



Jenkins: Shared Libraries in CI/CD Pipelines

5. September 2025

@ Noser E-Days, Europapark





Andreas Nussberger

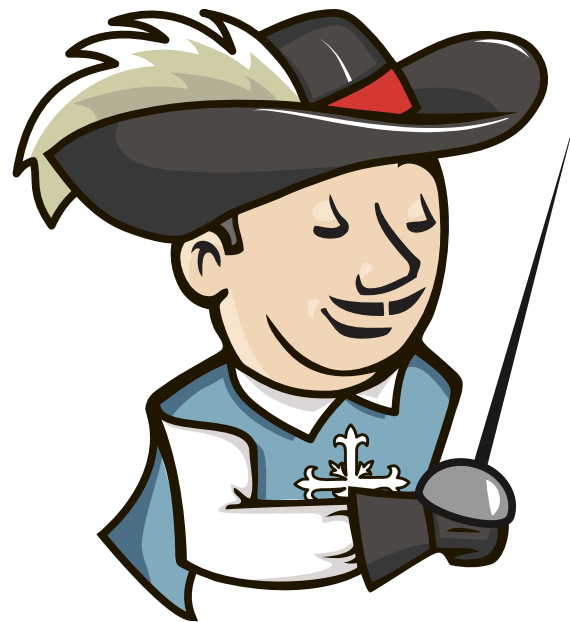
✉ andreas.nussberger@noser.com / ✉ andreas@devminds.ch

- **Senior Software Engineer and IoT Team Lead**
at Noser Engineering AG
- **Trainer and Founder** of *devminds GmbH*
- Education: **Dr. sc. ETH Zürich** (Computer Vision)
- **Highlights**
 - Senior developer in aviation, medical and security industry
 - Leading and pushing research projects in aviation industry
 - PhD thesis: Aerial Object Tracking from an Airborne Platform



Agenda

- **Declarative vs. Scripted Pipelines**
- Using Shared Libraries in Pipelines
- Creating own Steps and Stages
- Summary



Jenkins pipeline: Jenkinsfile



Jenkins pipelines are declared within a `Jenkinsfile` typically located at the root of the repository.



Pipelines can be defined in two flavors: **declarative** and **scripted**

Example **declarative** pipeline

```
1 pipeline {
2     agent any
3     stages {
4         stage('Example stage') {
5             steps {
6                 echo 'Hello world!'
7             }
8         }
9     }
10 }
```

Example **scripted** pipeline

```
1 node {
2     stage('Example stage') {
3         echo 'Hello World'
4     }
5 }
```



Scripted pipelines allow to use Groovy code directly in your `Jenkinsfile` and support integrating *Jenkins Shared Libraries*.



What are scripted pipelines?

Scripted Pipeline, like Declarative Pipeline, is built on top of the underlying Pipeline sub-system. Unlike Declarative, **Scripted Pipeline is effectively a general-purpose DSL built with Groovy.**



Most functionality provided by the Groovy language is made available to users of Scripted Pipeline, which means it can be a very expressive and flexible tool with which one can author continuous delivery pipelines.



 jenkins.io/doc/book/pipeline/syntax/#scripted-pipeline



Scripted pipeline Groovy examples

With scripted pipelines **Groovy** can be used directly within the `Jenkinsfile` :

if/else

```
1  node {
2    stage('Example stage') {
3      if (env.BRANCH_NAME == 'main') {
4        echo 'we are on main branch!'
5      } else {
6        echo 'some other branch...'
7      }
8    }
9  }
```

try/catch

```
1  node {
2    stage('Example stage') {
3      try {
4        echo 'this will force an error'
5        sh 'exit 1'
6      }
7      catch (exc) {
8        echo 'something failed!'
9        throw
10     }
11   }
12 }
```

Declarative vs Scripted pipelines



Scripted pipeline

```
1 pipeline {
2   agent any
3   options {
4     disableConcurrentBuilds()
5   }
6   parameters {
7     booleanParam(
8       name: 'FLAG',
9       defaultValue: true,
10      description: 'Flag... '
11    )
12  }
13  triggers {
14    pollSCM('H/5 * * * *')
15  }
16  stages {
17    stage('Example stage') {
18      steps {
19        echo "Hello world: ${params.FLAG}"
20      }
21    }
22  }
23 }
```

```
1 node {
2   properties([
3     disableConcurrentBuilds(),
4     parameters([
5       booleanParam(
6         name: 'FLAG',
7         defaultValue: true,
8         description: 'Flag... '
9       )
10    ]),
11   pipelineTriggers([
12     pollSCM('H/5 * * * *')
13   ])
14 ])
15 stage('Example stage') {
16   checkout scm
17   echo "Hello world: ${params.FLAG}"
18 }
19 }
```

Comparison of Declarative and Scripted pipelines



Scripted pipelines...

