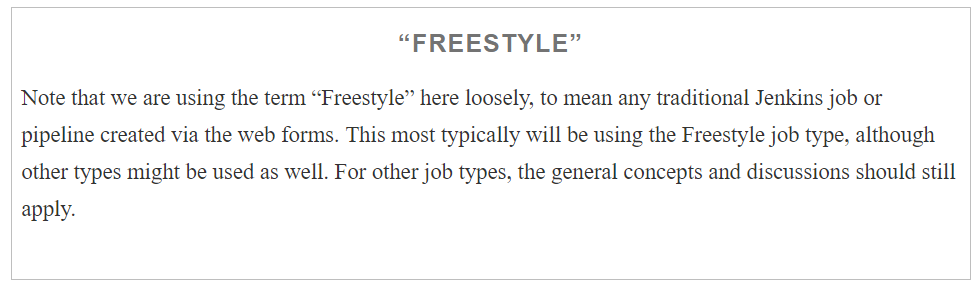
**Chapter 10. Conversions**

With the advent of Jenkins 2, the Jenkins user now has many options for ways to create and express pipelines. They include the traditional Freestyle jobs, pipeline code in the Jenkins application itself, and pipeline code stored in Jenkinsfiles. Additionally, pipeline code can be written either in the Scripted Pipeline syntax or the Declarative Pipeline syntax. With all of these ways to define pipelines, it is highly likely that the user will need, or want, to do some sort of conversion between the various forms at some point. This chapter will provide guidelines on accomplishing some of these conversions.

In particular, we’ll focus on three main types of conversions:

* Converting from Freestyle jobs to a pipeline in the Jenkins application
* Converting from a Scripted Pipeline to a Jenkinsfile
* Converting from a Scripted Pipeline to a Declarative Pipeline



Rather than attempt to provide every detail about how to do a conversion, we’ll focus on guidelines and some selected examples to illustrate the approach and principles involved for each of these categories. While these do not cover every possible case, they should cover enough to give you a good grasp of how to handle the other cases.

# Common Preparation

Before beginning a conversion, there are a few general things to consider. While not an absolute requirement, this may save you some work later on. Most of the items descriubed here are in the form of questions, designed to remind you of information you may want to gather up front for the existing pipeline.

## Logic and Accuracy

It may go without saying, but before you convert from an existing pipeline of one form or another, you want to make sure that the existing pipeline runs as expected and completes successfully. That doesn’t mean you can’t redesign or refactor parts of the pipeline as you convert it, but ensuring you have an existing pipeline that works will give you a reference to test against and compare results and logic to.

## Project Type

Jenkins 2 introduces a number of different project types and structures that were not previously available. It is worth considering at this point whether your converted pipeline jobs might better fit into a Jenkins folder structure, a Multibranch Pipeline project (if you can make use of a Jenkinsfile and multiple branches), or a GitHub Organization or Bitbucket Team/Project project (if you have one of those already set up).

## Systems

Next, consider what nodes the pipeline currently uses. Will the new pipeline have access to these, or do new ones need to be set up? What are the labels of each system that is used? Is anything running on the master node? If so, is it appropriate to be run on the lightweight executor there? Do you need to add any additional labels to the node configurations to fit your new pipeline?

## Access

What access to resources or user permissions are needed for the parts of the pipeline to run? Are certain credentials required, or do new/additional ones need to be defined and set up?

Another use case might be transitioning from a Freestyle project to a Multibranch Pipeline or GitHub Organization/Bitbucket Team project. In those cases, you might need to ensure you have access to the code in the external repository and set up supporting pieces like the webhook for a GitHub project (as discussed in [Chapter 8](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch08.html#CH_Understanding_Project_Types)).

Also, if you choose to create or use shared libraries, you will want to consider whether they should be global or not, and who should have access to update them

## Global Configuration

Luckily, telling Jenkins where global tools are located still involves the same basic process. In the Global Tool Configuration (or System Configuration, depending on the tool), you add an entry for the tool and specify a name and installation location. No significant changes are needed in most cases for this part. However, it’s worth reviewing the configuration to see whether any newer (or different) versions are warranted. This also serves to refresh your knowledge of what’s available and how it can be accessed.

## Plugins

Since Jenkins derives most of its functionality from plugins, the correct ones need to be installed. Are there updates that need to be done? If converting from a Freestyle to a Pipeline project, do the operations done in the Freestyle job have corresponding pipeline DSL commands?

In order for plugins to be compatible with the new Jenkins 2 features, they must be updated from the traditional versions. There are primarily two criteria:

* They must be able to survive restarts (be serializable).
* They need to provide steps that can be integrated with pipeline DSL code.

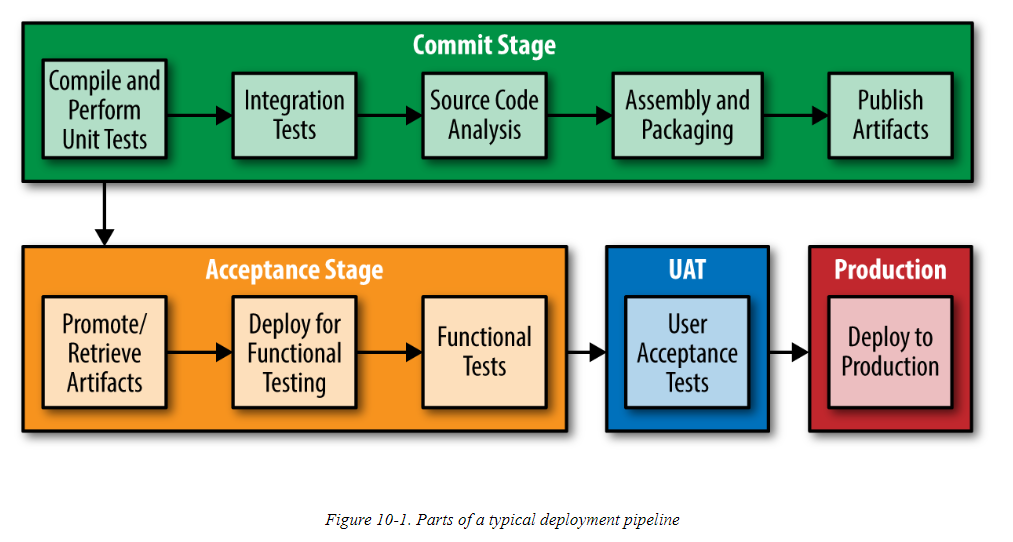
So, the first order of business when looking to migrate the specific functionality of some technology in a Jenkins pipeline is to ensure that you have an updated plugin version installed that is compatible with the pipeline DSL.

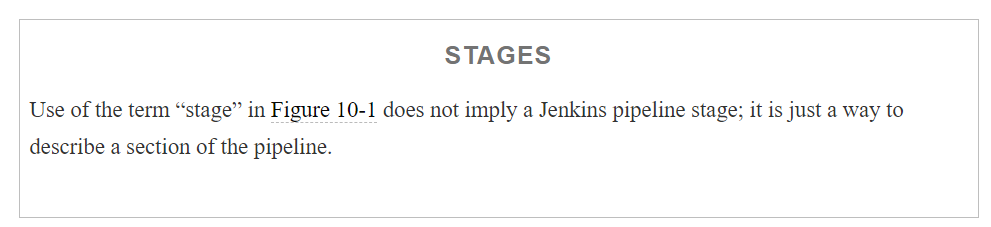
## Shared Libraries

Shared libraries are a convenient way to compartmentalize code that needs to be reused, or that needs to contain complexity, or that must be separated out for security purposes. Consider whether there is such functionality in your existing pipeline that you want to move into a shared library. If so, it would be advisable to work on coding your shared library early to ensure it will work as you think and can be called from your code.

# Converting a Freestyle Pipeline to a Scripted Pipeline

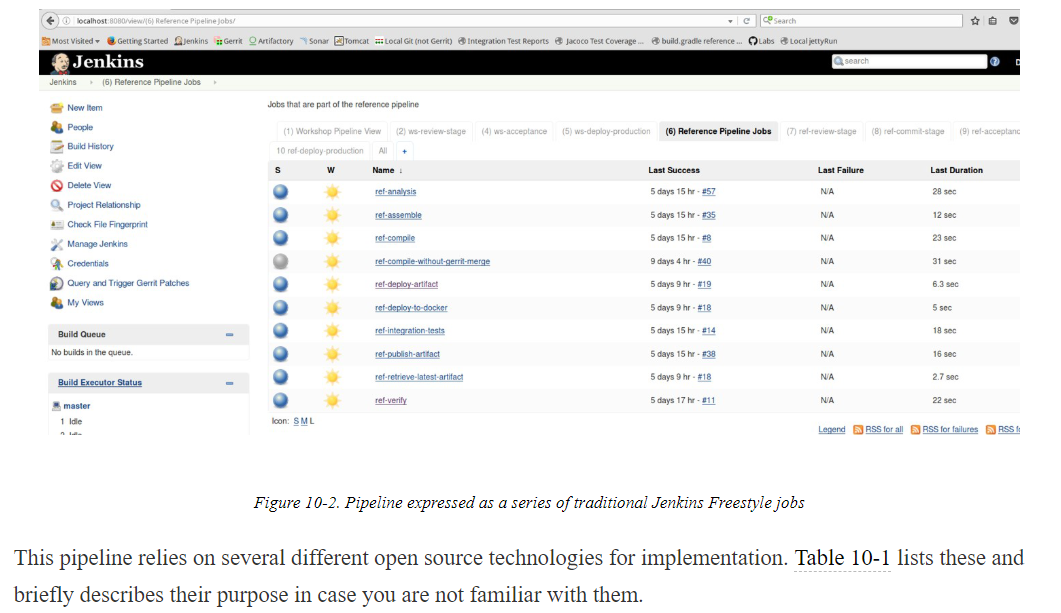
Now that we’ve covered the prerequisites and migration considerations, let’s actually walk through (at a high level) a conversion of an example Freestyle pipeline to a Scripted Pipeline. [Below](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch10.html#fig_parts_of_typ_dep_pip) figure  shows a typical example of a deployment pipeline and the pieces associated with it.

