Cahier de TP
« Jenkins pipeline»

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

- **Git**
- **JDK17**
- **Docker**
- **Kubernetes** :
  - ***kubectl*** : https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/#install-kubectl-binary-with-curl-on-linux
  - ***kind*** : https://kind.sigs.k8s.io/docs/user/quick-start/#installing-from-release-binaries
  - ***helm*** : https://get.helm.sh/helm-v3.13.1-linux-amd64.tar.gz
- **Ansible** : https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html
- **VisualStudio Code**

Table des matières

Ateliers 1: Mise en place, Installation, Free-style Job

Objectifs

- Prendre en main la distribution de Jenkins
- Avoir un aperçu des configurations administrateur
- Concrétiser l'architecture Maître/Esclave de Jenkins

1.1 Script de démarrage

Télécharger la distribution générique de Jenkins : https://www.jenkins.io/download/

Écrire un script de démarrage qui positionne en variables d'environnement :

- JAVA_HOME
- JENKINS_HOME
- la mémoire de la JVM
- Éventuellement, le port d'écoute de jenkins
- La propriété *hudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT* permettant de travailler avec des dépôts localux

Voici un exemple d'un tel script :

```
#!/bin/sh
export JAVA_HOME=/usr/lib/jvm/java-1.17.0-openjdk-amd64
export JENKINS_BASE=/home/dthibau/Formations/Jenkins/MyWork/jenkins
export JENKINS_HOME=${JENKINS_BASE}/.jenkins

java -Dhudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true -Xmx2048m \
-jar ${JENKINS_BASE}/jenkins.war --httpPort=8082
```

Faire un premier démarrage, se connecter à l'adresse d'écoute et compléter l'installation en :

- Installant les plugins recommandés
- Définissant un login/password administrateur (*admin/admin* par exemple)

Visualiser ensuite l'arborescence de JENKINS_HOME

1.2 Configuration générale et outils

1.2.1 Configuration système : l'exemple du serveur de mail

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

Ensuite aller dans la configuration système :

« Administrer Jenkins → Configurer le système»

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)

Aller dans la configuration globale des outils :
« Administrer Jenkins → Configuration Globale des outils »

- Dans la section JDK, indiquer un JAVA_HOME pré-installé
- Dans la section Maven, utiliser l'installation automatique
- Dans la section Git, indiquer l'installation par défaut

1.3 Mise en place Agent SSH

Installer si nécessaire le plugin ***SSH Build Agents***

Vérifier qu'un serveur sshd est démarré :

```
$ ssh localhost
```

Si ce n'est pas le cas, installer OpenSSH Servers, (*openssh-server*)

Dans l'interface d'administration Jenkins : créer un nœud démarré en *ssh*, lui affecter 4 exécuteurs et les labels suivant :

```
docker jdk17
```

Indiquer « *Non Verifying Verification Strategy* »

Vérifier la bonne connexion entre le maître et l'esclave via la console d'administration

Configurer le maître afin qu'il ne prenne aucun exécuteur

1.4 Test de la config avec job simple :

1.4.1 Mise en place dépôt Git

Reprendre les source du projet et les décompresser dans un répertoire de travail

Initialiser le dépôt et committer les fichiers sources
Dans votre répertoire de travail exécuter :

```
git init
git add .
git commit -m 'Initial commit'
```

1.4.2 Création job et test

De retour sur la page d'accueil, Activer le lien « Créer un nouveau job »

- Donner un nom ***1_freestyle*** et choisir job freestyle
- Configurer le SCM afin qu'il point vers un dépôt git local
- Indiquer le label ***jdk*** par exemple concernant les contraintes sur le noeud
- Indiquer que le build est déclenché à chaque changement dans GIT et que le repository est interrogé toutes les 5 mns. (*/5 * * * *)
- Ajouter une première étape de build affichant toutes les variables d'environnement disponibles

```
env
```

- Ajouter ensuite une étape de build qui invoque la cible Maven ***clean package***
- Ajouter une étape « post-build » permettant d'afficher les résultats des tests : « Publier les rapports JUnit »
  Si l'action n'est pas disponible vous devez ajouter le plugin ***junit***
- Ajouter une action d'archivage des artefacts

Vérifier que le job démarre automatiquement :

- Observer les logs sur le maître
- Visualisez ensuite le répertoire de travail du nœud esclave.

1.4.3 Modification de code

Modifier le fichier source ***library/src/test/java/hello/service/MyServiceTest.java*** afin que le test échoue.

Committer vos modifications et observer le déclenchement de la pipeline.

Observer le statut de la pipeline.

Modifier la configuration du job et l'étape maven en ajoutant l'option :
-Dmaven.test.failure.ignore=true

Relancer manuellement la pipeline.

Observer le statut du build

Ateliers 2 : Pipelines, syntaxe déclarative, script et librairies

Objectifs

- Distinguer et prendre en main les 2 syntaxes
- Mettre au point son environnement de développement de JenkinsFile
- Utiliser les multi-branches
- Visualiser les pipelines avec le plugin *BlueOcean*

2.1 Déclaratif / Script

Installer le plugin ***Blue Ocean***

Créer un job Pipeline « *First_Pipeline* »

Dans le champ d'édition de la pipeline, choisir un exemple avec la syntaxe déclarative

Modifier le script afin qu'il effectue les tâches suivantes :

- Checkout du dépôt des sources de l'atelier précédent
- Définition de l'outil Maven défini dans notre installation de Jenkins
- Exécution de la cible Maven ***clean package***
- Étape de Post-build de Publication des résultats de tests
- Archivage des artefacts

Modifier ensuite l'exemple de la version script et effectuer les mêmes opérations

2.2 Jobs multibranches et implémentation déclarative

2.2.1 Multibranch Job

Pour la mise en place de la pipeline, vous pouvez :

- Utiliser la fonction Replay
- Installer l'extension Jenkins ***Pipeline Linter Connector*** dans Vscode qui permet de valider le JenkinsFile avant de committer

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

```
git checkout -b dev
```

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

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

Observer l'exécution automatique des JenkinsFile dans la branche dev

2.2.2 Implémentation déclarative

Pré-requis : Démarrage d'un serveur SonarQube et intégration Maven

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

- Se connecter à localhost:9000 avec admin/admin
- Changer le mot de passe
- Dans le Menu en haut à droit ***My Account → Security***
- Générer un nouveau jeton,
- Le déclarer dans Jenkins sous forme de crédentiels :
  - Type « Secret Text »

Le fichier *JenkinsFile* déclare 3 phases ou stage.