- start with a `node` keyword instead of `pipeline`
- do **NOT** checkout the SCM automatically
- do **NOT** use the `options`, `parameters` and `triggers` keywords - instead everything is handled using `properties`
- do **NOT** use the `stages` keyword
- do **NOT** use the `steps` keyword
- 🚀 use the same steps with the same syntax as the declarative pipelines
- 🚀 allow Groovy usage within the Jenkinsfile

```
1  node {
2      properties([
3          disableConcurrentBuilds(),
4          parameters([
5              booleanParam(
6                  name: 'FLAG',
7                  defaultValue: true,
8                  description: 'Flag... '
9              )
10         ]),
11         pipelineTriggers([
12             pollSCM('H/5 * * * *')
13         ])
14     ])
15     stage('Example stage') {
16         checkout scm
17         echo "Hello world: ${params.FLAG}"
18     }
19 }
```

... and some more differences as we will see



Scripted Jenkins agent labels

Declaring agent labels is handled differently between declarative and scripted pipelines.

Declarative pipeline

In declarative pipelines, the agent label is specified with the `agent` keyword:

```
1 pipeline {
2   agent label: 'label1 || label2'
3   stages {
4     stage('Example stage') {
5       steps {
6         echo 'Hello world!'
7       }
8     }
9   }
10 }
```

Scripted pipeline

In scripted pipelines, the agent label is directly specified within the `node` keyword:

```
1 node('label1 || label2') {
2   stage('Example stage') {
3     echo 'Hello world!'
4   }
5 }
```

Scripted Jenkins Docker agents: Docker images



Docker agents are handled differently between declarative and scripted pipelines. Make sure to check the corresponding documentation: [🌐 jenkins.io/doc/book/pipeline/docker](https://jenkins.io/doc/book/pipeline/docker)

Declarative pipeline

```
1 pipeline {
2     agent {
3         docker {
4             image 'ubuntu:24.04'
5             args '-v /path/on/host:/path/in/container'
6         }
7     }
8     stages {
9         stage('Example stage') {
10             steps {
11                 echo 'Hello world!'
12             }
13         }
14     }
15 }
```

Scripted pipeline

```
1 node {
2     docker
3     .image('ubuntu:24.04')
4     .inside('-v /path/on/host:/path/in/container') {
5         stage('Example stage') {
6             echo 'Hello world!'
7         }
8     }
9 }
```

Scripted Jenkins Docker agents: Dockerfiles



i For scripted pipelines using Dockerfiles we have to manually checkout the SCM first!

Declarative pipeline

```
1  pipeline {
2      agent {
3          dockerfile {
4              filename 'Dockerfile'
5              dir 'build'
6              args '-v /path/on/host:/path/in/container'
7          }
8      }
9      stages {
10         stage('Example stage') {
11             echo 'Hello world!'
12         }
13     }
14 }
```

Scripted pipeline

```
1  node {
2      // First we require to checkout the SCM
3      // as it contains the Dockerfile
4      stage('Checkout SCM') {
5          checkout scm
6      }
7      def customImage = docker.build(
8          'my-image:latest', '-f Dockerfile ./'
9      )
10     customImage.inside() {
11         stage('Example stage') {
12             echo 'Hello world!'
13         }
14     }
15 }
```



Parallel builds with Scripted pipelines

The  `parallel` keyword allows to execute stages in parallel for Declarative and Scripted pipelines:

Declarative pipeline

```
1  stage('Parallel build') {
2      parallel {
3          stage('first') {
4              // ...
5          }
6          stage('second') {
7              // ...
8          }
9      }
10 }
```

Scripted pipeline

```
1  stage('Parallel build') {
2      parallel first: {
3          // use a node, stage or step here...
4      },
5      second: {
6          // use a node, stage or step here...
7      }
8  }
```

 The `parallel` keyword must be called as child of a `stage` !

