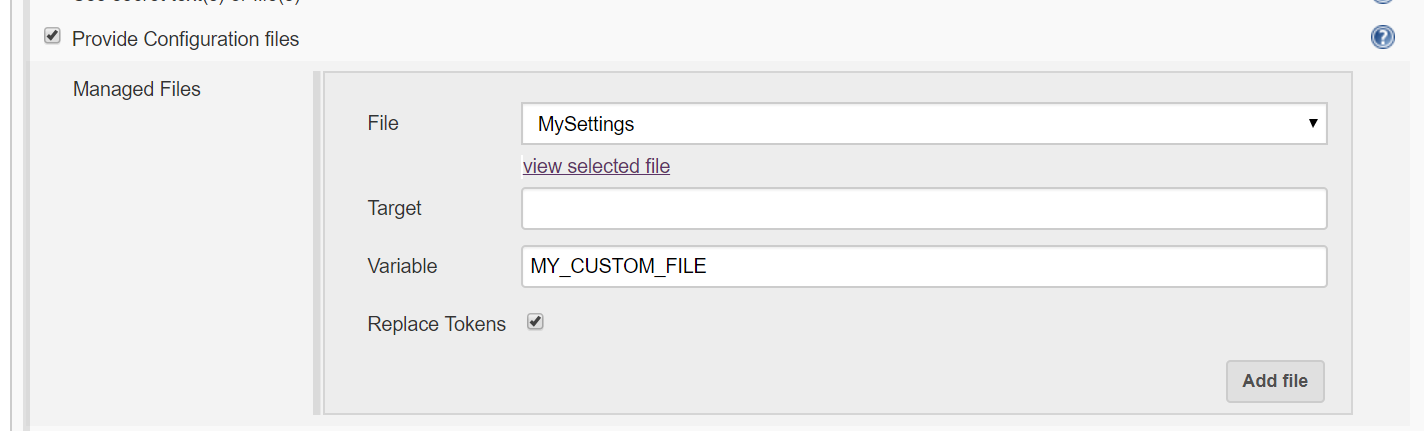
After you click Submit here, you move on to filling in the file’s actual content



Once you are done with that, you’ll have a screen where you can edit or delete your new file



Now that you’ve completed the global setup for this, you can use the file in your project. After selecting the “Provide Configuration files” option in the project, you’re presented with a dialog like the one in



The File field allows you to select a file that you have previously configured globally as we just described.

The Target field allows you to specify where the file should be created on a node. If this is left blank, then the file will be created in a temporary location.

The Variable field allows you to define an environment variable name to reference the file in job steps. This also gives you a handle to get to the file in the temporary location if the Target field was left blank.

Finally, the Replace Tokens option replaces environment variables set by Jenkins and specified in yourconfiguration file with their values. (This relies on the Token Macro plugin.) The syntax to use in your config files for token replacement is:

${ENV, var="<variable-name>"}

where <variable-name> is replaced with the name of the variable you want to get the value of (such as JOB\_NAME).

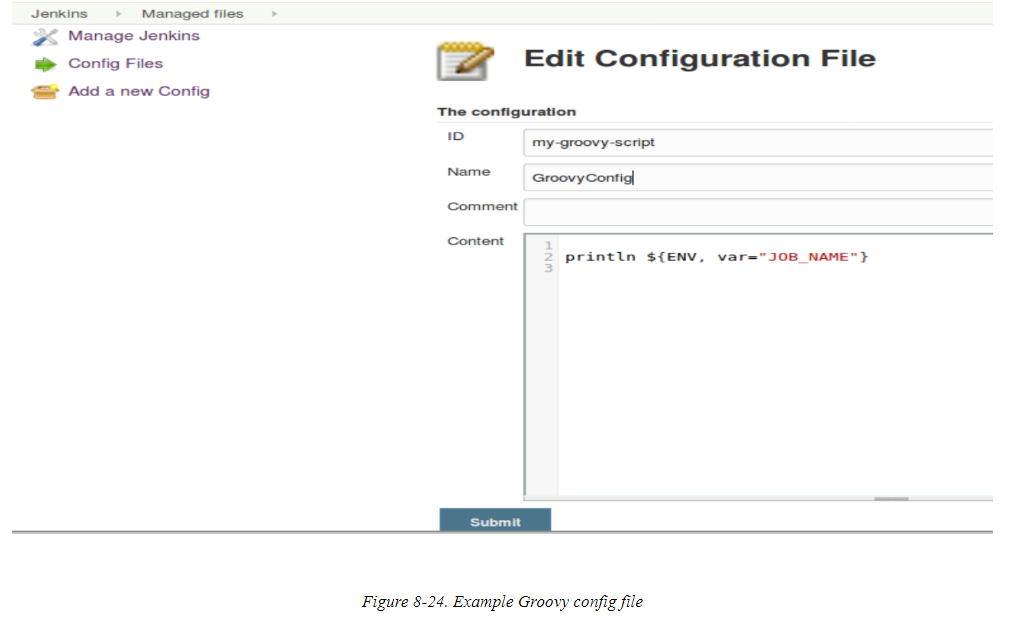
<https://stackoverflow.com/questions/25277866/maven-command-line-how-to-point-to-a-specific-settings-xml-for-a-single-command>

For sample example In our project for the tps Jenkins instance has some issue with the trs Jenkins global maven settings.xml . So, We can create a settings.xml and pass the configuration file as the parameter to the mvn command as in the link

#### Managing configuration files in a pipeline

There is a configFileProvider step that you can use in your pipeline code. This is another block step, meaning that you invoke the step with some context, and it provides a closure in which you execute other code. So, for example, if you were using this functionality to access a properties file or custom XML file, you would first configure the file globally (as described previously). Then, in your pipeline code, you would invoke the step, passing in the file ID and related information. Then, within the step block, you could invoke other pipeline commands that use the properties file, the XML file, or their data.

For example, suppose we have a config file set up as shown in [Figure 8-24](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_sample_groovy_config).



We could then add a step in our pipeline like the following to use it:

 configFileProvider(

        [configFile(fileId: 'my-groovy-script',

        variable: 'MY\_GROOVY\_SCRIPT',

        replaceTokens:**true**)]) {

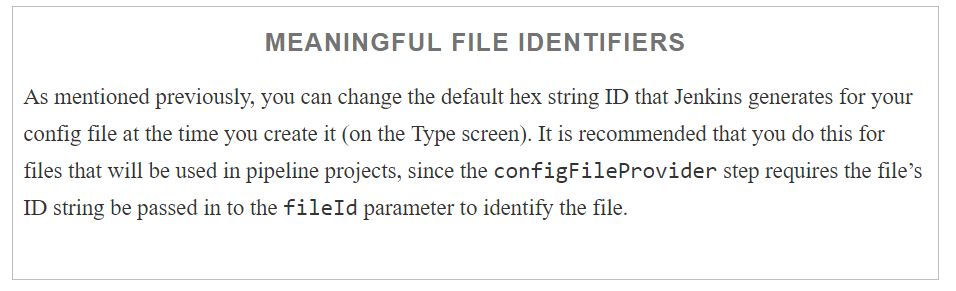
             sh "cat ${MY\_GROOVY\_SCRIPT}"

        }

Notice the syntax here. We first have the configFile parameter, which takes arguments that correspond to our Freestyle ones:

fileId

This is the file ID that was set up when you globally configured the file.



variable

This is a variable you can use to access the file itself on the node.

replaceTokens

If set to true, this tells Jenkins to replace known environment variables with their actual values in the configuration file. (See the preceding section on Freestyle usage for syntax.)

Running this step in the pipeline would yield results like the following:



Note that when we ran the shell cat command in the block, we used the variable we defined in the step’s invocation. And when the contents of the file were printed out, because we have the replaceTokens value set to true, the environment variable string inside the configuration file was replaced with the value of the environment variable in the output—in this case, the job name.

One other point about this step is that multiple config files can be specified in the step using an array syntax, as shown in the following example:

configFileProvider(

        [configFile(fileId: 'my-custom-file',

        variable: 'MY\_CUSTOM\_FILE',

        replaceTokens:**true**),

        configFile(fileId: 'my-groovy-script',

        variable: 'MY\_GROOVY\_SCRIPT',

        replaceTokens:**true**)]) {

             sh "cat ${MY\_GROOVY\_SCRIPT}"

        }

### ABORT THE BUILD IF IT’S STUCK

This setting allows you to specify a timeout strategy and related values to stop the build if it appears to be taking too long. The main parameters are a timeout value in minutes and the choice of strategy to use for determining when a build is stuck.

As defined in the help for the setting, the following strategies are available:

Absolute

Abort the build based on a fixed timeout.

Deadline

Abort the build based on a deadline time specified in HH:MM:SS or HH:MM (24-hour time) format.

Elastic

Define the time to wait before killing the build as a percentage of the mean of the duration of the last *n*successful builds.

Likely stuck

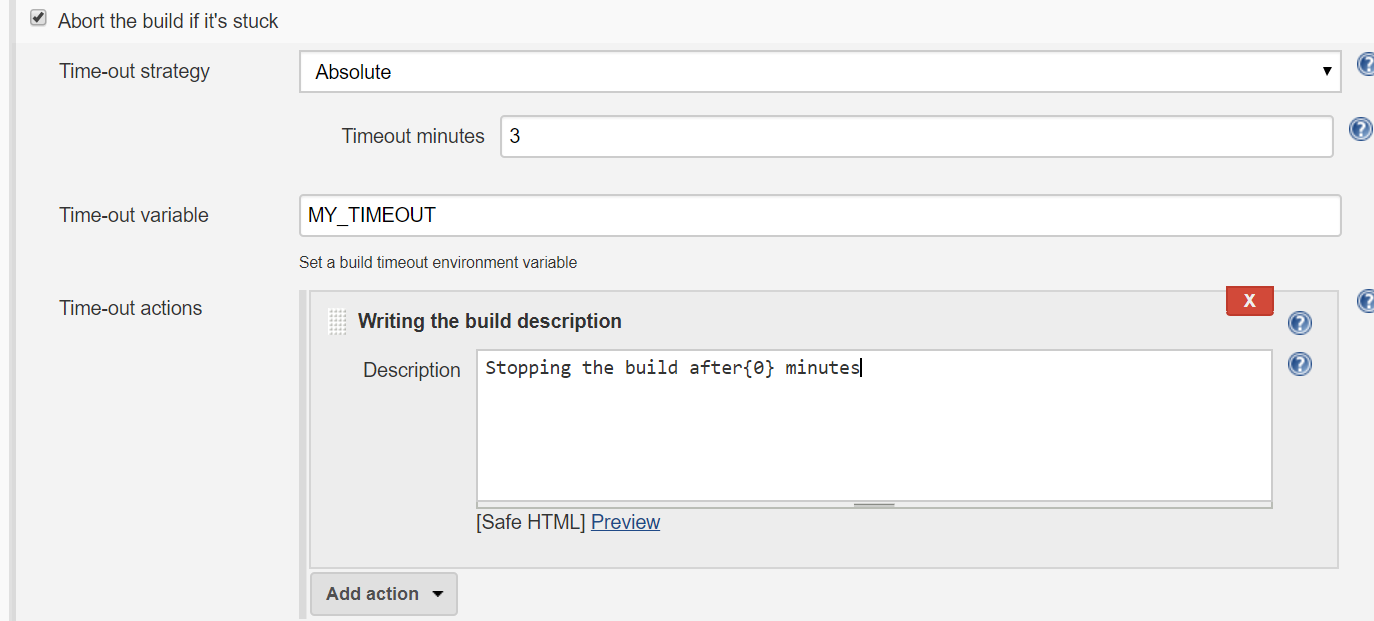
Abort the build when the job has taken many times longer than previous runs.

