How To Build Your Own Jenkins Shared Library

Part Two of my series on Jenkins Shared Libraries

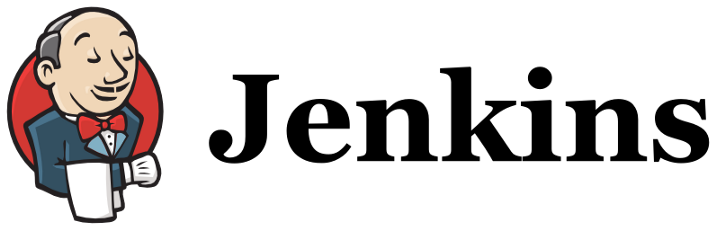
[Jake Wernette](https://medium.com/@werne2j?source=post_page-----9dc129db260c----------------------)

[Jake Wernette](https://medium.com/@werne2j?source=post_page-----9dc129db260c----------------------)

Follow

[Sep 19, 2019](https://medium.com/@werne2j/how-to-build-your-own-jenkins-shared-library-9dc129db260c?source=post_page-----9dc129db260c----------------------) · 9 min read

https://miro.medium.com/max/27/1*1xY4qH6IqSRQ7SJnVJDDZw.png?q=20



<https://jenkins.io/>

In the [first entry of my Jenkins Shared Library series](https://medium.com/@werne2j/jenkins-shared-libraries-part-1-5ba3d072536a), I described what a shared library is and why you should start taking advantage of them. In this entry, I will be doing a step-by-step walkthrough of how you can build your own [Jenkins Shared Library](https://jenkins.io/doc/book/pipeline/shared-libraries/).

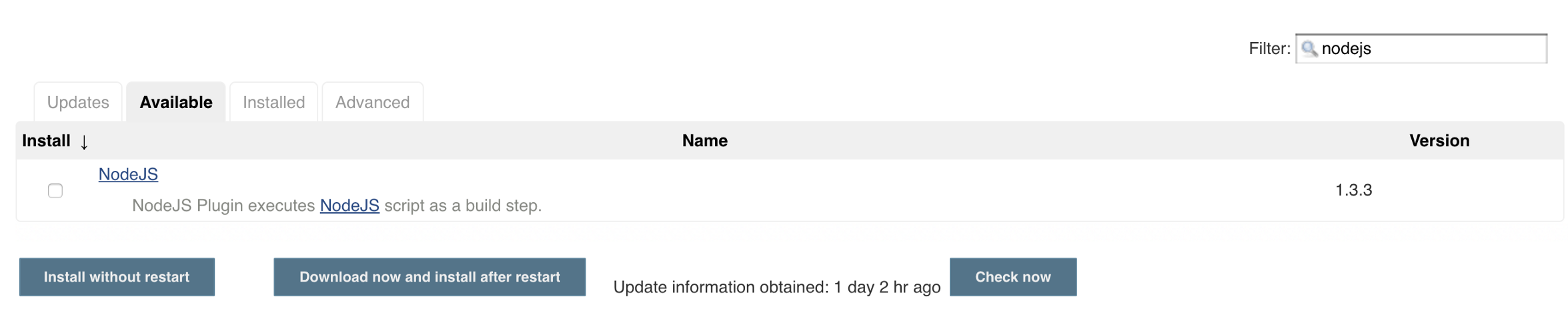
**DISCLAIMER**: I will not be showing you how to setup Jenkins in this tutorial. For the rest of the post I will assume you have Jenkins running. You can follow the [official Jenkins documentation](https://jenkins.io/doc/book/installing/) for getting Jenkins setup

**Prerequisites**

This tutorial requires that we have Node JS installed on our Jenkins instance. So, before we dive into the creation of the Jenkins Share Library, I am going to show you how we can get that setup.

