Extending with Shared Libraries

Table of Contents

As Pipeline is adopted for more and more projects in an organization, common patterns are likely to emerge. Oftentimes it is useful to share parts of Pipelines between various projects to reduce redundancies and keep code "DRY" [1].

Pipeline has support for creating "Shared Libraries" which can be defined in external source control repositories and loaded into existing Pipelines.

Defining Shared Libraries

A Shared Library is defined with a name, a source code retrieval method such as by SCM, and optionally a default version. The name should be a short identifier as it will be used in scripts.

The version could be anything understood by that SCM; for example, branches, tags, and commit hashes all work for Git. You may also declare whether scripts need to explicitly request that library (detailed below), or if it is present by default. Furthermore, if you specify a version in Jenkins configuration, you can block scripts from selecting a *different* version.

The best way to specify the SCM is using an SCM plugin which has been specifically updated to support a new API for checking out an arbitrary named version (*Modern SCM* option). As of this writing, the latest versions of the Git and Subversion plugins support this mode; others should follow.

If your SCM plugin has not been integrated, you may select *Legacy SCM* and pick anything offered. In this case, you need to include `${library.yourLibName.version}` somewhere in the configuration of the SCM, so that during checkout the plugin will expand this variable to select the desired version. For example, for Subversion, you can set the *Repository URL* to `svnserver/project/${library.yourLibName.version}` and then use versions such as `trunk` or `branches/dev` or `tags/1.0`.

