

[Home](#) > [Operations & Sysadmin](#)

How to Create a Jenkins Shared Library (Tutorial + Example Repo)

Written by [Tom Donohue](#) 

Updated: 16 February 2022

You're getting up to speed with Jenkins, and you're mastering the *art of the pipeline*. But now that you've got a few projects up and running, how do you avoid repeating the same code in many different pipelines?

As you've probably realised by now, copying and pasting the same code into different Jenkins pipelines can become an absolute *headache*.

There's a point at which some voice in your head starts telling you: "D.R.Y." (Don't Repeat Yourself).

But I often need to write code that I will need to use in several different pipelines in Jenkins.

How can you avoid this?

You can store your "reusable bits" (*technical term!*) in a **Shared Library** in Jenkins.



You only need to write your code **once**, and then you can share the same code with **all** of your pipelines. **BAM.** 🌟

This tutorial will tell you the basics of setting up a Jenkins pipeline shared library, how to load the library, and how to use it in your pipelines.

Let's go.

ON THIS PAGE

- [Can you reuse code in Jenkins?](#)
- [What is a Shared Library in Jenkins?](#)
 - [What goes inside a Shared Library?](#)
- [Example: Creating and using a Jenkins shared library](#)
- [What if you don't have Jenkins admin access?](#)
- [Summary \(TL;DR\)](#)

LEGALES



Can you reuse code in Jenkins?

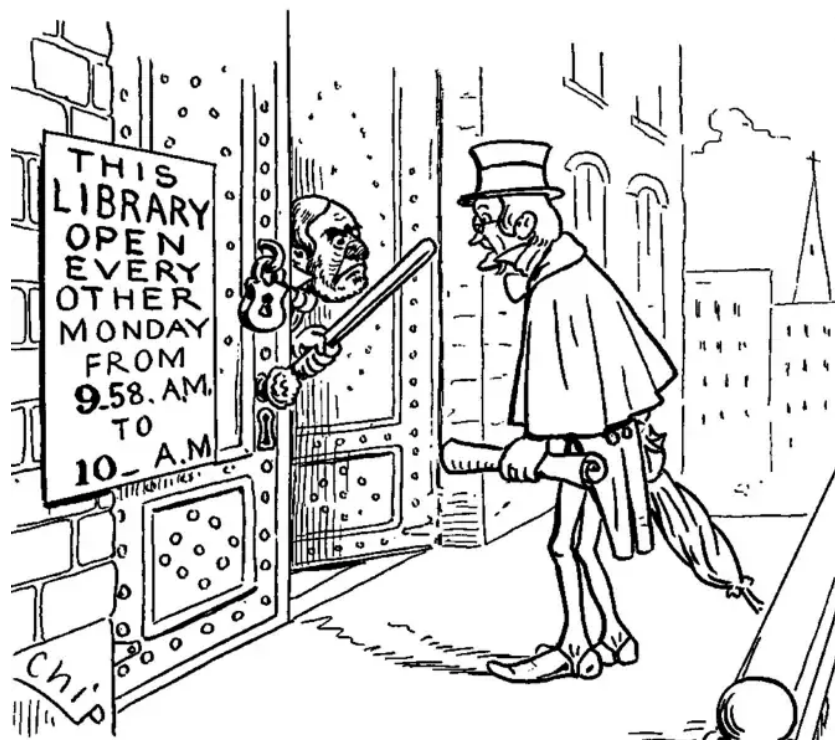
If you're already writing **custom code** in your pipelines to do things like this:

- read a configuration file
- perform a code review of the application (e.g. using a tool like SonarQube)
- do some checks on the target environment
- perform a deployment

...then you're probably better off adding this code into a shared library, so that other projects can use it.

Over time, you'll build up a collection of these reusable functions in your Library.

People will admire your library, and will come to visit to see the marvellous works within.



The library is closed.

What is a Shared Library in Jenkins?