No Activity

Trigger a timeout when the specified number of seconds have passed since the last log output.

Additionally, we can define an environment variable that is automatically filled in with the timeout value (in milliseconds) and can be referenced in our jobs. And finally, we can define what actions Jenkins should take when the timeout is hit. Options include failing the build, aborting the build, and writing information to the run’s description. For the information that goes in the description, the special value “{0}” will be filled in with the timeout in minutes.

As an example, suppose we have a job where the properties are configured like in



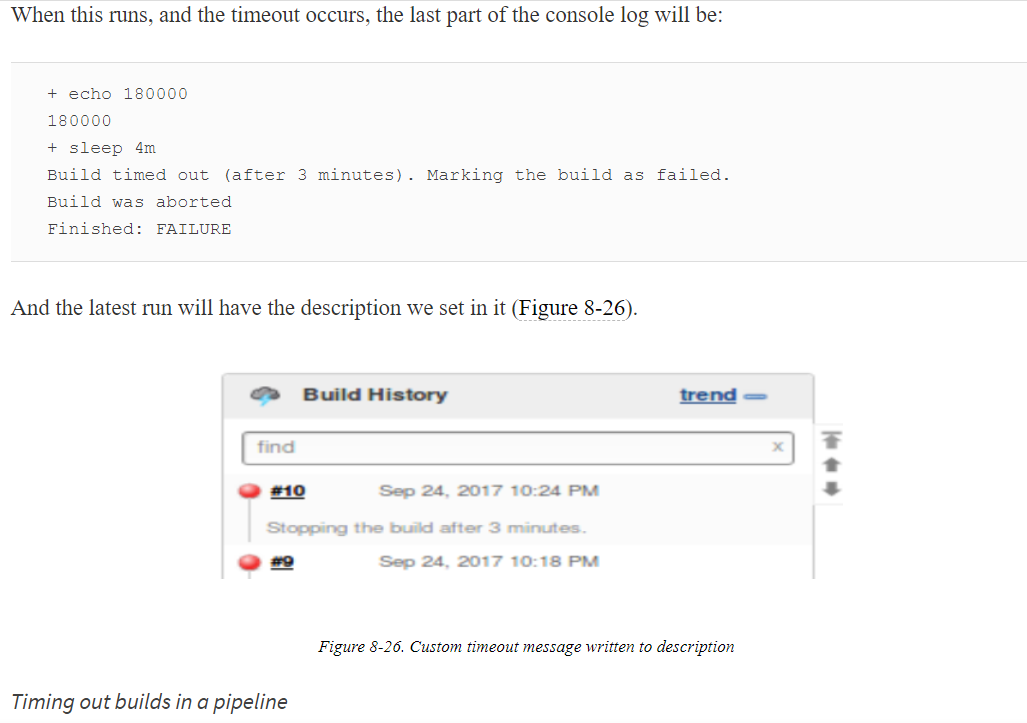
Here we are telling Jenkins to do an absolute timeout after the job has been running for three minutes. We’ve defined an environment variable named MY\_TIMEOUT that we can reference in our job, and we’ve added some actions to happen after the timeout occurs. We will be writing the string “Stopping the build after {0} minutes” (using the special variable) and then failing the build.

An extremely simple job to test this could be a shell command that executes something like:

echo $MY\_TIMEOUT

sleep 4m

When this runs, and the timeout occurs, the last part of the console log will be:



The pipeline DSL has a simple timeout step that provides similar functionality. This step is a block step, meaning it wraps around a set of code. It takes a default parameter of a number of minutes to wait for the code in the block to time out. If you want to use a different unit than minutes, you need to specify that as an additional parameter. A simple example is shown here (see [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow) for more explanation and related examples):

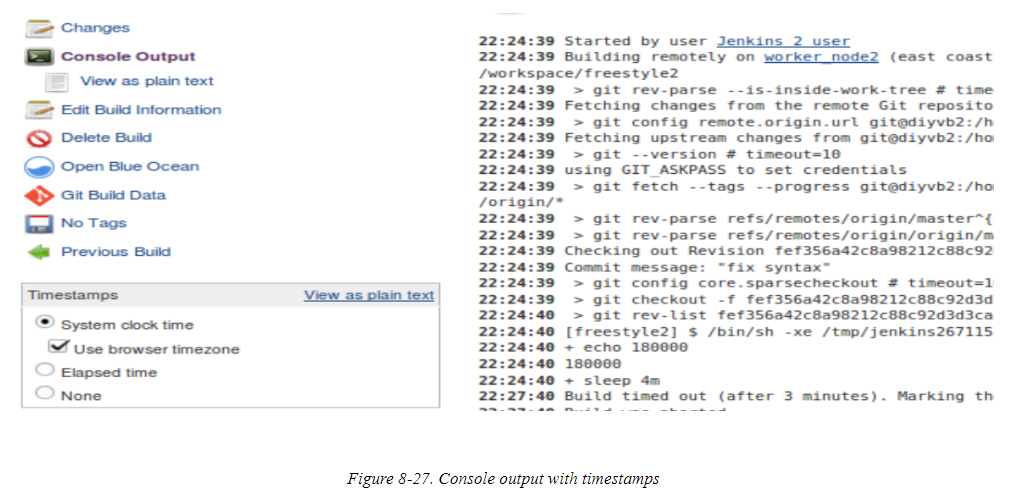
timeout(time: 1, unit: 'HOURS') {

*// block of code*

}

### ADD TIMESTAMPS TO CONSOLE OUTPUT

As the name implies, this setting will print timestamps in the console log as parts of your job are executed. An example of the default output is shown in [Figure 8-27](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_console_output_timed).



Note that this option also adds a dialog on the console log screen with controls that allow you to modify the timestamps to show elapsed time (instead of clock time) or even turn off displaying the timestamps.

#### Adding timestamps to a pipeline

The timestamps step in the pipeline provides similar functionality. This is another block step that wraps around a block of code and generes timestamps in the console output for that block. The syntax is straightforward:

timestamps {

*// block of code*

}

### OTHER BUILD ENVIRONMENT OPTIONS

Depending on what other plugins you have installed in Jenkins and what other applications you are running on your system, you may have more environment options here. For example, if you are using the Artifactory plugin, you may have an option here to configure Artifactory integration with Ant, Gradle, or Maven.

Because of the number of possibilities we won’t try to cover all of them here, but you can generally find out the details you need by clicking on the help buttons next to the options and/or going to the plugin’s web page.

As far as corresponding pipeline steps, many are covered in related chapters of this book.

## **Build**

The Build section of the configuration is where the main logic for your job goes. For many of the traditional job types that Jenkins supports, this is where the projects most extensively differentiate from one another—from the wide-open Freestyle project to the more specialized ones like Maven and Ivy. Depending on the type of project and the set of plugins and other applications you are using, you may have many different choices on this page; rather than attempting to detail all of them here, we will cover the most significant parts of each respective project type in later sections. For remaining items not covered in the project-specific sections in the chapter, refer to the help associated with each step (available via the blue help buttons as well as the plugins’ web pages).

For corresponding pipeline functionality, see the other chapters of this book.

## **Post-Build Actions**

The final configuration section allows us to select specific post-build actions for a job. These are actions that are always run after the build finishes—in some cases whether successfully or not.

Again, there are too many options based on plugins and integrations to cover here. See the help for a particular option or the plugin’s web page to find out more about a particular available action.

### POST-BUILD ACTIONS IN A PIPELINE

Post-build actions are not built in for Scripted Pipelines. [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow) describes how you can use the try-catchJava/Groovy construct to create a workflow with similar actions.

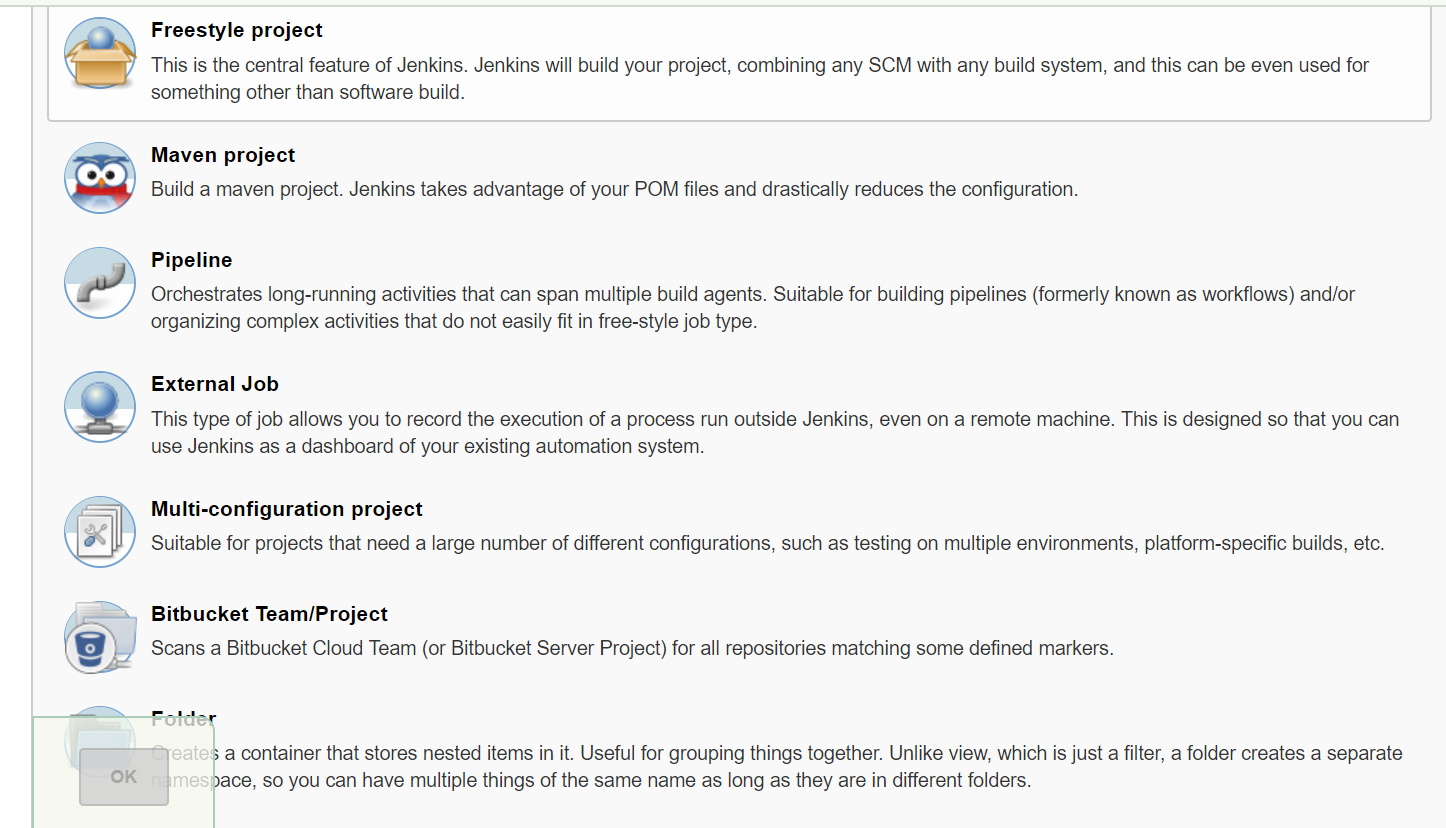
For Declarative Pipelines, there is a specific post section that can be put in the pipeline to provide this functionality. See [Chapter 7](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch07.html#CH_Declarative_Pipelines) for more information.