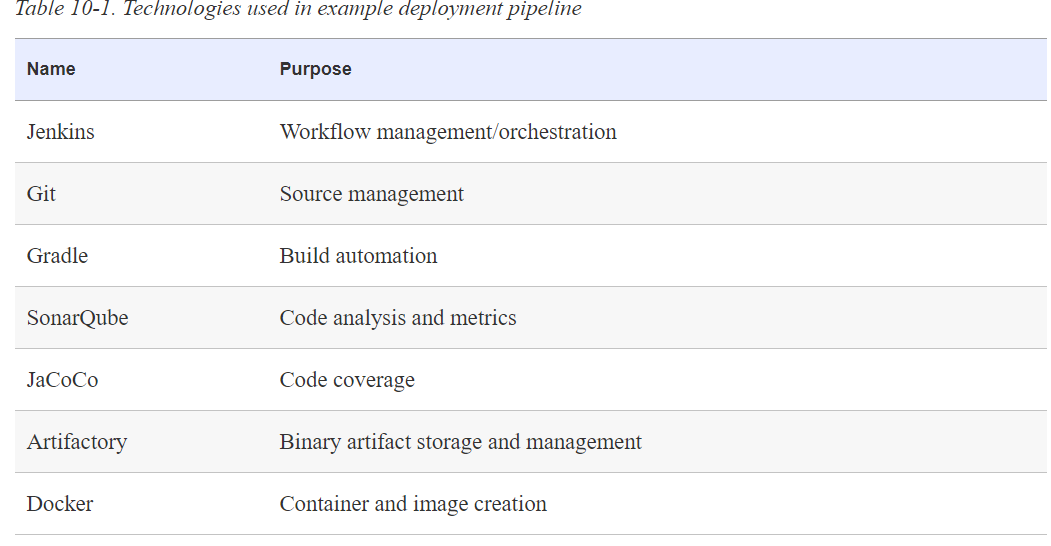




For some of my training courses, I have implemented this type of pipeline with Freestyle jobs in Jenkins. Essentially, each block was implemented by a single Jenkins job that, if successful, chained to the next job.

[Figure 10-2](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch10.html#fig_pip_as_tradJenk_freestyle) shows this set of Freestyle jobs in a traditional Jenkins list view. Note that each job has a descriptive name that maps to a part of the pipeline.

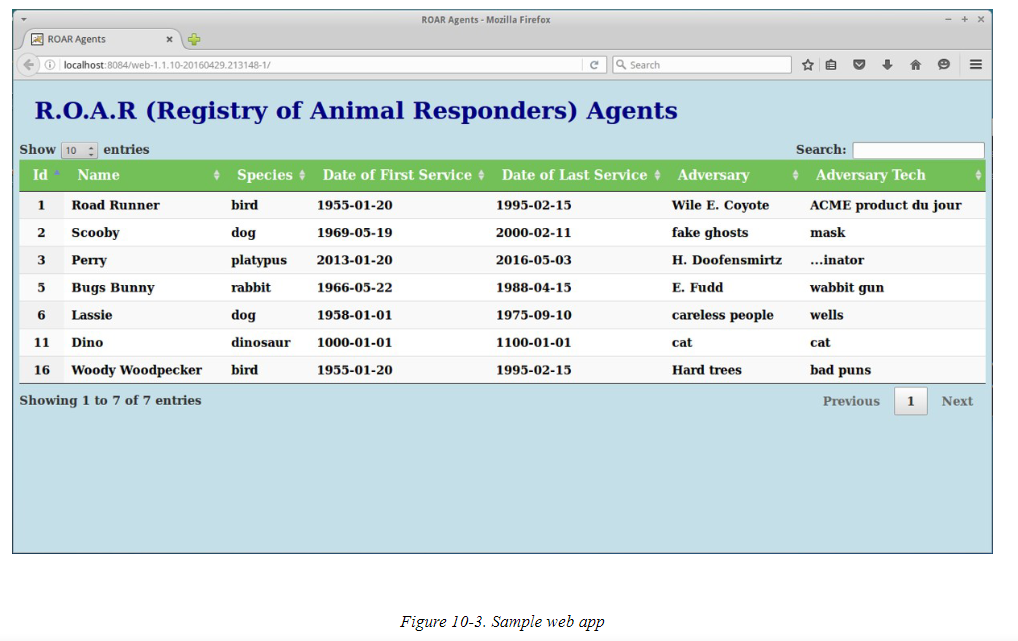




The pipeline performs the following tasks:

* Gets the designated source
* Compiles the source and runs unit tests
* Runs a simplified integration test (using a test database)
* Does code analysis with SonarQube (metrics) and Jacoco (code coverage)
* Assembles an artifact
* Publishes the artifact into the artifact repository (Artifactory)
* Gets the latest artifact out
* Deploys it to a container in Docker for functional testing
* Deploys it for public use

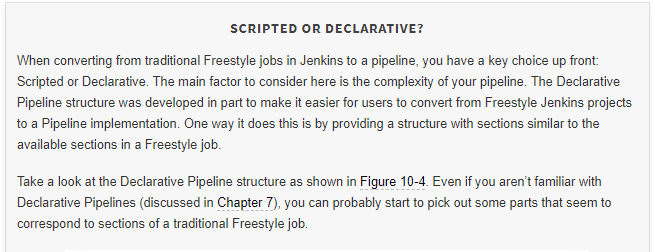
The application itself is a simple web app that uses an underlying MySQL database and exposes a simple REST API. An example of the web app running is shown in below

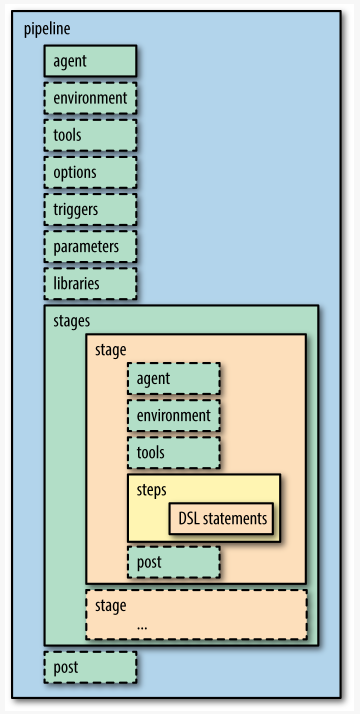


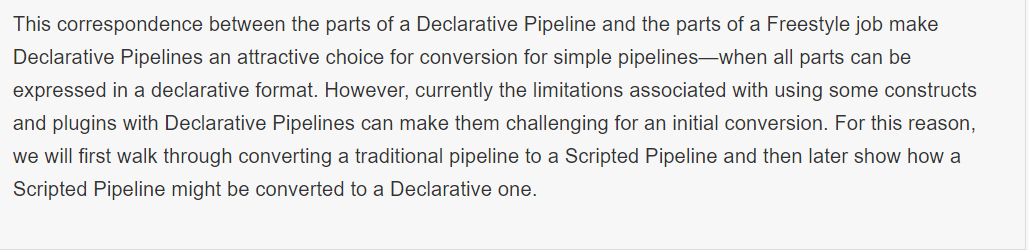
The underlying Gradle project is made up of four separate subprojects: one for the API, one for data access, one for utility code, and one for the web-centric code.

Obviously, this is a very simplistic and contrived pipeline example, but it serves to illustrate the main parts of a continuous delivery pipeline/workflow.

Let’s now dive in and look at converting some of the Freestyle jobs into corresponding stages in a Scripted Pipeline.

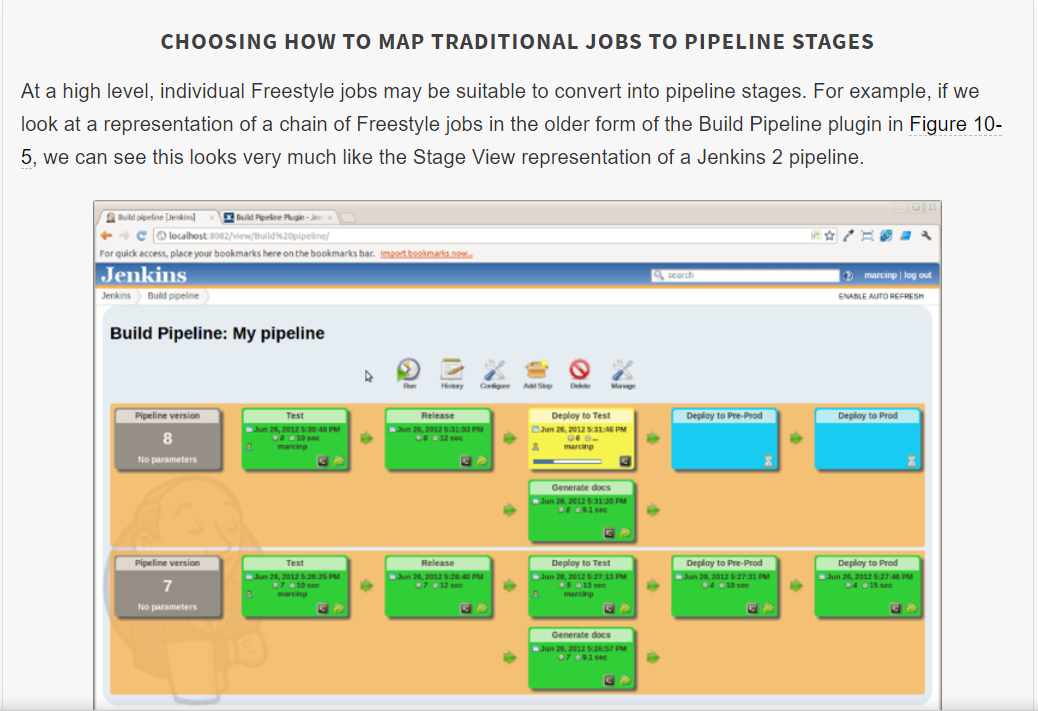






## Source

When you first start looking to convert a Freestyle pipeline, you’ll want to find the section that pertains to the pipeline stage you’re interested in creating. For example, if we wanted to create a Source stage to pull down the source for our pipeline, we would first find the SCM section in our Freestyle project. (In the sample project, retrieving the source code was tied in with another job in the original version, but it works well to have it as its own stage in our pipeline.)



In general, a good guideline if you have multiple Freestyle jobs chained together is to create a corresponding stage for each Freestyle job.

This assumes, however, that your Freestyle jobs are each set up to do one operation. That may not always be the case. For example, some users might have a single job that pulls down source, does the build, and runs unit tests. Another user might have three separate Freestyle jobs chained together for those functions. Both are legitimate use cases, and each may work better in one situation or another.

Both of these cases can also be modeled in pipelines—either as a single stage that pulls down source, does the build, and runs unit tests, or as three separate stages, each doing one of those functions.

When you are first learning about and starting out with pipelines, the recommended approach is to create more separate stages to isolate each type of operation/function rather than trying to do multiple kinds of operations in each stage. The reason for this is to allow for focusing on getting the pipeline code correct for each kind of operation without mixing in other variables. You are going from a guided web form interface to a programming interface, and breaking the process down into smaller chunks can simplify the transition.

