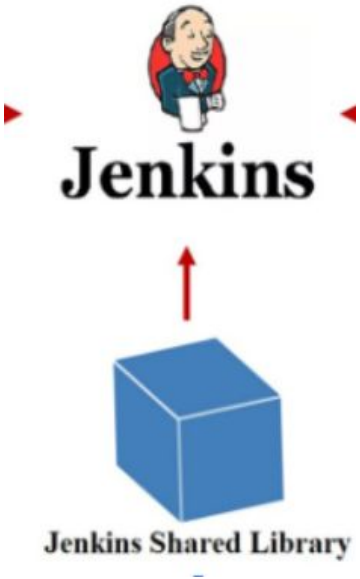


Jenkins Shared Libraries (JSL)

Ajay Kakumanu

Understanding Jenkins Shared Libraries

- Jenkins Shared Libraries are a powerful feature that allows you to define reusable code and functions that can be used across multiple Jenkins pipelines.
- Follow the principle of DRY (Don't repeat yourself).
- You only need to write your code once, and then you can share the same code with all of your pipelines
- Load a library dynamically rather than using @Library syntax.
- Shared library is a collection of independent Groovy scripts which you pull into your Jenkinsfile at runtime.
- Library can be stored, like everything else, in a Git repository.
- Can do version, tag to that library.

Advantages of Shared Libraries

1. Code reusability
    - Shared Libraries enable you to encapsulate common functionality, such as build steps, deployment procedures, or custom logic, into reusable modules. This reduces duplication of code and ensures consistency across pipelines.
2. Centralized maintenance
    - By placing shared code in a central repository, you can make updates and improvements in one location. This makes it easier to maintain and update your Jenkins pipelines as your project evolves
3. Abstractions
    - Shared Libraries allow you to create higher-level abstractions that simplify pipeline scripts. This is especially beneficial for complex workflows, enabling pipeline authors to focus on business logic rather than intricate implementation details.
4. Security and Compliance
    - You can include security checks, access controls, and compliance measures within your shared libraries to ensure that pipelines adhere to organizational policies
5. Version Control
    - Shared Libraries are version-controlled, just like any other code. This makes it easy to roll back to previous versions or track changes over time.
6. Extensibility
    - Jenkins Shared Libraries can be extended and customized to match your specific project requirements. This flexibility allows you to tailor the libraries to fit your team's needs.

Directory structure

```
(root)
+- src                     # Groovy source files
|   +- org
|       +- foo
|           +- Bar.groovy  # for org.foo.Bar class
+- vars
|   +- foo.groovy          # for global 'foo' variable
|   +- foo.txt             # help for 'foo' variable
+- resources               # resource files (external libraries only)
|   +- org
|       +- foo
|           +- bar.json    # static helper data for org.foo.Bar
```

The **src** directory should look like standard Java source directory structure.
The **vars** directory contains script files (.groovy) that are exposed as a variable in Pipelines
So if you had a file called vars/log.groovy with a function like def info(message)... in it, you can access this function like log.info "hello world" in the Pipeline. You can put as many functions as you like inside this file. Read on below for more examples and options.

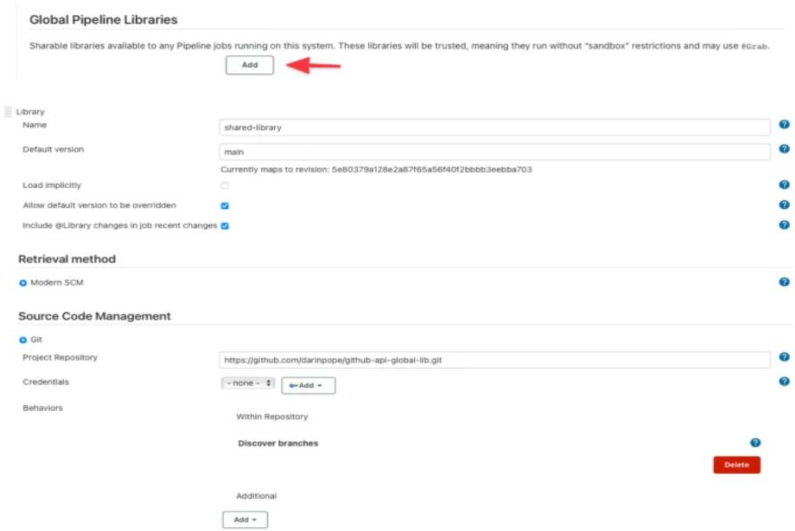
Types of shared Libraries based on scope

1. Global Shared Libraries
2. Folder-level Shared Libraries
3. Automatic Shared Libraries

Global Shared Libraries

- These libraries will be globally usable.
- Any Pipeline in the system can utilize functionality implemented in these libraries
- Manage Jenkins » System » Global Pipeline Libraries as many libraries as necessary can be configured.