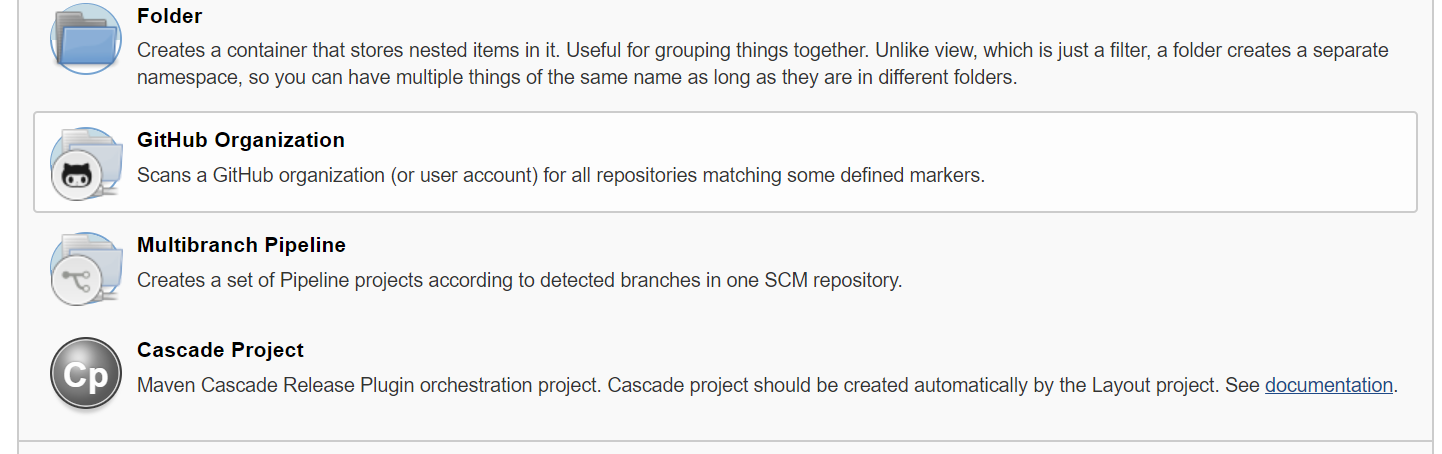
# Types of Projects

Now that we have a basic understanding of the sections and options that are common to many of the Jenkins projects, we’ll look at how those projects differ from one another.

The differentiation between project types can be based on one or multiple criteria, including:

* Open configuration to do any task: Freestyle, Pipeline projects
* Specialization for an application: Maven, Ivy projects
* Specialization for an advanced or challenging use case: Multiconfiguration, External Job projects
* Organizational purposes: Folder, Multibranch Pipeline, GitHub Organization, Bitbucket Team/Project projects
* Automated configuration and building: Multibranch Pipeline, GitHub Organization, Bitbucket Team/Project projects





## **Freestyle Projects**

Freestyle projects are the traditional working base for most Jenkins jobs. The name “Freestyle” refers to the relatively open way that these projects can be constructed to do many different tasks. Prior to Pipeline projects, Freestyle projects were considered the most flexible. They were also considered the easiest to set up, at least for individual projects.

As pointed out at the start of the chapter, for the traditional Jenkins project types, what differentiated them mostly was the Build section. For Freestyle projects, probably the most common item in the Build section is a call to the shell. The Build section provides options to execute a shell call as well as a Windows batch command.

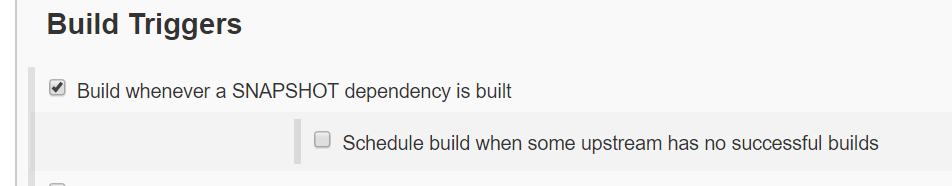
These steps are pretty straightforward; just type the command into the dialog box after selecting the type of shell step you want.

## The Maven Project Type

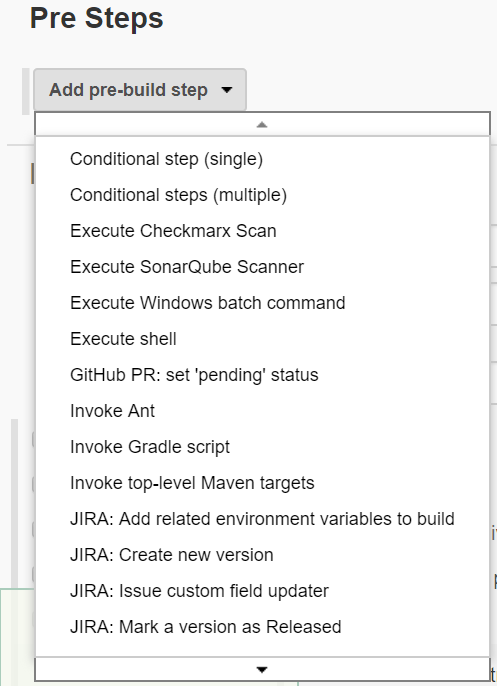
In addition to the Freestyle project type, Jenkins also offers some project types customized for different applications. Probably the best-known legacy one is the Maven project type.

The Maven project type is intended to simplify some common tasks, such as triggering downstream dependencyjobs, deploying artifacts to a Maven repo, optionally rebuilding only changed modules, and breaking out test results by module.

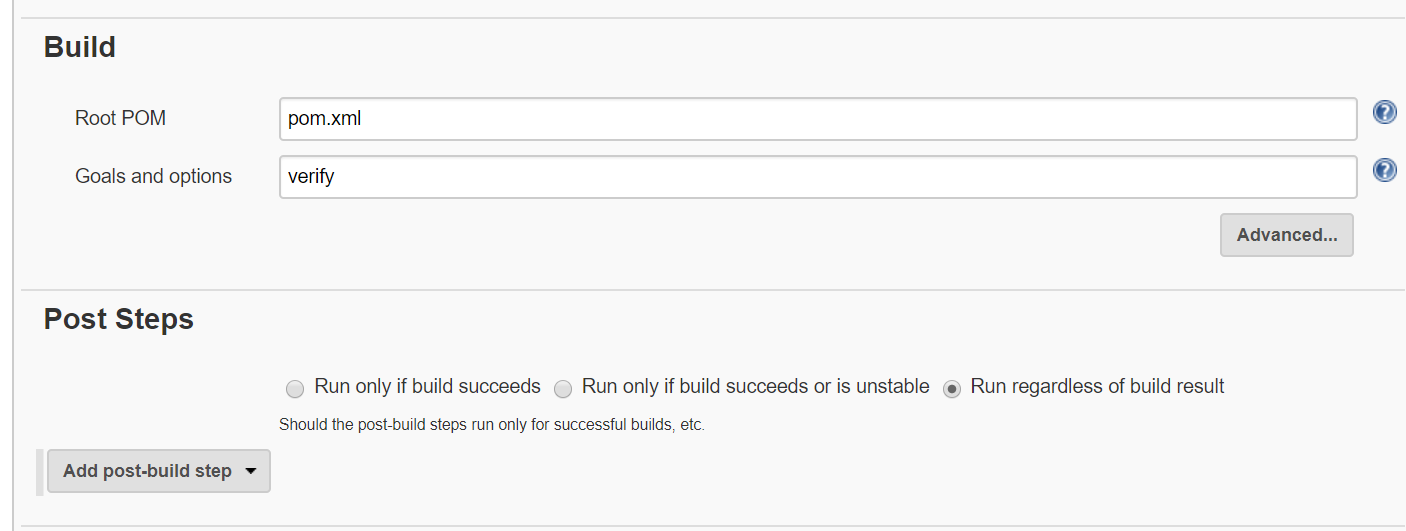
This type of project has a few additional options, such as a build trigger that you can set up to have the projectbuild if dependencies are built on the same system



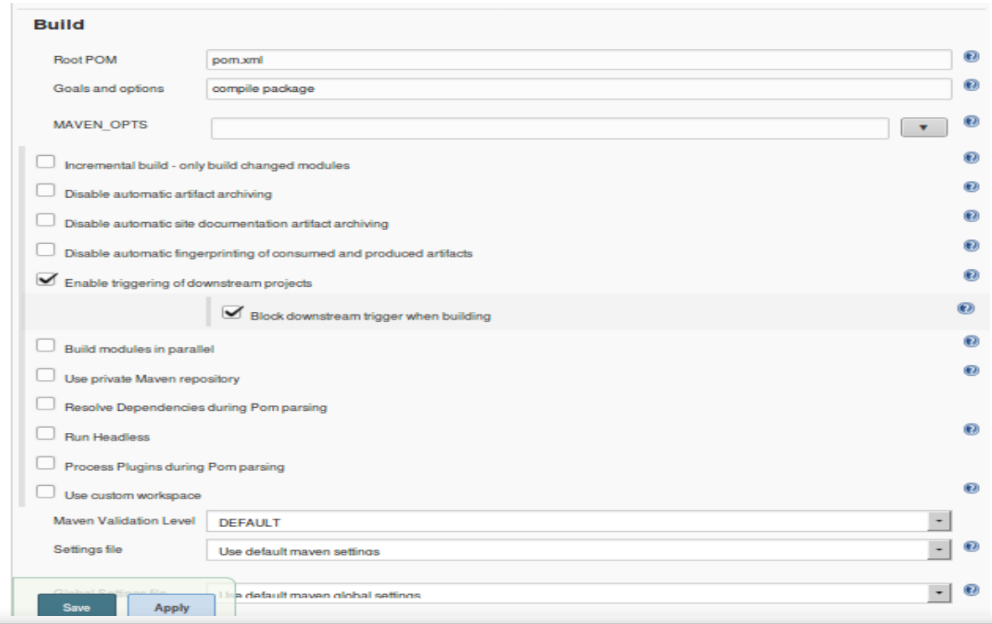
Some traditional non-Maven build steps are moved to sections named Pre Steps ([Figure 8-30](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_maven_proj_presteps)) and Post Steps. (The same set of steps appears in both.)



