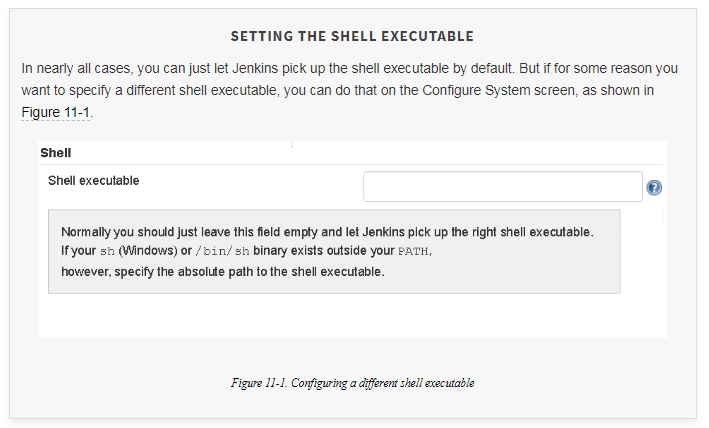
**Chapter 11. Integration with the OS (Shells, Workspaces, Environments, and Files)**

# Using Shell Steps

We’ll begin by looking at a set of steps that allow for passing commands to the operating system for execution. As you might imagine, there are separate steps for Linux/Unix and Windows. However, they are nearly identical in the options they supply.



## **The sh Step**

Probably the most general-purpose step available to pipelines is the sh step. If there isn’t a specific step that does what you need, or that integrates a particular application, you can typically come up with a shell command and arguments to do it. The sh step can then be used to execute that command.

The default syntax is straightforward:

sh '<shell command string>'

The default form of the step does not provide much integration with the pipeline in terms of returning information. We’ll explore some useful options to help with that. Afterwards, we’ll explore some ways to set context around the command and even run scripts in other programming languages. To begin, let’s look at the set of options for the step:

script

The operations to execute, expressed as strings. This is the default parameter, so script doesn’t have to be specified if that’s the only parameter you’re using. Multiple lines are allowed, but you’ll need to enclose them in triple quotes.

encoding

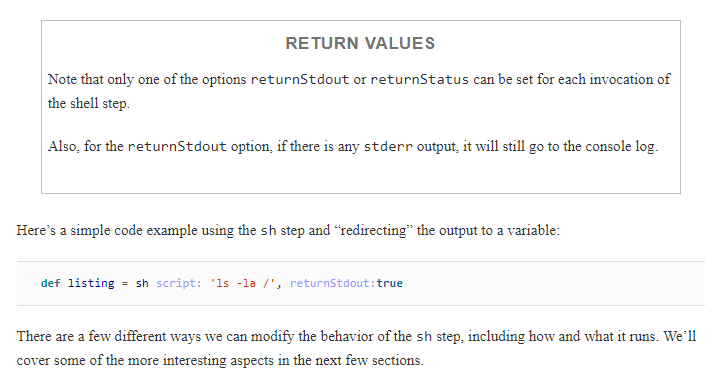
The encoding of the output, expressed as a string. You only need to set this if you need to use something other than the default value of UTF-8.

returnStdout

A Boolean. If this is set to false (the default), then stdout is just printed to the console log. If it’s set to true, stdout is returned from the step as a string. (Hint: You can use trim() to strip a trailing newline if needed.)

returnStatus

A Boolean. If this is set to false (the default), then a nonzero status code will cause the step to fail and throw an exception. If this is set to true, then the status code will instead be the return code from the step. You can take that return code and check it and act accordingly.