A shared library is a **collection of independent Groovy scripts** which you pull into your *Jenkinsfile* at runtime.

The best part is, the Library can be stored, like everything else, in a Git repository. This means you can version, tag, and do all the cool stuff you're used to with Git.

Not sure if you know Groovy? You've probably already it in Jenkins. It's a Java-based scripting language, with a forgiving syntax.

Most things that you can do in Java, you can also do in Groovy. You don't need to be a Groovy expert, but if you want to [start fully learning Groovy, then you can find some learning resources here](#).

Here's how you create a Jenkins library, step-by-step:

- 1 First you **create your Groovy scripts** (see further below for details), and add them into your Git repository.
- 2 Then, you **add your Shared Library into Jenkins** from the *Manage Jenkins* screen.
- 3 Finally, you **pull the Shared Library** into your pipeline using this annotation (usually at the top of your *Jenkinsfile*):

```
@Library('your-library-name')
```

What goes inside a Shared Library?

Inside your Library you'll probably have two types of common code:



Steps.

- **Steps:** These are called *Global Variables* in Jenkins terminology, but these are the custom steps that you want to be available to all your Jenkins pipelines.

For example: you might write a standard step to deploy an application, or perform a code review. To do this, add your code into `vars/YourStepName.groovy` and then implement a `def call` function, like this:

```
#!/usr/bin/env groovy
// vars/YourStepName.groovy

def call() {
    // Do something here...
}
```

- **Other common code:** This might include helper classes, or common code that you might want to include inside pipeline steps themselves (very meta!). You could also use it as a place to store **static constants** which you want to use throughout your pipelines.

Code like this needs to go in the `src/your/package/name` directory, and then you can use normal Groovy syntax.

For example: Here's a common file, `xyz/tomd/GlobalVars.groovy` (note the package `xyz.tomd` at the top):

```
#!/usr/bin/env groovy
package xyz.tomd

class GlobalVars {
    static String foo = "bar"
}
```

You can then `import` this class into your *Jenkinsfile* and reference the static variable like `GlobalVars.foo`.

Example: Creating and using a Jenkins shared library

So now the only thing left to do in this guide is show you how to do it, for real.

In this section, we'll see how to set up a shared library in Jenkins, with a quick example.

1 Create the shared library

First you need to create a Git repository which will contain your library of functions (steps). (You can also use Subversion.)

In your repository, create a directory called `vars`. This will hold your custom steps. Each of them will be a different `.groovy` file underneath your `vars` directory, e.g.:

```
vars/  
  deployApplication.groovy  
  parseFile.groovy  
  sayHello.groovy  
  readSystemCredentials.groovy  
  doCodeReview.groovy
```

I'm using Git to store my repository. I've created [a sample repo on GitHub](#), which includes one function, the `sayHello` example from the [official Jenkins documentation](#).

See the example repository on GitHub

2 Add your custom steps

Each of your custom steps is a different `.groovy` file inside your `vars/` directory. In Jenkins terminology, these are called *Global Variables*, which is why they are located inside `vars/`.

Create a file for your custom step, and fill in the code. For example, a simple greeting function would look like this:

```
#!/usr/bin/env groovy  
  
def call(String name = 'human') {  
    echo "Hello, ${name}."  
}
```

Notice how the Groovy script must **implement the `call` method**.

After writing that, you should write your custom code within the braces `{ }`. You can also add parameters to

your method - the example above has one parameter called `name`, which has a default value of `human` (cos we're being really personal here.)

3 Set up the library in Jenkins

Now you've created your library with custom steps, you need to tell Jenkins about it.

You can define a shared library within a *Jenkinsfile*, or you can configure the library using the Jenkins web console. Personally, I think it's better to add from the web console, because you then you can share the library across all of your build jobs.