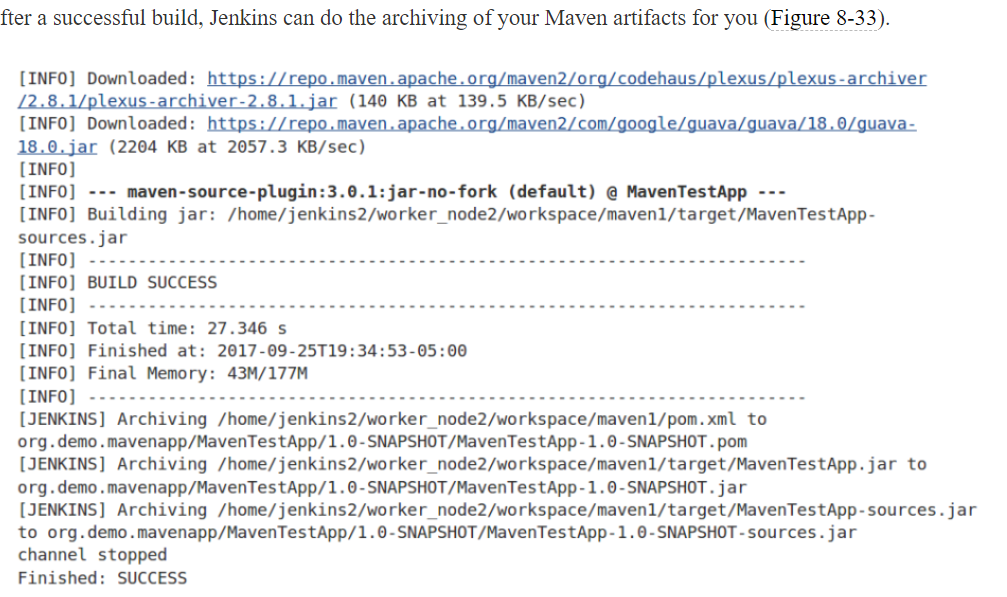
these Pre and Post Steps sections “sandwich” the main build area, where you can enter the root POM filename (if different from “pom.xml”), the Maven goals to build, and any Maven options (via the Advanced button).



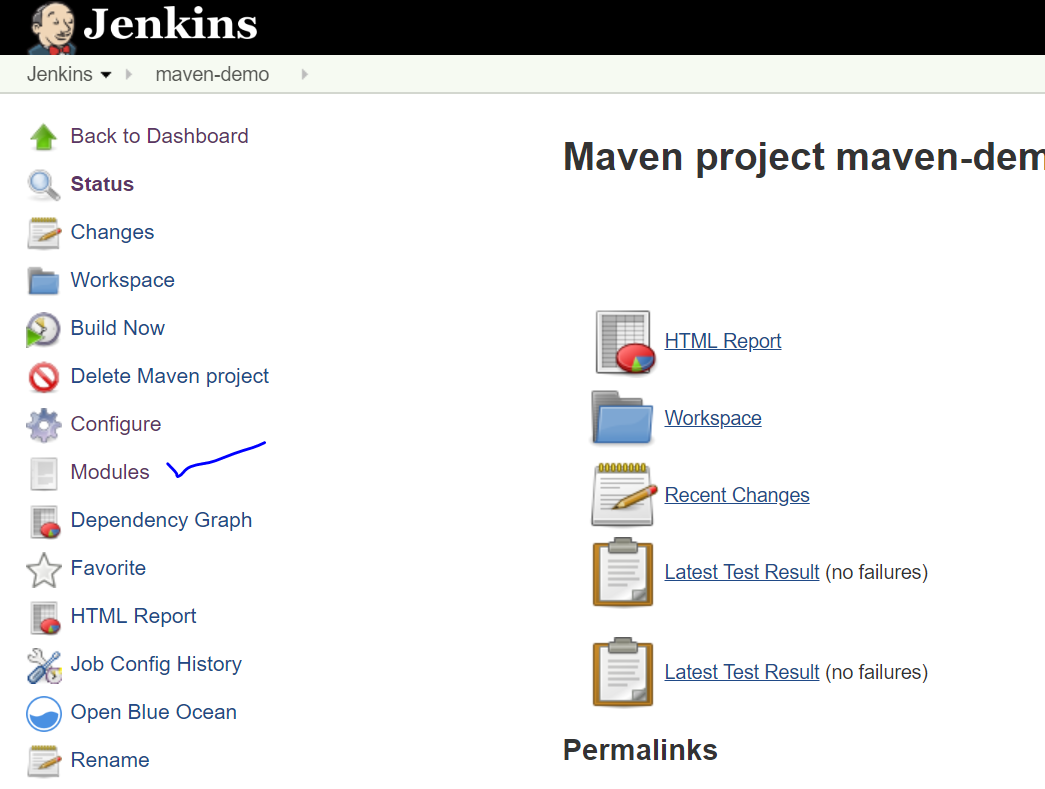
Clicking the Advanced button reveals a number of other options you can set for your build

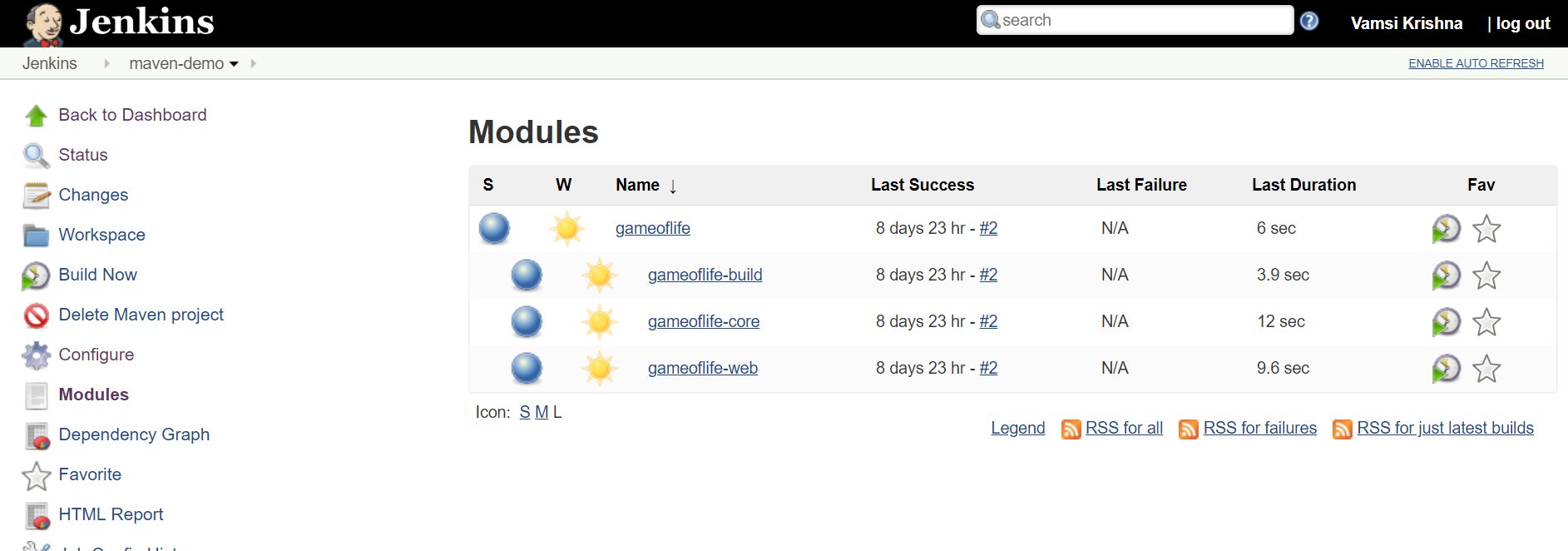


After a successful build, Jenkins can do the archiving of your Maven artifacts for you



And within the job output, you can easily get to the artifacts and even redeploy them if needed: simply click on the “modules” item in the lefthand menu on the build status page and drill down to get to the various artifacts/modules





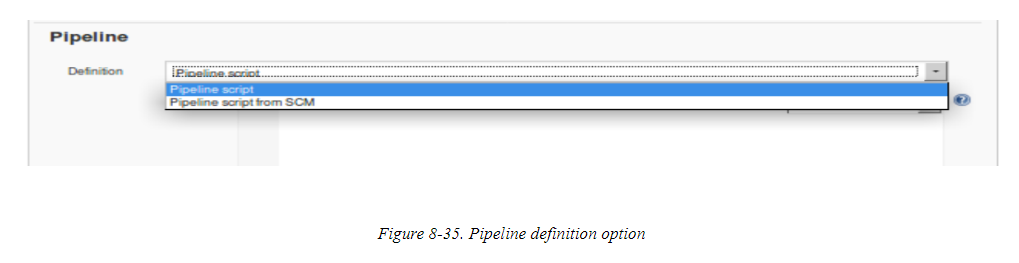
## The Pipeline Project Type

Pipeline projects are the main focus of this book, so we won’t go into too much detail on them here. The simple way to define a Pipeline project is as a Jenkins project type where the steps and logic are specified in a structured Groovy script instead of in a web form. That script can be structured in a declarative or scripted form. It can also be entered as part of a Jenkins Pipeline project or stored externally in a file named Jenkinsfile.

Since our focus here is on the configuration aspects of the various project types, it is worth briefly calling out some of the ways that the Pipeline project configuration overlaps with the actual pipeline scripts themselves.

On the Pipeline project configuration page, the area where you can type in the pipeline script is located in a dedicated tab/section named Pipeline (just like the General, Build Triggers, and other tabs/sections).

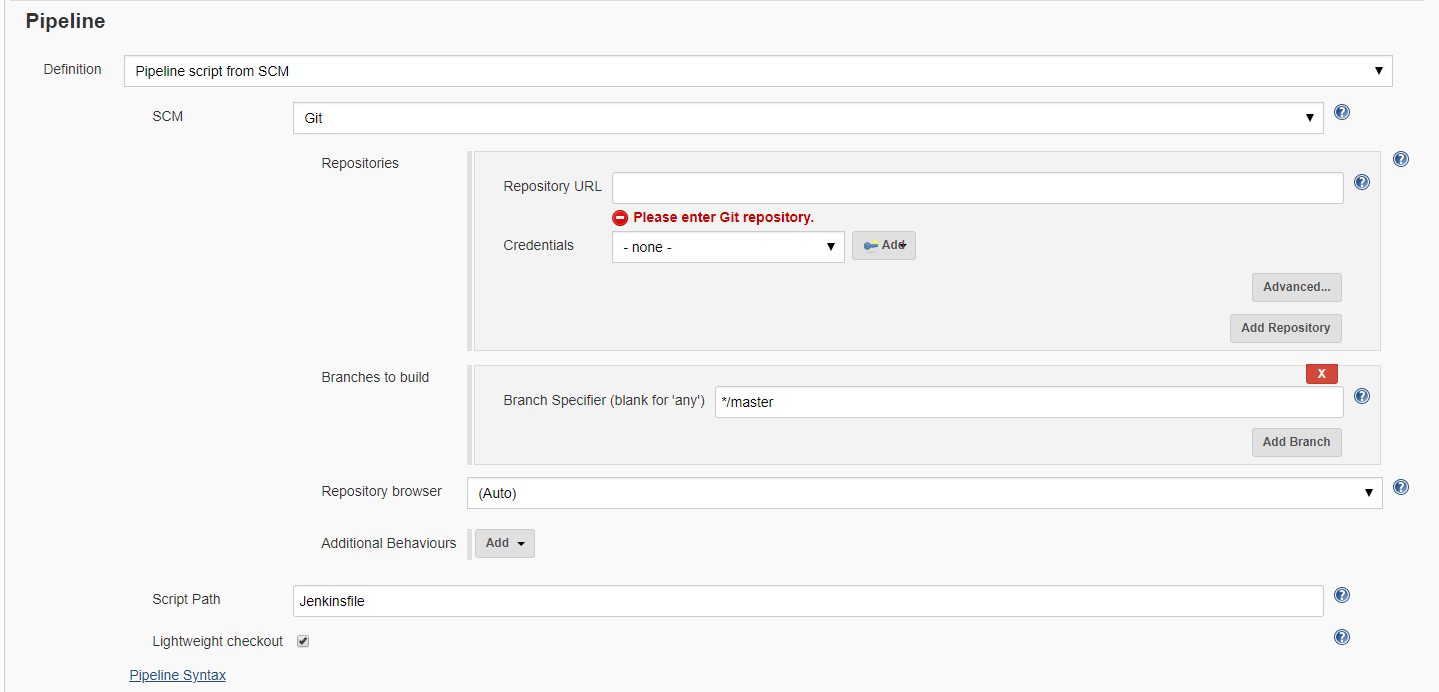
At the top of this section, there is a configurable option—the Definition field. The choices here are either “Pipeline script” or “Pipeline script from SCM” ([Figure 8-35](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_piplin_defopt)).