Configure System takes you to a very long page. Scroll down until you see Global Pipeline Libraries.



- **Shared-library** is the name of library
- **main** for the default version
- A default version can be a branch name, a tag, or even a git commit hash

Global Shared Libraries -continued

```
@Library('shared-library') _
def config = [name: 'Newman', dayOfWeek: 'Friday']
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                helloWorld(config)
            }
        }
    }
}
```

- @Library loads the Shared Library. You need to follow the function call with an underscore (_) operator to tell Groovy to import the entire library namespace. Usually, your Shared Library would be smaller than the sample code, or you would only import a subset of the library namespace.
- the script defines a map with the two parameters helloWorld() expects—name and dayOfWeek.Finally, replace the call to sh in the build step with helloWorld() with the config map.
  This is console output

```
[Pipeline] { (Example)

[Pipeline] sh

+ echo Hello Newman. Today is Friday.

Hello Newman. Today is Friday.
```

Folder-level Shared Libraries

- Any Folder created can have Shared Libraries associated with it. This mechanism allows scoping of specific libraries to all the Pipelines inside of the folder or subfolder.

Automatic level Shared Libraries

- Add libraries on the fly to the Jenkins job **without any additional configuration.**

Pipeline: GitHub Groovy Libraries

Allows Pipeline Groovy libraries to be loaded on the fly from public repositories on GitHub. Unlike regular library definitions, no preconfiguration at the global or folder level is needed.

Example:

```
@Library('github.com/jglick/sample-pipeline-library') _
if (currentBuildExt().hasChangeIn('src')) {
  return
}
node {
  sh 'make'
}
```

Loading libraries dynamically

- As of version 2.7 of the Pipeline: Shared Groovy Libraries plugin, there is a new option for loading (non-implicit) libraries in a script: a library step that loads a library dynamically, at any time during the build.
- Syntax is library 'my-shared-library'

- The shared library itself is configured in Jenkins to be loaded dynamically from a version control repository (e.g., Git) or a shared location whenever a pipeline run requires it.

- The dynamic loading approach provides flexibility and the ability to update the shared library code independently of the pipeline scripts. Changes made to the shared library are automatically available to all pipelines that reference it, without requiring the pipelines themselves to be updated.

vars Vs src Folder

- vars Folder: Contains files with step and function definitions that can be directly called from pipeline scripts. These files enable high-level abstractions for pipeline steps.

- src Folder: The src folder is optional and is used to store supporting code or classes that are used by the functions defined in the **vars** folder. It's meant for code that doesn't directly represent steps or functions callable from pipeline scripts.
  For example, if your vars step requires some additional classes or utility functions (like JsonUtils, MapUtils), you can organize them within the src folder. These supporting files enhance the functionality of your library steps without being directly called from pipeline scripts.
  For example the below Jsonutility class and methods used by Groovy files in Var folder

```
class jsonutility implements Serializable {

    static String groovyObjectJsonString(object) {
        return groovy.json.JsonOutput.print(groovy.json.JsonOutput.toJson(object))
    }

}
```

Best practises

1. Follows OOPs standards (Object-Oriented Programming)
    a. Modularity → Design your library with modular functions and classes.
    b. Abstraction →using separate classes for each functionality
    c. Inheritance →re-use class by extends the existing class
    d. Polymorphism → write methods to follow Methods overloading standards which is used for backward compatibility
2. Global Variables
    a. Use global variables to store configurations or settings that can be easily changed without modifying pipeline scripts
3. Versioning
    a. Use semantic versioning (SemVer) to indicate changes in your shared library
4. Clear Documentation
    a. Provide clear and comprehensive documentation for your shared library.

Best practises

1. Follows OOPs standards (Object-Oriented Programming)
    a. Modularity → Design your library with modular functions and classes.
    b. Abstraction →using separate classes for each functionality
    c. Inheritance →re-use class by extends the existing class
    d. Polymorphism → write methods to follow Methods overloading standards which is used for backward compatibility
2. Global Variables
    a. Use global variables to store configurations or settings that can be easily changed without modifying pipeline scripts
3. Versioning
    a. Use semantic versioning (SemVer) to indicate changes in your shared library
4. Clear Documentation
    a. Provide clear and comprehensive documentation for your shared library.

References

1. https://www.jenkins.io/doc/pipeline/steps/workflow-cps-global-lib/
2. https://www.cloudbees.com/blog/getting-started-with-shared-libraries-in-jenkins