Directory structure

The directory structure of a Shared Library repository is as follows:

```
(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. This directory is added to the classpath when executing Pipelines.

The `vars` directory hosts scripts that define global variables accessible from Pipeline. The basename of each **`.groovy` file should be a Groovy (~ Java) identifier, conventionally `camelCased`. The matching `.txt`**, if present, can contain documentation, processed through the system's configured markup formatter (so may really be HTML, Markdown, etc., though the `txt` extension is required).

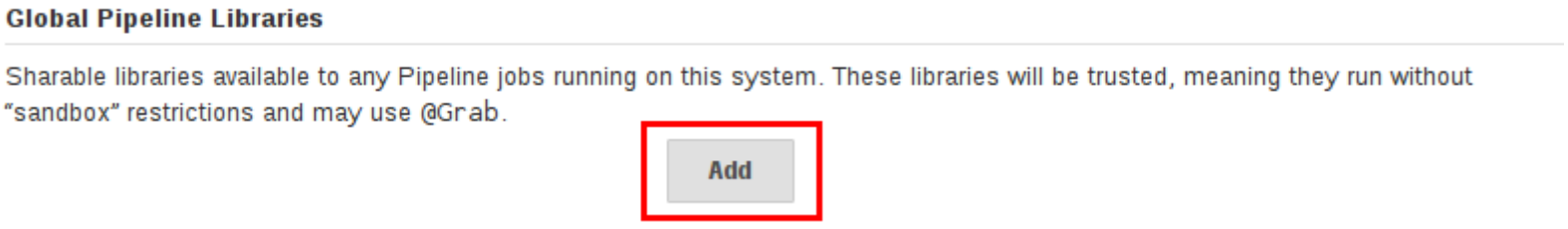
The Groovy source files in these directories get the same "CPS transformation" as in Scripted Pipeline.

A `resources` directory allows the `libraryResource` step to be used from an external library to load associated non-Groovy files. Currently this feature is not supported for internal libraries.

Other directories under the root are reserved for future enhancements.

Global Shared Libraries

There are several places where Shared Libraries can be defined, depending on the use-case. *Manage Jenkins » Configure System » Global Pipeline Libraries* as many libraries as necessary can be configured.



Since these libraries will be globally usable, any Pipeline in the system can utilize functionality implemented in these libraries.

These libraries are considered "trusted:" they can run any methods in Java, Groovy, Jenkins internal APIs, Jenkins plugins, or third-party libraries. This allows you to define libraries which encapsulate individually unsafe APIs in a higher-level wrapper safe for use from any Pipeline. Beware that **anyone able to push commits to this SCM repository could obtain unlimited access to Jenkins**. You need the *Overall/RunScripts* permission to configure these libraries (normally this will be granted to Jenkins administrators).

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.

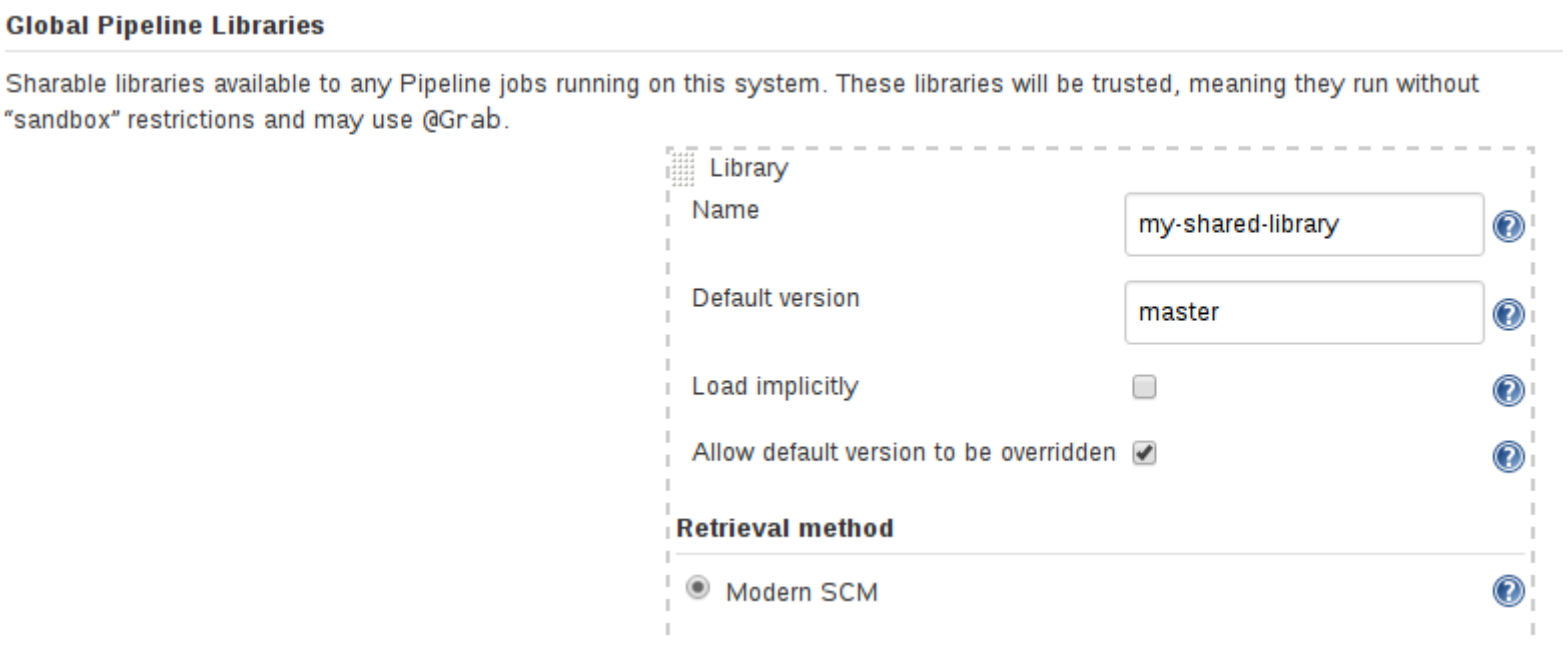
Folder-based libraries are not considered "trusted:" they run in the Groovy sandbox just like typical Pipelines.

Automatic Shared Libraries

Other plugins may add ways of defining libraries on the fly. For example, the GitHub Branch Source plugin provides a "GitHub Organization Folder" item which allows a script to use an untrusted library such as `github.com/someorg/somerepo` without any additional configuration. In this case, the specified GitHub repository would be loaded, from the `master` branch, using an anonymous checkout.

Using libraries

Shared Libraries marked *Load implicitly* allows Pipelines to immediately use classes or global variables defined by any such libraries. To access other shared libraries, the `Jenkinsfile` needs to use the `@Library` annotation, specifying the library's name:




```
@Library('my-shared-library') _
/* Using a version specifier, such as branch, tag, etc */
@Library('my-shared-library@1.0') _
/* Accessing multiple libraries with one statement */
@Library(['my-shared-library', 'otherlib@abc1234']) _
```

The annotation can be anywhere in the script where an annotation is permitted by Groovy. When referring to class libraries (with `src/` directories), conventionally the annotation goes on an `import` statement:

```
@Library('somelib')
import com.mycorp.pipeline.somelib.UsefulClass
```

For Shared Libraries which only define Global Variables (`vars/`), or a `Jenkinsfile` which only needs a Global Variable, the annotation pattern `@Library('my-shared-library') _` may be useful for keeping code concise. In essence, instead of

annotating an unnecessary `import` statement, the symbol _ is annotated.

It is not recommended to `import` a global variable/function, since this will force the compiler to interpret fields and methods as `static` even if they were intended to be instance. The Groovy compiler in this case can produce confusing error messages.

Libraries are resolved and loaded during *compilation* of the script, before it starts executing. This allows the Groovy compiler to understand the meaning of symbols used in static type checking, and permits them to be used in type declarations in the script, for example:

```
@Library('somelib')
import com.mycorp.pipeline.somelib.Helper

int useSomeLib(Helper helper) {
    helper.prepare()
    return helper.count()
}

echo useSomeLib(new Helper('some text'))
```

Global Variables however, are resolved at runtime.

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.

If you are only interested in using global variables/functions (from the `vars/` directory), the syntax is quite simple:

```
library 'my-shared-library'
```

Thereafter, any global variables from that library will be accessible to the script.

Using classes from the `src/` directory is also possible, but trickier. Whereas the `@Library` annotation prepares the "classpath" of the script prior to compilation, by the time a `library` step is encountered the script has already been compiled. Therefore you cannot `import` or otherwise "statically" refer to types from the library.

However you may use library classes dynamically (without type checking), accessing them by fully-qualified name from the return value of the `library` step. `static` methods can be invoked using a Java-like syntax:

```
library('my-shared-library').com.mycorp.pipeline.Utils.someStaticMethod()
```

You can also access `static` fields, and call constructors as if they were `static` methods named `new`:

```
def useSomeLib(helper) { // dynamic: cannot declare as Helper
    helper.prepare()
    return helper.count()
}

def lib = library('my-shared-library').com.mycorp.pipeline // preselect the package

echo useSomeLib(lib.Helper.new(lib.Constants.SOME_TEXT))
```

Library versions

The "Default version" for a configured Shared Library is used when "Load implicitly" is checked, or if a Pipeline references the library only by name, for example `@Library('my-shared-library') _`. If a "Default version" is **not** defined, the Pipeline must specify a version, for example `@Library('my-shared-library@master') _`.

If "Allow default version to be overridden" is enabled in the Shared Library's configuration, a `@Library` annotation may also override a default version defined for the library. This also allows a library with "Load implicitly" to be loaded from a different version if necessary.

When using the `library` step you may also specify a version:

```
library 'my-shared-library@master'
```

Since this is a regular step, that version could be *computed* rather than a constant as with the annotation; for example:

```
library "my-shared-library@$BRANCH_NAME"
```

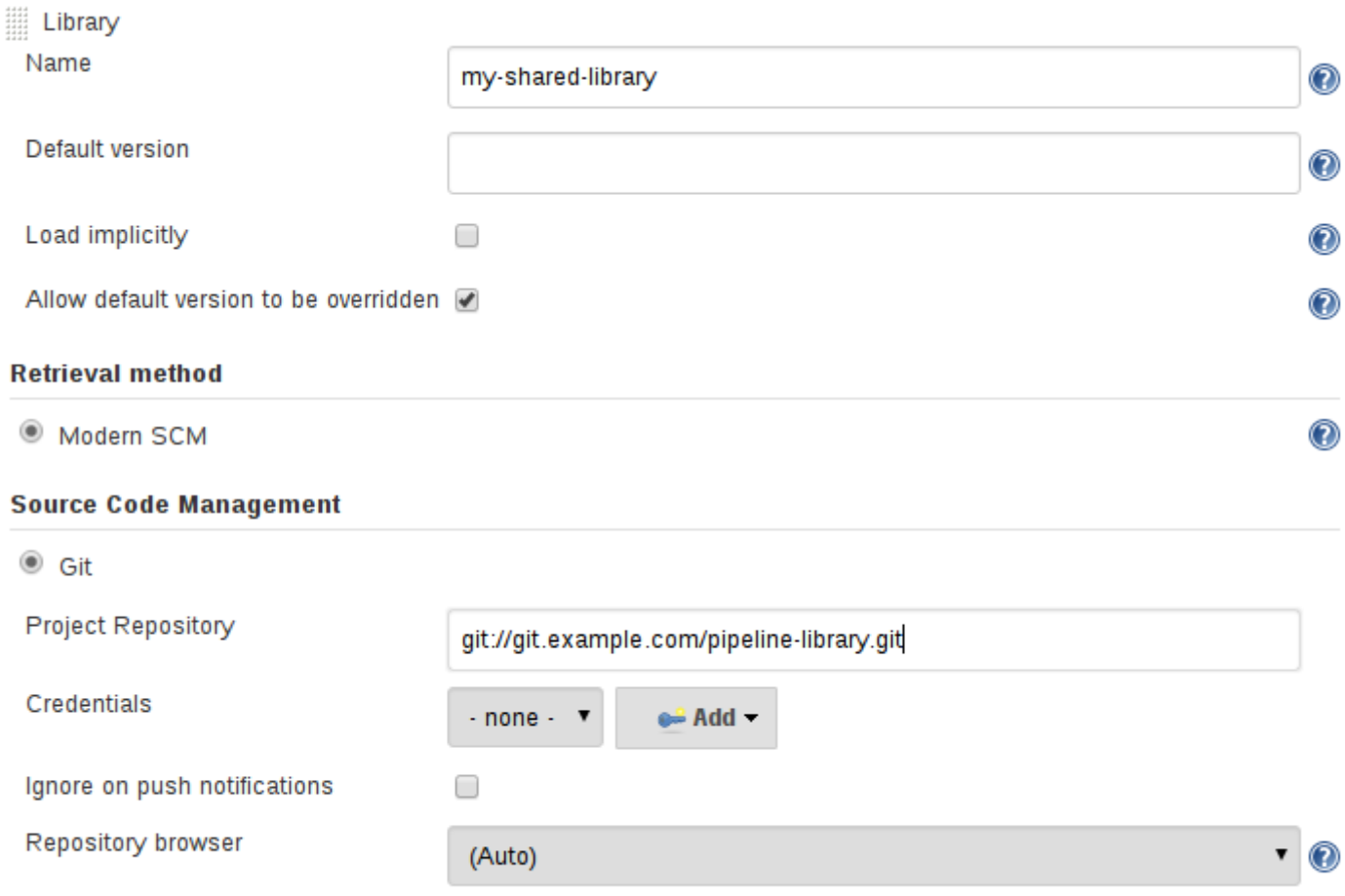
would load a library using the same SCM branch as the multibranch `Jenkinsfile`. As another example, you could pick a library by parameter:

```
properties([parameters([string(name: 'LIB_VERSION', defaultValue: 'master')])])
library "my-shared-library@${params.LIB_VERSION}"
```

Note that the `library` step may not be used to override the version of an implicitly loaded library. It is already loaded by the time the script starts, and a library of a given name may not be loaded twice.

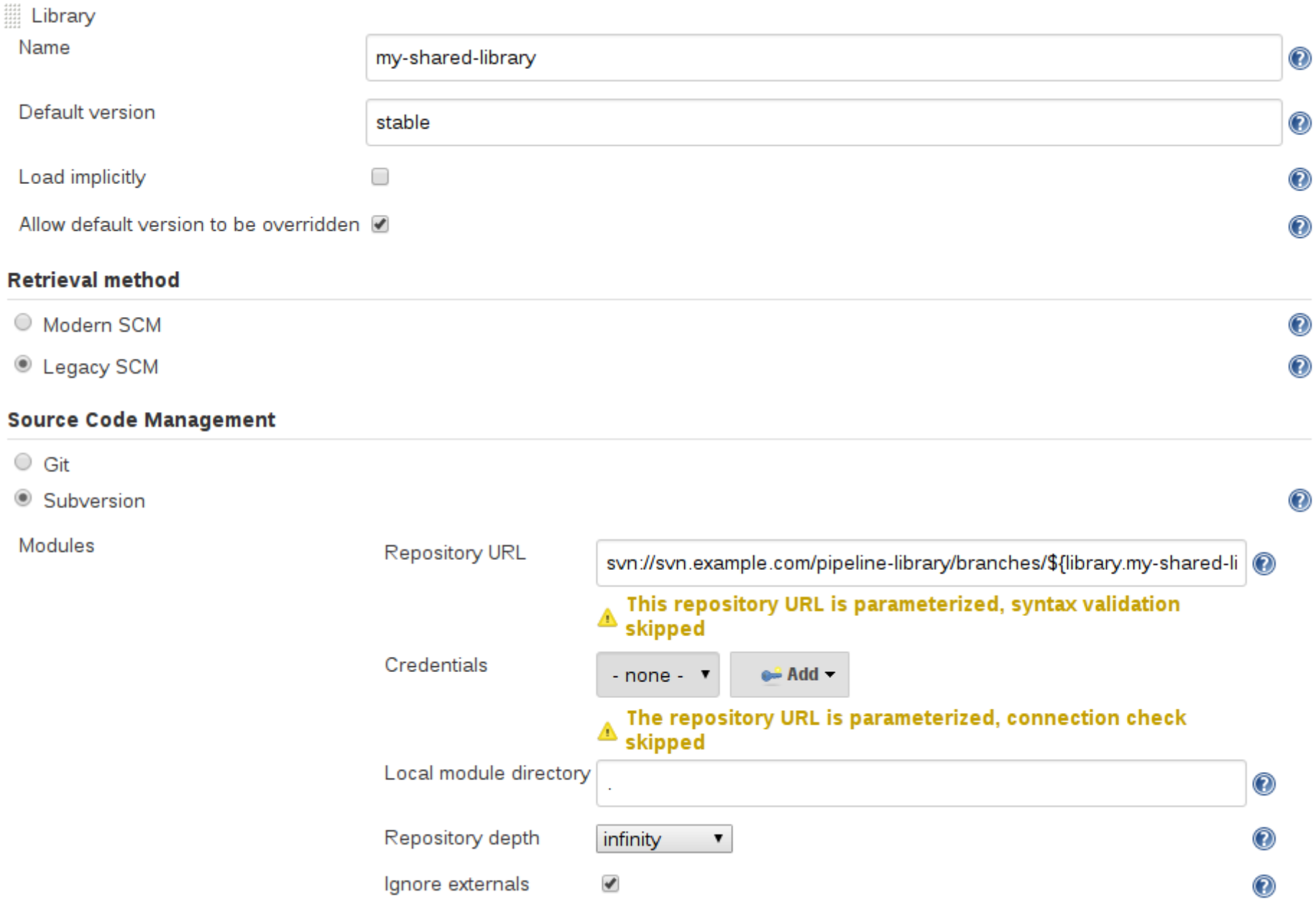
Retrieval Method

The best way to specify the SCM is using an SCM plugin which has been specifically updated to support a new API for checking out an arbitrary named version (**Modern SCM** option). As of this writing, the latest versions of the Git and Subversion plugins support this mode.



Legacy SCM

SCM plugins which have not yet been updated to support the newer features required by Shared Libraries, may still be used via the **Legacy SCM** option. In this case, include `${library.yourlibrarynamehere.version}` wherever a branch/tag/ref may be configured for that particular SCM plugin. This ensures that during checkout of the library's source code, the SCM plugin will expand this variable to checkout the appropriate version of the library.



Dynamic retrieval

If you only specify a library name (optionally with version after `@`) in the `library` step, Jenkins will look for a preconfigured library of that name. (Or in the case of a `github.com/owner/repo` automatic library it will load that.)

But you may also specify the retrieval method dynamically, in which case there is no need for the library to have been predefined in Jenkins. Here is an example:

```
library identifier: 'custom-lib@master', retriever: modernSCM(
  [$class: 'GitSCMSource',
   remote: 'git@git.mycorp.com:my-jenkins-utils.git',
   credentialsId: 'my-private-key'])
```

It is best to refer to **Pipeline Syntax** for the precise syntax for your SCM.

Note that the library version *must* be specified in these cases.

Writing libraries

At the base level, any valid Groovy code is okay for use. Different data structures, utility methods, etc, such as:

```
// src/org/foo/Point.groovy
package org.foo

// point in 3D space
class Point {
  float x,y,z
}
```

Accessing steps

Library classes cannot directly call steps such as `sh` or `git`. They can however implement methods, outside of the scope of an enclosing class, which in turn invoke Pipeline steps, for example:

```
// src/org/foo/Zot.groovy
package org.foo

def checkOutFrom(repo) {
  git url: "git@github.com:jenkinsci/${repo}"
}

return this
```

Which can then be called from a Scripted Pipeline:

```
def z = new org.foo.Zot()
z.checkOutFrom(repo)
```

This approach has limitations; for example, it prevents the declaration of a superclass.

Alternately, a set of `steps` can be passed explicitly using `this` to a library class, in a constructor, or just one method:

```
package org.foo
class Utilities implements Serializable {
  def steps
  Utilities(steps) {this.steps = steps}
  def mvn(args) {
    steps.sh "${steps.tool 'Maven'}/bin/mvn -o ${args}"
  }
}
```

When saving state on classes, such as above, the class **must** implement the `Serializable` interface. This ensures that a Pipeline using the class, as seen in the example below, can properly suspend and resume in Jenkins.

```
@Library('utils') import org.foo.Utilities
def utils = new Utilities(this)
node {
  utils.mvn 'clean package'
}
```

If the library needs to access global variables, such as env, those should be explicitly passed into the library classes, or methods, in a similar manner.

Instead of passing numerous variables from the Scripted Pipeline into a library,

```
package org.foo
class Utilities {
```

```
  static def mvn(script, args) {
    script.sh "${script.tool 'Maven'}/bin/mvn -s ${script.env.HOME}/jenkins.xml -o ${args}"
  }
}
```

The above example shows the script being passed in to one `static` method, invoked from a Scripted Pipeline as follows:

```
@Library('utils') import static org.foo.Utilities.*
node {
  mvn this, 'clean package'
}
```

Defining global variables

Internally, scripts in the `vars` directory are instantiated on-demand as singletons. This allows multiple methods to be defined in a single `.groovy` file for convenience. For example:

vars/log.groovy

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

def warning(message) {
    echo "WARNING: ${message}"
}
```

Jenkinsfile

```
@Library('utils') _

log.info 'Starting'
log.warning 'Nothing to do!'
```

Declarative Pipeline does not allow global variable usage outside of a `script` directive (JENKINS-42360).

Jenkinsfile

```
@Library('utils') _

pipeline {
    agent none
    stage ('Example') {
        steps {
            script { (1)
                log.info 'Starting'
                log.warning 'Nothing to do!'
            }
        }
    }
}
```

1 `script` directive required to access global variables in Declarative Pipeline.

A variable defined in a shared library will only show up in *Global Variables Reference* (under *Pipeline Syntax*) after Jenkins loads and uses that library as part of a successful Pipeline run.

Avoid preserving state in global variables

Avoid defining global variables with methods that interact or preserve state. Use a static class or instantiate a local variable of a class instead.

Defining custom steps

Shared Libraries can also define global variables which behave similarly to built-in steps, such as `sh` or `git`. Global variables defined in Shared Libraries **must** be named with all lower-case or "camelCased" in order to be loaded properly by Pipeline. [2]

For example, to define `sayHello`, the file `vars/sayHello.groovy` should be created and should implement a `call` method. The `call` method allows the global variable to be invoked in a manner similar to a step:

```
// vars/sayHello.groovy
def call(String name = 'human') {
    // Any valid steps can be called from this code, just like in other
    // Scripted Pipeline
```

```
    echo "Hello, ${name}."
}
```