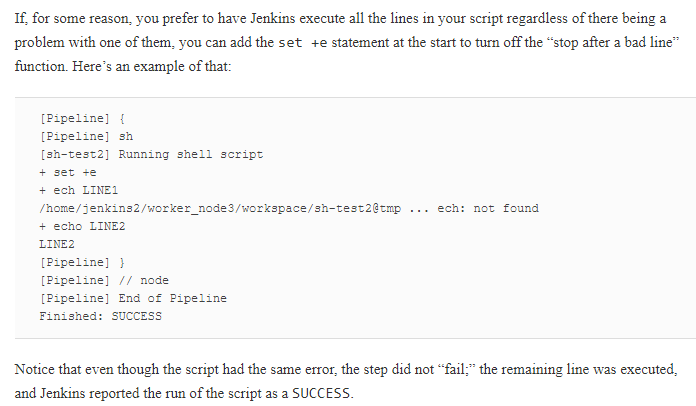
### SET OPTIONS

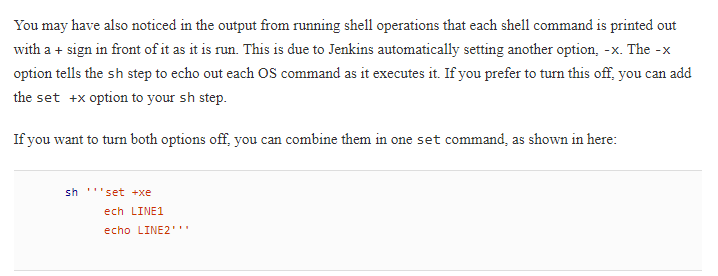
By default, the shell will not stop if there is an error in your script. It will happily try to execute all of the lines. However, this is not usually what you want—especially if you are using shell commands as part of a pipeline script.

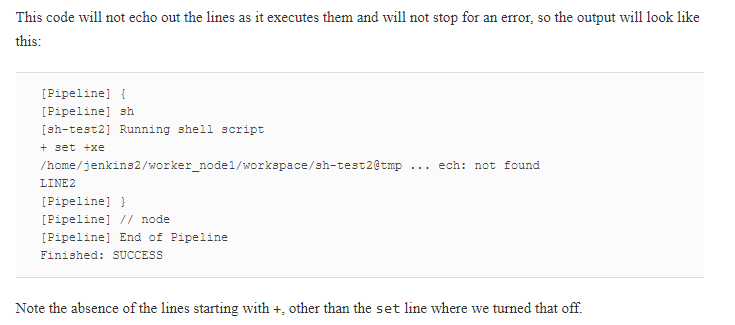
So, the sh step in Jenkins automatically includes a set -e option. This tells the shell to stop execution and not run the rest of the script lines if it encounters an error in a line.

