

Shared Libraries dans Jenkins

June-27-17 9:04 AM

Cette tâche a pour but de définir le setup requis sur un poste de travail pour pouvoir développer de façon efficace un plugin Jenkins. Il faut déterminer:

- Les logiciels requis :
Éclipse pour définir le répertoire source du dépôt de "Shared Libraries" avec les fichiers .groovy permettant de définir les structures de données, les méthodes et autres
- Les librairies requises (avoir la bonne version Java et autres).
- Les fichiers à générer (quels sont les outputs du projet Éclipse ? Juste un fichier qui représente le plugin? Ou il y a plusieurs fichiers ?)
- Comment gérer le code source du plugin (Créer un dépôt git pour chaque plugin ou un seul dépôt pour tous nos plugins ? Inclure les fichiers "projets" d'Éclipse dans les dépôts git?)

Introduction

L'utilisation des **Shared libraries** au sein d'une entreprise permet de partager des parties de pipeline entre plusieurs projets. L'objectif visé est alors de réduire les redondances et de maintenir un code concis. Les Shared libraries peuvent être définies dans des répertoires de code source externe et ensuite être chargés dans les pipelines existants. Une **shared library** se définit via un nom, un gestionnaire de code source tel que SCM et optionnellement une version par défaut. La version peut être une **branche**, un **tag** ou tout élément supporté par le SCM utilisé. Le meilleur choix de SCM serait un dont le plugin Jenkins permet de checker un code à version arbitraire; c'est le cas - pour l'heure - de **Git** et **Subversion** (plus de détails dans la documentation <https://jenkins.io/doc/book/pipeline/shared-libraries/>).

Définir des Shared libraries

Structure

La structure d'un répertoire de shared library est la suivante :

```
(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
```

- ✓ Le dossier **src** devrait ressembler à un répertoire standard de fichiers sources **Java**; il est ajouté au **classpath** lors de l'exécution des Pipelines.
- ✓ Le dossier **vars** contient les scripts qui définissent les variables globales accessibles depuis le Pipeline. Le nom de chaque fichier ***.groovy** doit être un identifiant **Groovy**(~ **Java**) **camelCased** et le fichier ***.txt** correspondant, s'il existe, pourrait contenir de la documentation qui sera traité par le "formateur de balise(markup formatter)" du système(il peut donc s'agir en réalité d'un contenu **HTML**, **Markdown**, etc., mais l'extension **txt** est obligatoire).
- ✓ Le dossier **ressources** permet au **step Jenkins libraryResource** d'être utilisée à partir de librairies externes pour charger des fichiers non Groovy; pour l'heure, cette fonctionnalité n'est pas encore supportée pour les librairies internes.

Les autres sous-répertoires de root sont réservées pour de futures améliorations.

Shared libraries Globales

On peut définir des Shared libraries à différents endroits selon l'usage projeté. Dans l'onglet **Administrer Jenkins >> Configurer le système >> Global Pipeline Libraries**, on peut configurer autant de librairies que nécessaire.

Global 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.

Add

Global 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

You must enter a name.

Default version

Load implicitly

☐

Allow default version to be overridden

☒

Retrieval method

☐ Modern SCM

☐ Legacy SCM

Supprimer

Ajouter

Les Shared libraries globales vont être **utilisables partout**, donc tous les Pipelines du système pourront disposer des fonctionnalités implémentées dans ces librairies. Elles sont considérées "**sûres**" c'est à dire qu'ils peuvent exécuter n'importe quelle méthode en Java, Groovy, les APIS internes et plugins de Jenkins ou autres librairies tiers. Ceci permet de définir des librairies qui encapsulent différentes APIs, considérées précaires individuellement, dans une enveloppe (paquetage, couche) haut niveau plus sûre à utiliser dans un Pipeline. Il faudra s'assurer que toute personne capable de faire des commits dans le répertoire SCM contenant ces librairies, obtienne un accès illimité à Jenkins. Pour configurer ce type de librairies, on a besoin de la **permission Overall/RunScripts** (généralement celle-ci est accordée par les administrateurs Jenkins).

