DevOps Shack

Jenkins Shared Library: A Comprehensive Guide

Introduction

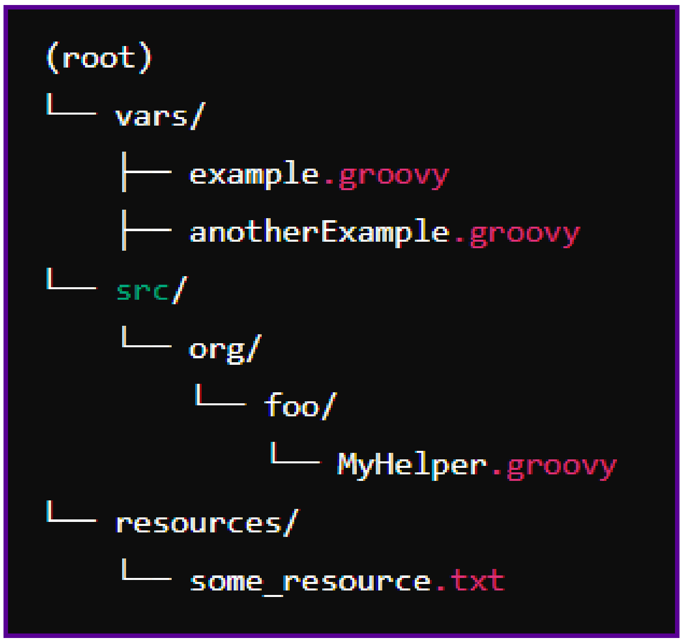
Jenkins Shared Library is a powerful feature that allows you to define reusable code, enabling you to centralize your Jenkins Pipeline scripts in a common location. This approach promotes code reuse, standardization, and easier maintenance across multiple Jenkins pipelines. By using Shared Libraries, you can store and manage your pipeline logic in a version-controlled repository, making it accessible to all your Jenkins jobs.

This guide will provide a detailed overview of Jenkins Shared Library, including setup instructions, usage examples, and advanced features.

Setting Up a Jenkins Shared Library

Step 1: Create a Shared Library Repository

1. **Create a new Git repository** where the shared library code will be stored. This repository will contain all the reusable pipeline code, scripts, and helper functions.
2. **Structure your repository** according to Jenkins Shared Library conventions. The basic structure should include the following directories:



```
(root)
└── vars/
    ├── example.groovy
    ├── anotherExample.groovy
└── src/
    └── org/
        └── foo/
            └── MyHelper.groovy
└── resources/
    └── some_resource.txt
```

- **vars/**: Contains global variables or functions accessible from any Jenkinsfile. Each .groovy file in this directory represents a global variable or a function.
- **src/**: Contains Groovy classes and scripts used by your shared library. This directory should follow the standard Java package structure.
- **resources/**: Stores resource files that can be loaded and used within your pipeline scripts, such as text files, configuration files, etc.

Step 2: Configure the Shared Library in Jenkins

1. **Navigate to Jenkins Dashboard** and click on Manage Jenkins -> Configure System.
2. **Scroll down to the section** titled Global Pipeline Libraries.
3. **Add a new library** by clicking on Add.
4. **Provide the following details**:
   - **Name**: A name for the shared library (e.g., my-shared-library).
   - **Default Version**: The branch or tag to be used by default (e.g., main).
   - **Source Code Management**: Configure the Git repository details (URL, credentials).
   - **Load implicitly**: Check this option if you want the library to be loaded automatically in every pipeline. If unchecked, you'll need to load the library explicitly in your Jenkinsfiles.
5. **Save the configuration**.

Using Jenkins Shared Library in a Pipeline

Example 1: Basic Usage of Global Variables

Consider the following structure in your vars/ directory:

- example.groovy:

```
def call(String name) {

  echo "Hello, ${name}!"

}
```

- **Using the library in your Jenkinsfile**:

```
@Library('my-shared-library') _

pipeline {

  agent any

  stages {

    stage('Greet') {

      steps {

        example('Jenkins')
```

```
                }
            }
        }
    }
```

Explanation:

- The @Library('my-shared-library') _ directive loads the shared library.
- The example function, defined in example.groovy, is invoked in the pipeline to print a greeting message.