To add your shared library (I'm using my demo repository on GitHub as an example):

In Jenkins, go to Manage Jenkins → Configure System. Under *Global Pipeline Libraries*, add a library with the following settings:

- Name: `pipeline-library-demo`
- Default version: Specify a Git reference (branch or commit SHA), e.g. `master`
- Retrieval method: **Modern SCM**
- Select the **Git** type
- Project repository: `https://github.com/tutorialworks/pipeline-library-demo.git`

4 Use the library in a pipeline

To use the shared library in a pipeline, you add `@Library('your-library-name')` to the top of your pipeline definition, or *Jenkinsfile*. Then call your step by name, e.g. `sayHello`:

```
@Library('pipeline-library-demo')_

stage('Demo') {
    echo 'Hello world'
    sayHello 'Dave'
}
```

NOTE: The underscore (`_`) is **not a typo!** You need this underscore if the line immediately after the `@Library` annotation is not an `import` statement.

If you're using declarative pipeline, the syntax looks slightly different:

```
libraries {
    lib('pipeline-library-demo')
}

pipeline {
    // Your pipeline would go here....
}
```

```
}


```

- 5 Run the pipeline above, and the output should look something like this:

```
Jenkins > lib-test > #2
Console Output

Started by user Admin
Loading library pipeline-library-demo@master
> git rev-parse --is-inside-work-tree # timeout=10
Setting origin to https://github.com/monodot/pipeline-library-demo
> git config remote.origin.url https://github.com/monodot/pipeline-library-demo # timeout=10
Fetching origin...
Fetching upstream changes from origin
> git --version # timeout=10
> git fetch --tags --progress origin +refs/heads/*:refs/remotes/origin/*
> git rev-parse master^{commit} # timeout=10
> git rev-parse origin/master^{commit} # timeout=10
Cloning the remote Git repository
Cloning repository https://github.com/monodot/pipeline-library-demo
> git init /Users/tdonohue/.jenkins/workspace/lib-test@libs/pipeline-library-demo # timeout=10
Fetching upstream changes from https://github.com/monodot/pipeline-library-demo
> git --version # timeout=10
> git fetch --tags --progress https://github.com/monodot/pipeline-library-demo
+refs/heads/*:refs/remotes/origin/*
> git config remote.origin.url https://github.com/monodot/pipeline-library-demo # timeout=10
> git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
> git config remote.origin.url https://github.com/monodot/pipeline-library-demo # timeout=10
Fetching upstream changes from https://github.com/monodot/pipeline-library-demo
> git fetch --tags --progress https://github.com/monodot/pipeline-library-demo
+refs/heads/*:refs/remotes/origin/*
Checking out Revision 5606c9baa9130ec594aec257bbda3d567778715e (master)
Commit message: "First commit"
> git config core.sparsecheckout # timeout=10
> git checkout -f 5606c9baa9130ec594aec257bbda3d567778715e
First time build. Skipping changelog.
[Pipeline] stage
[Pipeline] { (Demo)
[Pipeline] echo
Hello world
[Pipeline] echo
Hello, Dave.
[Pipeline] }
[Pipeline] // stage
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Sample job output from Jenkins

Sample job output from Jenkins by Tutorial Works is licensed under
CC BY-SA 4.0 

And you're done.

Now you can start adding all of your custom steps to the library.

What if you don't have Jenkins admin access?

You might find that you don't have administrator access to Jenkins, so you can't see the *Manage Jenkins* area.

(I think this is pretty bad, because you should be able to manage your own Jenkins instance)

If this is a blocker for you, then you can import a library explicitly in a pipeline, instead.

Here's an example **declarative pipeline** which uses a shared library, which is stored in a protected (private) Git repository.

The pipeline imports the library by using the `library` command, and giving a `retriever`, which is basically the details of your Git repo.



An actual retriever

The *retriever* will authenticate to the repository using the credentials identified by `your-credentials-id`:

```
library identifier: 'mylibraryname@master',  
    // 'mylibraryname' is just an identifier, it can be anything you like  
    // 'master' refers to a valid git ref (branch)  
retriever: modernSCM([  
    $class: 'GitSCMSource',  
    credentialsId: 'your-credentials-id', // remove this if it's public!  
    remote: 'https://git.yourcompany.com/yourrepo/private-library.git'  
])  
  
pipeline {  
    agent any  
    stages {  
        stage('Demo') {  
            steps {  
                echo 'Hello world'  
                yourCustomStep 'your_arg'  
            }  
        }  
    }  
}
```

Summary (TL;DR)

That's it for my intro to Shared Libraries in Jenkins! As you can see they're a useful way to share common code that you might use across different *Jenkinsfiles*.

Here are the important things to remember:

- **You need to use Groovy** to write your custom functions or steps
- To write pipeline steps that you can call from your pipelines (e.g. `deployApplication`, `readConfigFile`, etc.):
 - Create a file in the `vars/` directory in your repository, with the name of your custom step
 - The file should implement the `def call()` method; you can also define parameters to your step

- To write other common Groovy code, add it into `src/`
- Add your Shared Library in Jenkins using the Configure System screen

Thanks for reading! I hope you've found this example Jenkins pipeline library useful.

How are you using Jenkins pipelines? Any feedback on this article? You're very welcome to post your thoughts in the comments section below.

This article was originally posted on [my blog](#).

Thanks for reading. Please don't let this be the end. 💔

It took us this long to find each other. Before you close your browser and forget all about this article, let's agree to stay in touch? **Join my email newsletter.** I'll email you my latest tutorials and guides, so you can read at your leisure! 🇺🇸 (No spam, unsubscribe whenever you want.)

Join the newsletter

Comments

Got any thoughts on what you've just read? Anything wrong, or no longer correct? Sign in with your GitHub account to **leave a comment**.

(All comments get added as Issues in [our GitHub repo here](#), using the open source comments tool [Utterances](#))

2 Comments - powered by [utteranc.es](#)

tarun9715m commented 3 months ago

awesome

KhafazovPavlo commented 5 days ago

thank you, the part about import a library explicitly in a pipeline was very useful for me.
please advice, how can I import static constants using this approach? I tried like this: `import com.deng.dengVars`.
but got `MissingPropertyException` error

Want more? Read these articles next...

You've found the end of another article! Keep on learning by checking out one of these articles:

- [DevOps Project Ideas](#): A career in DevOps is all about building a broad skill base and understanding. Use these project ideas to invest in yourself and get that...
- [The Best Places to Learn & Try Kubernetes Online](#): Learning Kubernetes can seem challenging. But fear not! Here's a boatload of resources that will help you get there.
- [14 best practices for containerising your Java applications](#): Best practices to follow when building and running your Java application in a Docker container
- [Spring Boot app metrics - with Prometheus and Micrometer](#): How to configure and publish metrics from your application, and define your own custom metrics
- [The Honest \(and Opinionated\) Guide to Java IDEs](#): Choosing an IDE is like buying a house: don't buy a set of old problems. Find out our top IDE pick here.

Tutorial Works is a website to help you navigate the world of IT, and grow your tech career, with tips, tutorials, guides, and *real opinions*.

Thanks for being here today! 🍌

[Join our newsletter](#)

[About this website](#)

[Privacy policy](#)

[Contact us](#)

[CAREER](#)

[Getting into DevOps](#)

[DevOps Project Ideas](#)

TECHNOLOGIES

[Learn Kubernetes](#)

[Top Linux Commands](#)

[How Docker containers communicate](#)

[Kubernetes Courses](#)

TOPICS

[What is DevOps?](#)

[DevOps Roadmap](#)

[DevOps Books](#)

[Containers](#)

Copyright © 2022 Tom Donohue. All rights reserved, except where stated.

Tutorial Works is a participant in the Amazon.com Services LLC Associates Program. As an Amazon Associate we earn from qualifying purchases. Amazon and the Amazon logo are trademarks of Amazon.com, Inc. or its affiliates.