# Cahier de TPs

Jenkins

**Pré-requis :**
Poste développeur avec accès réseau Internet libre
Linux (Recommandé) ou Windows 10
Pré-installation de :

- Git
- JDK11
- Docker
- Editeur fichier .yaml (Exemple VisualStudio Code)

## TP1 : Installation, Configuration, Premier job

### *Objectifs*

- Installation de Jenkins en service, configuration du service
- Configurer le système
- Configurer les outils de base
- Créer un premier job freestyle

### 1. 1 Installation

#### *1.1.a : Installation en service*

En fonction de votre OS, exécuter la procédure d'installation en service adéquat.

Avec un navigateur, accéder à localhost:8080 et continuer la procédure **sans** installer de plugin

Éditer le fichier */etc/default/jenkins* :

- modifier les paramètres de la JVM

Effectuer un redémarrage du service :

```
sudo service jenkins restart
```

Visualisez l'arborescence de JENKINS_HOME

#### *1.1.b : Script de démarrage*

Télécharger la distribution générique de Jenkins et écrire un script de démarrage qui positionne le JENKINS_HOME, la mémoire de la JVM et le port d'écoute de jenkins

Visualiser l'arborescence de JENKINS_HOME

## 1.2 Configuration générale et outils

### *1.2.1 Configuration serveur de mail*

Si nécessaire installer le plugin ***mailer***

Ensuite aller dans la configuration système « *Administrer Jenkins* → *Configurer le système*» et renseigner les paramètres du serveur smtp

login : stageojen@plbformation.com

password : stageojen

smtp : smtp.plbformation.com

(port smtp 587)

Tester l'envoi de mail

### *1.2.2 Configuration des outils (JDK,  Maven, Git)*

1. Cliquer sur « *Administrer Jenkins* → *Configuration Globale des outils* »
2. Dans la section JDK, indiquer soit un JAVA_HOME pré-installé soit un installateur automatique
3. Dans la section Maven, utiliser l'installation automatique
4. Dans la section Git, indiquer votre installation de Git
   (Si la section Gitt n'est pas présente installer le plugin ***Git Client)***
5. Enregistrer vos modifications

## 1.3 : Création de job

### *Mise en place dépôt Git*

Vérifier l'installation de git sur votre machine
Installer le plugin Jenkins « ***git*** »

Reprendre les source du projet et les décompresser.
Initialiser le dépôt et committer les fichiers sources :
*git init*
*git add .*
*git commit -m 'Initial commit'*

### *Création job et test*

1. De retour sur la page d'accueil, Activer le lien « Créer un nouveau job »
2. Donner un nom et choisir job freestyle
3. Configurer le SCM afin qu'il point vers un dépôt git local
4. Indiquer que le build est déclenché à chaque changement dans GIT et que le repository est interrogé toutes les 5 mns. (*/5 * * * *)
5. Ajouter une étape de build affichant toutes les variables d'environnement disponibles
6. Ajouter ensuite une étape de build qui invoque la cible Maven « ***clean package*** », indiquer le

chemin vers le *pom.xml*

7. Ajouter une étape « post-build » permettant d'afficher les résultats des tests : « Publier les rapports *JUnit* »
   Nécessite le plugin ***junit***
8. Ajouter une autre étape « post-build » pour archiver tous les jars produits
9. Lancer le build manuellement et observer la page d'accueil du projet

## *Déclenchement de job*

1. Modifier un fichier du projet provoquant une erreur dans les tests, committer le changement dans le repository
   Voir fichier *library/src/test/java/hello/service/MyServiceTest.java*
2. Attendre le déclenchement du job
3. Observer la page d'accueil qui doit afficher un graphique de tendance sur l'exécution des tests.
4. Ensuite restaurer votre modification

## *Plugin Maven (Optionnel)*

1. Installer le plugin ***Maven Integration***
2. Créer un job Maven nommé ***1_package*** effectuant les cibles ***clean package***
3. Exécuter manuellement le build et observer les résultats
4. Observer l'archivage automatique et les artefacts archivés, la configuration multi-modules

# TP2 : Architecture maître/esclave

Dans ce TP, nous mettons en place une architecture maître/esclaves permettant de distribuer la charge

.