The “Pipeline script” option represents the default: defining the script in the text entry box below the Definition field. The option that appears underneath the text entry box—“Use Groovy Sandbox”—is explained in [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow).

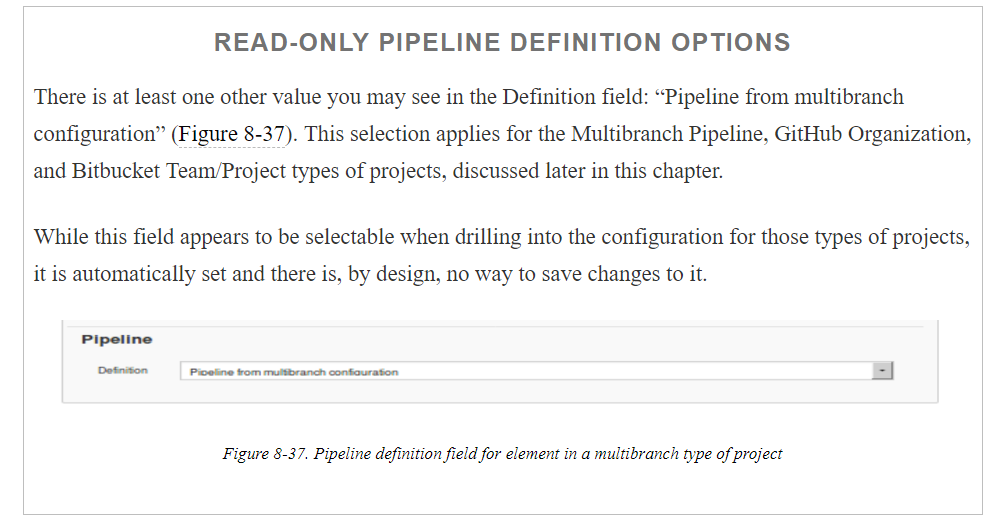
If you instead choose the “Pipeline script from SCM” option, this will allow you to specify the location in a source management system of a Jenkinsfile to use with this job instead of entering the script in the text entry area.

Once you have selected the “Pipeline script from SCM” option, you’ll be presented with additional fields to indicate where to get the script from. These fields are the typical SCM type of fields for a location, revision, etc. (see [Figure 8-36](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_pipeline_script_from_SCM)).



While the Script Path field is editable here, unless you have a particular reason to use something else, the recommended approach is to stay with the Jenkinsfile in the root of the project.

The “Lightweight checkout” option refers to telling the SCM plugin to try to check out only the Jenkinsfile initially instead of the entire project. This is an efficiency to avoid checking out the entire project twice—once to get the Jenkinsfile and once when the Jenkinsfile executes the checkout scm statement. Note that this option may not be supported by all SCM plugins and so may not appear in all cases.



One other interaction to be aware of between the Jenkins Pipeline project configuration page and the script that defines your Jenkins pipeline has to do with setting options on the Jenkins configuration screen. In many cases, the options can be set in the configuration web interface and will define behavior for your script even though there are no lines explicitly defining or setting those options in your script.

As an example, you can select the “This project is parameterized” option on the configuration web page and define parameters through that interface. Those parameters will then be accessible in the pipeline script you define in the Pipeline section.

This behavior is both convenient and inconvenient. It is convenient while you are running your script within the context of a Pipeline project in the Jenkins application itself; you don’t have to add the code in your pipeline to define those parameters. It is inconvenient if you want to use your pipeline script as a Jenkinsfile, separate from the Jenkins application itself. Then you need to go back and update the code in the script to explicitly define the parameters.

[Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow) discusses this particular interaction with parameters in more detail, but it is good to be aware of the interdependence between options defined only in the Jenkins application for a Pipeline project’s configuration, and how those options are referenced in the pipeline script itself. A best-practice approach is to define all such options and functionality in the script

## **The Multiconfiguration Project Type**

This type of project is designed to simplify running a set of project builds that only differ in terms of parameters. For example, suppose you needed to run a test build against a set of five different browsers (IE, Firefox, Safari, etc.) and across a set of five different operating systems (Debian, Centos, Windows, etc.).

Without the Multiconfiguration project type, you would need 25 jobs (5 browsers tested against each of 5 operating systems) to accomplish this. With the Multiconfiguration type, you only need one job that does the work of executing the various possible combinations for you.

The way this works is that you define your base job to do whatever you need to do based on parameters that represent each of the different “axes” you are using. For the example just mentioned, one axis would be the set of browsers and the second would be the set of operating systems.

Like the other project types we’ve discussed, the Multiconfiguration project has the common setup, environment, build, post-processing, and other configuration. sections. But it also includes a separate new Configuration Matrix section. This is where you define the axes that you want to include in the job. There are three types of axes that you can create. Each one takes a name that will become an environment variable (which you can use in the build step) and a definition. The types of axes that can be added to the Configuration Matrix are:

Slaves

This type of axis definition allows you to specify either a node’s name or a label on a node to include in the set of nodes to iterate over. (As discussed elsewhere in the book, a label is simply a tag or identifying name that we can attach to one or more nodes. Then we can select one or multiple nodes by specifying a label that they have.)

Label expression

This type of axis definition allows you to use advanced syntax to choose which set of nodes to include. For example, you can combine node labels and operators, as in label1&&label2, to indicate that only a node having both labels is eligible to be included.

User-defined axis

This type allows you to specify a set of items as values to iterate over in building the set of jobs.

### MULTICONFIGURATION EXAMPLE

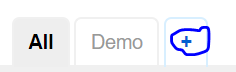
Let’s consider a use case for this type of project. We have some jobs to build to create web pages for a set of company job families in each of several different regions (where each region has a dedicated node).

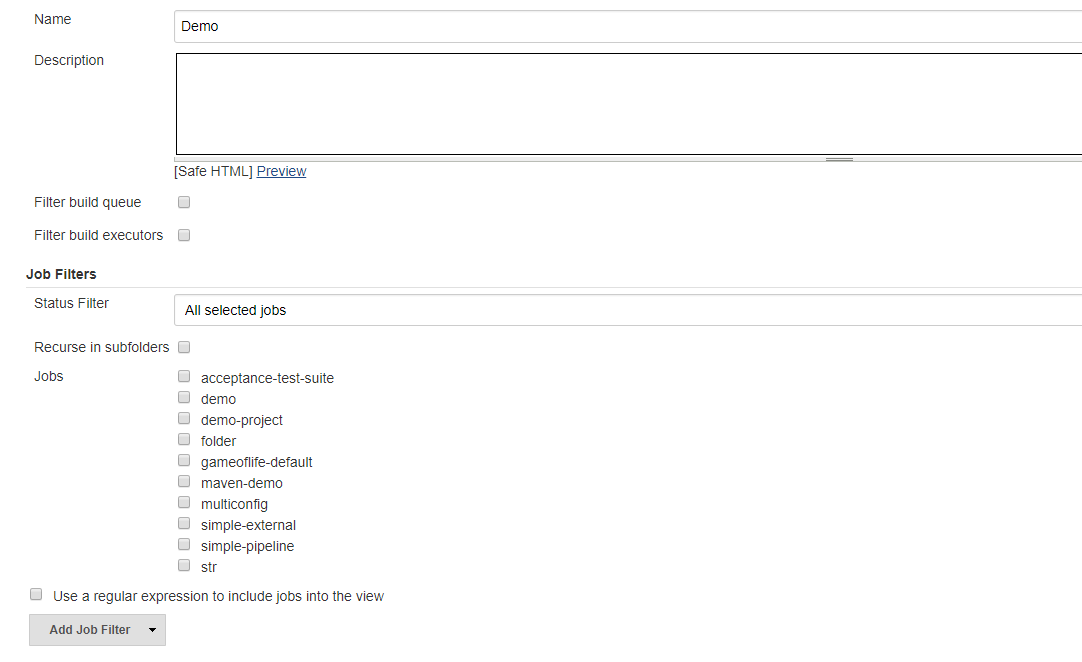
In our setup, we have three worker nodes available, with various labels, defined like as in [Table 8-1](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#available_worker_nodes).

|  |
| --- |
|  |
|  |

## Folders

One of the newer types of items you can create in Jenkins 2 is a folder. As the name implies, this is an organizing structure rather than a job or project. Traditionally, views have been used in Jenkins to filter lists of items on the dashboard. Views offered the ability to create limited lists of jobs via configuration (by clicking on the “+” tab at the top of the main project list). [Figure 8-52](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_config_typi_JenkLV) shows the configuration screen for a typical list view.





Unlike views, folders actually add the ability to group items together into a common namespace, structure, and environment. Specifically, a folder allows a set of jobs to share:

A container

Creating a folder creates a container to hold a set of jobs. As noted previously, this is different from a traditional Jenkins view, which only allowed filtering a list of jobs to restrict which jobs were visible.

A namespace

This namespace also becomes part of the path to the job.