Environment variables in Scripted pipelines



i Scripted pipelines use `withEnv` instead of the `environment` keyword!

Declarative pipeline

```
1 pipeline {
2     agent any
3     environment {
4         GLOBAL_ENV = 'hello global'
5     }
6     stages {
7         stage('Example stage') {
8             environment {
9                 STAGE_ENV = 'hello stage'
10            }
11            steps {
12                sh 'printenv'
13            }
14        }
15    }
16 }
```

Scripted pipeline

```
1 node {
2     withEnv(['GLOBAL_ENV="hello global"']) {
3         stage('Example stage') {
4             withEnv(['STAGE_ENV="hello stage"']) {
5                 sh 'printenv'
6             }
7         }
8     }
9 }
```

Jenkins credentials in Scripted pipelines



i Scripted pipelines use `withCredentials` instead of the `credentials` keyword!

Declarative pipeline

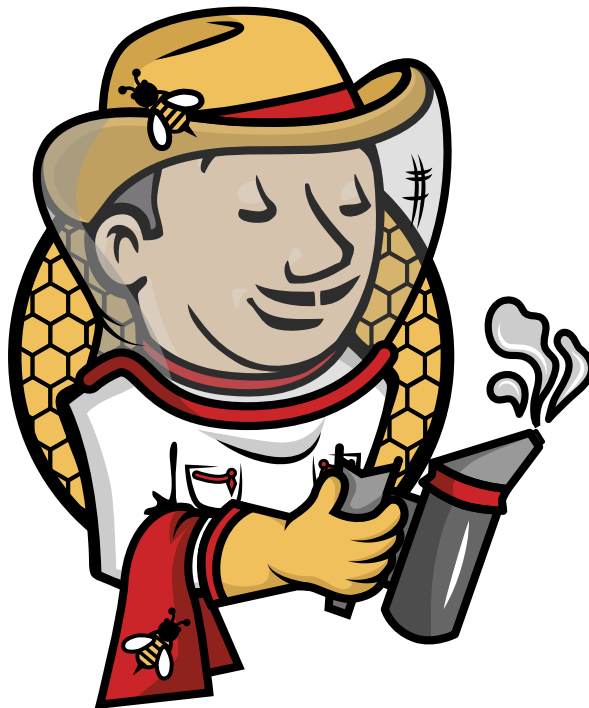
```
1 pipeline {
2   agent any
3   stages {
4     stage('Example stage') {
5       environment {
6         CREDENTIALS =
7           credentials('my-credentials')
8       }
9       steps {
10        // do something with:
11        // CREDENTIALS
12        // CREDENTIALS_USR
13        // CREDENTIALS_PSW
14      }
15    }
16  }
17 }
```

Scripted pipeline

```
1 node {
2   stage('Example stage') {
3     withCredentials([
4       usernamePassword(
5         credentialsId: 'my-credentials',
6         usernameVariable: 'CREDENTIALS_USR',
7         passwordVariable: 'CREDENTIALS_PSW'
8       )
9     ]) {
10      // do something with:
11      // CREDENTIALS_USR
12      // CREDENTIALS_PSW
13    }
14  }
15 }
```

Agenda

- Declarative vs. Scripted Pipelines
- **Using Shared Libraries in Pipelines**
- Creating own Steps and Stages
- Summary





Jenkins shared libraries

Shared Libraries allow to share parts of pipelines between various projects to reduce redundancies and keep code "DRY".

jenkins.io/doc/book/pipeline/shared-libraries

Introduction to Shared Libraries



Jenkins shared libraries:

- are defined in a SCM repository
- can be configured on the Jenkins controller
- have a unique name for each SCM repository
- can be used within pipelines

Check the documentation for more information:

 jenkins.io/doc/book/pipeline/shared-libraries

Shared Library example usage

```
1  @Library('shared-library@main') _  
2  
3  log.info 'Staring ... '
```

The Shared Library used in this example contains a `vars/log.groovy` with the following function inside:

```
1  def info(message) {  
2      echo "INFO: ${message}"  
3  }
```

Required Shared Library directory structure



For Jenkins shared libraries to be picked up by the Jenkins controller, the following directory structure has to be used:

```
1  .
2  ├── resources
3      └── org
4          └── foo
5              └── bar.json
6  ├── src
7      └── org
8          └── foo
9              └── Bar.groovy
10 ├── vars
11     ├── foo.groovy
12     └── foo.txt
```

resources

The `resources` folder allows to store static resource files used by source files from the `src` folder.

src

The `src` folder is where Groovy source files are stored. These files are typically used by functionality implemented in the `vars` folder.