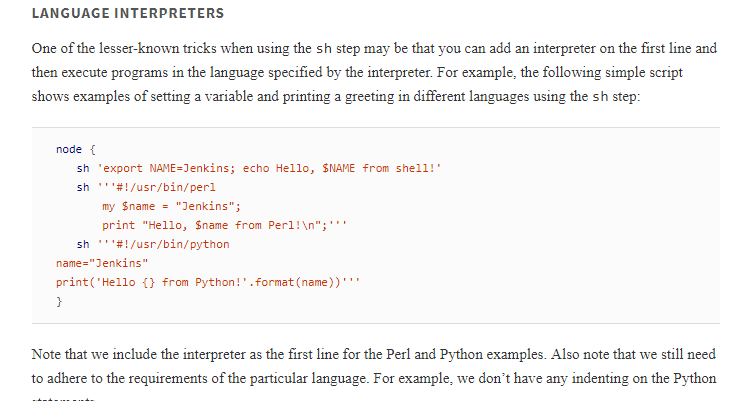


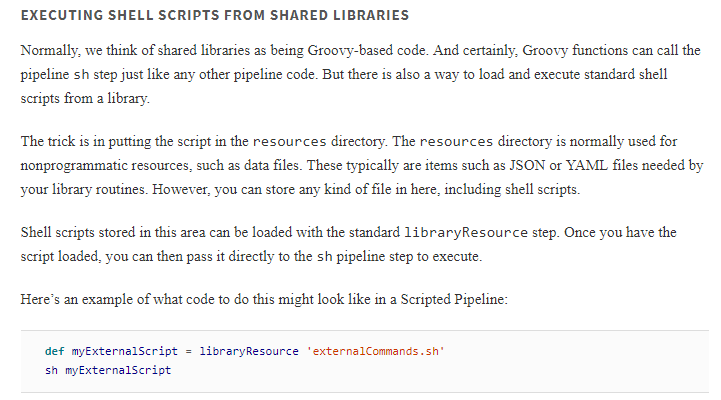


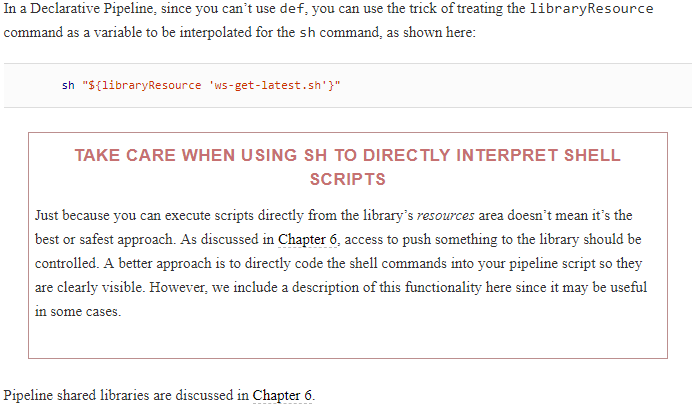


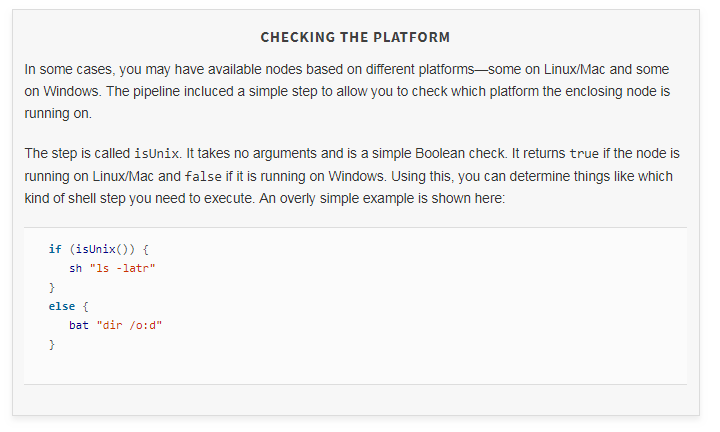












Next up is bat step which we are currently not using

# Working with Workspaces

Most of the time when we work with Jenkins, we don’t even think about workspaces. Jenkins just manages them for us. If there’s a problem with our processing, we may go out and find the workspace and look in it to try to understand what went wrong, but otherwise we’re happy to let Jenkins manage things. However, you may come across a time when you need or want more control over the workspaces you’re using. For those times, Jenkins provides a couple of pipeline steps that can be useful.

## **Creating a Custom Workspace**

Using the node or agent directive gives you a workspace automatically. However, if you need or want a custom workspace to work in, you can use the pipeline ws step. This step takes a single argument—the directory you want to use for the workspace—and attempts to lock it for exclusive use. The path supplied as the argument can be relative to the node area or an absolute path. The directory will be created if it doesn’t exist.

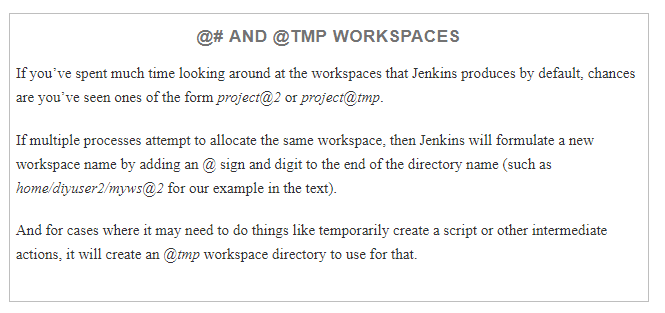
An example of the basic syntax is shown here:

ws ('home/diyuser2/myws') {

*// block of code to execute in workspace*

}

Note that this step is actually a block step, defining a closure in which you can put the code that you want to execute in the custom workspace.

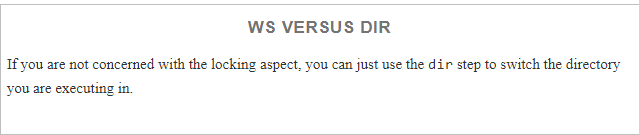


In this script, we’re using the ws step to create a new workspace to work in. And then, within that, we’re asking for the same workspace again. Within each workspace area, we’re using the pwd step so we can print out the current working directory.