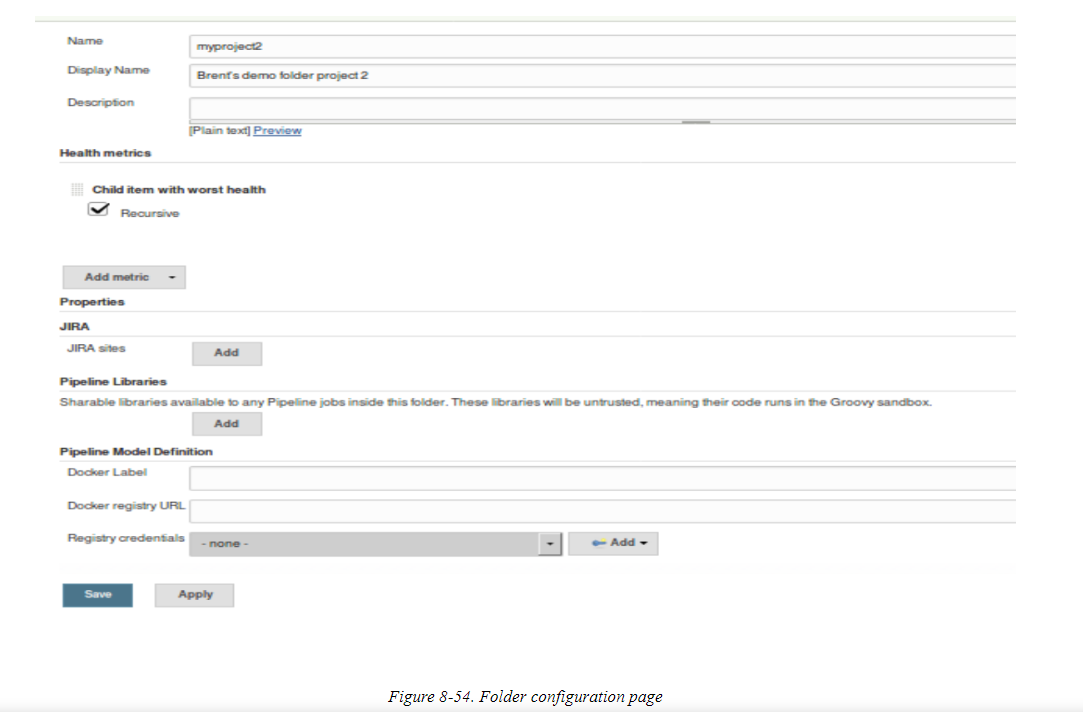
Shared libraries

A folder can have its own set of shared libraries just for the projects in the folder.

Separate permissions

These are available provided the Role-Based Authorization Strategy plugin is installed and role-based permissions are configured. More details on this are included later in this section.

All of these elements allow for new ways in Jenkins to organize jobs and restrict the environment in which they run. This could be used, for example, to separate or group the projects for a department or larger effort.



At the top, you can enter user-facing details such as a separate display name to show for the folder and a description.

Below that is a section for adding “health metrics”—that is, identifying properties of items in the folder that should contribute to an overall health indication (how successful or not builds for items in the folder have been). As of the time of this writing, the only available health metric is “Child item with worst health.” There is also a Recursive option to indicate whether items in subfolders should contribute to this metric.

Next is a Properties section. You may or may not have anything in this section, depending on what plugins you have installed. The idea is to provide a place to define tools or setups specific to items in this folder or its subfolders (if it has any). An example here might be a JIRA project configuration for items in the folder.

Further down on the page is the section where you can configure a shared library to be available to all jobs in the folder structure (this folder and any subfolders). The same configuration fields and settings are available as for global shared libraries (see [Chapter 6](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch06.html#CH_Extending_Your_Pipeline) for details and examples); the only differences are that these libraries are not trusted (so they cannot make unapproved calls or method invocations, as global shared libraries can), and they are only available to the items in the folder structure.

Finally, we have the Pipeline Model Definitions section. This one requires some additional explanation. (Like for shared libraries, there is also a section for this in the global Jenkins Configure System screen, so this can be configured at different granularities.)

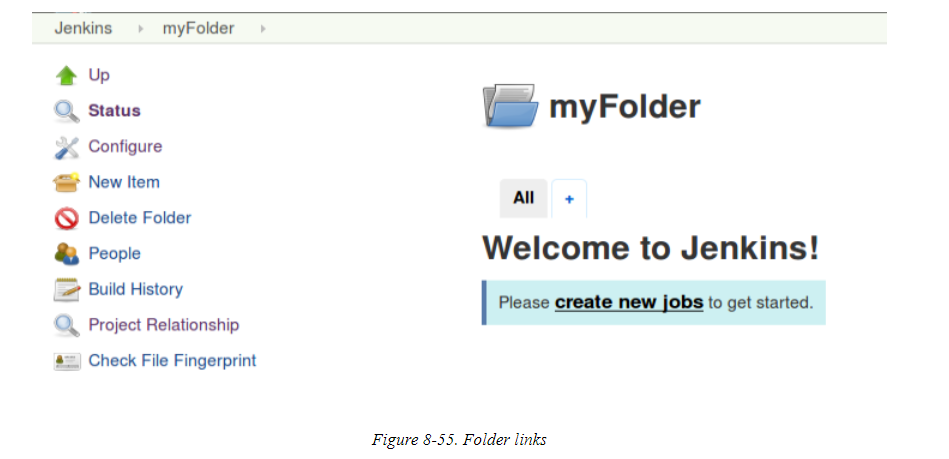
By default, Jenkins pipelines make the assumption that all agents are able to run Docker pipelines. (See [Chapter 14](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch14.html#CH_Integrating_Containers) for information on using Docker and Docker-based agents in your pipelines.) However, in some cases, such as if you’re running on Windows, where you traditionally can’t run the Docker daemon directly, this assumption can be incorrect. So, if you don’t explicitly specify an agent that can run Docker in your pipeline, and you get one of the agents that can’t, your pipeline won’t work.

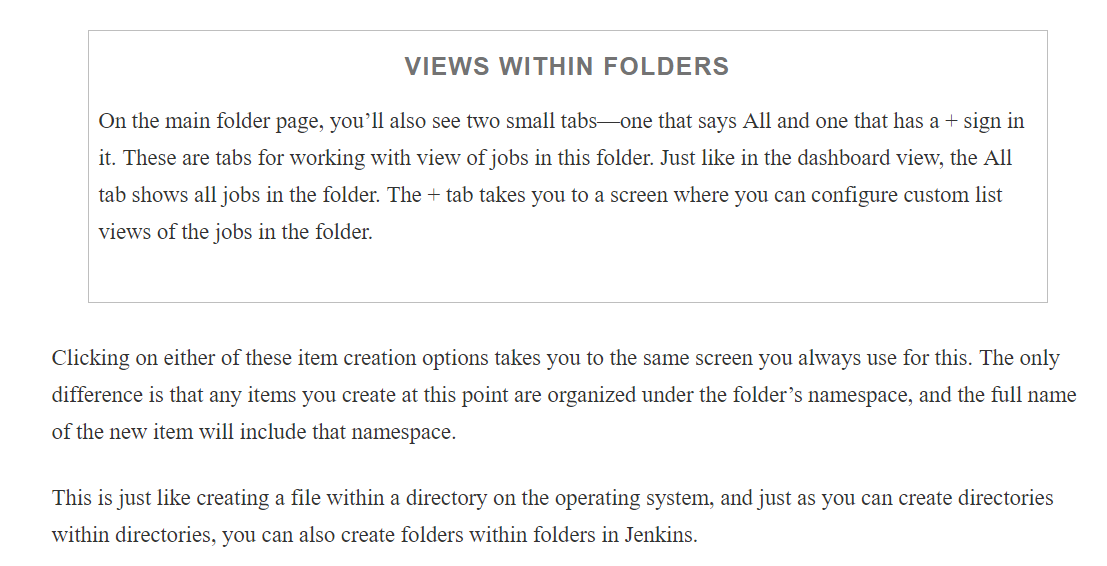
Assuming you have a label that identifies one or more of your agents as being capable of running Docker, you can specify that label here. This tells Jenkins to use one of those agents for any folder items that need Docker, but don’t directly specify an agent that can run it.

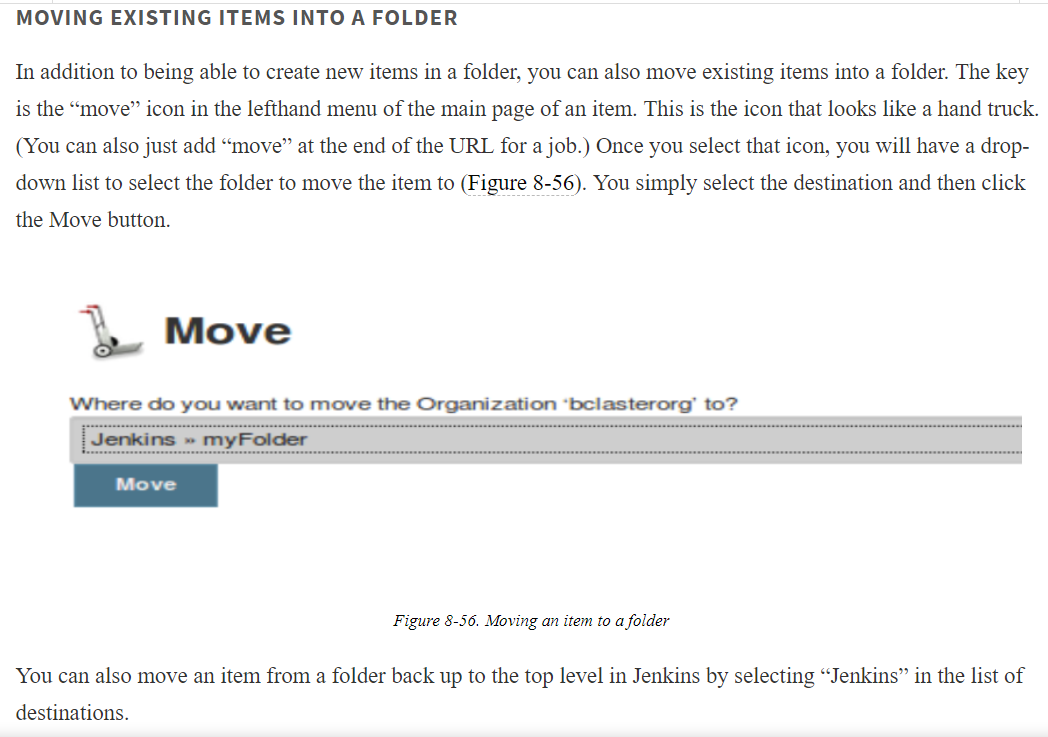
Likewise, you can specify a Docker registry to use here that is scoped to just the items in the folder.

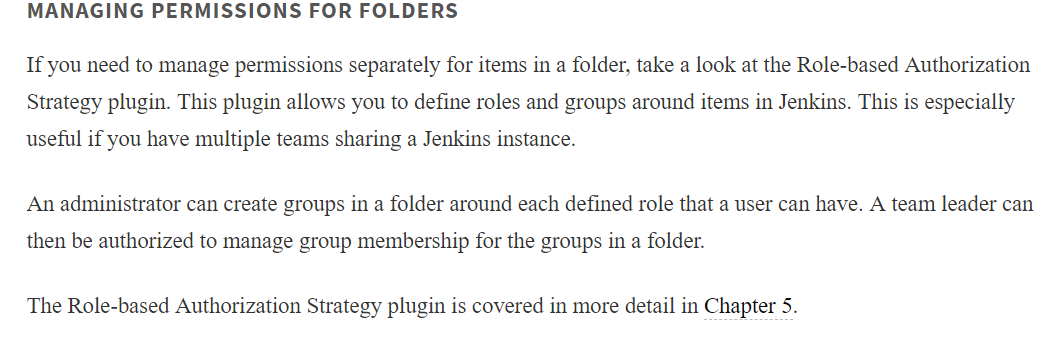
### CREATING ITEMS IN A FOLDER

Once you’ve created a new folder in Jenkins 2, you can create new items in it just like you’ve always done. When you switch to a Folder project, you have a link in the center of the page to “create new jobs” as well as the New Item link in the left hand menu ([Figure 8-55](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_folderlinks)). (Note that there is also a Delete Folder item in the left hand menu.)