Shared libraries pour Répertoire

Tout répertoire créé peut être associé à des shared libraries. Ce mécanisme permet d'étendre la portée de librairies spécifiques à tous les Pipelines d'un répertoire ou sous-répertoire.

Les shared libraries pour répertoire ne sont **pas considérées "sûres"**: ils s'exécutent dans la **sandbox Groovy** comme les Pipelines en général.

Shared libraries Automatiques

Certains plugins offrent de nouvelles façons de définir des librairies à la volée. Par exemple, le plugin **GitHub Branch Source** contient une option **GitHub Organization Folder** permettant à un script de Pipeline d'utiliser une librairie "**considérée pas sûre**" telle que github.com/someorg/somerepo sans configuration supplémentaire. Dans ce cas, le répertoire GitHub spécifié sera chargé à partir de la branche **master** via un checkout anonyme. (Voir <https://go.cloudbees.com/docs/cloudbees-documentation/cje-user-guide/index.html#github-branch-source> pour plus de détails).

Jenkins

Jenkins > All

New Item

People

Build History

Manage Jenkins

Support

Credentials

Pooled Virtual Machines

Build Queue

No builds in the queue.

Build Executor Status

1 idle

2 idle

Wasted Minutes

0 ms were wasted because you didn't have enough executors.

Item name

TestGitHubOrg

☐ Freestyle project

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and thi

☐ Maven project

Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

☐ Workflow

Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipelines and/or organizing

☐ Auxiliary Template

Auxiliary templates are used to create nested structures embedded within other templates as attribute values. For exan

"tracks" that contain a list of "Track"s.

☐ Backup

Performs back up of Jenkins

☐ Bitbucket Team

Scans a Bitbucket team for all repositories matching some defined markers.

☐ Builder Template

Builder template lets you define a custom builder by defining a number of attributes and describing how it translates to t

the definition of the translation without redoing all the use of the template.

☐ External Job

This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is

[details](#)

☐ Folder

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, i

folders.

☐ Folder Template

A folder template is useful to model a domain-specific concept that contains other "stuff" in it.

☒ GitHub Organization

Scans a GitHub organization (or user account) for all repositories matching some defined markers.

Name: TestGitHubOrg

Display Name:

Description:

[Safe HTML] [Prévisualisation](#)

Projects

GitHub Organization: Credentials: CiBuildServer1Opalnt/***** (GitHub access) [Ajouter](#)

Owner: TestGitHubOrg

Behaviours: Discover branches

Strategy: Exclude branches that are also filed as PRs

Utilisation des librairies

Les shared libraries marquées "Charger implicitement" (**Load implicitly**) permettent aux Pipelines d'utiliser immédiatement les classes ou variables globales qui y sont définies.

Global 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: my-shared-library

Default version: master

Load implicitly: ☒

Allow default version to be overridden: ☒

Retrieval method: Modern SCM

Pour accéder aux autres shared libraries, le **JenkinsFile** doit utiliser l'annotation **@Library** en spécifiant le nom de la librairie :

```
@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']) _
```

L'annotation peut être placée partout où une annotation est autorisée (par Groovy) dans le script de Pipeline. Lorsqu'on fait référence aux classes de librairies (celles dans le répertoire **src**), l'annotation se place par convention sur une instruction **import**.

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

Remarque

Pour les shared libraries qui définissent uniquement des variables globales (**vars**/), ou un **JenkinsFile** qui n'a besoin que d'une variable globale, le patron d'annotation **@Library('my-shared-library') _** aide à maintenir le code concis. En effet, au lieu d'annoter l'instruction **import inutile ici**, le symbole **_** est annoté.

Il n'est pas recommandé d'**importer** une variable/fonction globale, vu que ceci va obliger le compilateur à interpréter des champs et méthodes comme étant **statiques** même s'ils sont supposés n'être que des instances. Le compilateur Groovy, dans ce cas, produirait des messages d'erreur dues à des confusions.

