What Are Jenkins Shared Libraries And Why You Should Use Them

Part One of my series on Jenkins Shared Libraries

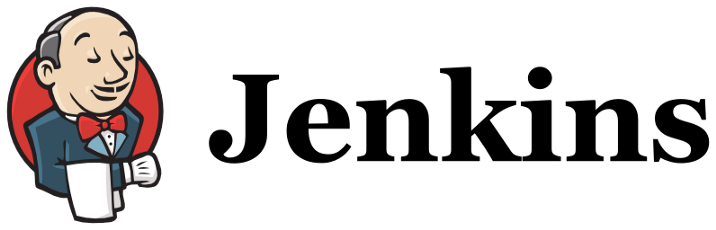
[](https://medium.com/@werne2j?source=post_page-----5ba3d072536a----------------------)

[Jake Wernette](https://medium.com/@werne2j?source=post_page-----5ba3d072536a----------------------)

Follow

[Aug 15, 2019](https://medium.com/@werne2j/jenkins-shared-libraries-part-1-5ba3d072536a?source=post_page-----5ba3d072536a----------------------) · 6 min read

Jenkins



<https://jenkins.io/>

Jenkins is one of the most common continuous integration/deployment servers being used today. Jenkins can express continuous integrations or deployment tasks as a “pipeline” comprised of one or more “stages” and is, typically, written to a file commonly named “Jenkinsfile”. A Jenkinsfile contains the order of stage execution and can contain logic for these stages. A Jenkinsfile can also be checked into source control.

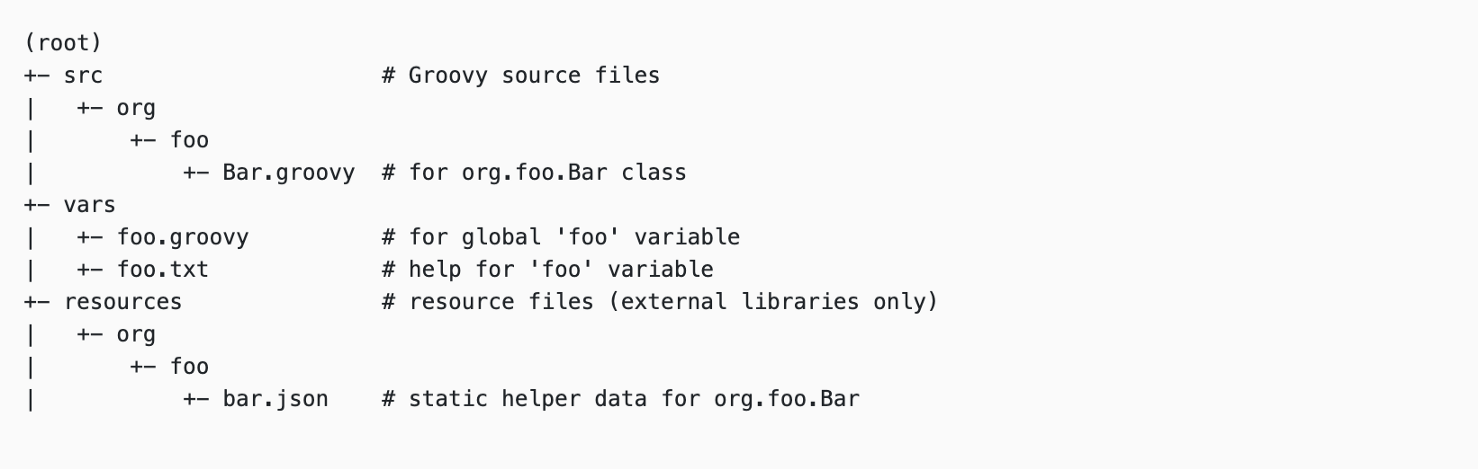
When writing a Jenkinsfile, it is very easy for it to get bloated and can end up being many hundreds of lines of code. A Jenkinsfile does not have the concept of hierarchy or inheritance, so any common functions or logic must be copied from one file to another in order to share that logic. Hopefully this all sounds terrible. Don’t worry, I am going to show you how this problem can be fixed with Jenkins Shared Libraries.