## *Objectifs*

- Mettre en place 1 nœud esclave
- Donner des étiquettes aux nœuds

## Mise en place Agent SSH

Installation le plugin « ***SSH Build Agents*** »

Installation de OpenSSH Servers, (**openssh-server** ) vérifier que le serveur sshd démarre

Vérifier l'accès avec *ssh localhost*

Dans l'interface d'administration Jenkins ; créer un nœud démarré en ssh, lui affecter 4 exécuteurs et des labels

Arrangez vous pour que ce nœud exécute un job. Visualisez ensuite le répertoire de travail du nœud esclave.

# TP3 : Projet multi-jobs

## *Objectifs*

- Utilisation des paramètres
- Faire une job multi-configuration
- Exécuter une pipeline Legacy

## 3.1 Job paramétré

1. Installer le plugin ***Git Parameter Plugin***
2. Définir un paramètre pour le job ***1_package*** dont la liste des valeurs est les différents hash de commit du repository Git
3. Démarrer le job manuellement

## 3.2 Job multi-configuration

Nécessite le plugin ***matrix-project***

Définir un deuxième JDK (il peut pointer sur le même JAVA_HOME) et définir un premier axe avec le JDK

Créer un second axe BD avec comme valeurs possibles :

- H2
- MySQL
- Postgres

Créer un nouveau projet multi-configuration ***1_package_multi*** avec les 2 axes précédents à partir du job ***1_package***

## 3.3 Intégration Sonar et Chaînage de jobs

### *3.3.1 Intégration Sonar*

Démarrer Sonarqube via docker :

```
docker run -d --name sonarqube -p 9000:9000 sonarqube
```

Accéder à localhost:9000 , se connecter avec admin/admin et définir un nouveau mot de passe

#### 3.3.1.1 Option1 : Plugin SonarQube Scanner

Installer le plugins *SonarQube Scanner*

Configurer Sonar dans :

- *Configuration Système :* Indiquer l'URL du serveur Sonar
- *Configuration des outils :* Définir un scanner s'installant automatiquement

Créer un nouveau job ***2_qualite*** sur le même modèle que ***2_integration***, dans la partie build appeler le Scanner Sonar en indiquant le fichier de configuration Sonar fourni (*sonar.properties*)

#### 3.3.1.2 Option2 : Plugin Maven

Sans installer le plugin *SonarScanner*, indiquer les propriétés de connexion au serveur Sonar dans le pom.xml avec les 2 propriétés suivantes :

***sonar.login*** et ***sonar.password***

Tester en exécutant la commande suivante à la racine du projet :

***mvn clean verify sonar:sonar***

Créer un job Maven nommé ***2_qualite*** invoquant la cible Maven ***clean integration-test***

### 3.3.2 Chaînage de jobs

Après avoir créé le job ***2_qualite*** avec 1 des 2 options :

Chaîner ce build avec le build précédent, faire en sorte que les 2 builds soient sur le même dépôt Git