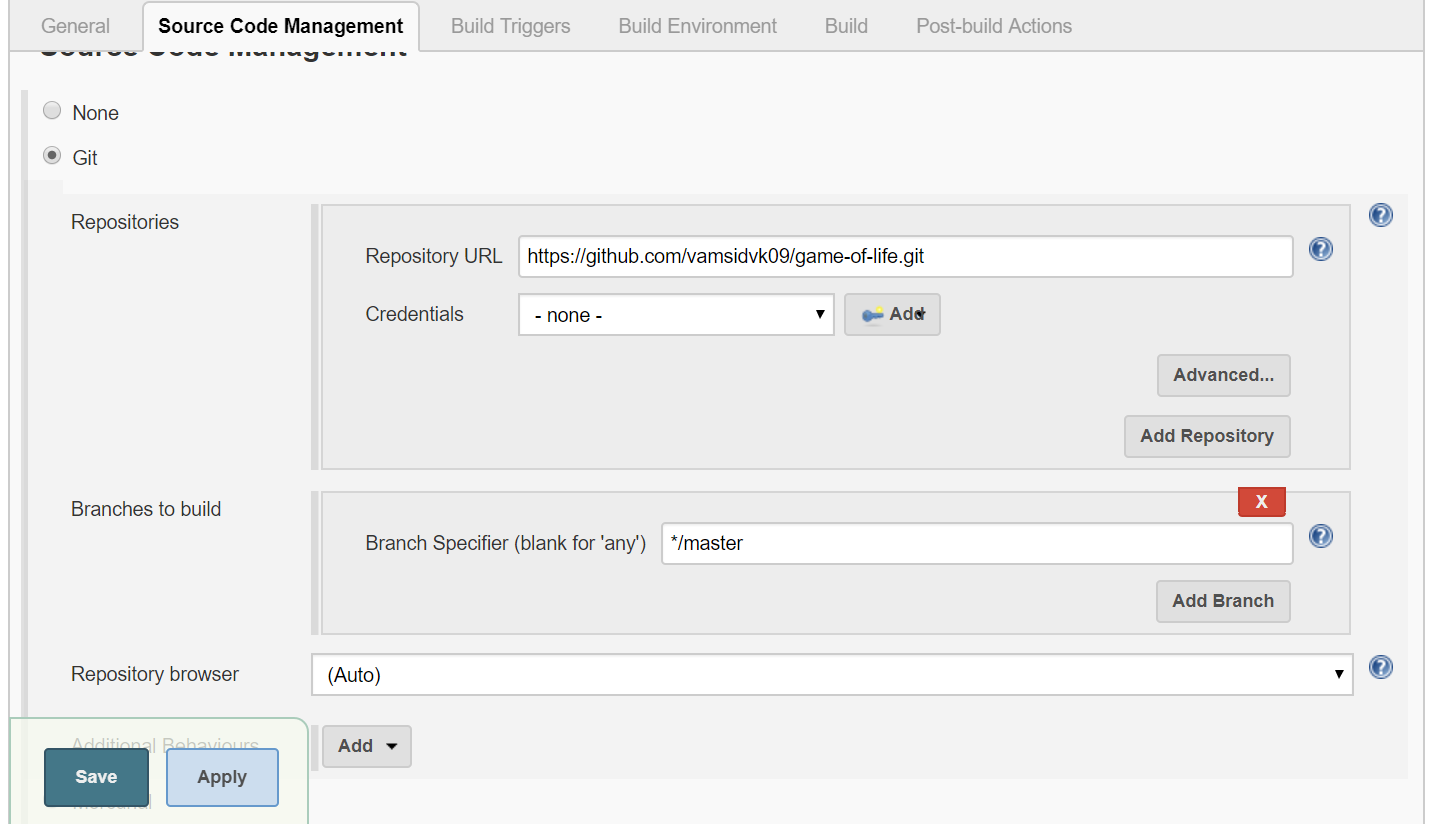
Here, Declarative Pipelines can offer an advantage since they more closely resemble the parts of a Jenkins web form. However, as noted in other places in this chapter, that advantage comes at the price of flexibility. Smaller chunks of functionality also make it easier to quickly isolate problems if a stage fails.

Unfortunately, there isn’t really a great way in Jenkins pipelines yet to temporarily disable a stage, short of commenting it out or removing it during a Replay attempt (see [Chapter 2](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch02.html#CH_The_Foundations) for more information on the Replay functionality). Plowing through error reports/tracebacks can also be challenging, so the more granular approach to isolating functionality in stages can pay off at debug time as well.

This approach can have its challenges, especially if tooling tries to perform multiple functions for you automatically. You may have to override the tooling or specifically force it to do less in a stage. As an example of that, consider the Gradle build tool and its (usually) convenient approach of convention over configuration. For Java projects, if your Java source files are in a standard Maven-style directory structure, Gradle can detect those and automatically build them without you having to tell it about where they are.

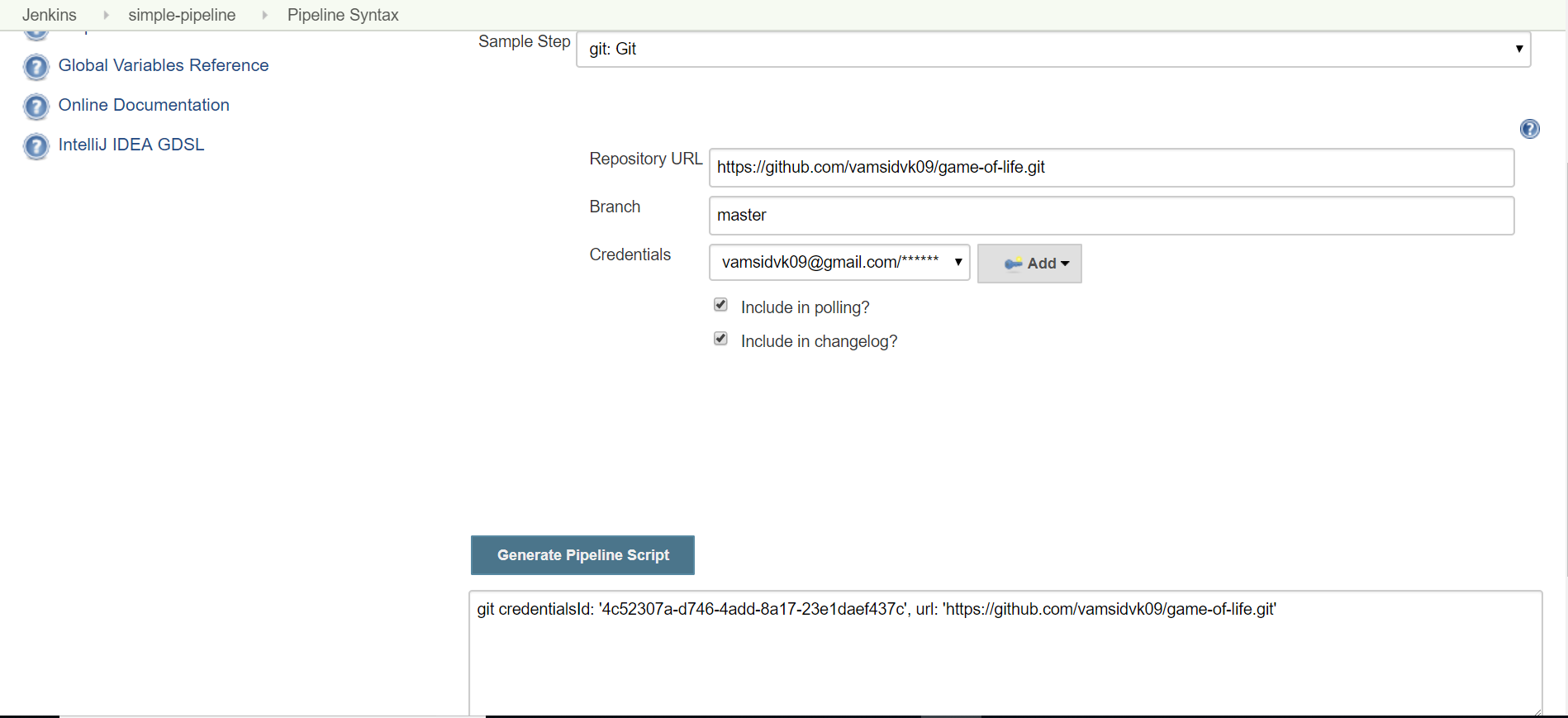
Likewise, if Gradle detects files in a corresponding directory structure for tests, it will assume those are unit tests and build and execute them as part of the same task. So, a build task for Gradle automatically includes an operation to build and execute unit tests if Gradle detects test files in the expected structure. This kind of effect can typically be mitigated via the application. For example, you can supply a −x option to tell Gradle not to run a particular task even if it thinks it can and should.

Below shows the section for a GitSCM configuration in a Freestyle project. (The setup would be similar for other SCMs.)



From this, we can identify the parameters that we need. A reasonable first question when considering converting this to a new pipeline is whether there is an existing DSL step for this functionality that we can leverage in our pipeline.

To determine the answer to that question, we can go to the Snippet Generator (via the Pipeline Syntax link in the left menu of any pipeline job screen) and look for a step with a related name. In this case, we’ll find one named “git” that looks promising. Selecting that step gives us a form with similar fields to the ones we are using. We can then plug in those values and click the Generate Pipeline Script button to get a step for our pipeline.



We can now take the code from the Snippet Generator, wrap it in a stage closure, and wrap that in a simple node step in a pipeline job to try it out. The code could look something like this (assuming we are plugging this into a new pipeline project that we are working on):

node ('master'){

stage('Source'){

git branch: 'master', credentialsId: '4c52307a-d746-4add-8a17-23e1daef437c',

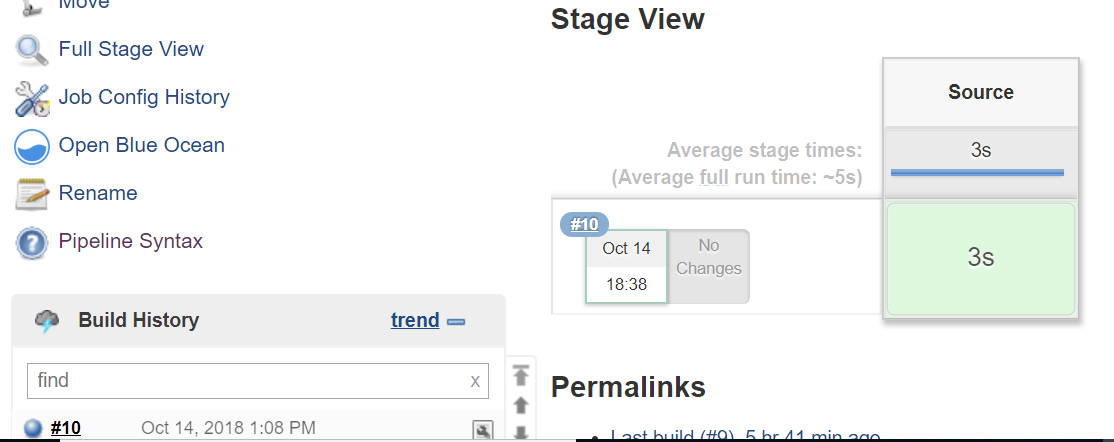
url: 'https://github.com/vamsidvk09/game-of-life.git'

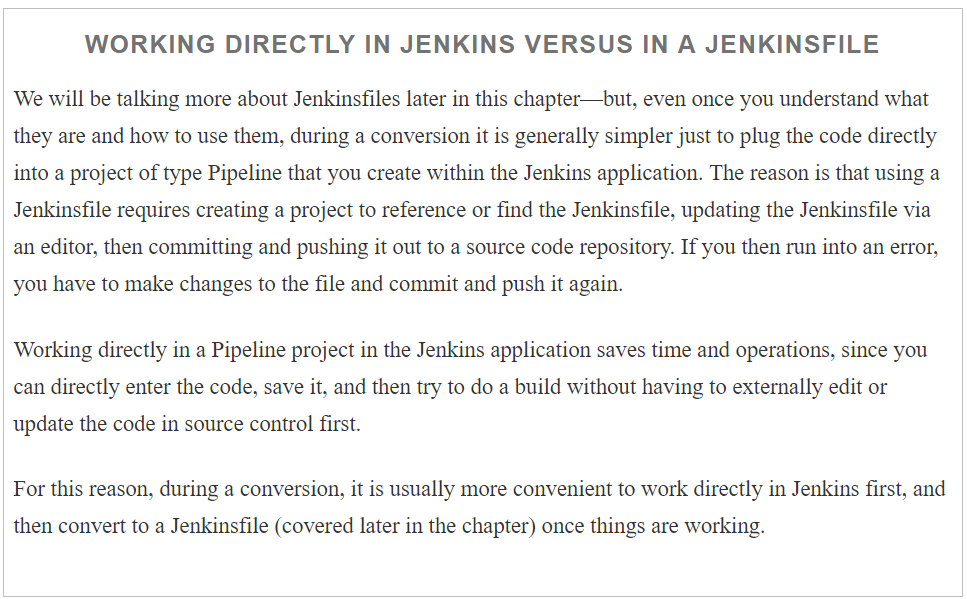
}

}

If we are working directly in Jenkins and putting the code into a Pipeline project, then we can simply save it and tell Jenkins to try to build it now. Jenkins will immediately report any syntax errors, and if there are none, it will execute and build the stage.