Les librairies sont traitées et chargées au moment de la compilation du script, avant qu'il ne commence à s'exécuter. Ceci permet au compilateur Groovy de :

- ✓ comprendre la signification des symboles utilisés lors du contrôle statique des types,
- ✓ et d'autoriser leur utilisation lors des déclarations de types dans le script.

Exemple:

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

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

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

En revanche, les variables globales sont traitées lors de l'exécution.

Chargement dynamique des Librairies

Avec la version 2.7 du Pipeline, via le plugin Shared Groovy Libraries, il y a une nouvelle option pour charger les librairies (non-implicites) dans un script : un **step library** qui charge les librairies de façon dynamique, à tout moment durant le build.

Si on s'intéresse uniquement à l'utilisation des variables/fonctions globales (du dossier **vars/**), la syntaxe est assez simple :

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

Par la suite, toutes les variables globales de cette librairie seront accessibles au script.

Il est également possible d'utiliser les classes du dossier **src/**, mais c'est plus complexe. En effet, tandis que l'annotation **@Library** prépare le "chemin d'accès à la classe (classpath)" du script avant compilation, au moment où un **step library** est rencontrée, le script a déjà été compilé. Par conséquent, on ne peut pas importer (faire d'**import**) ou encore faire référence de façon statique aux types dans la librairie.

Cependant, on peut utiliser les classes de librairies dynamiquement (sans contrôle de type), en y accédant par leur nom complet à partir de la valeur de retour du **step library**. Les méthodes **static** peuvent être invoquées en utilisant une syntaxe similaire à Java :

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

Ainsi pour l'annotation **@Library** précédente, *si la méthode prepare était statique*, l'écriture avec le **step library** équivaldrait à : **library('somelib').com.mycorp.pipeline.Helper.prepare()**