***Les 2 jobs travaillent-ils sur le même commit ?***

## 3.4 Passage de paramètres

Objectif : Passage de la clé sha1 de Git entre ***1_package*** et le job aval

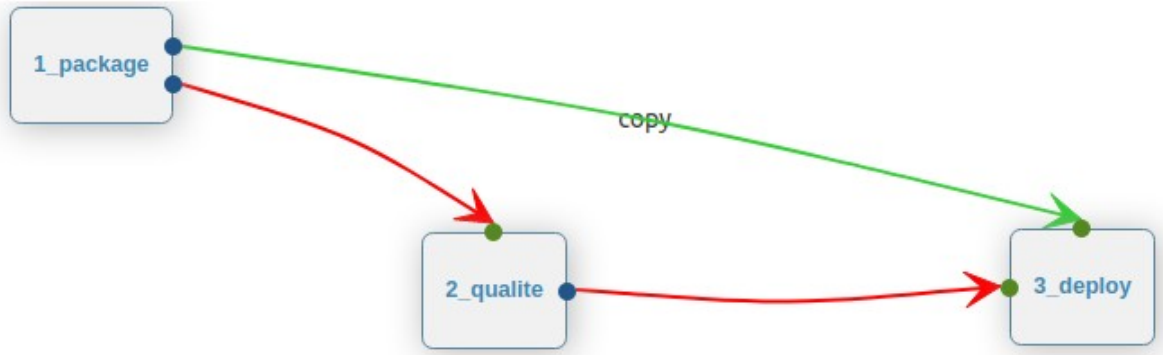
- Tester le job manuellement, afficher les variables d'environnement
- Installer le plugin ***Parameterized Trigger***
- Modifier le lancement du builds ***2_integration*** afin que le paramètre du premier job soit passé
- Utiliser ce paramètre pour effectuer un checkout du même commit

## 3.6 Copy Artifact

Installer les plugins ***Dependency Graph Viewer, Copy Artifact***

Créer un nouveau job ***3_deploy_integration*** qui se déclenchera à la suite de ***2_qualite***

Ce job se contentera d'exécuter un shell effectuant une copie du war créé lors de ***1_package*** dans un répertoire de votre choix



# TP4 : Premières pipelines

## *Objectifs*

- Distinguer les 2 syntaxes
- Se familiariser avec les 2 façons de mettre au point des pipelines

Installer le plugin ***pipeline*** et ***BlueOcean***

A partir des exemples fournis par Jenkins, créer une Pipeline « ***First_Pipeline*** » qui définit 3 phases :

- Préparation :
  - Checkout du scm
  - Définition de l'outil Maven défini dans Jenkins
- Build
  - Exécution de la cible Maven ***clean package***
- Post-build
  - Publication des résultats de tests
  - Archivage des artefacts

# TP5 : Multi-branche, Syntaxe déclarative

.

## *Objectifs*

- Syntaxe déclarative
- Multi-branche pipeline

## 5.1 Création d'un job Multibranch

Créer la branche ***dev*** dans votre repository GIT

Récupérer le fichier *Jenkinsfile* fourni et le committer dans la branche dev

Créer un job ***multi-branch pipeline*** pointant sur le dépôt dans Jenkins

Observer l'exécution automatique des *JenkinsFile* dans la branche dev

## 5.2 Implémentation de la pipeline

Dans la première phase effectuer la commande maven

*mvn -Dmaven.test.failure.ignore=true clean package*

Puis dans les étapes de post-build :

- Publier toujours les résultats des tests
- En cas de succès : Archiver les artefacts
- En cas d'erreur : Envoyer un mail

Dans la seconde phase, effectuer en parallèle :

- Une tâche exécutant les tests d'intégration
- Une tâche exécutant une analyse SonarQube

Dans la dernière phase, Poser une question à l'utilisateur et récupérer un paramètre et l'utiliser pour déployer

# TP6 : Librairies partagées

## *6.1 Tutoriel*

*Référence :*
*https://dev.to/jalogut/centralise-jenkins-pipelines-configuration-using-shared-libraries*

## *Dépôt pour la librarie partagées*

Créer un nouveau dépôt Git

Y créer un répertoire vars et y déposer le fichier `standardPipeline.groovy` fourni.

Committer

## *Configuration Jenkins*

Dans Jenkins, Configurer une Global Pipeline Library pointant sur le précédent dépôt.

## *Utilisation*

Créer ensuite un JenkinsFile utilisant la libairie définie.