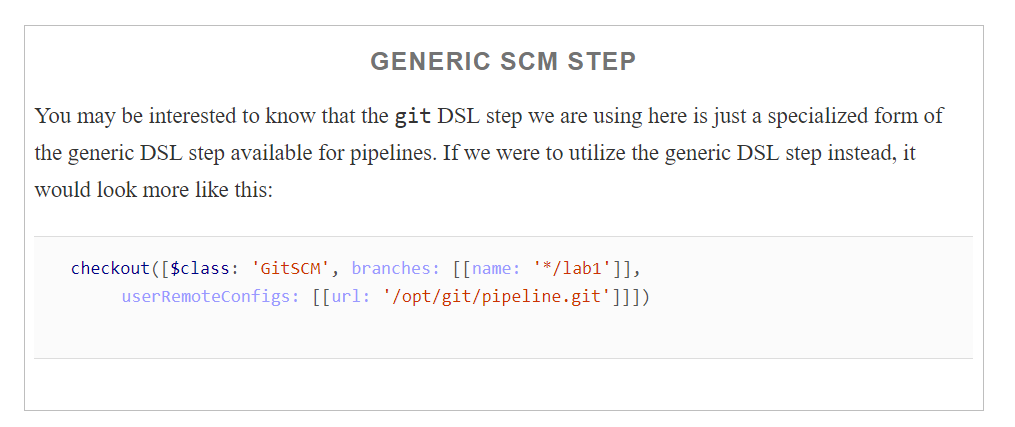
You can easily tell whether the code worked via the Stage View or the Console Log if you need more detail.





As you can see, working with this approach of taking the parameters from the Freestyle job, plugging them into the Snippet Generator, and then putting the result into a stage closure and trying it out is fairly straightforward. It won’t always be that simple, but for plugins that have contributed simple DSL steps (with data for the Snippet Generator), this can often get you close.

More complicated cases may require multiple steps, especially if there is additional configuration to be done, or an environment to use for the operation. In the latter case, you may often have a “with...” DSL block of some kind to use as well. We’ll look at these more complicated cases as we go along.

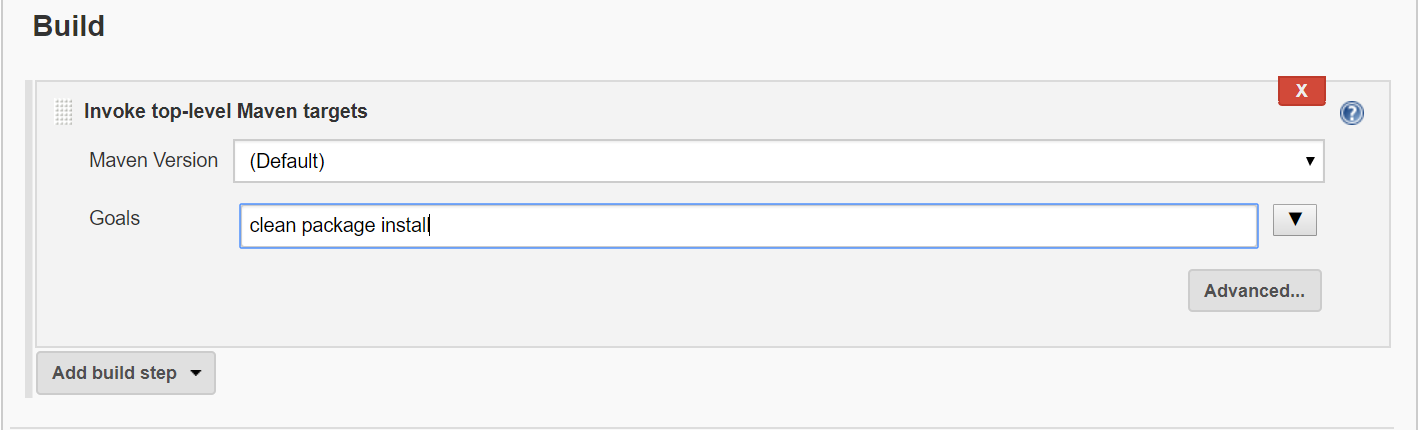


Next, we’ll look at a simple compile step.

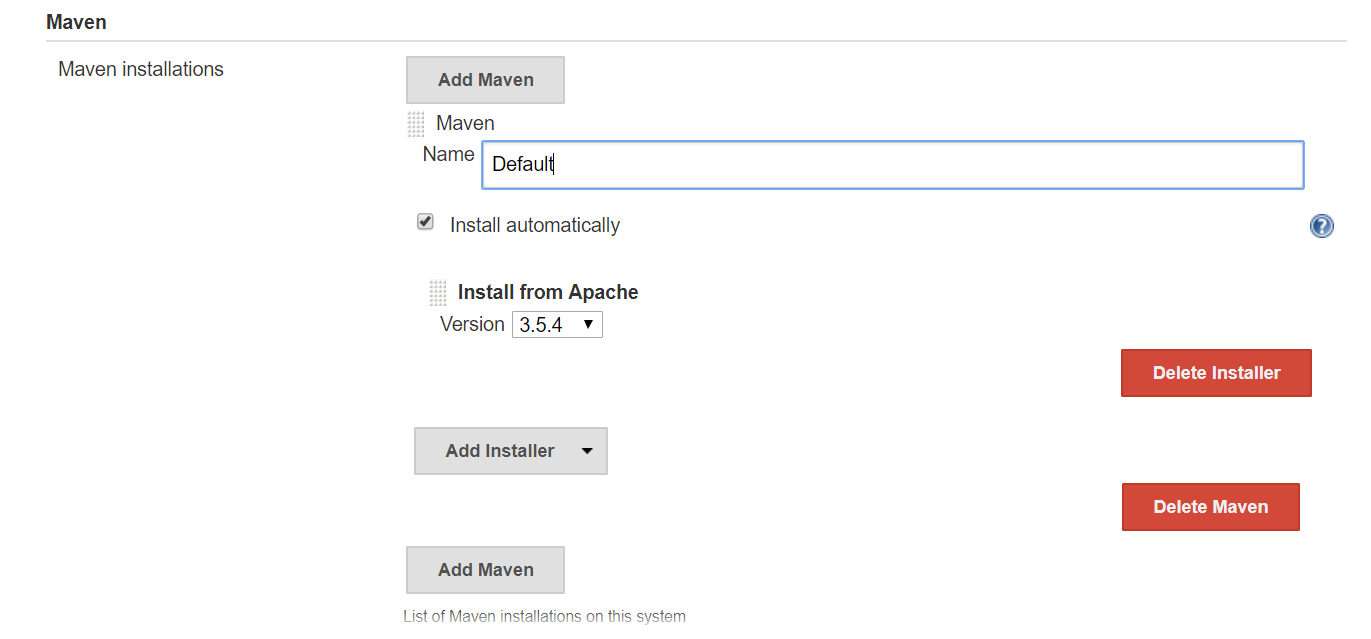
## Compile

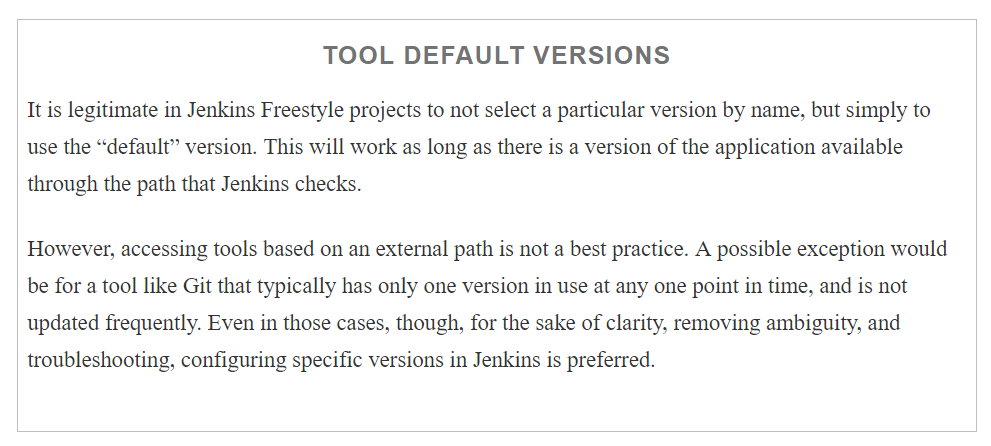
After pulling down the source, most pipelines will have a “build” stage of some sort. In some cases, this may involve more than just a compile action. It could also create deliverables and/or execute defined unit tests, for example.

shows an example of a Freestyle job invoking the build tool Maven to run a series of “tasks” as part of pipeline . We want to look at couple of details here.