On peut également accéder à des champs **static**, et appeler des constructeurs comme si il s'agissait de méthodes **static** nommées **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))
```

Ci-dessus, le champ statique est **Constants.SOME_TEXT** qui est un champ appartenant à la classe **Helper**.

Versions de Librairies

La "version par défaut" d'une Shared Library est utilisée lorsque :

- ✓ l'option "Charger implicitement (**Load implicitly**)" est cochée,
- ✓ ou qu'un Pipeline référence cette librairie uniquement par son nom, par exemple **@Library('my-shared-library')** _.

Si une "version par défaut" n'est pas définie, le Pipeline doit spécifier une version, par exemple :

@Library('my-shared-library@master') _.

Si l'option "Autoriser la version par défaut à être modifiée (**Allow default version to be overridden**)" est activée lors de la configuration d'une Shared Library, une annotation **@Library** pourra également modifier la version par défaut définie pour cette librairie. Cette option permet aussi à une librairie avec l'option "Charger implicitement" d'être chargée à partir d'une version différente si nécessaire. Lorsqu'on utilise le **step library**, on peut aussi spécifier une version en utilisant :

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

Vu qu'il s'agit d'un **step** normal, cette version pourrait être générique plutôt qu'une constante comme c'est le cas dans les annotations; **exemple** :

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

Cette instruction chargerait une librairie en utilisant la même branche **SCM** que la multi-branche **JenkinsFile**. Comme autre exemple, on pourrait choisir une librairie via un paramètre (choix paramétrable):

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

Noter que le **step library** peut ne pas être utilisée pour modifier la version d'une librairie implicitement chargée. En effet, au moment

où le script démarre, cette librairie est déjà chargée; et une librairie ayant un nom donné ne peut être chargée deux fois.

Méthode d'accès(Retrieval Method)

La meilleure façon de spécifier un SCM est d'utiliser un plugin (Jenkins) pour SCM qui a été spécifiquement mis à jour pour supporter le contrôle (checkout) de versions arbitraires (option **Modern SCM**). Actuellement, les dernières versions des plugins (Jenkins) Git et Subversion supportent ce mode.

The screenshot shows the Jenkins Shared Library configuration page. The 'Name' field is 'my-shared-library'. The 'Retrieval method' section has 'Modern SCM' selected and highlighted with a blue box. The 'Source Code Management' section has 'Git' selected. The 'Project Repository' field contains 'git://git.example.com/pipeline-library.git'. The 'Repository browser' is set to '(Auto)'.

SCM hérité(Legacy SCM)

Les plugins SCM qui n'ont pas encore été mis à jour pour supporter les nouvelles caractéristiques requises par les Shared Libraries, peuvent toujours être utilisés via l'option **Legacy SCM**. Pour cela, il faut inclure la variable `${library.yourlibrarynamehere.version}` partout où une **branche/tag/réf** doit être configuré pour ce plugin SCM. En effet, ceci permet au plugin SCM de remplacer cette variable par la version appropriée de la librairie, lors du contrôle (checkout) de code source.

The screenshot shows the Jenkins Shared Library configuration page with 'Legacy SCM' selected and highlighted with a blue box. In the 'Source Code Management' section, 'Subversion' is selected. The 'Repository URL' field contains 'svn://svn.example.com/pipeline-library/branches/\${library.my-shared-li}' and is highlighted with a red box. Below the URL, there are two yellow warning messages: 'This repository URL is parameterized, syntax validation skipped' and 'The repository URL is parameterized, connection check skipped'.

Accès dynamique

Lorsqu'on spécifie uniquement le nom d'une librairie (avec éventuellement un @ suivie de la version) dans le **step library**, Jenkins va rechercher une librairie préconfigurée portant ce nom. (Dans le cas d'une librairie automatique `github.com/owner/repo`, comme celle que nous avons présentée tantôt, Jenkins chargera cette librairie via cet "url").

Par ailleurs, on peut également spécifier la méthode d'accès (**SCM**) dynamiquement, auquel cas il n'est pas nécessaire que la librairie ait été prédéfinie dans Jenkins. Voici un exemple :

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

Il est préférable de se référer au **Pipeline Syntax** pour la syntaxe précise à utiliser pour un **SCM** donné. Noter que la version de librairie doit être spécifiée dans ces cas.

Écrire(Créer) des librairies

A la base, tout code Groovy valide peut être utilisé. Différentes structures de données, méthodes utilitaires, etc., telles que :

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

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

Accéder aux step

Les classes de librairies ne peuvent pas directement appeler des **step** telles que **sh** ou **git**. Ils peuvent cependant implémenter des méthodes, en dehors de la portée de la classe englobante, et qui à leur tour invoquent des **step** de Pipeline, par exemple avec le **step** git dans la classe Zot:

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

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

Par la suite, on peut appeler cette méthode dans un script de Pipeline :

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

Cette approche a des limites; par exemple, elle empêche la déclaration de superclasse.

Comme alternative, un ensemble de **step** peut être passé explicitement à une classe de librairie, dans un constructeur ou une simple méthode.

```
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}"
    }
}
```

Lorsqu'on enregistre des états sur des classes, comme ci-dessus, la classe doit implémenter l'interface **Serializable**. Ceci permet à un Pipeline utilisant cette classe, comme démontré dans l'exemple ci-dessous, puisse être proprement suspendu(suspended) et réactivé(resumed) dans Jenkins.

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

Par analogie, si la librairie a besoin d'accéder à des variables globales, telles que **env**, celles-ci devraient être explicitement passées dans les classes de librairies, ou méthodes. Au lieu de passer plusieurs variables à une librairie depuis le script d'un pipe line, on peut passer un script dans une méthode statique, qui sera ensuite invoquée depuis un Pipeline comme le montre l'exemple ci-dessous:

```
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}"
    }
}
```

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

Définir des variables globales

En interne, les scripts dans le dossier **vars** sont instanciés à la demande comme des singletons. Ceci permet à plusieurs méthodes ou propriétés d'être définies dans un unique fichier **.groovy** et d'interagir les unes avec les autres; exemple :