Required Shared Library directory structure



The most powerful feature of Shared Libraries are Groovy files within the `vars` subfolder!

These Groovy files are exposed as **global variables** and...

- ... functions inside can be accessed using:

```
<file-name>.<function-name>
```

Example for file `vars/log.groovy` containing `def info(message) {}`:

```
@Library('shared-library@main') _  
  
log.info 'Staring... '
```

- ... allow to implement **custom steps** by implementing the `call` function:

```
def call() {  
    // do something  
}
```

Using Shared Libraries in pipelines

For Shared Libraries to be available within pipeline scripts they have to be registered at the Jenkins controller:

Register a Shared Library

Manage -> System -> Global Trusted Pipeline Libraries:

- `https://<jenkins>/manage/configure`
- Name: `shared-library`
- Default version: `tag or branch name`
- Load implicitly: automatically load this shared library
 - Otherwise: pipelines have to load the library explicitly using `@Library`



Global Trusted Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

Library

Name ?

shared-library

Default version ?

main

Currently maps to revision: 9f664087bc7055ae3f120346bcfd359bdd8f802

☐ Load implicitly ?

☒ Allow default version to be overridden ?

☒ Include @Library changes in job recent changes ?

☐ Cache fetched versions on controller for quick retrieval ?

Retrieval method

Modern SCM

Loads a library from an SCM plugin using newer interfaces optimized for this purpose. The recommended option when available.

Source Code Management

Git

Project Repository ?

http://gitea.lan:3000/admin/shared-library.git

Credentials ?

- none -

+ Add

Behaviors

Discover branches ?



Using Shared Libraries in pipelines

For Shared Libraries to be available within pipeline scripts they have to be configured with **load implicitly** or must be explicitly loaded using `@Library` :

Use a Shared Library

Example library load statements:

```
1 // Loading latest version of default branch
2 @Library('shared-library') _
3 // Loading a specific branch
4 @Library('shared-library@main') _
5 // Loading a specific tag
6 @Library('shared-library@1.0.0') _
```

Use specific imports

If the `_` is omitted we can also import specific classes (**NOT** global variables!) to be used:

```
1 @Library('shared-library')
2 import com.mycorp.pipeline.somelib.UsefulClass
```



The trailing `_` (underscore) is a placeholder to **import everything** of the Shared Library

If global variables are used we should stick to the `_` placeholder syntax!



Versioning of Shared Libraries

Example workflow

- Create branch in Shared Library repository
- Create branch in dev repository containing Jenkinsfile
- Modify `@Library()` to use the new branch
- Change the Shared Library code
- Tag the new Shared Library release
- Modify `@Library()` to use new new tag
- **Fast-forward merge** the Shared Library branch into main - *otherwise the Git hash will change!*
- Merge the dev repository branch into main

Versioning

- **Make sure to use tags of Shared Libraries:**

```
@Library('shared-library@1.0.0') _
```

- Use  semantic versioning for smooth updates



If the Shared Library version is not locked to a specific release, old tags or branches will likely fail over time in Jenkins as their Jenkinsfile is not compatible anymore with the current Shared Library code!