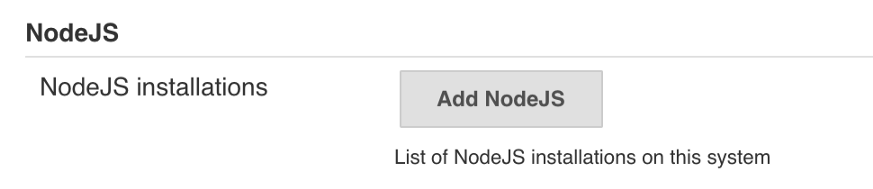
* From the Jenkins home screen, navigate to Manage Jenkins and then Manage Plugins.
* Click the Available tab, and then using the filter, filter by nodejs.

https://miro.medium.com/max/27/1*BqaU52-6U4wp_rWWelYq4Q.png?q=20



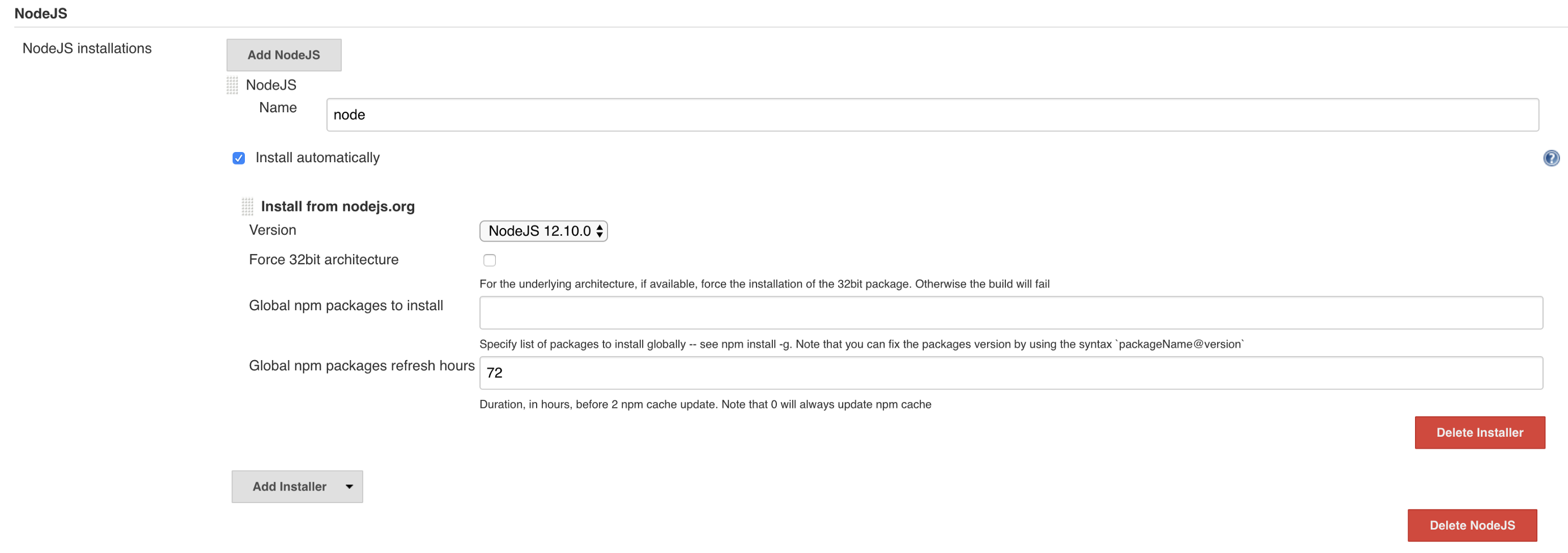
* Select the checkbox under where it says Install and then click, Install without restart.
* When the installation finishes, navigate back to Manage Jenkins.
* Navigate to Global Tool Configuration and then scroll down until you see Node JS.

https://miro.medium.com/max/27/1*XltMFQiMyaOECHvUbH18ig.png?q=20



* Click Add NodeJS.
* Fill out the form to look like this:

https://miro.medium.com/max/27/1*g7XjOBlYnmurwFyvlGZcIg.png?q=20



* Click Save. Now Node JS should be available for us to use on Jenkins!

**Getting Started**

The first thing to do is to plan out the variables for our shared library. In this example, we are going to look at a couple of different scenarios:

* The first scenario we are going to handle is sharing a full pipeline. For a team that may have a similar deployment strategy, sharing a full pipeline can be beneficial. This can allow teams to have Jenkinsfile as simple as one line of code.
* The second scenario is going to be an example of sharing some sort of helper code. This is useful in situations where different teams may not share the same Jenkinsfile but they may share blocks of code.