```
// vars/acme.groovy
def setName(value) {
    name = value
}
def getName() {
    name
}
def caution(message) {
    echo "Hello, ${name}! CAUTION: ${message}"
}
```

Dans l'exemple ci-dessus, **name** ne fait pas référence à un champ (même si on écrit **this.name** !), mais à une entrée créée à la demande dans un **Script.binding**. Pour préciser les données que l'on veut enregistrer et leur type, on peut faire à la place une déclaration explicite de classe (le nom de la classe devrait correspondre au nom de fichier et les **step** de Pipeline peuvent être invoqués uniquement si **steps** ou **this** sont passés à la classe ou méthode):

```
// vars/acme.groovy
class acme implements Serializable {
    private String name
    def setName(value) {
        name = value
    }
    def getName() {
        name
    }
    def caution(message) {
        echo "Hello, ${name}! CAUTION: ${message}"
    }
}
```

Le pipeline peut alors invoquer ces méthodes qui seront définies sur l'objet (instance) **acme** :

```
acme.name = 'Alice'
echo acme.name /* prints: 'Alice' */
acme.caution 'The queen is angry!' /* prints: 'Hello, Alice. CAUTION: The queen is angry!' */
```

Remarque

Une variable définie dans une Shared Library n'apparaîtra dans la Référence des Variables Globales (sous la **Pipeline Syntax**) que lorsque Jenkins aura chargé et utilisé cette librairie lors d'une exécution réussie (successful run) de Pipeline.

Définir des step

Les Shared Libraries peuvent également définir des variables globales qui se comportent de façon similaire aux **step** natifs, telles que **sh** ou **git**. Les variables globales définies dans les Shared Libraries doivent avoir un nom minuscule ou "**camelCased**" afin d'être correctement chargés par le Pipeline.

Par exemple, pour définir la variable globale **sayHello**, le fichier **vars/sayHello.groovy** doit être créé et implémenter une méthode **call**. La méthode **call** permet à la variable globale d'être invoqué tel un **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}."
}
```

Le Pipeline serait alors capable de référencer et invoquer cette variable comme un **step** natif:

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

Lorsqu'elle est appelée sous forme de "bloc", la méthode **call** reçoit comme argument une [Closure](#) (instance de **Closure**). Le type devrait être défini explicitement afin de préciser le but du **step**, par exemple :

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

Le Pipeline peut alors utiliser cette variable comme tout **step** natif pouvant être appelé sous forme de "**bloc**" :

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

Définir un DSL plus structuré

Lorsqu'on a plusieurs Pipelines qui sont assez similaires, on peut utiliser le mécanisme de variable globale qui fournit un outil pratique permettant de capturer la similarité et de concevoir une **DSL** de niveau supérieur . Par exemple, vu que tous les plugins Jenkins sont compilés et testés de la même façon, nous pourrions créer un **step** nommé *buildPlugin* :

```
// vars/buildPlugin.groovy
def call(body) {
    // evaluate the body block, and collect configuration into the object
    def config = [:]
    body.resolveStrategy = Closure.DELEGATE_FIRST
    body.delegate = config
    body()

    // now build, based on the configuration provided
    node {
        git url: "https://github.com/jenkinsci/${config.name}-plugin.git"
        sh "mvn install"
        mail to: "...", subject: "${config.name} plugin build", body: "..."
    }
}
```

En supposant que le script a soit été chargé comme Shared Library globale ou Shared Library pour répertoire, le JenkinsFile résultant sera beaucoup plus simple :

```
Jenkinsfile (Scripted Pipeline)
buildPlugin {
    name = 'git'
}
```

Utiliser des librairies tiers

Il est possible d'utiliser des librairies tiers Java, que l'on retrouve généralement dans [Maven Central](#), à partir de code de librairie "**sûre**". Pour cela, on utilise l'annotation **@Grab**. Se référer à la [Grape documentation](#) pour les détails, exemple :

```
@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"
    }
    // ...
}
```

Les librairies tiers sont par défaut conservées dans le cache `~/groovy/grapes/` sur le Jenkins master.