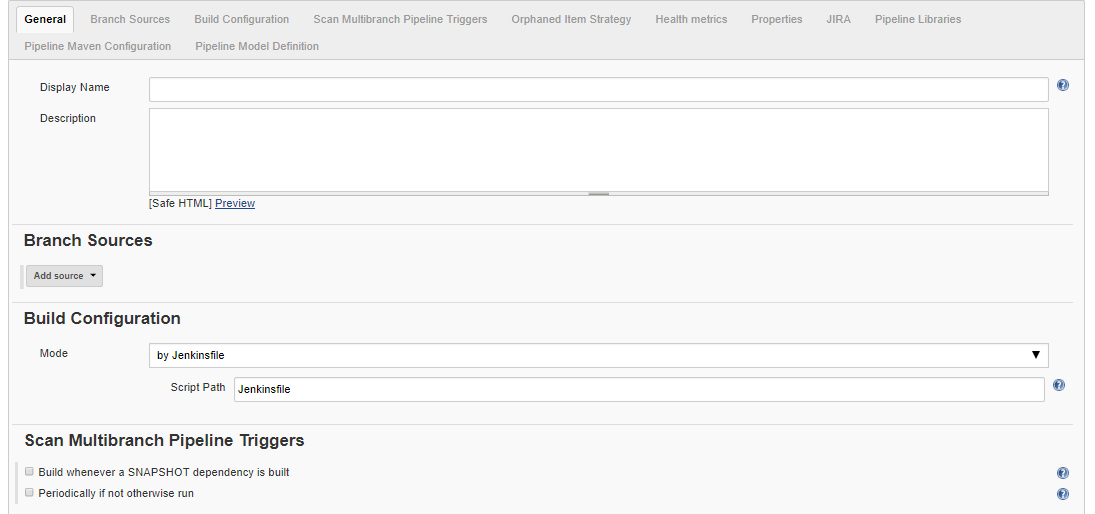
## Multibranch Pipeline Projects

One of the other new project types in Jenkins 2 is the Multibranch Pipeline project. The primary feature of this type of project is that Jenkins can automatically manage and build branches of projects managed in a source control management system if it recognizes them as Jenkins projects. It can also create new Pipeline projects for each branch it detects in the source control repository.

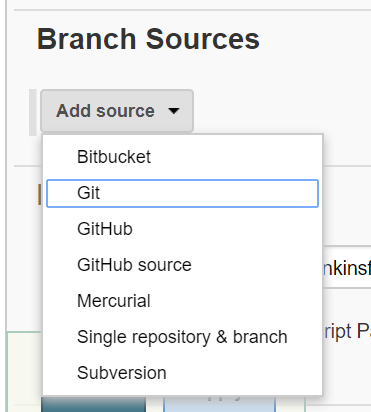
You can effectively think of this type of project as a Folder project with different jobs in the folder for each branch of a source project. Creating and automatically building these jobs is possible by using the presence of a Jenkinsfile as a marker and utilizing a scanning process known as branch indexing.

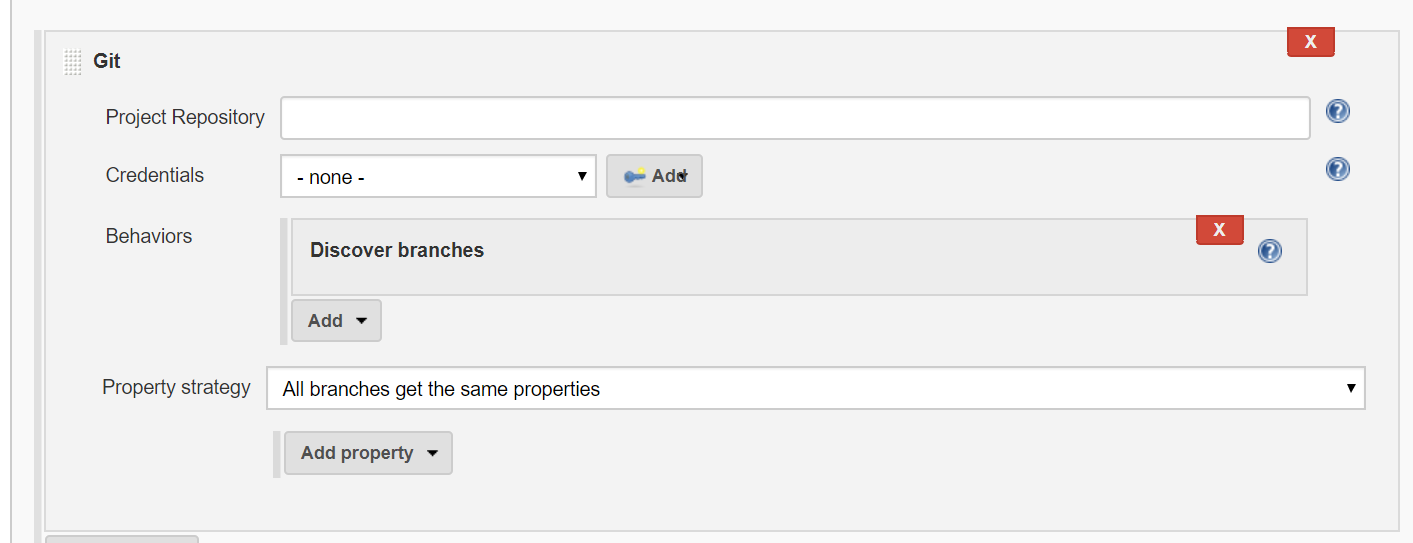
### CONFIGURATION

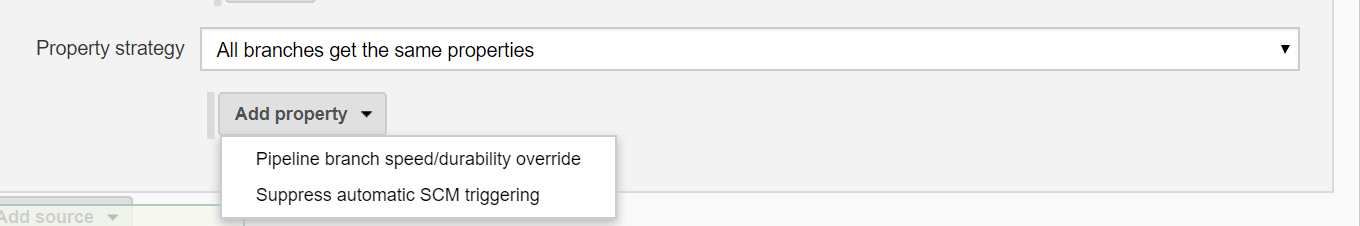
When you create a new Multibranch Pipeline project, you will typically point the job to an SCM repository instead of to a specific branch of a project. Below shows an example of the configuration screen for this type of project.



The first few settings here are pretty standard. However, notice that in the Behaviors section under Branch Sources, there is a default behavior of “Discover branches.” This is one of the key elements of a Multibranch Pipeline project: the ability to look into the SCM repository, figure out what branches are there, and set up jobs for them. Other typical behaviors (as provided by the particular SCM plugin) can be added with the Add button. For Git, these might include, for example, ignoring branches based on patterns, specifying options when cloning, and cleaning out workspaces.

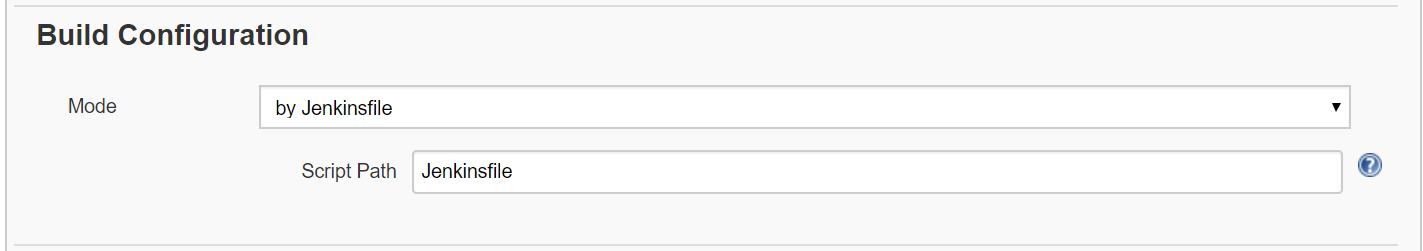






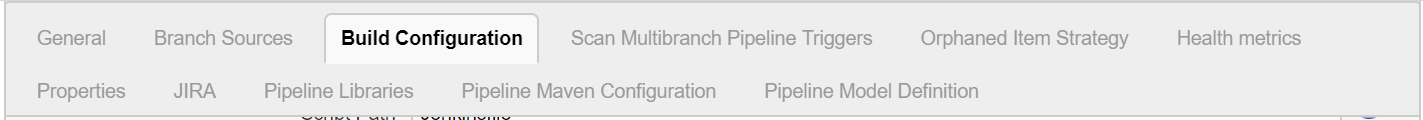
Underneath that is the “Property strategy” section. For Multibranch Pipeline projects, this is either “All branches get the same properties” or “Named branches get different properties.” Selecting the latter allows you to specify one or more named branches (in a “Branch name” field) and choose a property to apply. Currently the onlyavailable property is “Suppress SCM triggering,” which suppresses the normal commit trigger for Jenkins in that branch.

In the Build Configuration section, we have only one option currently: “by Jenkinsfile.” This is the functionality we’ve already talked about where Jenkins will look for a file named Jenkinsfile in the root of the checked-out project to see if it can automatically build the branch of a project. While you could change the path for the Jenkinsfile in the Script Path field underneath, it’s best to just leave it as the default for standardization.