In the first part of this series, I described the directory structure that is required for shared libraries. We will need a src and vars directory.

To start, open your terminal and change into the directory that you want the code of your shared library. I like to have all my code in a Developer directory. Once in your directory of choice, we will make the directories that are necessary for our shared library and then cd into the project directory.

An example of this is:

mkdir -p ~/Developer/jenkins-shared-library/{src, vars}  
cd ~/Developer/jenkins-shared-library

The src directory should look like a standard Java directory structure. So for our example we will add the subdirectories org/example to src. To do this, run mkdir -p src/org/example.

This is what your directory structure should look like at this point:

├── jenkins-shared-library  
│ ├── src  
│ │ ├── org  
│ │ │ ├── example  
│ ├── vars

**Creating the Pipeline Variable**

The first variable we are going to create is going to be called buildJavascriptApp, if this looks familiar, that is because this is an example that I used in the first part of the series.

With your editor of choice, create a file inside the vars directory called buildJavascriptApp.groovy and paste in this code:

def call(Map config=[:], Closure body) {  
 node {  
 git url: "<https://github.com/werne2j/sample-nodejs>" stage("Install") {  
 sh "npm install"  
 } stage("Test") {  
 sh "npm test"  
 } stage("Deploy") {  
 if (config.deploy) {  
 sh "npm publish"  
 }  
 } body()  
 }  
}

Don’t worry if some of this does not make sense, we will walk through every piece of code.

Starting at the top of the file we have the call() method. This is required by Jenkins, it tells Jenkins what method to run when the var is called from a Jenkinsfile. So all variables must have a call method.

We have two parameters for our variable, a map called config and a closure called body. The config parameter allows us to pass named parameters to the call method. In this example, we only care about a parameter called deploy, this tells us if we want to run the publish command or not.

What this would look like when being called would be:

buildJavascriptApp deploy: true

The other parameter is a [Groovy closure](https://groovy-lang.org/closures.html). This allows consumers the ability to “extend” this variable by passing a block of code we would want to execute, and this can be done by calling body() just like a normal method.

When calling the variable from a Jenkinsfile and passing a closure, it would look like this:

buildJavascriptApp {  
 stage(“extra step”) {  
 …  
 }  
}

The rest of the variable consists of basic Jenkins steps. We are using the git step to checkout the source code of our sample Node JS project. Then we use the sh step to execute npm install and npm test. The last stage checks if the deploy variable has been set too true and if it is, runs the npm publish command.

That is it for the first variable!

**Creating the Helper Variable**

For this variable, we are going to create a notify variable that wraps code that sends notifications via Slack or email. In your editor of choice, create a file called notify.groovy under the vars directory. In this example we are are not going to write a full functioning notification variable but instead some simple if/else logic, however the variable could be easily adjusted to handle complex real world logic.

For this variable we are going to take advantage of some code that will live in the src directory. In src/org/example, create a file called Constants.groovy, this is going to be a class that exports constants for us to use in our variable.

Open, Constants.groovy and add this code:

package org.exampleclass Constants {  
 static final String SLACK\_MESSAGE = "Sending Slack Notification"  
 static final String EMAIL\_MESSAGE = "Sending Email"  
}

The src directory can be used for things like this or even for housing most of the logic for your shared library, you can read more about that in part 3 of the series.

Now, back to the variable, open notify.groovy and add this code:

import org.example.Constantsdef call(Map config=[:]) {  
 if (config.type == "slack") {  
 echo Constants.SLACK\_MESSAGE  
 echo config.message  
 } else {  
 echo Constants.EMAIL\_MESSAGE  
 echo config.message  
 }  
}

This variable accepts a parameter called config. Like the first example, this is a map and allows us to pass named parameters to the variable.

We are also importing the Constants class from the src directory, so that we can access the constants that we created.

The logic in this variable is very simple, If the type of notification we want to send is slack, we will print out Sending Slack Notification, otherwise we will assume that an email is to be sent and will print out Sending Email. We also print out a message that is passed to the variable.