First we have selected a specific maven version(Default) in the Maven version field 

This maps back to a particular version of Maven installed on our system, identified by the “Default” name in our Global Tool Configuration. This follows the traditional approach of installing an application: installing the plugin in Jenkins and then giving a name to the global installation to reference that particular installation. An example of the global configuration in Jenkins is shown in

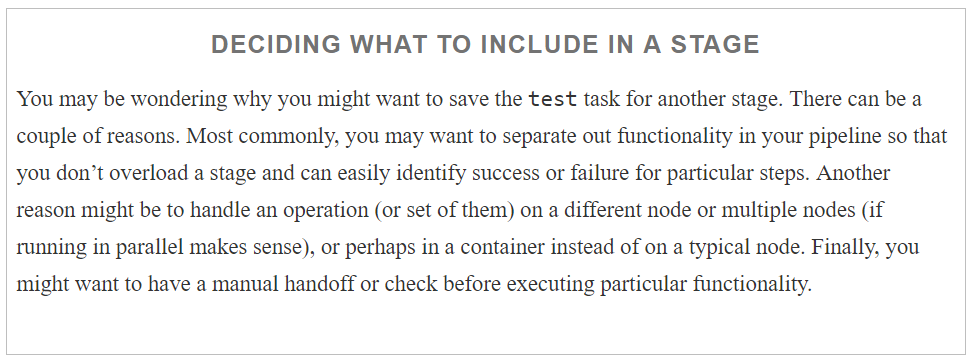




We are also referencing multiple Gradle tasks here (clean, compile, test, artifactoryPublish). A full explanation of each one of these is beyond the scope of this book. However, you can probably tell from the names of several of them what they do. Here is a quick explanation:

* clean cleans out build output.
* compile compiles our Java source.
* test attempts to compile and test any Java test cases that it finds.
* artifactoryPublish attempts to publish designated build types (such as a JAR or WAR for Java) to an “archive repository” such as Artifactory.

So, to convert this section to a pipeline script, we need to first consider whether we want to do exactly the same set of operations in our stage. For simplicity, let’s say that we are only going to execute the clean and compile tasks in our Compile pipeline stage. We do not want it to do the test task (we will save that for another stage) nor will the artifactoryPublish task be needed yet.



So, taking all of this into account, our actual Maven invocation becomes this:

Command : mvn -B clean compile

This looks like a fairly straightforward command (step) to add to our pipeline script, assuming there is a mvn command provided by the Maven plugin for the DSL. To check this, we can go to the Snippet Generator again and look through the list of steps that are available.

As of the time of this writing, there is no step named maven. However, you may notice a step named build. This looks promising at first glance. When you find a step that you think you can use, it is important to confirm that it will do what you think. The easiest way to do that is by clicking on the help icon (the blue button with the question mark in it) that is closest under the step. Doing that in this case shows the explanation of the step as



Looking at this, we can see that this is not a generic step for invoking build tools. Rather, it is a step designed to kick off building entire Jenkins jobs—not what we want.

How then do we invoke our Maven command without a DSL step to do it? In most cases, if you have an executable to run (such as Maven here) and you don’t have a DSL step with that name, that’s an indicator that you need to fall back to running it by using a shell call. And fortunately, the DSL has two commands for executing shell steps:

sh

The command used for executing shell calls on Unix-type systems

bat

The command used for executing shell calls on Windows systems

You can find further information on both in the Snippet Generator (and in [Chapter 11](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch11.html#CH_Integration_with_the_OS)). In our case, we want to leverage the sh DSL step. If we go into the Snippet Generator, find the sh step, and then fill it in with what we think our command should be, we get this generated Groovy script command:

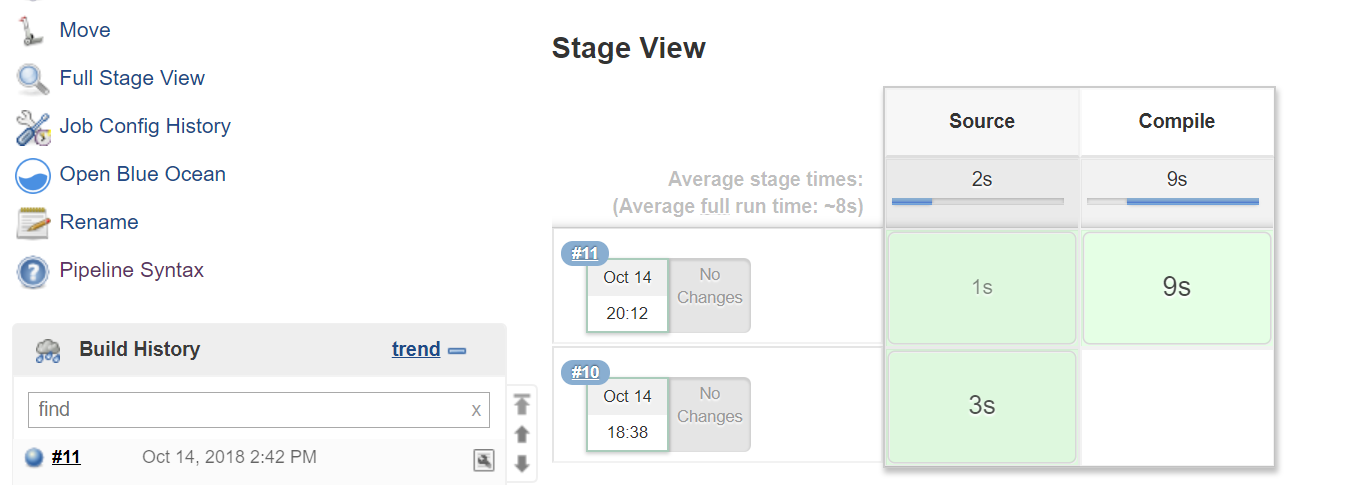
sh 'mvn -B clean compile'

This will work in our pipeline if we have Maven in a path where Jenkins can always find it. However, recall that in our original Freestyle project we were referencing a specific Gradle installation (one is defined globally in our Jenkins system).

stage('Compile'){

sh 'mvn -B clean compile'

}

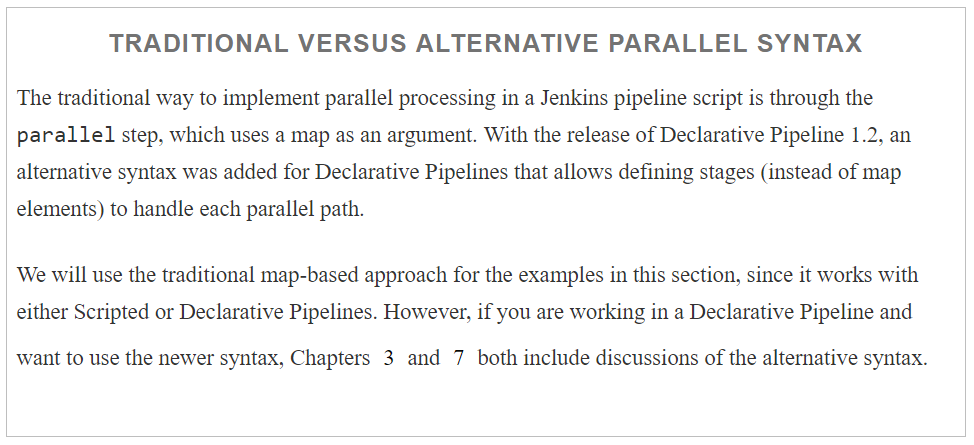


Unit Tests

## Historically, one of the challenges with managing multiple projects in Jenkins was running any of them in parallel. Certain plugins, such as Join and Build Flow, had some mechanisms to support this, but they were not necessarily straightforward to configure. One of the benefits of working in a pipeline environment is the ability to easily script parallel processing using the parallel DSL step.

One case that usually lends itself to this approach is processing large batches of unit tests, especially if they can be broken down into multiple independent sets.

[“Dealing with Concurrency”](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#SEC_deal_with_concurrency) discusses setting up parallel processing in more detail, but we’ll touch on the main points here and see how we might apply it to a large set of simple tests.



The key to working with the traditional parallel DSL step is understanding that it takes a map as an argument. The programmatic keys to the map are just labels to identify the different branches, while the values contain the actual code blocks to execute. As a means of distributing the load, we can use a node block around each codeblock to ensure each branch runs on a different node.

Consider, for example, a set of tests for a subproject gameoflife-core of our sample gradle project.  For simplicity, these unit tests are written in Java programs named **WhenYouCreateACell.java & WhenYouCreateAGrid.java.** If we have two defined nodes avilable, node1 and node2, we might choose to run all Using Gradle, we can pass the set of tests to run via a system parameter using the -D test.single=<pattern> option.

Wrapping the parallel step in a stage (currently if parallel is used, it should be the only step in a stage) could result in code like the following:

stage('Unit Test') {

tester1: { node ('master'){

sh "'${tool 'gradle4.8'}/bin/gradle' -D test.single=WhenYoucreate\*

:core:test"

}}

tester1: { node ('master'){

sh "'${tool 'gradle4.8'}/bin/gradle' -D test.single=WhenYouP\*

:core:test"

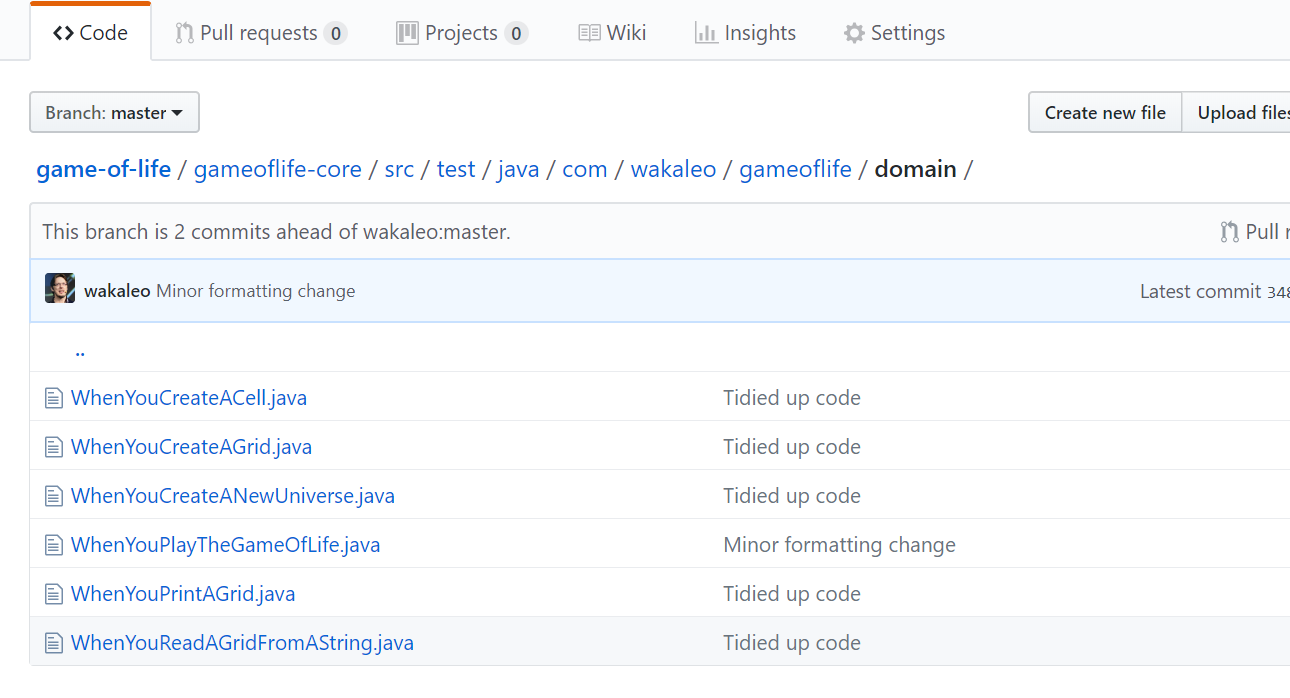
}}

}

Although this sample code that has errors

Notice that we simply call the parallel DSL step and pass the map to it. Our map consists of two branches with the keys tester1 and tester2 and the blocks of code as the values. Each code block, in turn, consists of a node specification and then a call to the shell to run the specific Gradle command. The Gradle command identifies a subset of tests and calls the test task in the *api* subproject.