DEMO

- Creating a shared library
- Registering the shared library
- Using the shared library

Agenda

- Declarative vs. Scripted Pipelines
- Using Shared Libraries in Pipelines
- **Creating own Steps and Stages**
- Summary



Shared Library custom steps



The Groovy files exposed as **global variables** within the `vars` folder allow to implement custom pipeline steps - **this allows to share code across pipelines and keep code DRY!**

Example custom step

File `vars/log.groovy` :

```
1  def call(String message = 'empty') {  
2      echo "Logging: ${message}"  
3  }
```

Pipeline usage

Usage within a scripted pipeline `Jenkinsfile` :

```
1  @Library('shared-library@main') _  
2  
3  log 'some message to be logged'
```

Output: `Logging: some message to be logged`

Execute with default arguments:

```
1  log()
```

Output: `Logging: empty`

Shared Library custom steps with Map parameters

If we use a `Map` as function parameter of the `call` function **we are able to implement a higher level DSL for our pipeline scripts:**

Example custom step

File `vars/customStage.groovy`:

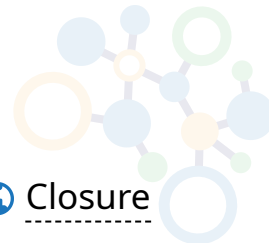
```
1 // Custom stage
2 // title: Stage title
3 // message: Stage message
4 def call(Map config) {
5     stage("${config.title}") {
6         echo "${config.message}"
7     }
8 }
```

Pipeline usage

Usage within a scripted pipeline `Jenkinsfile`:

```
1 @Library('shared-library@main') _
2
3 node {
4     // Agent preparation, checkout, etc. ...
5     customStage(
6         title: 'Custom Stage',
7         message: 'Hello world!'
8     )
9 }
```

Shared Library custom steps with Closures



If a custom step is called with a block as argument, the `call` method will receive a Groovy  Closure

This allows to wrap pipeline code with some context:

Example custom step

File `vars/linux.groovy` :

```
1  def call(Closure body) {
2      node('linux') {
3          body()
4      }
5  }
```

Pipeline usage

Usage within a scripted pipeline `Jenkinsfile` :

```
1  @Library('shared-library@main') _
2
3  linux {
4      echo 'this will be executed on a Linux node'
5  }
```



Example: CMake build stage

Workflow

Building a CMake project in Jenkins typically involves the following steps:

- Run CMake configure:

```
cmake --preset <preset-name>
```

- Run CMake build:

```
cmake --build --preset <preset-name> --target <target-name>
```

- Record compiler issues using Jenkins

```
recordIssues
```

- Archive artifacts using Jenkins `archiveArtifacts`

Example Jenkinsfile

```
1  node {
2      // Agent preparation, checkout, etc. ...
3      stage('Build CMake project') {
4          sh "cmake --preset gcc-release"
5          sh ""cmake --build
6              --preset gcc-release
7              --target cplusplus_training_project
8          ""
9          recordIssues(
10             sourceCodeRetention: 'LAST_BUILD',
11             tools: [
12                 gcc()
13             ]
14         )
15         archiveArtifacts(
16             artifacts: "build/gcc-release/bin/ ... ",
17             onlyIfSuccessful: true
18         )
19     }
20 }
```

Example: CMake build stage



```
1 // Custom CMake build stage
2 // preset:      CMake preset to use for the build
3 // target:      CMake target to name to build
4 // artifacts:   [optional] path to artifacts to be archived
5 def call(Map config) {
6     if (!config.containsKey('artifacts')) {
7         config['artifacts'] = null
8     }
9     stage("Build: ${config.target} [${config.preset}]") {
10         sh "cmake --preset ${config.preset}"
11         sh """cmake --build
12             --preset ${config.preset}
13             --target ${config.target}
14             """
15         // recordIssues ...
16         if (config.artifacts) {
17             archiveArtifacts(
18                 artifacts: config.artifacts,
19                 onlyIfSuccessful: true
20             )
21         }
22     }
23 }
```

Example Jenkinsfile

```
1 @Library('shared-library@main') _
2 node {
3     // Agent preparation, checkout, etc. ...
4     stageCMakeBuild(
5         preset: "gcc-release",
6         target: 'cplusplus_training_project',
7         artifacts: "build/gcc-release/bin/ ..."
8     )
9 }
```

Note about implementing reusable custom stages

-  If a custom stage is calling either `recordIssues()` from the `warnings-ng` plugin or `recordCoverage()` from the `coverage` plugin we have to make sure multiple calls to the same function do not reuse the default tool `id` !