## *6.2 Ajout de fonctions Groovy*

En vous inspirant du tutoriel précédent, créer 2 nouvelles fonctions dans la librairie partagée :

- ***unitTest.groovy*** : Rassemblera les steps nécessaires à l'exécution de la cible Maven compile test et à la publication des résultats de tests. Il s'appuiera sur l'outil Maven défini sur le serveur Jenkins
  Cette fonction ne prendra pas de paramètres
- ***sonar.groovy*** : Rassemblera les steps nécessaires à l'exécution du scanner Sonar en utilisant les configurations relatives au serveur Sonar et à l'outil scanner défini dans Jenkins
  Cette fonction prendra un paramètre : le fichier de configuration Sonar

Tester l'utilisation de ces fonctions dans la pipeline déclarative.

# TP7 : Utilisation de docker

## *Objectifs*

- Utiliser une image docker pour effectuer la partie Maven de votre build
- Utiliser les volumes et le cache pour ne pas repartir à zéro
- Construire une image Docker lors d'une étape de build
- Publier l'image vers un registre Docker

Vérifier l'installation des plugins ***Docker*** et ***Docker pipeline***

## 7.1 Utilisation d'un agent docker

Utiliser un agent docker pour le premier stage de la pipeline

## 7.2 Construction et *push* d'une image

Reprendre le fichier ***Dockerfile*** et le comprendre, le mettre dans le répertoire *application/src/main/docker*

Reprendre également le fichier *pom.xml* du module application

Modifier le fichier ***Jenkinsfile*** afin de construire une image docker et la publier vers un registre distant

## 7.3 Image Jenkins

Pousser le dépôt local vers un dépôt public

Mettre au point un fichier *Dockerfile* qui installe les plugins pipeline et BlueOcean au dessus de l'image *dind:dind*

Exécuter la pipeline précédente dans cette nouvelle installation

# TP9 : Sécurité

.

## *Objectifs*

- Mise en place de la sécurité
- Définition des autorisations par rôle, surcharge au niveau projet
- Autorisation dans l'étape manuelle d'une pipeline

## Sécurité globale via des rôles

Activer la sécurité dans Jenkins
Installer le plugin *Role-based Authorization Strategy*
Définir un rôle admin, pouvant tout faire au niveau global
Définir un rôle ***marketing*** avec un accès en lecture uniquement
Créer un nouvel utilisateur et lui assigner le rôle ***marketing***

Se logger avec le nouvel utilisateur et vérifier votre configuration.

## Sécurité projet

Créer un *Folder Legacy* et y déplacer les projets freestyle des premiers TPs
Définir un nouveau rôle projet avec comme expression régulière Legacy.* et ayant tous les droits sur les jobs (Create, Build, Configure, ...)
Affecter ce rôle à l'utilisateur

Se logger avec le nouvel utilisateur et vérifier qu'il peut démarrer les jobs legacy.

## Pipeline avec approbation manuelle

Dans la pipeline avec approbation manuelle, ajouter des permissions sur l'action de déploiement afin que l'utilisateur et l'admin puisse activer le bouton

# TP10 : Exploitation, Jenkins CLI, API Rest

Dans ce TP, nous abordons les aspects d'exploitation

.

## *Objectifs*

- Découvrir les plugins d'exploitation et de surveillance
- Effectuer des sauvegardes et migration
- Automatiser le backup et restore d'un jobs
- Déclenchement de build via API Rest

## Monitoring

Installer les plugins *Disk Usage* et *Monitoring*

Visualiser les vues associées

## Backup and Restore

Démarrer un second serveur Jenkins (autre port http)

Effectuer un backup de la première instance et la restaurer dans la seconde instance

Comparer avec un déplacement du répertoire JENKINS_HOME

Utiliser le *Thin Backup* Plugin, effectuer des backup de la configuration toutes les 5 min

## Jenkins CLI

Créer un token pour l'utilisateur *admin*.

Télécharcher jenkins-cli.jar via la console d'administration

Utiliser le token pour afficher l'aide :

*java -jar jenkins-cli.jar -s http://localhost:8080 -auth admin:<token> list-jobs > jobs.txt*

Via le cli récupérer la liste des jobs et leur configuration au format XML

## Rest API

Déclencher un build paramétré via une requête htt