## Jenkins Shared Library Designs

Jenkins provides several strategies for implementing shared libraries, allowing users to organize and reuse code across multiple pipelines. Shared libraries promote code reuse, maintainability, and consistency in Jenkins pipelines. Below are some common strategies for structuring and managing shared libraries in Jenkins.

### 1. Single Shared Library Repository

**Strategy:**

* Store all shared library code in a single Git repository.

**Pros:**

* Simple to set up and manage.
* Centralized codebase for all shared library functions.

**Cons:**

* Can become unwieldy as the library grows.
* Potential conflicts and difficulties in versioning.

### 2. Repository per Shared Library

**Strategy:**

* Create a separate Git repository for each shared library.

**Pros:**

* Easier to manage and version individual libraries independently.
* Teams can have ownership of specific libraries.

**Cons:**

* Requires additional setup and coordination for managing multiple repositories.

### 3. Monorepo with Multiple Libraries

**Strategy:**

* Use a single Git repository containing multiple shared libraries.

**Pros:**

* Centralized management of multiple libraries.
* Easier to coordinate changes across related libraries.

**Cons:**

* Can become complex and challenging to manage as the number of libraries grows.

### 4. Plugin-Based Libraries

**Strategy:**

* Develop shared libraries as Jenkins plugins.

**Pros:**

* Encapsulation of functionality in plugins.
* Can be versioned and updated independently.

**Cons:**

* Requires additional effort to create and maintain plugins.
* Limited flexibility compared to using pure Groovy shared libraries.

### 5. Global Shared Libraries

**Strategy:**

* Global shared libraries are defined at the Jenkins master level and are available across all Jenkins jobs.
* Suitable for libraries that are common across different projects.

### 6. Folder-Level Shared Libraries

**Strategy:**

* Shared libraries can be defined at the folder level.
* Libraries defined at this level are available for use within pipelines in that specific folder and its subfolders.

**Pros:**

* Useful for organizing shared libraries based on project structures.

**Cons:**

* Requires careful management to ensure libraries are accessible to the right teams.

### 7. Global Variables (vars/ directory)

* A global variable can be defined in the vars/ directory as test1.groovy.
* Accessible directly in pipelines without explicit import.

### 8. Java-Style Code Organization (src/ directory)

* Shared libraries can include Java-like structured Groovy code inside the src/ directory.
* Promotes modularization and better organization.

**Choosing the Right Strategy**

When selecting a strategy, consider:

* Size of the organization
* Team structure
* Complexity of pipelines
* Need for versioning and independent release cycles

By aligning shared library strategies with organizational needs, Jenkins pipelines can be more maintainable, scalable, and efficient.

## Best practices about JSL strategies

### 1. Avoiding Large Global Variable Declaration Files

Instead of maintaining one huge file with all variables, split them into **environment-specific files** and **load only what is needed**.

**✅ Example: Using Environment-Specific Variable Files**

* **Before (Inefficient Global File – vars/globalVars.groovy)**
* return [
* dev: [
* url: "http://dev.example.com",
* db: "dev\_db"
* ],
* staging: [
* url: "http://staging.example.com",
* db: "staging\_db"
* ],
* prod: [
* url: "http://prod.example.com",
* db: "prod\_db"
* ]
* ]

**Issue:** Every pipeline loads **all** environments, even when it only needs one.

* **After (Efficient, Environment-Specific Files)**
  + **vars/devVars.groovy**
  + return [
  + url: "http://dev.example.com",
  + db: "dev\_db"
  + ]
  + **vars/stagingVars.groovy**
  + return [
  + url: "http://staging.example.com",
  + db: "staging\_db"
  + ]
  + **vars/prodVars.groovy**
  + return [
  + url: "http://prod.example.com",
  + db: "prod\_db"
  + ]
  + **Jenkinsfile Usage:**
  + def envConfig = load "vars/${ENV}.groovy"
  + echo "Deploying to ${envConfig.url} using DB ${envConfig.db}"

**✅ Benefit:** Only the required file is loaded, reducing memory usage and improving performance.

### 2. Avoiding Large Shared Libraries

Instead of having **one huge library**, create **smaller modular shared libraries**.

**✅ Example: Using Modular Libraries**

* **Before (Large Library)**
  + **src/org/company/utils.groovy**
  + package org.company
  + class Utils {
  + static void buildApp() {
  + println "Building the application..."
  + }
  + static void deployApp(String env) {
  + println "Deploying to ${env}..."
  + }
  + static void runTests() {
  + println "Running tests..."
  + }
  + }
  + **Issue:** Even if a job only needs deployApp(), it must load the entire Utils class, increasing memory overhead.
* **After (Efficient, Split by Functionality)**
  + **src/org/company/buildUtils.groovy**
  + package org.company
  + class BuildUtils {
  + static void buildApp() {
  + println "Building the application..."
  + }
  + }
  + **src/org/company/deployUtils.groovy**
  + package org.company
  + class DeployUtils {
  + static void deployApp(String env) {
  + println "Deploying to ${env}..."
  + }
  + }
  + **src/org/company/testUtils.groovy**
  + package org.company
  + class TestUtils {
  + static void runTests() {
  + println "Running tests..."
  + }
  + }
  + **Jenkinsfile Usage:**
  + @Library('my-shared-lib') \_
  + import org.company.BuildUtils
  + import org.company.DeployUtils
  + import org.company.TestUtils
  + pipeline {
  + agent any
  + stages {
  + stage('Build') {
  + steps {
  + script {
  + BuildUtils.buildApp()
  + }
  + }
  + }
  + stage('Deploy') {
  + steps {
  + script {
  + DeployUtils.deployApp('staging')
  + }
  + }
  + }
  + stage('Test') {
  + steps {
  + script {
  + TestUtils.runTests()
  + }
  + }
  + }
  + }
  + }

**✅ Benefit:** Only the required functions are loaded into memory, making the pipeline **faster and more efficient**.

### 3. Using Folder-Level Shared Libraries

Instead of defining **global shared libraries** for all jobs, define them at the **folder level**.

* **Jenkins Global Shared Library Configuration (Not Recommended)**
* @Library('global-shared-library') \_
* **Folder-Level Shared Library Configuration (Recommended)**
* @Library('folder-specific-library') \_

**✅ Benefit:** Jobs in a folder load **only** the required library, reducing memory usage and improving execution speed.

## Reference links

<https://docs.google.com/presentation/d/1Q0fBqoFQQXVozxO4urNd75hb1c7rmiWK2nt6MpLfJU4/edit#slide=id.g260ab00d3e7_0_63>

<https://www.jenkins.io/doc/book/pipeline/shared-libraries/>

<https://docs.cloudbees.com/docs/cloudbees-ci/latest/pipelines/pipeline-best-practices>