Therefore we have to specify the `id` parameter of the corresponding function manually:

warnings-ng

```
1 recordIssues(  
2     name: "GCC Issues",  
3     id: "<unique-ID>",  
4     sourceCodeRetention: 'LAST_BUILD',  
5     tools: [  
6         gcc()  
7     ]  
8 )
```

coverage

```
1 recordCoverage(  
2     name: "GCC Coverage",  
3     id: "<unique-ID>",  
4     tools: [  
5         [parser: 'JUNIT', pattern: "report-gcc.xml"],  
6         [parser: 'COBERTURA', pattern: "coverage-gcc.xml"]  
7     ]  
8 )
```



DEMO

- Creating a CMake build step

Example: C++ test execution



Workflow

Executing tests for a C++ project in Jenkins typically involves the following steps:

- Build the project with code coverage enabled
- Execute the binary
- Import test results into Jenkins using `junit`
- Post process code coverage data
- Import code coverage into Jenkins using `recordCoverage`
- Optionally: publish HTML coverage reports in Jenkins using `publishHTML`

Example Jenkinsfile

```
1  node {
2    // Agent preparation, checkout, etc. ...
3    stage('Test C++ project') {
4      warnError("Executing tests failed") {
5        sh "..."./path/to/executable
6          --gtest_output='xml:build/test-report.xml'
7        ""
8      }
9      junit(
10         testResults: "build/test-report.xml"
11       )
12      sh "gcovr ..." // Post-process coverage data
13      recordCoverage(
14         name: "Code coverage",
15         tools: [[
16           parser: 'COBERTURA',
17           pattern: "build/test-coverage.xml"
18         ]]
19      )
20    }
21  }
```

Example: C++ test execution



```
1 // Custom test execution stage
2 // run:  binary under test to be executed
3 // name: the test name
4 def call(Map config) {
5     stage("Run test: ${config.name}") {
6         warnError("${config.name} tests failed") {
7             sh """${config.run}
8                 --gtest_output='xml:build/test-report.xml'
9                 """
10        }
11        junit(
12            testResults: "build/test-report.xml"
13        )
14        sh """gcovr -f src . --root ./
15            --exclude-unreachable-branches
16            --xml-pretty --print-summary
17            -o 'build/test-coverage.xml'
18            """
19        // ...
20    }
21}
```

```
1 // ...
2 recordCoverage(
3     name: "Coverage: ${config.name}",
4     id: "<unique-name>",
5     tools: [[
6         parser: 'COBERTURA',
7         pattern: "build/test-coverage.xml"
8     ]]
9 )
10 // ...
```

Example Jenkinsfile

```
1 @Library('shared-library@main') _
2 node {
3     // Agent preparation, checkout, etc. ...
4     stageRunTests(
5         run: "./path/to/executable",
6         name: "Calculate test [gcc]"
7     )
8 }
```



DEMO

- Creating a test execution step

Agenda

- Declarative vs. Scripted Pipelines
- Using Shared Libraries in Pipelines
- Creating own Steps and Stages
- **Summary**



Summary



Jenkins Shared Library benefits:

- 👍 Avoid duplicate Groovy code across pipelines
- 👍 Clean and readable Jenkinsfiles

Remember:

- ⚠️ make sure to reference fixed versions of Shared Libraries!
- ⚠️ think twice about how to test / debug pipeline code!

Keep your pipeline logic as simple as possible and move complex logic to scripts:

- 👍 Pipeline uses the same logic as the developer
- 👍 Scripts can be executed / debugged on the host

Appendix

Links



- **Jenkins user documentation:** jenkins.io/doc
- **Jenkins shared library documentation:** jenkins.io/doc/book/pipeline/shared-libraries
- **Pipeline examples:** github.com/jenkinsci/pipeline-examples
- **Shared library examples:** github.com/darinpope/github-api-global-lib
- **Unit-testable Jenkins shared libraries:**
dev.to/kuperadrian/how-to-setup-a-unit-testable-jenkins-shared-pipeline-library-2e62
- **Continuous delivery:** continuousdelivery.com
- **Jenkins logos:** © by the [Jenkins project](https://jenkins.io) and licensed under [CC BY-SA 3.0](https://creativecommons.org/licenses/by-sa/3.0/)

Books

- **Accelerate: The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations**, 2018, Nicole Forsgren, Jez Humble, Gene Kim



devminds
enabling people



Trainings

 Ansible for developers

 Docker for developers

 Git

 GitLab CI/CD

 Jenkins CI/CD

Contact

 info@devminds.ch

 devminds.ch