Here’s the output from this script:



As you can see, Jenkins switched to the requested workspace the first time we asked for it. On the second request, it appended the @2 to the workspace name. In most cases, this probably won’t matter, but if it did (if you needed the exact name), you could check the output of a pwd() call and exit or wait depending on what you wanted to do.



## **Cleaning a Workspace**

Workspaces in Jenkins are not automatically cleaned. However, as we’ve discussed in other chapters, the pipeline supplies a cleanWs step to clean out a workspace. By default, the step removes all files in the workspace, regardless of the build result, and fails the build if the cleanup fails. But there are a number of options to customize the steps’ behavior, summarized in the next few sections.

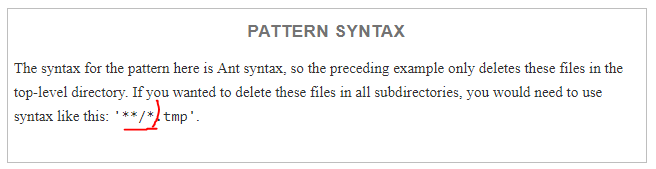
### **FILE PATTERNS TO BE DELETED**

By default, all files in the workspace will be deleted. However, you can add file patterns to include or exclude from deletion. This is specified using the patterns argument and an array of pattern and type pairs (where pattern is the file pattern and type is include or exclude).

For example, to only delete the \*.bak and \*.tmp files from the workspace, you could use the following syntax:

cleanWs patterns: [[pattern: '\*.bak', type: 'INCLUDE'],

[pattern: '\*.tmp', type: 'INCLUDE']]



Above marked one is Ant syntax

One thing that may not be obvious here is that you can use the INCLUDE and EXCLUDE types together. Why would you do this? One reason would be to add specific exclusions within the context of a larger inclusion—in other words, to keep a specific item or items when other things of that type are being removed.

In the following code, we are deleting all of the *\*.tmp* files except the one named *keep.tmp*:

cleanWs(patterns: [[pattern: '\*.tmp', type: 'INCLUDE'],

  [pattern: 'keep.tmp', type: 'EXCLUDE']])

### OTHER ARGUMENTS

The remaining arguments for the cleanWs step are all Booleans for various aspects. Remember that the default form of the step is just cleanWs, so these arguments only need to be specified if you don’t want the default behavior. The available arguments are:

cleanWhenAborted

Default is true; if set to false, the step will not clean the workspace when the build status is aborted.

cleanWhenFailure

Default is true; if set to false, the step will not clean the workspace when the build status is failed.

cleanWhenNotBuilt

Default is true; if set to false, the step will not clean the workspace when the project was not built.

cleanWhenSuccess

Default is true; if set to false, the step will not clean the workspace when the build status is success.

cleanWhenUnstable

Default is true; if set to false, the step will not clean the workspace when the build status is unstable.

deleteDirs

Default is false; if set to true, will delete directories also. Note that if patterns are supplied (as outlined in the earlier section), this will only delete directories that have names that also match those patterns.

notFailBuild

Default is true; if set to false, this will fail the overall build if the cleanup step fails.

# File and Directory Steps

Finally in this chapter, we’ll discuss the Jenkins pipeline steps provided for working with files and directories. We’ll start with the ones specific to files.

## **Working with Files**

Files are another way we can pass information to and from Jenkins. Pipeline DSL contains simple steps for the most common operations for working with files: reading, writing, and checking for existence. We’ll cover those operations in this section.

### **READING FILES**

The step for reading a file into a pipeline is readFile. It reads in the contents of the file and returns those as a string.

readFile has two possible parameters. The first is file, which is the relative path to the desired file from the current directory. Most often, this will be relative to the workspace directory, since that’s where the script will be running and thus will be the default current directory. Path components should be separated by forward slashes (/). This parameter is required.

The second parameter is the file encoding, such as UTF-8. This parameter is optional.