In a Jenkinsfile, this var can then be called like:

notify type: “slack” message: “a slack notification”

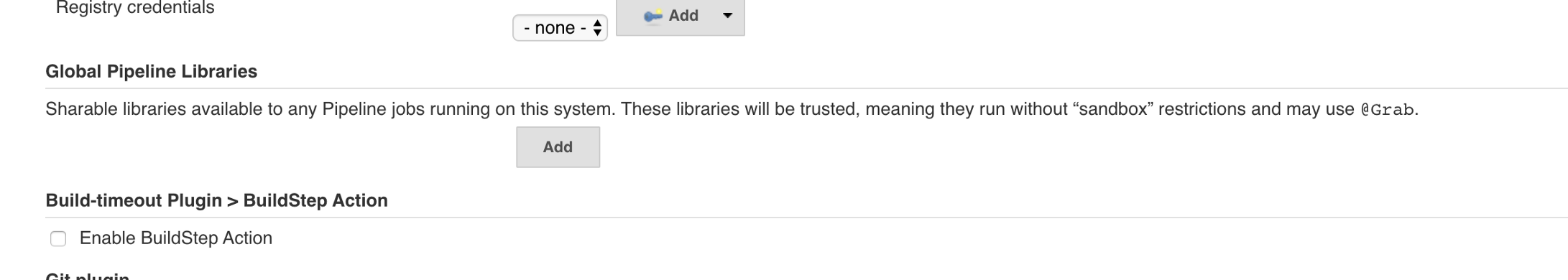
That is it for our second variable! We now have two different variables that we can use!

**Loading our Shared Library into Jenkins**

Now is time to load the shared library into Jenkins and see if they work!

* Click on Manage Jenkins -> Configure System.
* Scroll until you see Global Pipeline Libraries, this is where we will load the library.

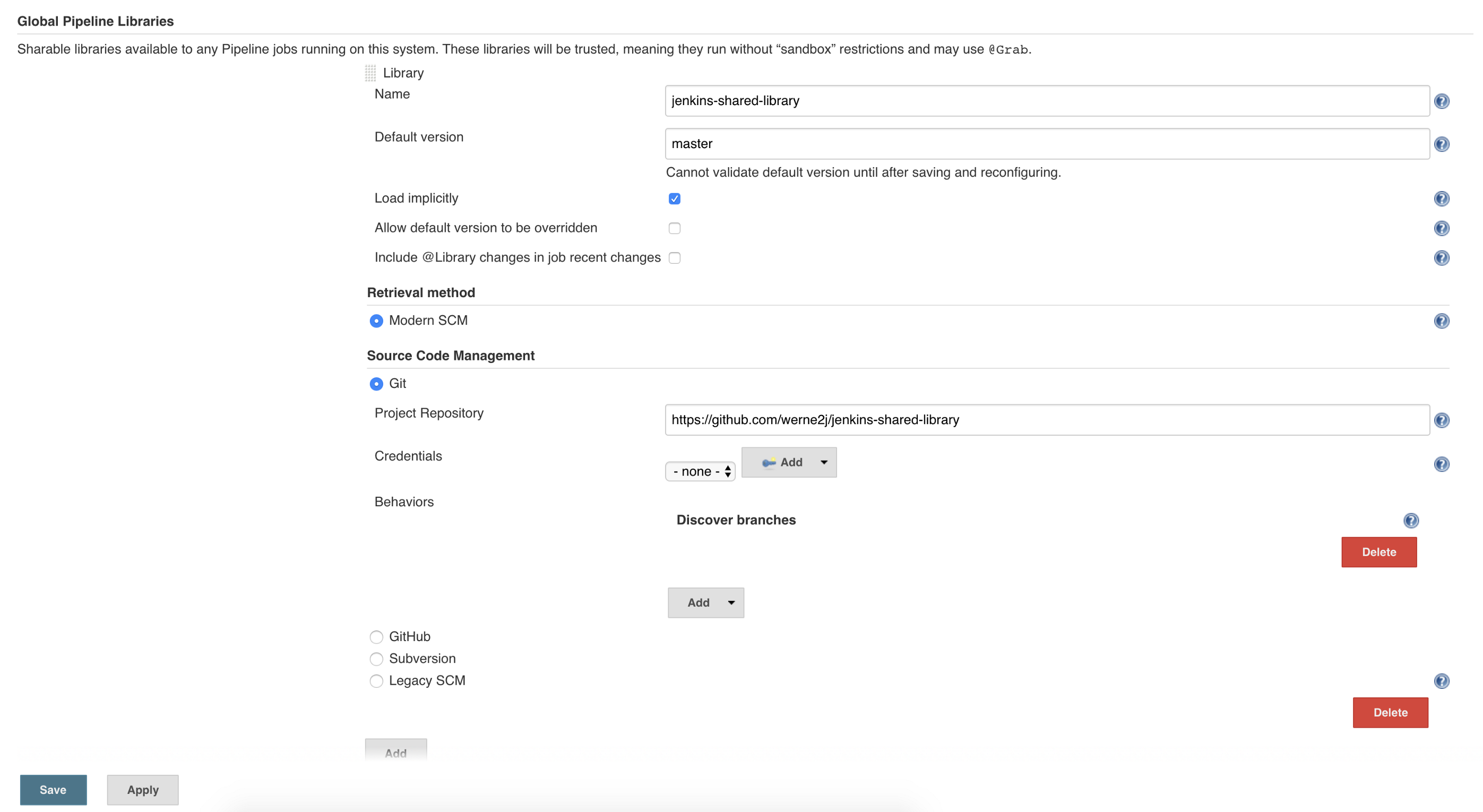
https://miro.medium.com/max/27/1*GXjnQCujBsLSVMP9ZWJ2cg.png?q=20



