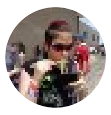
A Hacky Hacker's Guide To Jenkins Scripted Pipelines, Part 4

Taking it to the next level with Jenkins Shared Libraries



Eva Gonciarczyk Follow
May 3, 2018 · 5 min read

A few weeks ago, Jason Arcala commented on part three of this series, suggesting shared libraries as the next topic. It was my first time hearing about Jenkins shared libraries, but I looked it up and it looked wonderful (thanks Jason!).

Before I get into it, I just want to say right now: this is going to be dense, and there are a lot of questions I still don't have the answers to. Shared libraries are, more or less, straight Groovy code (with some Jenkins flavor!), and it took a ton of frustrated Googling and trial and error to get even the basics up and going, but as soon as I got a successful build I knew I had to try and share it here.

If you're new to Jenkins scripted pipelines, you might want to go back and start at part 1.

. . .

Why shared libraries?

Generally, you don't have just one Jenkins pipeline—you have many, but they tend to end up following the same patterns. As you develop more features, and try to refine your process, updating each pipeline individually gets to be a pain. They often get out of sync, some might have typos from repeated manual changes, and it's very counter to commonly accepted DRY (don't repeat yourself) practices in code. Plus, it's a big time suck, and super frustrating to have to go through all the pipelines repeatedly.

Enter the shared library: a separate repository where you can define chunks of reusable code and import them into your pipelines as needed.

Getting set up in Jenkins

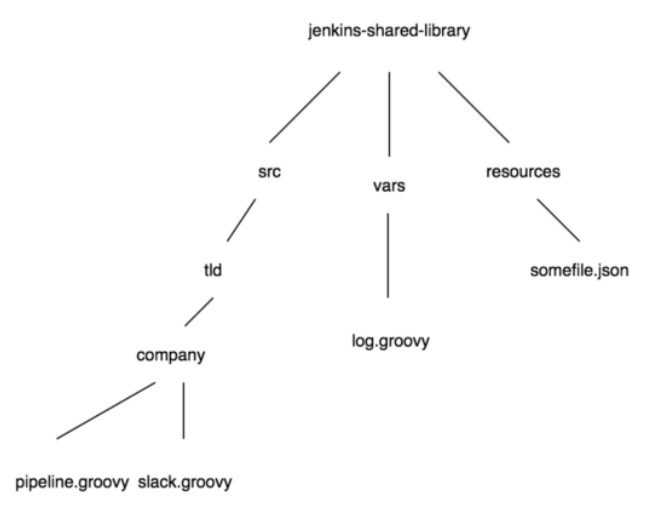
First order of business: creating a jenkins-shared-library repository (like many, I use Github, but whatever SCM you're using for Jenkins is fine). It doesn't need to be called that, and doesn't need to have anything in it yet, but you'll need to add that repository to Jenkins, under *Manage Jenkins » Configure System » Global Pipeline Libraries*. You'll either need administrative access to Jenkins or you'll have to get the people who do on board. Since there's a potential security risk (anyone who can commit to this repository can push code to run on your Jenkins), a private repository is best.

Choosing to 'Load Implicitly' means you don't have to use the `@Library` tag in your pipelines to access your library, but also means your library will be loaded in every single pipeline whether you want it or not (you can see the SCM checkout and last commit of your shared library towards the beginning of your pipeline).

The 'Allow default version to be overridden' option is useful—it lets you specify a different branch (or tag, or other identifier) than the defaults so you can try out changes in one place before pushing them on everything, which is important when any change you make to the library has the chance to affect every pipeline that uses it.

Creating the library

Shared libraries expect a very specific directory structure, shown by this very crudely-drawn diagram:



There are three top-level directories:

`src/` , for your classes/methods. Most of your code goes here. A lot of guides on the internet have most, if not all, of theirs in `vars/` , either because they're using declarative pipelines (which have limitations) or because that's how they got it to work. Like AimTheory, though, I wanted to do it right, so the examples I'll be showing will reflect that.

Inside the `src/` directory the structure is `[io|org|com|other tld]/companyname/` , and then your `.groovy` files. This convention comes from Java, and it's pretty important to follow.

`vars/` is for, well, global variables. I haven't seen a whole lot of examples of them in practice, though, and it never seems to be a regular `var=value` type of variable. The most common examples I've seen start with `def call` . I looked to Fabric8's shared library for guidance, since it's the most robust public one I found, but their `vars/` are full-blown chunks of code too.

`resources/` are your non-Groovy files, if you have them. I haven't needed to add any extra files yet, but if you need to, that's where.

Creating classes and using them in your pipelines

Now comes the tricky part: creating and using your library. Let's say your pipelines have the same basic structure:

```
Jenkinsfile

node {

  stage('build') {

    //build code (ex: mvn deploy)

  }

  stage('deploy to qa') {

    //deploy code (ex: aws cloudformation create-stack)

  }

  stage('test') {

    //test code (ex: sh /path/to/test/script.sh)
  }
  stage('deploy to staging') {
    //deploy code (ex: aws cloudformation create-stack)
  }
  stage('test') {
    //test code (ex: sh /path/to/test/script.sh)
  }
  stage('deploy to prod') {
    //deploy code (ex: aws cloudformation create/update-
stack)
  }
```

It boils down to a build step, three deploys, and two tests. Each deploy is pretty much the same thing, just with different environments, and the test s only vary by name—and these chunks of code are repeated across five, 10, 30 pipelines! If you wanted to make a change, like adding a try/catch block or a new test, you'd have to go through each pipeline, find that block, and update it. What if, instead, your pipeline steps were defined in the shared library?

```
src/io/abc/pipeline.groovy
#!/usr/bin/groovy
package io.abc;

def build() {
  mvn clean deploy -U
}

def test(name) {
  sh "/usr/local/bin/${name}"
}

def deploy(env,app) {
  aws cloudformation create-stack \
  --stack-name ${env}_{app)
  --parameters \
    ParameterKey=Env,ParameterValue=${env}
  ...(rest of the params go here)
}

// AimTheory have a recommendation and explanation about
this here
return this
```

Now all you have to do is pass the variables to the method calls in your super-stripped-down Jenkinsfile:

```
Jenkinsfile
```

```
// import
@Library("jenkins-shared-libraries@release")
import io.abc.*

// instantiate
pl = new pipeline()

// work
pl.build()
pl.deploy(qa,api)
pl.test(regression)
pl.deploy(staging,api)
pl.test(load)
pl.deploy(prod,api)
```

Note: `pipeline()` refers to my `src/io/abc/pipeline.groovy` file. If I had called the file in the shared library `buildTestDeploy.groovy` , the instantiation would've been `pl = new buildTestDeploy()` .

. . .

That's all for now—I've been doing some work on setting up dynamic, multi-module pipelines with shared libraries, and once it's a bit more polished I'll write it up too.