Example 2: Using Helper Classes from src/ Directory

Consider the following structure in your src/ directory:

- src/org/foo/MyHelper.groovy:

```
package org.foo


class MyHelper {
    static String toUpperCase(String input) {
        return input.toUpperCase()
    }
}
```

- **Using the helper class in your Jenkinsfile**:

```
@Library('my-shared-library') _
import org.foo.MyHelper


pipeline {
    agent any
    stages {
        stage('Convert to Uppercase') {
            steps {
                script {
                    def result = MyHelper.toUpperCase('jenkins shared library')
                    echo "Converted String: ${result}"
                }
            }
        }
    }
}
```

Explanation:

- The MyHelper class is defined in the src/org/foo/ directory.
- The toUpperCase method is used within the pipeline to convert a string to uppercase.

Example 3: Using Resources from the resources/ Directory

Consider a text file stored in your resources/ directory:

- resources/some_resource.txt:

```
This is a sample resource file.
```

- **Accessing the resource in your Jenkinsfile**:

```
@Library('my-shared-library') _
pipeline {
    agent any
    stages {
        stage('Read Resource') {
            steps {
                script {
                    def resourceContent = libraryResource('some_resource.txt')
                    echo "Resource Content: ${resourceContent}"
                }
            }
        }
    }
}
```

Explanation:

- The libraryResource function is used to read the content of the some_resource.txt file from the shared library's resources/ directory.

Advanced Features of Jenkins Shared Library

1. Loading Libraries Dynamically

In some scenarios, you may need to load different versions of a shared library dynamically within the same Jenkinsfile:

```
@Library('my-shared-library@development') _

pipeline {
  agent any
  stages {
    stage('Use Development Version') {
      steps {
        example('Development Jenkins')
      }
    }
  }
}


@Library('my-shared-library@production') _

pipeline {
  agent any
  stages {
    stage('Use Production Version') {
      steps {
        example('Production Jenkins')
      }
    }
  }
}
```

Explanation:

- The @Library('library-name@version') directive allows specifying the version or branch of the library to load.

2. Defining Custom Step Functions

Shared libraries can define custom step functions, enabling you to encapsulate complex pipeline logic:

- vars/deploy.groovy:

```
def call(String environment) {
    echo "Deploying to ${environment} environment..."
    // Deployment logic goes here
}
```

- **Using the custom step in your Jenkinsfile**:

```
@Library('my-shared-library') _
pipeline {
    agent any
    stages {
        stage('Deploy') {
            steps {
                deploy('staging')
            }
        }
    }
}
```

Explanation:

- The deploy function encapsulates deployment logic and can be reused across multiple pipelines.

3. Handling Complex Logic with Groovy Classes

Shared libraries allow you to organize complex logic using Groovy classes, which can be stored under the src/ directory:

- src/org/foo/DeployHelper.groovy:

```
package org.foo


class DeployHelper {
    static void deployToEnvironment(String environment) {
        println "Deploying to ${environment}"
        // Add more complex logic here
    }
}
```

- **Using the helper class in your Jenkinsfile**:

```
@Library('my-shared-library') _
import org.foo.DeployHelper


pipeline {
    agent any
    stages {
        stage('Deploy') {
            steps {
                script {
                    DeployHelper.deployToEnvironment('production')
                }
            }
        }
    }
}
```

Explanation:

- This example shows how to organize deployment logic in a reusable class and call it within a pipeline.

Testing and Debugging Jenkins Shared Libraries

1. Unit Testing with JenkinsPipelineUnit

To ensure the quality of your shared library code, you can write unit tests using the JenkinsPipelineUnit framework, which allows testing of pipeline scripts:

- Add a test class:

```
import org.foo.MyHelper
import org.junit.Test
import static org.junit.Assert.*


class MyHelperTest {
  @Test
  void testToUpperCase() {
    assertEquals("JENKINS", MyHelper.toUpperCase("jenkins"))
  }
}
```

- Run your tests locally or in a CI pipeline to validate the shared library functionality before deploying it to production.

2. Logging and Debugging

- Use println or echo statements in your Groovy scripts to log messages and debug your shared library code.
- Review Jenkins logs to identify any issues during the execution of your pipelines.

Conclusion

Jenkins Shared Library is an essential feature for any Jenkins-based CI/CD environment. By centralizing reusable pipeline code, it promotes consistency, reduces redundancy, and simplifies pipeline management. With the examples provided in this guide, you can start building your own shared libraries to enhance your Jenkins pipelines' efficiency and maintainability.

Whether you're working on simple scripts or complex deployment logic, Jenkins Shared Library offers the flexibility and scalability needed to manage your CI/CD workflows effectively.