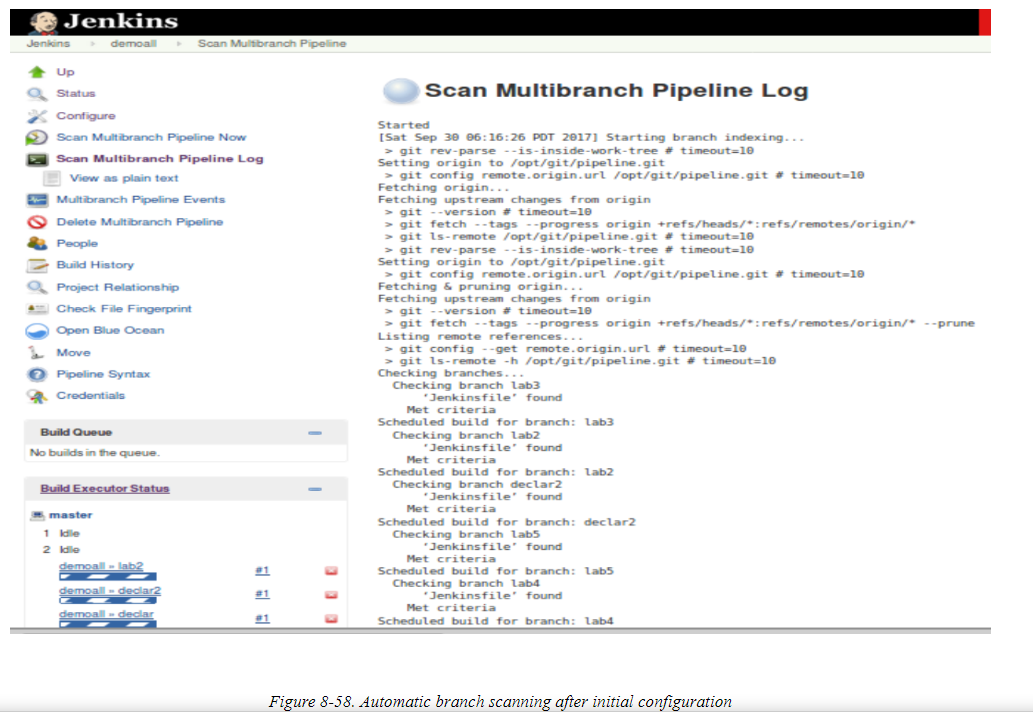
Next on that page is a Scan Multibranch Pipeline Triggers setting. This can be set to “Periodically if not otherwise run” if desired. Basically, if set, this is a fallback in case one of the standard notification mechanisms (commit trigger, etc.) doesn’t work. The idea is that you can set a time interval here that specifies the longest period you’re willing to wait to check for changes if an event doesn’t automatically trigger Jenkins

The remaining sections on the configuration page are the same as the standard ones for a Folder project, such as “Health metrics,” Pipeline Libraries, and Pipeline Model Definition. These are discussed in [“Folders”](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#SEC_Folders).

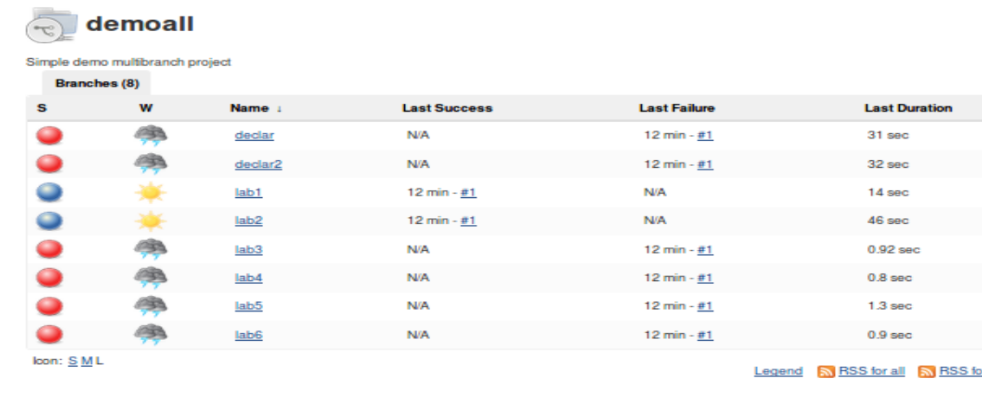


### BRANCH INDEXING

After initial configuration, Jenkins will run a “branch indexing” function to look for the presence of a Jenkinsfile in the branches of the project. If it finds a Jenkinsfile in any of the branches, it will automatically create a job for those branches and build them. [Figure 8-58](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_autobranch_scan_initconfig) shows what this looks like in the console output for the overall job. Notice the places in the log where Jenkins is checking to see if the branch meets the criterion of having a Jenkinsfile and, if so, kicking off a build for it. You can see the builds running in the lower-left build section.



After the branch indexing completes, you’ll have individual jobs for each of the matching branches within your Multibranch Pipeline project



### INDIVIDUAL JOB OUTPUT AND CONFIGURATION

You can drill into each of the individual jobs created automatically for the project and see the output/build results page in the Stage View form.

There is also a View Configuration link on that page. If you click that link, it will take you to a configuration page for the individual job. On that page you will see some of the common sections we have talked about previously, such as General and Build Triggers. You can check boxes in these sections, type things in, etc. However, this is a bit misleading as there is no Save or Apply button at the bottom of the page. As the menu item implies, you can view the configuration (which isn’t particularly useful in this case), but you can’t modify it. It is generated by the branch indexing functionality of the higher-level Multibranch Pipeline project.

Not being able to configure the individual jobs here might seem like a disadvantage, but remember that you can manage your pipeline through the Jenkinsfile instead of through the job’s configuration.

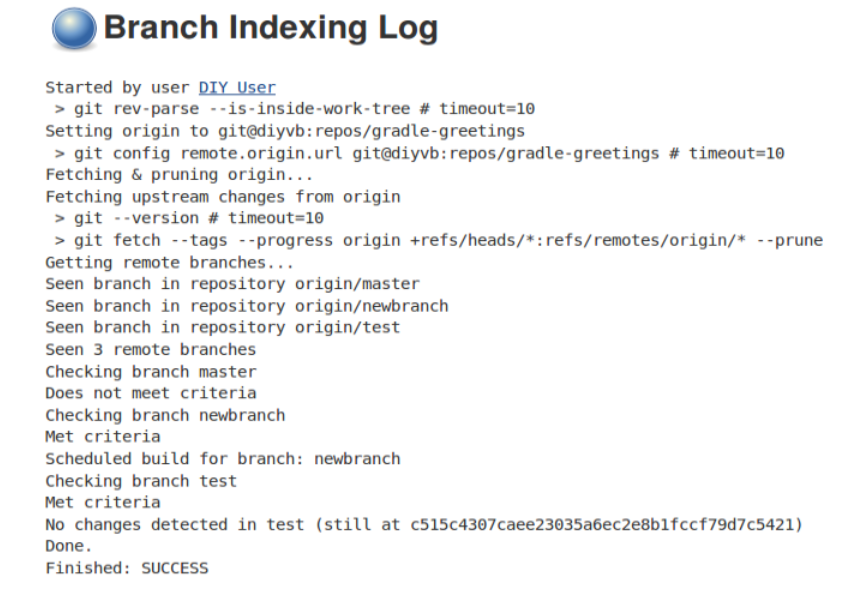
### INCORPORATING NEW BRANCHES

Once you have a Multibranch Pipeline project set up, Jenkins can automatically detect new branches and create corresponding jobs for them as well. Let’s look at an example.

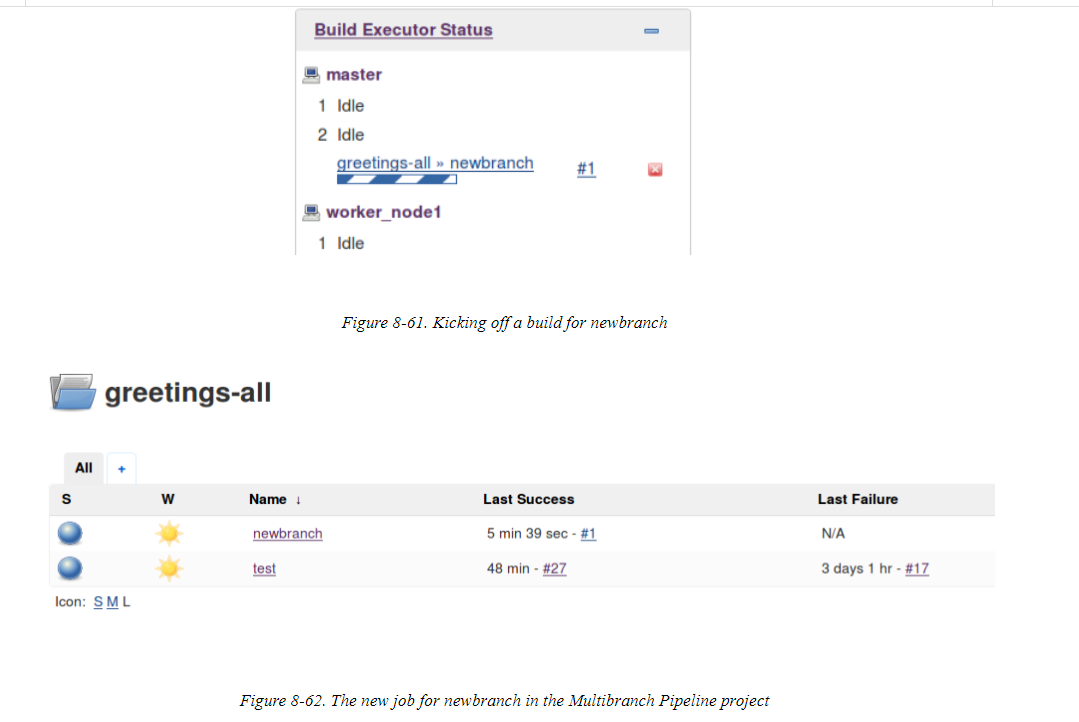
Suppose that you have a Multibranch Pipeline project set up in Jenkins for a local Git location. In your repository, you have a master branch that does not have a Jenkinsfile, and a branch named test that has a Jenkinsfile in it. Since you have set up a Multibranch Pipeline project, you have a job for test in Jenkins that was created automatically. There isn’t a job for master, because it did not have a Jenkinsfile.

Now suppose you clone that repository down and create a new branch called newbranch from test. newbranch inherits all of the files from test, including the Jenkinsfile.

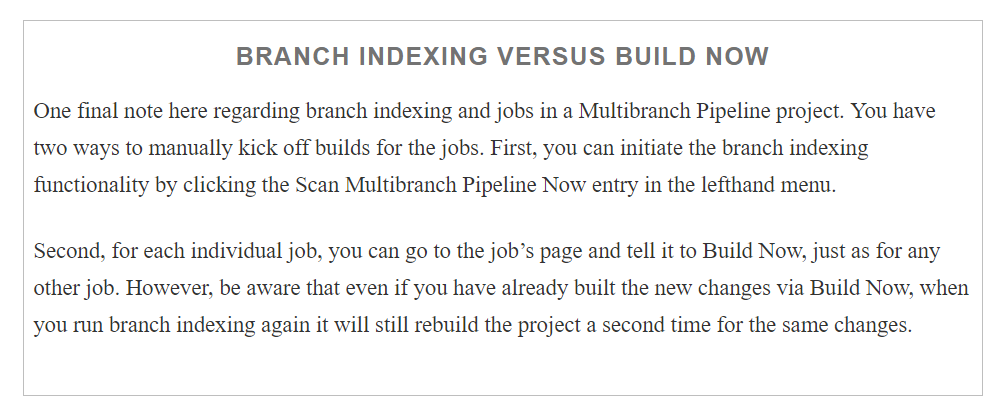
Next, you push the changes back to the remote Git repository. At this point, if you go back into Jenkins and tell it to run the branch indexing, it will go out to the repository and check each branch. [Figure 8-60](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_branchind_aft_newbran) shows the branch indexing running.



Jenkins identifies that the new branch “Met criteria.” This means that it has a Jenkinsfile. So, Jenkins creates a new job for it ([Figure 8-61](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#fig_kickoff_buildnewbranch)) and starts up a build for it



The nice thing about this setup is that it allows you to create branches in Git as you need them (for experimentation, for example) and automatically have corresponding Jenkins jobs created to execute the pipeline on those branches.



## GitHub Organization Projects

Read from the book & internet