Another way to code this would be to declare a map, then run through some code to fill in the map elements. Afterwards, the parallel step can be invoked, passing just the name of the map. (See [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow) for an example of this.)



### DISTRIBUTING CONTENT ACROSS NODES

When coding something to run in parallel, it makes sense to use different nodes (or node classes) for the different branches to distribute the load. However, this also presents a requirement that you may not have thought about—how to get the same content on multiple nodes so that all the needed pieces are there. Of course, one solution would be to have a repeated Source step on each node to pull down the code. However, this is redundant and expensive in terms of cycles and resources.

Fortunately, the DSL provides a simple solution—the stash and unstash steps. (We discussed these in [Chapter 3](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch03.html#CH_Pipeline_Execution_Flow), but will do a brief repeat of stash and related topics here for ease of reference.) As the names imply, we can use these commands to create a “stash” of content from one node and then “unstash” that content onto other nodes. The syntax is straightforward. The basic form of the stash step takes a set of comma-separated includes (or excludes) and a name:

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

The idea here is that we designate a set of included or excluded files via names and/or patterns. The stash itself is also given a name to refer to it by. To simplify things, we can just add the stash step immediately after we do the source code retrieval within the Source stage:

stage('Source'){

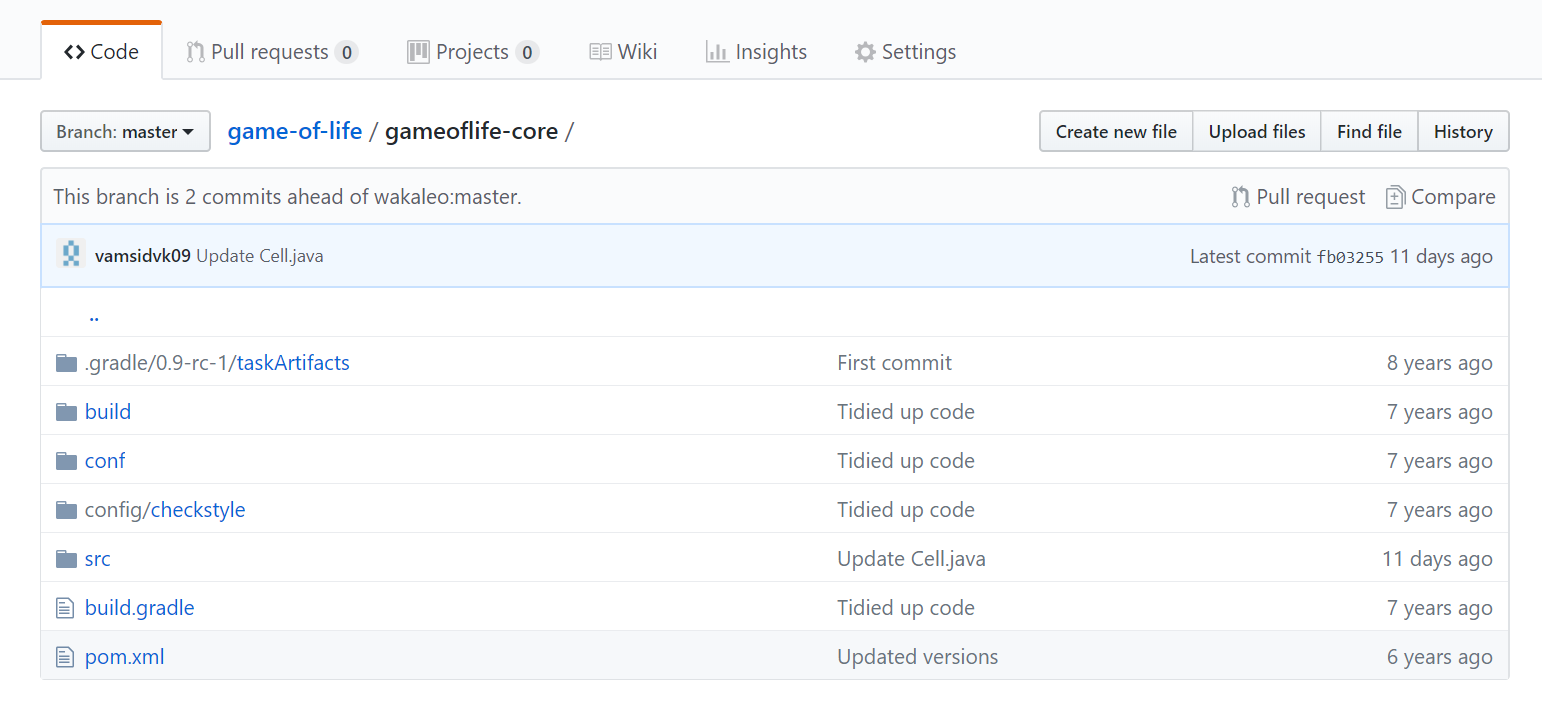
      git branch: 'master', credentialsId: '4c52307a-d746-4add-8a17-23e1daef437c',

       url: 'https://github.com/vamsidvk09/game-of-life.git'

    stash includes: 'gameoflifecore/\*\*, build.gradle

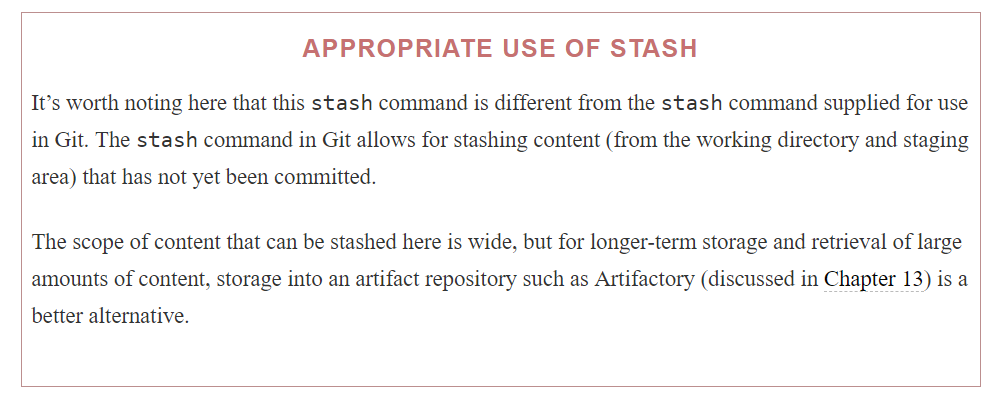
', name: 'testreqs'

}



Then, when we need to retrieve the set of files in any other part of our pipeline, we can simply pass the name of the stash to the unstash command. This can be done in a different stage, node, or branch of a parallelstatement. The format is simply:

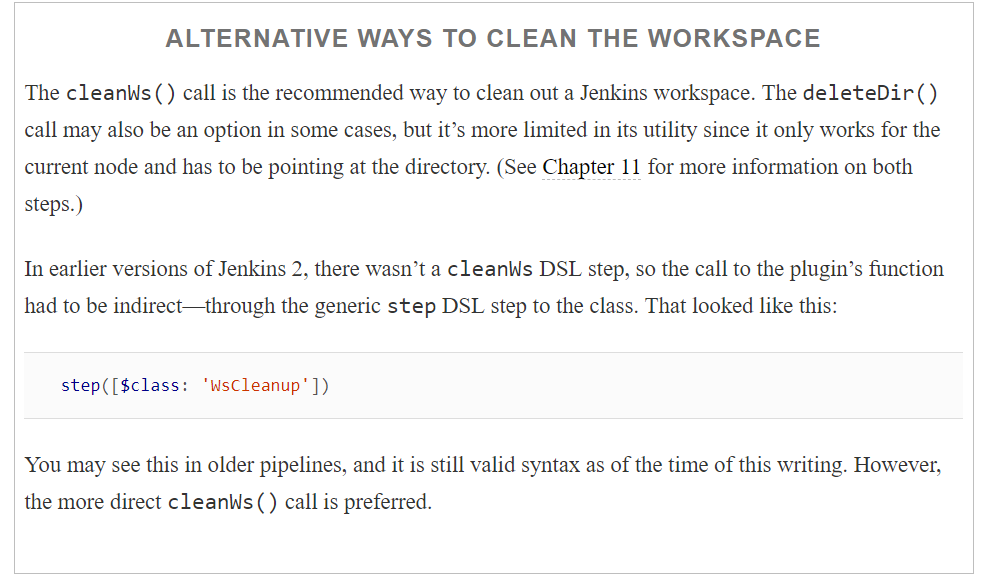
unstash "<name>"



### CLEANING OUT WORKSPACES

When using commands like stash and running across multiple nodes, it’s a good idea to clean out the workspace each time first. Jenkins does not guarantee that workspaces will be clean or that they will persist over time.

If we have the [Workspace Cleanup plugin](https://plugins.jenkins.io/ws-cleanup) installed, we can use the cleanWs step to accomplish this.

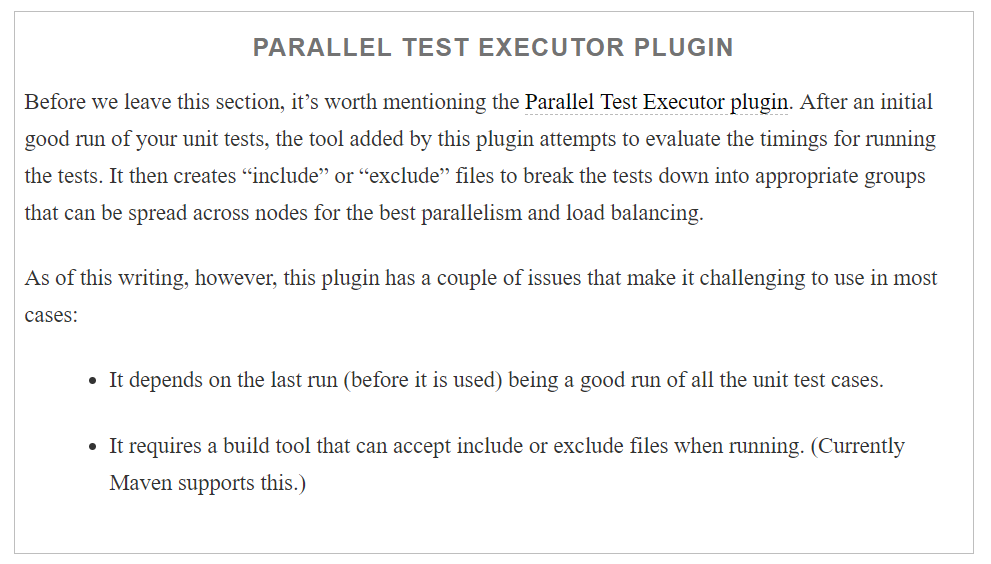


Adding in the elements to clean up the workspace and unstash the needed pieces results in our parallel unit testing stage looking like the following



Above is just a sample example

When executing this part of the pipeline, if you look in the Console Log, you’ll be able to see the interspersed commands for the tester1 and tester2 branches as they execute in parallel.