**What is a Jenkins Shared Library?**

Jenkins Shared Libraries are written in [Groovy](https://groovy-lang.org/) and allow you to create common sets of logic, and share that among teams/projects/organizations. Instead of “copy and pasting” the code from some other Jenkinsfile, you can simply load a library in to Jenkins and every pipeline job on that Jenkins master has access to that shared library.

When writing a shared library, the code must be structured in a [specific way](https://jenkins.io/doc/book/pipeline/shared-libraries/#directory-structure).

https://miro.medium.com/max/60/1*YXEY9RIxW5RnWHACZFbNeA.png?q=20



<https://jenkins.io/doc/book/pipeline/shared-libraries/#directory-structure>

The “vars” directory contains groovy scripts that, in Jenkins parlance, are called “global vars”. These are the steps that you want to expose to be used in the Jenkinsfile Let’s say you want to create a shared step that handles the deployment of some code, you would add a file called “deploy.groovy” to the “vars” directory.

(root)  
src/  
 ...  
vars/  
 deploy.groovy  
 deploy.txt

This would allow the deploy step to be called like

node() {  
 stage(‘deploy’){  
 deploy() // deploy var from shared library  
 }  
}

**Why you should be using Jenkins Shared Libraries in Your Pipelines**

Imagine that within your team you have three projects that all deploy code in the same way. You wouldn’t want to have to write that logic three different times. This is a violation of the DRY (Don’t repeat youreslf) principal. This means that, if the process changes, then each team has to go and update their Jenkinsfile to accommodate the changes. This usually includes copying what someone else has done, and then pasting the code into their project and then making small tweaks to accommodate their project.

With a shared library, the code to deploy a project is written once and then made available to all other teams as a simple update to the version of the library. This separation of concerns will allow teams to put all of their focus on writing code, instead of worrying about how to write the code to deploy code, do automatic releases, etc, which in turn saves time and money.

Another advantage of shared libraries is that they promote collaboration between teams. Sometimes teams can have their heads down and not be aware of what others are doing. This often leads to the same code being written multiple times. Shared libraries can bridge the gap between teams that do similar things and allow those teams to work together on a shared piece of code that can benefit them as well as any other team.

Let’s take a look at a Jenkinsfile to deploy a javascript application.

**Original Jenkinsfile**

node {  
 checkout scm stage('Install') {  
 sh 'npm install'  
 } stage('Test') {  
 sh 'npm test'  
 } stage('Deploy') {  
 if (deploy == true) {  
 sh 'npm publish'  
 }  
 }  
}

Now this alone does not look too bad. The problem comes when your team has 3 different projects that all deploy a javascript application the same way. So instead of just copying and pasting the code between the 3 projects, we can take advantage of a shared library.

We can take the code from the Jenkinsfile and wrap it in a global variable so that this whole pipeline is available as a Jenkins step called “buildJavascriptApp” for anyone else in their own Jenkinsfile.

def call() {  
 node {  
 checkout scm stage('Install') {  
 sh 'npm install'  
 } stage('Test') {  
 sh 'npm test'  
 } stage('Deploy') {  
 if (deploy == true) {  
 sh 'npm publish'  
 }  
 }  
 }  
}

Using this global variable turns that original Jenkinsfile into:

**New Jenkinsfile**

buildJavascriptApp deploy: true

We just turned a 14 line Jenkinsfile into a 1 line Jenkinsfile. This may not seem like much now, but when the Jenkinsfiles start to grow in size, it quickly become clear how advantages shared libraries can be.