The following code snippet shows the simple form of the step and then a version that includes the encoding:

readFile 'dir1/dir2/filename'

readFile encoding: 'UTF-8', file: 'dir1/dir2/filename'

### **WRITING FILES**

Like reading files, writing files is fairly straightforward. The step is writeFile. It takes a required parameter for the path to the file to write. That parameter is named file. The path for the file parameter is relative to the current directory which, like with readFile, is usually going to be the current workspace. Here too, path components should be separated by slashes.

Also required is the text string to write out to the file, specified by the parameter text.

Finally, an optional encoding parameter can be specified if needed.

An example call for this step is:

writeFile encoding: 'UTF-8', file: 'dir1/dir2/file.out',

text: 'Output from build'

### **CHECKING FOR FILE EXISTENCE**

The last file operation is one that checks for the existence of a file. Not surprisingly, the name of the step is fileExists. It only takes one argument: the path with the name of the file to check for. Like the other file operations, this path is expected to be relative to the current directory (usually the workspace directory when a job is running) and with the components separated by slashes.

An example is:

fileExists 'build/reports/index.html'

Now, we’ll move on to the steps that support working with directories.

## **Working with Directories**

The pipeline DSL provides several steps related to directories that can be useful. The function of most of these may be obvious based on the name, but some of them can have specialized usage in the context of a pipeline.

### **DIR**

As the name suggests, the dir step allows you to switch the current working directory. This step is a block step, meaning that you supply a directory and that directory is the current directory for any other steps in the block.

For example:

dir('/home/user') {

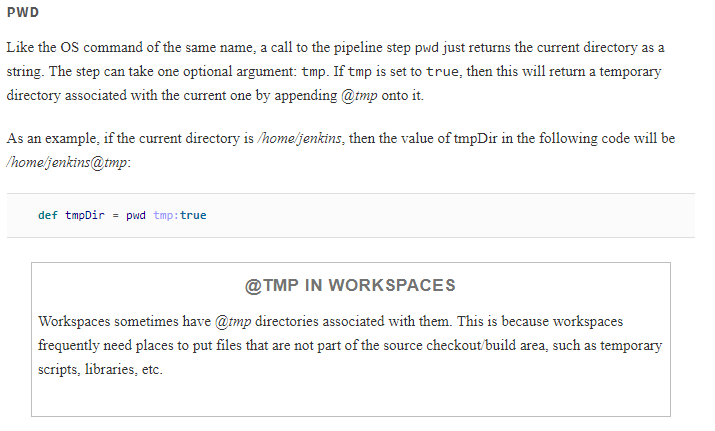
*// some steps*

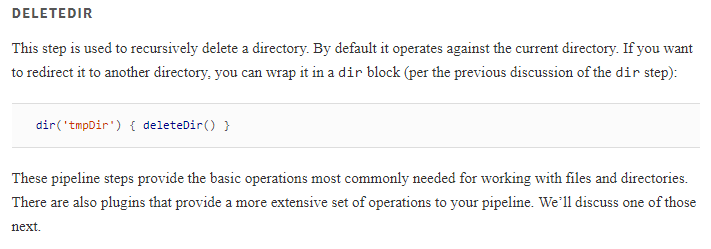
}

Here are a few points to be aware of when using this command:

* The path you supply to the step can be either absolute or relative.
* If the directory does not exist, Jenkins will attempt to create it, but it must have appropriate permissions to do so.
* If a step inside the block uses a relative path, it will be relative to the directory set in the step.

You may be wondering what the difference is between using this command to switch directories for a workspace and the ws command. As mentioned briefly earlier in the chapter, the ws command provides locking functionality, such that multiple jobs cannot use the same directory as the workspace at the same time. dir does not.





## **Doing More with Files and Directories**

As with nearly any kind of built-in functionality in Jenkins, plugins have been written that can extend the set of functionality for working with files and directories. One of those is the [File Operations plugin](https://plugins.jenkins.io/file-operations).

When installed, the File Operations plugin adds a new fileOperations step that has a number of suboperations to assist with file and directory manipulations. Most of the functionality should be obvious from the names, but for more information on what these do, please see the plugin’s documentation.