Charger des ressources

Les librairies externes peuvent charger des fichiers auxiliaires d'un répertoire *ressources/* en utilisant le **step** *libraryResource*. L'argument du **step** est un nom de chemin relatif (**pathname**), semblable au chargement de ressources en Java :

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

Le fichier est chargé tel une chaîne de caractères (via son **path**), ce qui est adéquat pour le passer à certains API ou l'enregistrer dans un workspace en utilisant *writeFile*. Il est conseillé d'utiliser une structure de package unique afin de ne pas rentrer accidentellement en conflit avec d'autres librairies.

Pré-tester les changements de librairies

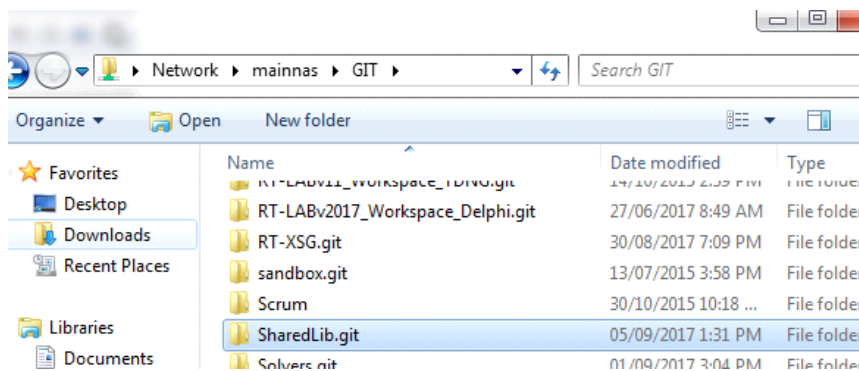
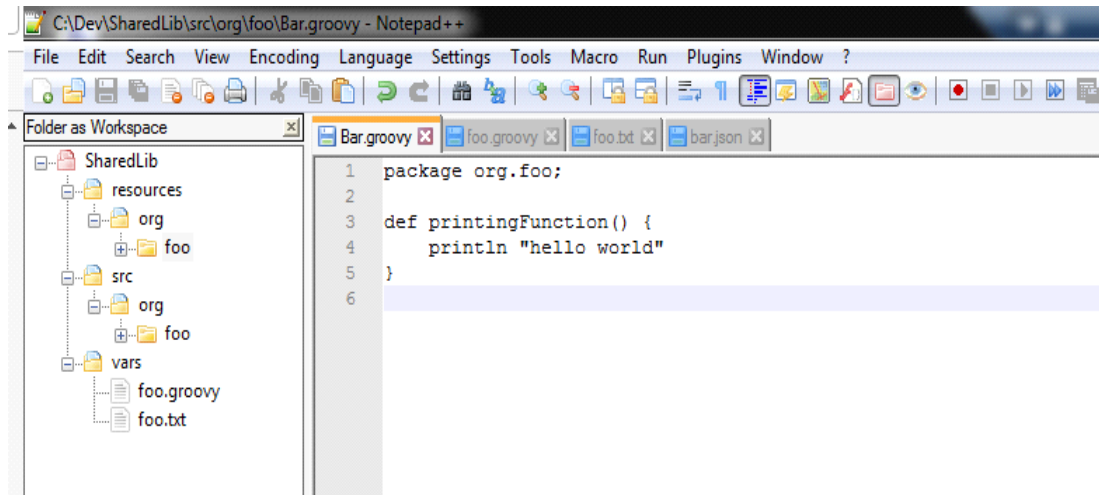
Si vous remarquez une erreur dans un build en utilisant une librairie "**pas sûre**", il faut juste cliquer sur le lien **Replay** pour essayer d'éditer un ou plusieurs de ses fichiers sources, et voir si le build résultant se comporte comme prévu. Une fois satisfait par le résultat, il suffit de suivre le lien **diff** de la page du statut de build, appliquer le diff au répertoire de la librairie et faire un commit.