Another way to do this is to take advantage of [Groovy Closures](http://www.groovy-lang.org/closures.html). This allows a project to use the global shared pipeline while also adding to it. So if your project performs the javascript build process but also sends a message to slack to notify when a build has succeeded or failed, you can extend the library step and pass the logic you want to perform after the common logic finishes.

def call(Closure body) {  
 node {  
 checkout scm stage('Install') {  
 sh 'npm install'  
 } stage('Test') {  
 sh 'npm test'  
 } stage('Deploy') {  
 if (deploy == true) {  
 sh 'npm publish'  
 }  
 } body()  
 }  
}

We see that this global variable accepts a Closure as a parameter. This is the extra block of code that you would want to perform after the common stages. With a shared variable like this, we can use it like:

buildJavascriptApp deply: true {  
 stage('Notify') {  
 slackSend(...) // send a message to slack  
 }  
}

Hopefully these examples show how powerful shared libraries can be, as well as the flexibility they provide to teams and their continuous integration efforts using Jenkins.

**Caveats**

One of the perceived downfalls of writing a Jenkins Shared Library is that the code is written specifically for Jenkins. The argument goes that, if at some point you or your team wants to move to another CI system, it would require rewriting the code contained in the shared libraries. This process would have to happen, most likely, if moving to a new system. An argument could be made for writing shared logic in a way that can more easily be used by other systems such as shell scripts, Ansible playbooks, or CLI tools.

**Conclusion**

Jenkins Shared Libraries are a great way to help keep Jenkinsfile concise and readable. These libraries reduce the headache and time to manually update multiple Jenkinsfiles when something might change in a certain process. They also allow the Jenkinsfile of each project to focus on “what” needs to happen, and lets the shared libraries take care of “how” it happens.

Thank you for reading Part 1 of my Jenkins Shared Library series.

In the [2nd part of my Jenkins Shared Library series](https://medium.com/@werne2j/how-to-build-your-own-jenkins-shared-library-9dc129db260c) I will show you exactly how you can create your own shared library in a step by step tutorial.

**My Jenkins Shared Library Series**

**[What are Jenkins Shared Libraries and why you should use them](https://medium.com/@werne2j/jenkins-shared-libraries-part-1-5ba3d072536a)**

[Part 1 of my series on Jenkins Shared Libraries](https://medium.com/@werne2j/jenkins-shared-libraries-part-1-5ba3d072536a)

[medium.com](https://medium.com/@werne2j/jenkins-shared-libraries-part-1-5ba3d072536a)

**[How to build your own Jenkins Shared Library](https://medium.com/@werne2j/how-to-build-your-own-jenkins-shared-library-9dc129db260c)**

[Part Two of my series on Jenkins Shared Libraries](https://medium.com/@werne2j/how-to-build-your-own-jenkins-shared-library-9dc129db260c)

[medium.com](https://medium.com/@werne2j/how-to-build-your-own-jenkins-shared-library-9dc129db260c)

**[Unit Testing a Jenkins Shared Library](https://medium.com/@werne2j/unit-testing-a-jenkins-shared-library-9bfb6b599748)**

[Part Three of my series on Jenkins Shared Libraries](https://medium.com/@werne2j/unit-testing-a-jenkins-shared-library-9bfb6b599748)

[medium.com](https://medium.com/@werne2j/unit-testing-a-jenkins-shared-library-9bfb6b599748)

**[Collecting Code Coverage for a Jenkins Shared Library](https://medium.com/@werne2j/collecting-code-coverage-for-a-jenkins-shared-library-c2d8f502732e)**

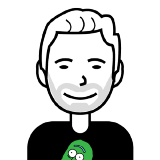
[Part Four of my series on Jenkins Shared Libraries](https://medium.com/@werne2j/collecting-code-coverage-for-a-jenkins-shared-library-c2d8f502732e)

[medium.com](https://medium.com/@werne2j/collecting-code-coverage-for-a-jenkins-shared-library-c2d8f502732e)

53

* [Jenkins](https://medium.com/tag/jenkins)
* [Jenkins Pipeline](https://medium.com/tag/jenkins-pipeline)
* [Shared Libraries](https://medium.com/tag/shared-libraries)

53 claps

[](https://medium.com/@werne2j?source=follow_footer--------------------------follow_footer-)

WRITTEN BY

[**Jake Wernette**](https://medium.com/@werne2j?source=follow_footer--------------------------follow_footer-)

Follow

https://github.com/werne2j