In the next section, we’ll look at how to incorporate credentials in the context of another common pipeline stage: integration testing.

## **Integration Testing**

Integration testing can take many forms. In our example Freestyle pipeline, we have a job that leverages Gradle SourceSets and defines an integrationTest task similar to the Gradle default test task that is provided by the Java plugin (we used the test task for the unit testing in the previous section).

We’ll have more to say about Gradle SourceSets shortly. But another technique that we’re leveraging here (which is more widely applicable) is using a test database for our web app to run against. In particular, we are creating a test database with a single command that redirects input into MySQL from an external SQL file. The basic command in our Freestyle job is a shell step and looks like this:

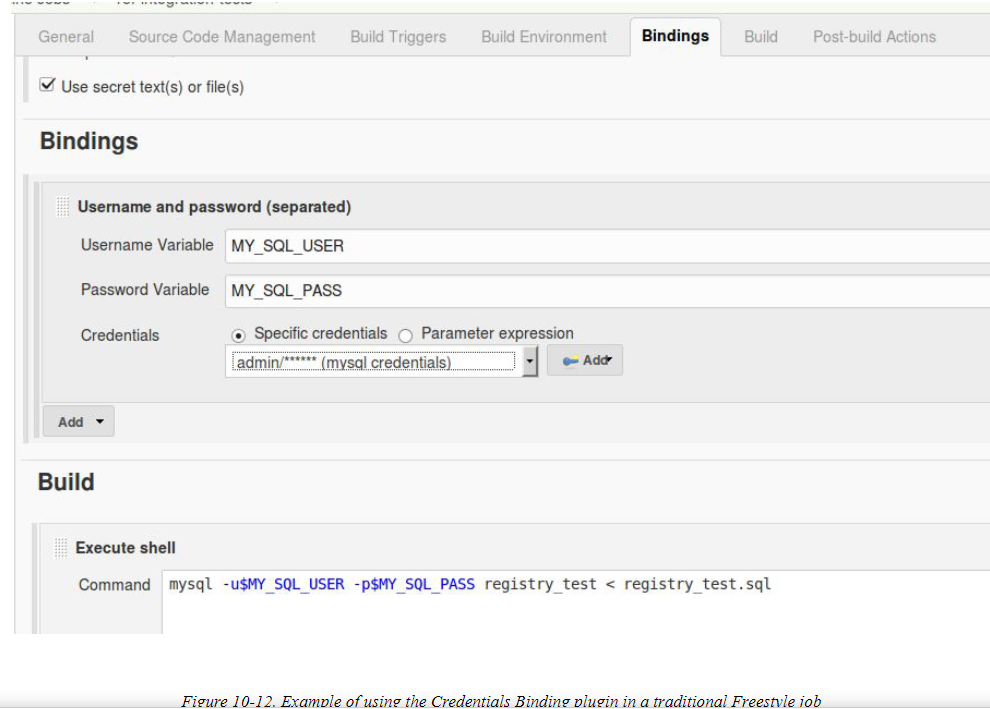
mysql -u<username> -p<password> registry\_test < registry\_test.sql

What is interesting about this is how we supply the credentials of username and password to the command. Traditionally we have had a few choices:

* Hardcode the username and password
* Manually set them as environment variables
* Supply them via parameters
* Read them from an external file
* Leverage injection via a plugin, such as the [Credentials Binding plugin](https://plugins.jenkins.io/credentials-binding)

Obviously, the first option is completely insecure and a bad practice. The second option is slightly better but still exposes too much information. The third option is dependent on input each time, which is less than ideal in an automated environment. The fourth option provides some isolation, but requires maintaining data outside of Jenkins.

The last option represents our most direct and secure way to use the credentials defined in Jenkins for this type of access. Essentially, the Credentials Binding plugin allows us to bind the credentials (such as username and password) that we have already set up in Jenkins to variables that we can then pass to our build steps. An example use case is shown in [Figure 10-12](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch10.html#fig_ex_using_credbind_plugin_TFJ).



The Jenkins pipeline DSL also includes a step that allows us to use the Credentials Binding plugin in a pipeline: the withCredentials step. Like the Freestyle version, this step takes a type of credentials binding to use and then allows the user to specify variables to receive the actual values of the credentials. The variables can then be used within the block in place of the credentials, preventing the values of the credentials from being exposed.(See [Chapter 5](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch05.html#CH_Access_and_Security) for more details and examples on creating and using credentials.)

In our case, we’ll assume that we’ve set up a credential named mysql\_credentials that contains the separated username and password for accessing MySQL databases on our system. We can then instantiate a step that uses that binding and dereferences it into two separate environment variables to be used where the credentials are needed in statements that we put in the enclosed block.

Translating our example from the Freestyle project would look like this:

withCredentials([usernamePassword(credentialsId: 'mysql\_credentials',

passwordVariable: 'MY\_SQL\_USER', usernameVariable: 'MY\_SQL\_PASS')])

{

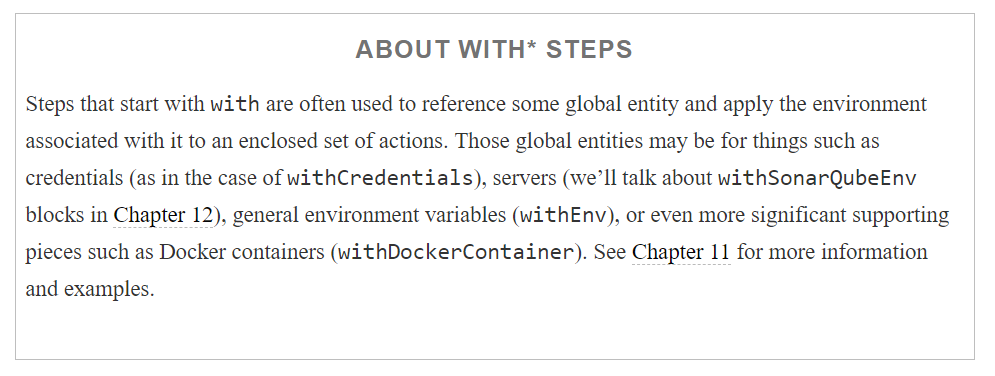
sh "mysql -u$MY\_SQL\_USER -p$MY\_SQL\_PASS registry\_test < registry\_test.sql"

}

withCredentials(...) {

sh "..."

}



The remaining piece of our integration testing stage relies on a mechanism called *SourceSets* that the Gradle build tooling supports. A SourceSet in Gradle is simply a way to define a set of source files with their own environment and structure. When working with Java files (and the corresponding Java plugin for Gradle), Gradle by convention is set up with two default SourceSets, one for the main project source (called main) and one for any associated test cases written in Java (called the test SourceSet). We used basic functionality of the testSourceSet in the Gradle invocations for the parallel unit test processing earlier in this chapter.

Gradle allows us to define the classpath, output path, directory structure, and so forth for a SourceSet so that Gradle can compile and access them correctly. One of the other abilities we have with Gradle SourceSets is the ability to create new SourceSets based on existing ones, with modified characteristics—a sort of SourceSet “inheritance.” For our pipeline with Gradle, we have created a new integrationTest SourceSet based on the default test SourceSet and a functionalTest SourceSet based on the new integrationTest SourceSet. We won’t go into more detail than that here since this isn’t a Gradle text, but the bottom line is that once we have the registry database in place for our integration testing (via the withCredentials step), we can execute our integration tests by invoking Gradle to run the new integrationTest task. We can do that simply by invoking it through a shell call:

sh "'${tool 'gradle3'}/bin/gradle' integrationTest"

Here again, note the use of the tool step to get the location of our Gradle\_HOME path, and the mixture of double and single quotes necessary to make this all work.

At this point, we have the core initial stages of our converted pipeline complete. We can pull down the source, build it, and test it on multiple levels. The primary remaining parts of our pipeline require more detailed integration with their respective external applications. To keep the scope and content of this chapter reasonable, we defer details on integrating/migrating with those applications to their own chapters—but we’ll briefly cover the high-level ideas and approach of working with these technologies in the next sections.

## Migrating the Next Parts of the Pipeline

Thus far, we’ve covered integration in two key technology areas, source management (with Git) and builds and testing (with Gradle). These stages form the minimum pipeline that we need in order to establish that our code is syntactically correct and the functionality works in isolated testing.

From here, we want to establish successive levels of confidence in our code by incorporating tools such as source code analysis (via gathering metrics with SonarQube) and deploying to more comprehensive environments for testing (such as a Docker container). Along the way, we will need to ensure we can store and retrieve versioned artifacts produced by our pipeline (done via Artifactory in our case).

Each of the technologies we use for these tasks and their respective integrations with Jenkins (and pipelines-as-code) deserve more extensive treatment than we can give them in this one chapter, so the book contains separate chapters for these. As such, we’ll only touch on these areas at a high level. For more details, refer to [Chapter 12](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#CH_Integrating_Analysis_Tools), which covers integration with SonarQube, and [Chapter 13](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch13.html#CH_Integrating_Artifact_Management), which covers integration with Artifactory.

### SOURCE CODE ANALYSIS

Although testing gives us some assurance that we have written code that does what we want, it doesn’t provide any feedback on the quality of the source code itself. Source code analysis can provide that for any code going through our pipeline.

Source code analysis generally refers to a set of quality metrics related to using best practices, producing code that is insulated against known failure conditions, assessing technical debt, determining code coverage through testing, etc.

The set of metrics is wide and varied. Scores on metrics are obtained by measuring compliance of the code against a set of rules. Thresholds can be defined for each metric area. A set of thresholds can be treated as a “quality gate”—a pass/fail criterion for code being analyzed in a pipeline.

SonarQube is one application that provides this kind of analysis. To integrate with Jenkins, we have to have a SonarQube server set up, the SonarQube plugin installed in Jenkins, and a standalone program called a “scanner” or “runner” installed and configured.

We can define a webhook in SonarQube to notify Jenkins after the analysis is complete. This same notification will let Jenkins know whether the code passed or failed the quality gate.

[Chapter 12](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch12.html#CH_Integrating_Analysis_Tools) describes in detail how to integrate with SonarQube. We’ll also look at how to use a code coverage tool called Jacoco (Java Code Coverage) that integrates with Jenkins to provide data on how well our tests are testing the source code in our projects.