(Même si la version requise pour la librairie était une **branche**, au lieu d'une version fixe tel qu'un **tag**, les builds rejoués vont utiliser l'exacte même révision que le build original : les sources de librairie ne seront pas contrôlés (checkout) de nouveau.)

Replay n'est pas actuellement supportée par les librairies "**sûres**". La modification des fichiers de ressource n'est actuellement pas supportée durant Replay.

Exemple de Shared library créé et utilisé dans un Pipeline de notre serveur Jenkins

Structure et contenu du répertoire de la Shared Library 'SharedLib'



Configuration de la librairie sur Jenkins

Aller à :

Administrer Jenkins >> Configurer le Système >> Global Pipeline Libraries >> Ajouter

Jenkins > configuration

Global 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
Default version
Load implicitly
Allow default version to be overridden
Retrieval method
Source Code Management
Repositories
Branches to build
Navigateur de la base de code

SharedLib
0.1
☐
☒
Retrieval method: Modern SCM, Legacy SCM (selected)
Source Code Management: Base ClearCase, CVS, CVS Projectset, Gerrit Repo, Git (selected)
Repository URL: file:///mainnas/GIT/SharedLib.git
Credentials: - aucun -
Add Repository
Branch Specifier (blank for 'any'): */master
Add Branch, Delete Branch
(Auto)

Enregistrer Appliquer

Créer un Pipeline utilisant la shared library référencée : SharedLib

jenkins:58080/job/sharedLibTest/configure

Jenkins

sharedLibTest

General

Build Triggers

Advanced Project Options

Pipeline

☐ [FSTrigger] - Monitor files
☐ [FSTrigger] - Monitor folder
☐ Désactiver le projet
☐ Période d'attente
☐ Déclencher les builds à distance (Par exemple, à partir de scripts)

Advanced Project Options

Pipeline

Definition

Pipeline script
















Script

```

1 node('JENKINS-SLAVE'){
2     @Library('SharedLib')
3     def bar = new org.foo.Bar()
4     bar.printingFunction()
5 }

```

Résultat :

-  [Back to Project](#)
-  [Status](#)
-  [Changes](#)
-  **Console Output**
 -  [View as plain text](#)
-  [Edit Build Information](#)
-  [Supprimer le build](#)
-  [Open Blue Ocean](#)
-  [Git Build Data](#)
-  [No Tags](#)
-  [Rebuild](#)
-  [Replay](#)
-  [Pipeline Steps](#)
-  [Embeddable Build Status](#)
-  [Previous Build](#)

Sortie de la console

```

Started by user Ndeye Anna Ndiaye
Loading library SharedLib@0.1
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url file:///mainnas/GIT/SharedLib.git # timeout=10
Fetching upstream changes from file:///mainnas/GIT/SharedLib.git
> git --version # timeout=10
> git fetch --tags --progress file:///mainnas/GIT/SharedLib.git +refs/heads/*:refs/remotes/origin/*
> git rev-parse "refs/remotes/origin/master^{commit}" # timeout=10
> git rev-parse "refs/remotes/origin/origin/master^{commit}" # timeout=10
Checking out Revision f622eabb0630e636ad93190d5d38014d7c5be1c9 (refs/remotes/origin/master)
Commit message: "Removing change in Bar.groovy"
> git config core.sparsecheckout # timeout=10
> git checkout -f f622eabb0630e636ad93190d5d38014d7c5be1c9
> git rev-list f622eabb0630e636ad93190d5d38014d7c5be1c9 # timeout=10
[Pipeline] node
Running on JENKINS-SLAVE in c:\jenkins\workspace\sharedLibTest
[Pipeline] {
[Pipeline] echo
hello world
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```