* Click Add.
* We will name the library jenkins-shared-library and set default version to master.
* Make sure to select the checkbox Load Implicitly. [Shared Libraries marked Load Implicitly allows Pipelines to immediately use classes or global variables defined by any such libraries.](https://jenkins.io/doc/book/pipeline/shared-libraries/#using-libraries)
* Select Modern SCM and then Git. I am going to be loading the library from [https://github.com/werne2j/jenkins-shared-library.git](https://github.com/werne2j/jenkins-shared-library). You can try and push your library to Github and load it in if you would like.
* Put the repository url of the library in Project Repository.
* Click Save.

The library is now loaded into Jenkins!

https://miro.medium.com/max/27/1*ViCAESE5W0VxZEASToPBWg.png?q=20

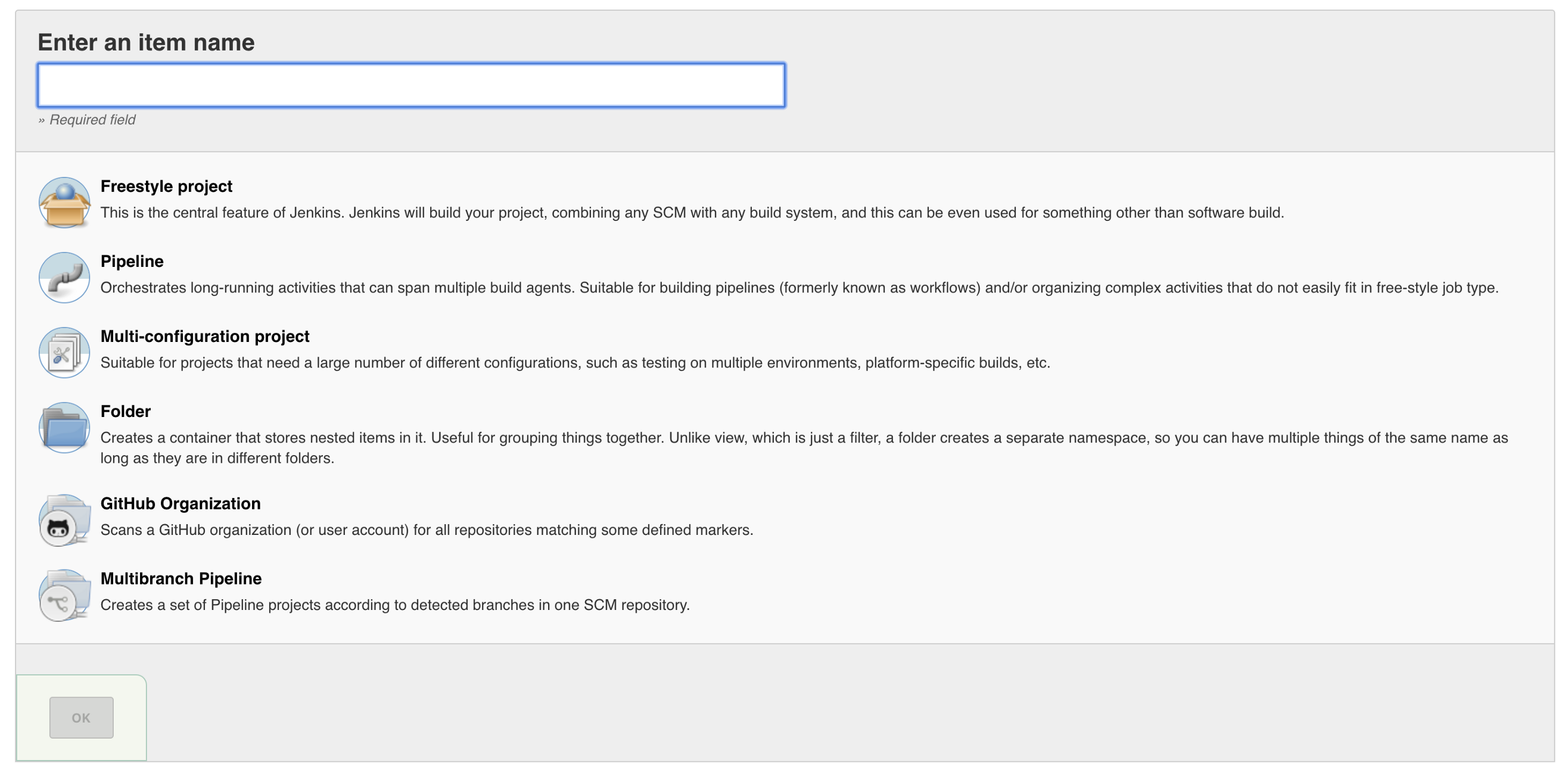


**Testing the Shared Library from a Jenkins Job**

The last thing to do is to create a Jenkins job and test the library.

* Click create new job on the homepage of Jenkins.

https://miro.medium.com/max/27/1*CnxWiktpwAQ3C-Y5DOdCkA.png?q=20



* Enter a name for your job, I will be using test-shared-library.
* Select Pipeline as the type of job and then click OK.
* Once the job is created, you are taken to the configuration page of that job. Scroll down until you see Pipeline.
* We are going to select Pipeline Script so we can test the library without having to load in a Jenkinsfile from Git.
* In the script area we are going to test the buildJavascriptApp variable while passing the notify variable as part of a closure. It will all make sense once it is written out.
* In the pipeline script area paste this code:

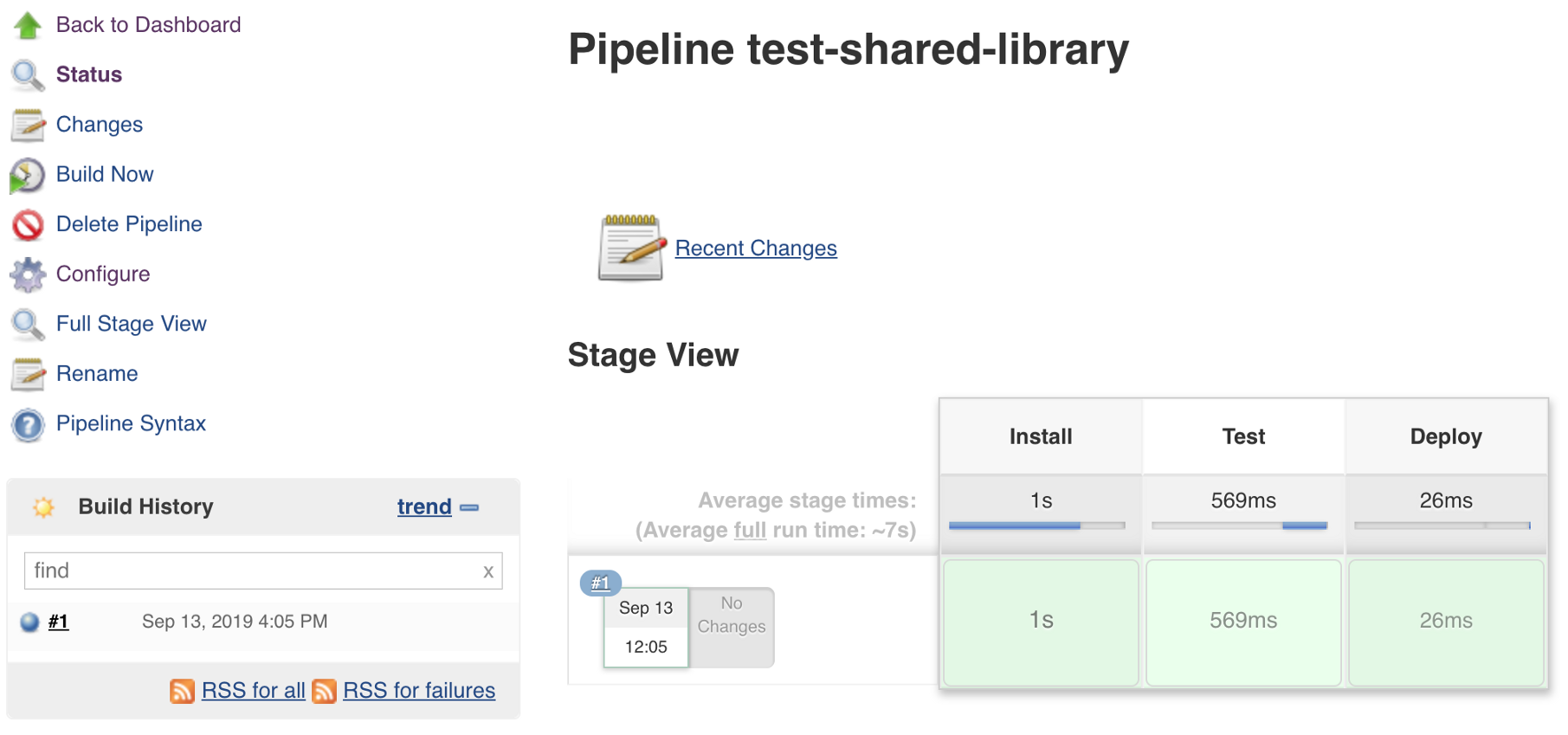
node {  
 env.NODEJS\_HOME = "${tool 'node'}"  
 env.PATH = "${env.NODEJS\_HOME}/bin:${env.PATH}"  
   
 buildJavascriptApp deploy: false, {  
 notify type: "slack", message: "Build succeeded"  
 }  
}

Hopefully, this code makes sense by now. The only thing that is different here is the two lines above our variable call. We need to tell Jenkins to use our Node JS configuration and we do that by setting the Node JS tool on the path environment variable.

* Click Save at the bottom of the screen. It is time to test!
* You will be taken to the homepage of the job. Once there, click Build Now to test the pipeline script.

After a few seconds you should see the pipeline appear, if you see all green, then congratulations, you have successfully created your first shared library!

https://miro.medium.com/max/27/1*jjeWlbDVuHUm71V18VFZVQ.png?q=20

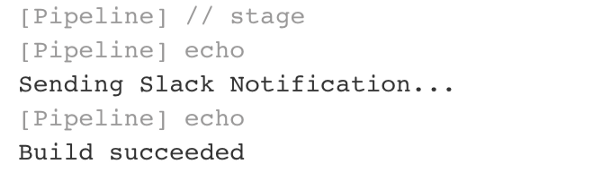


We can look at the console output to see what happened. Click the blue circle next to the build job number, in this case #1. This will take you to the console output of the build.

If you scroll down you should see Jenkins checking out the sample Node JS app from Github and then running npm install and npm test. This shows that these stages were executed just by including buildJavascriptApp in the pipeline script.

Down a little further, you see our other variable in action.

https://miro.medium.com/max/27/1*sAF66z8ePlwpW947UHXs8A.png?q=20



In the pipeline script, we stated we wanted to run notify with type slack inside a closure that was passed to buildJavascriptApp, and as we see here, we get the slack notification constant from the constants class in src as well as the message passed to notify.

Hopefully this tutorial was able to provide the skills for you to be able to go forth and start creating your own Jenkins Shared Libraries!

Thank you for reading part two of my Jenkins Shared Library series.

In the third part of my Jenkins Shared Library series, we will discuss how to write tests for the shared library.

**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)

13

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

13 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