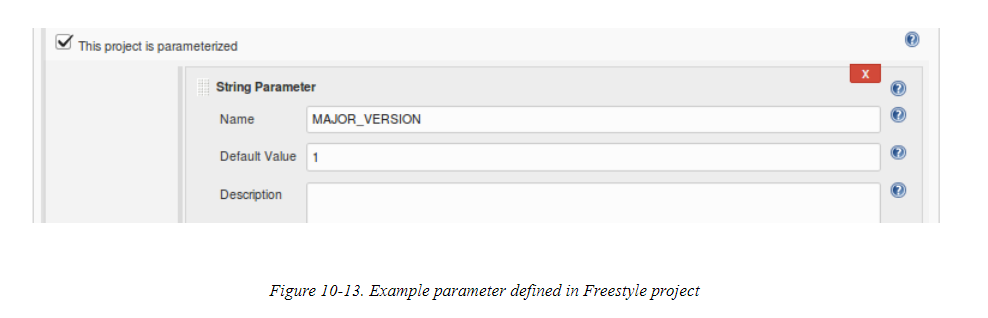
### INCORPORATING AN ARTIFACT REPOSITORY

An artifact repository is used to store, manage, and track binary artifacts, just as a source management repository does for source code. It allows users and automated processes, such as jobs in Jenkins or stages in a Jenkins pipeline, to ensure they are working with the desired version of an artifact.

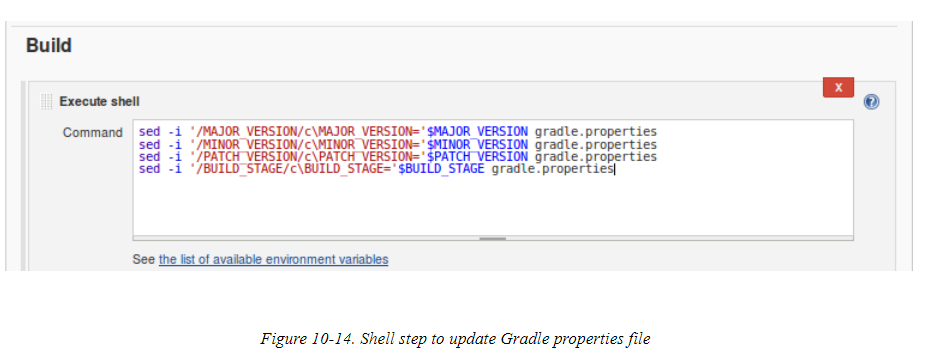
Artifacts in this case can be external dependencies that are needed for some operation or artifacts generated by the current processes for later use or distribution. Repositories that store the dependencies are referred to as resolution or resolver repositories. Repositories used to store generated content for later use or distribution are referred to as distribution repositories. These repositories may be in any of a number of standard formats, such as Maven, Ivy, or Gradle—the important aspect is the versioning. Let’s dive into more detail on that, as it demonstrates some other techniques that may be useful as you convert your pipeline.

#### **Setting version information with parameters**

In our original pipeline based on the Freestyle jobs, we used parameters as a way to override default versioning information

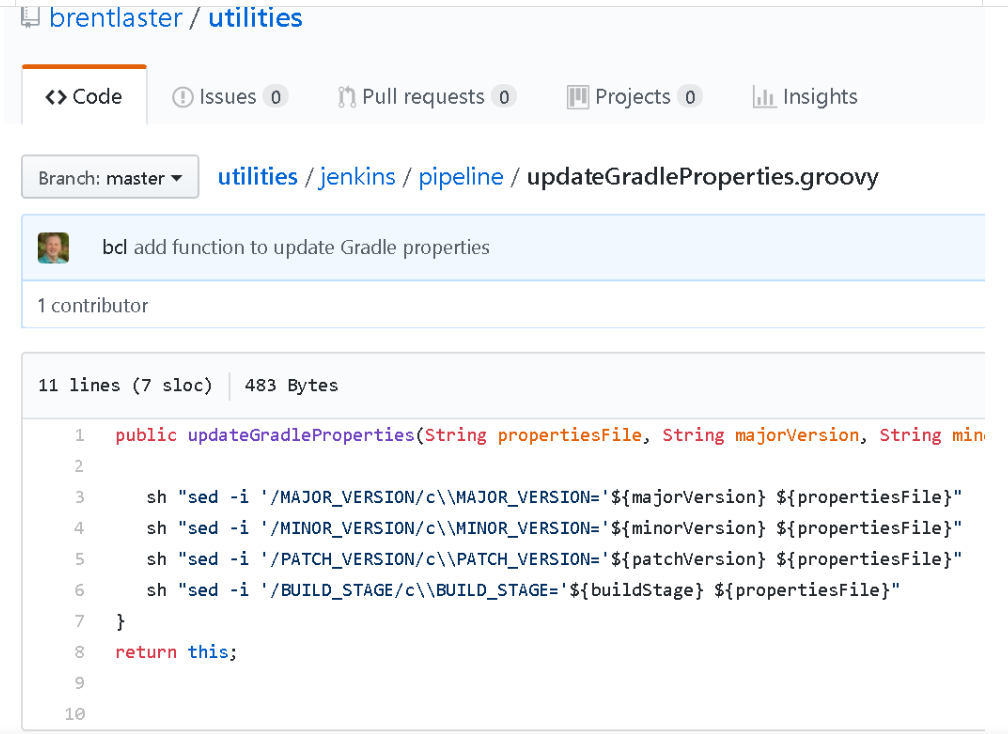


We then used those values (or the defaults, if not overridden) to set the values for the version of the generated WAR file that we placed in the artifact repository. This was done by manipulating properties for Gradle in a gradle.properties file. Ideally, we would have passed these values as properties to Gradle via some clear integration with the web form. However, the Gradle integration with Freestyle projects did not have a good way to do this. So, instead, we fell back to calling a set of shell commands that used the Unix sed utility. Basically, the commands did text substitution to get the desired values into the properties file. The step in the traditional Jenkins job looked like



In our Scripted Pipeline, we could also use a series of direct shell commands via the sh step. However, for this conversion, we’ve chosen to actually put these commands in a separate script and store that script in a different source code repository. This is mostly illustrative at this point, as it shows how commands can be stored and loaded from a remote site.

[Figure 10-15](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch10.html#fig_shell_steps_sepscrip_SCM_repo) shows the shell steps encapsulated in a file stored on GitHub.



This serves to abstract out operations in case we want to later change the implementation. It also avoids hardcoding steps in the pipeline and allows more open sharing of code.

From within our new pipeline, we can load the script from its remote location using the Pipeline Remote Loader plugin. With this plugin installed, the DSL has a fromGit method to load content stored in a Git repository. (There are other methods for other SCMs as well.) So, we can load the function and then execute a call to it in our pipeline. Putting this as the start of our Assemble stage would yield code like the following:

stage('Assemble') { *// assemble WAR file*

**def** workspace = env.WORKSPACE

**def** setPropertiesProc = fileLoader.fromGit('jenkins/pipeline/

      updateGradleProperties','https://github.com/brentlaster/utilities.git',

      'master', **null**, '')

      setPropertiesProc.updateGradleProperties(

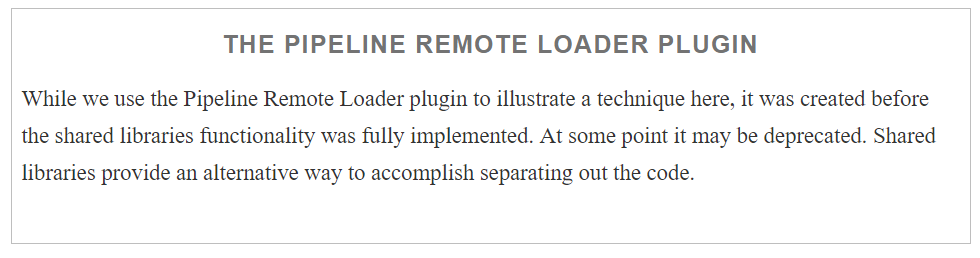
        "${workspace}/gradle.properties",

        "${params.MAJOR\_VERSION}",

        "${params.MINOR\_VERSION}",

        "${params.PATCH\_VERSION}",

        "${params.BUILD\_STAGE}")



Handling the versioning of artifacts is an important part of integration with an artifact repository. Beyond that is the overall integration with the application you choose.

For our pipeline examples in [Chapter 13](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch13.html#CH_Integrating_Artifact_Management), we will discuss integration with Artifactory Community Edition, the free version of one of the most common artifact management tools. We’ll see in that chapter how it is configured globally in Jenkins and integrated in the traditional web forms of a Freestyle job, including fields to define common elements such as resolution and deployment repositories, and options about what to publish into repositories.

To translate this to a pipeline environment, we define variables to point to the server and repositories that we want. Also, depending on the type of the integration, we have specific objects that represent the functionality of the combined Artifactory and build projects. These combined objects then allow us to invoke Artifactory functionality within the build application in direct calls.

[Chapter 13](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch13.html#CH_Integrating_Artifact_Management) contains all of the details on the Artifactory integration. [Chapter 14](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch14.html#CH_Integrating_Containers) contains details on the other main component of our pipeline, which we’ll look at briefly next—working with containers.

### USING CONTAINERS IN A PIPELINE

Containers are becoming more and more ubiquitous in terms of being used in pipelines today. By “containers” here, we really mean the higher-level orchestration applications for Linux Containers (LXC). These allow us to define and run multiple isolated Linux systems within a single Linux container, achieving many of the goals of a VM without the overhead.

Of course, the most popular of these applications for defining and using containers is Docker. In fact, there are whole pipeline applications that are built around only using Docker containers.

Historically, integration with Docker in a Freestyle project centered around using it as an agent via the Docker Cloud plugin or invoking it directly via shell commands. Within a Jenkins 2 pipeline, we now have four options for integration with Docker:

* Configured as a “cloud,” meaning running a standalone Jenkins agent provided by the Docker plugin
* Running as an agent via constructs provided by Declarative Pipelines
* Inside the pipeline, using the docker global variable (provided by the Docker Pipeline plugin)
* Directly invoking Docker via a shell call

The Docker (cloud) plugin is still available to pipeline creators, but within a pipeline, we can also create new Docker containers from images and execute commands in them easily

The traditional pipeline we are migrating from used Docker as an isolated, repeatable environment for deploying our artifact into for functional testing. The migration to our Scripted Pipeline goes further. [Chapter 14](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch14.html#CH_Integrating_Containers) shows how we can also create an image with a different version of a tool than the one we have configured globally in Jenkins and easily pass our pipeline commands to that container, for execution in the isolated environment it provides.

This is most easily done with a DSL with block called withDockerContainer, but the Docker integration with Jenkins 2 also provides a built-in Docker global variable that has an inside method that can be used. The nice thing about both of these constructs in the new Jenkins DSL is that they automatically handle a lot of the setup and teardown of using Docker for you. For example, they can automatically pull an image if not already available, start up containers, and mount Jenkins workspaces as volumes in the container (assuming filesystem access).

One other key aspect of using Docker in Jenkins 2 comes into play when working with Declarative Pipelines. The DSL provides a number of ways to easily define agents based on Docker containers. There are methods to create agents based on a particular Docker image, as well as a Dockerfile. These mechanisms greatly simplify integrating containerization more widely in your pipeline.