The Pipeline would then be able to reference and invoke this variable:

```
sayHello 'Joe'
sayHello() /* invoke with default arguments */
```

If called with a block, the `call` method will receive a `Closure`. The type should be defined explicitly to clarify the intent of the step, for example:

```
// vars/windows.groovy
def call(Closure body) {
    node('windows') {
        body()
    }
}
```

The Pipeline can then use this variable like any built-in step which accepts a block:

```
windows {
    bat "cmd /?"
}
```

Defining a more structured DSL

If you have a lot of Pipelines that are mostly similar, the global variable mechanism provides a handy tool to build a higher-level DSL that captures the similarity. For example, all Jenkins plugins are built and tested in the same way, so we might write a step named `buildPlugin`:

```
// vars/buildPlugin.groovy
def call(Map config) {
    node {
        git url: "https://github.com/jenkinsci/${config.name}-plugin.git"
        sh 'mvn install'
        mail to: '...', subject: "${config.name} plugin build", body: '...'
    }
}
```

Assuming the script has either been loaded as a Global Shared Library or as a Folder-level Shared Library the resulting `Jenkinsfile` will be dramatically simpler:

Jenkinsfile (Scripted Pipeline)

```
buildPlugin name: 'git'
```

There is also a “builder pattern” trick using Groovy’s `Closure.DELEGATE_FIRST`, which permits `Jenkinsfile` to look slightly more like a configuration file than a program, but this is more complex and error-prone and is not recommended.

Using third-party libraries

It is possible to use third-party Java libraries, typically found in Maven Central, from **trusted** library code using the `@Grab` annotation. Refer to the Grape documentation for details, but simply put:

```
@Grab('org.apache.commons:commons-math3:3.4.1')
import org.apache.commons.math3.primes.Primes
void parallelize(int count) {
  if (!Primes.isPrime(count)) {
    error "${count} was not prime"
  }
  // …
}
```

Third-party libraries are cached by default in `~/.groovy/grapes/` on the Jenkins master.

Loading resources

External libraries may load adjunct files from a `resources/` directory using the `libraryResource` step. The argument is a relative pathname, akin to Java resource loading:

```
def request = libraryResource 'com/mycorp/pipeline/somelib/request.json'
```

The file is loaded as a string, suitable for passing to certain APIs or saving to a workspace using `writeFile`.

It is advisable to use an unique package structure so you do not accidentally conflict with another library.

Pretesting library changes

If you notice a mistake in a build using an untrusted library, simply click the *Replay* link to try editing one or more of its source files, and see if the resulting build behaves as expected. Once you are satisfied with the result, follow the diff link from the build's status page, and apply the diff to the library repository and commit.

(Even if the version requested for the library was a branch, rather than a fixed version like a tag, replayed builds will use the exact same revision as the original build: library sources will not be checked out again.)

Replay is not currently supported for trusted libraries. Modifying resource files is also not currently supported during *Replay*.

Defining Declarative Pipelines

Starting with Declarative 1.2, released in late September, 2017, you can define Declarative Pipelines in your shared libraries as well. Here's an example, which will execute a different Declarative Pipeline depending on whether the build number is odd or even:

```
// vars/evenOrOdd.groovy
def call(int buildNumber) {
  if (buildNumber % 2 == 0) {
    pipeline {
      agent any
      stages {
        stage('Even Stage') {
          steps {
            echo "The build number is even"
          }
        }
      }
    }
  } else {
    pipeline {
      agent any
      stages {
        stage('Odd Stage') {
          steps {
            echo "The build number is odd"
          }
        }
      }
    }
  }
}
```

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

evenOrOdd(currentBuild.getNumber())
```

Only entire `pipeline`s can be defined in shared libraries as of this time. This can only be done in `vars/*.groovy`, and only in a `call` method. Only one Declarative Pipeline can be executed in a single build, and if you attempt to execute a second one, your build will fail as a result.

---

1. en.wikipedia.org/wiki/Don't_repeat_yourself
2. gist.github.com/rtyler/e5e57f075af381fce4ed3ae57aa1f0c2

⇐ Using Docker with Pipeline
⇑ Pipeline
Index
Pipeline Development Tools ⇒