Dans la première phase « Build and Tests » 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 une analyse des dépendances du projet :

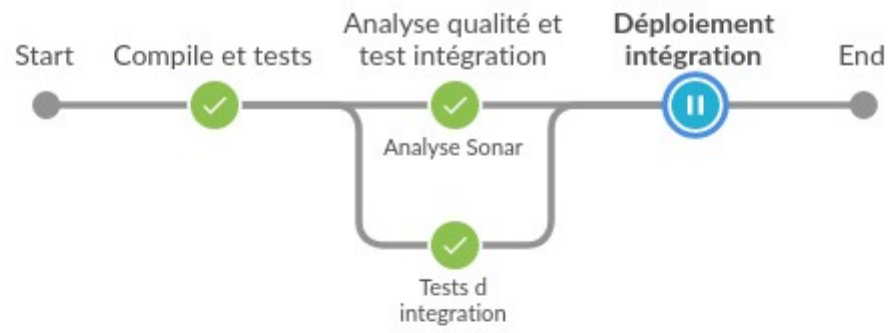
```
mvn -DskipTests verify
```

- Une tâche exécutant une analyse SonarQube

```
mvn -Dsonar.token=${SONAR_TOKEN} clean integration-test sonar:sonar
```

Dans la dernière phase :

- Poser une question à l'utilisateur :
  Dans quel Data Center, voulez-vous déployer l'artefact ?
- Lui proposer 3 choix. Par exemple : Paris, Lille, Lyon
- Récupérer le paramètre saisi et l'utiliser pour déployer l'artefact de la première phase dans un répertoire dynamique
- Finalement faire en sorte que cette phase ne s'exécute que sur la branche principale



2.3 Bloc script

Installer le plugin ***Pipeline Utility Steps***.

2.3.1 Construction boucle et variable globale

Nous voulons maintenant déployer vers plusieurs dataCenters.

La liste des datacenters sera fournie dans un fichier projet au format JSON.

Veillez également à que l'étape en attente d'approbation n'utilise pas d'agent.

Dans la phase de déploiement, effectuer une boucle sur la liste des datacenters et effectuer un déploiement sur chaque

2.3.2 Fonction et appel API

Visualiser la fonction Groovy fournie dans le fichier ***wait4Quality.txt***.

La comprendre, la déclarer dans la pipeline et l'utiliser à bon escient

2.4 Librairies

Référence :

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

2.4.1 Tutoriel

Créer un nouveau dépôt Git nommé ***GlobalLib***

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 librairie définie dans un nouveau projet.

2.4.2 Ajout de fonctions Groovy

En vous inspirant du tutoriel précédent, créer une nouvelle fonctions dans la librairie partagée ***tarGz.groovy*** . La fonction récupérera dans sa configuration :

- Un sous-répertoire du projet
- Une liste de suffixe de fichiers
- Un répertoire de sortie

Elle produira un tar qui contiendra tous les fichiers respectant les extensions

```
// my-shared-library/vars/createTarGz.groovy

def call(Map config) {
    def sourceDir = config.sourceDir
    def extensions = config.extensions
    def outputDir = config.outputDir

    echo "Création d'une distribution tar.gz à partir de $sourceDir avec les
extensions : ${extensions.join(', ')}"

    // Créez une chaîne de filtres pour rechercher plusieurs extensions
    def extensionFilter = extensions.collect { ext -> "--include=\"*.
$ext\"" }.join(' ')

    // Utilisez des commandes shell pour créer le tar.gz
    sh "mkdir -p $outputDir"
```

```
    sh "tar -czvf $outputDir/my_distribution.tar.gz -C $sourceDir
$extensionFilter ."
}
```

Utilisez la fonction dans la pipeline

2.5 Template engine

https://jenkinsci.github.io/templating-engine-plugin/2.5.3/tutorials/jte-the-basics/

Ateliers 3 : Intégration container

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
- Installer Jenkins dans un cluster Kubernetes
- Utiliser le cluster kubernetes pour exécuter les builds

Vérifier l'installation du plugin ***Docker pipeline***

3.1 Utilisation d'un agent docker

Utiliser un agent docker pour le premier stage de la pipeline.

Vous pouvez utiliser l'image ***openjdk:17-alpine***

Bien penser à mettre en place le cache des librairies téléchargées par Maven

3.2 Construction et push d'une image

Pré-requis : Compte Docker Hub

Visualiser le fichier Dockerfile à la racine et le comprendre

Ajouter une phase « *Push to Docker Hub* » avant la phase déploiement qui :

- Construit une image docker avec un tag préfixé par votre compte DockerHub
- Pousser le tag en ayant auparavant stocker un nouveau crédentiel dans Jenkins

3.3 Kubernetes

Pré-requis :
Compte Github ou Gitlab

Référence : https://www.jenkins.io/doc/book/installing/kubernetes/

3.3.1 Préparation du cluster

Créer un cluster kind nommé jenkins :

```
kind create cluster --name jenkins
```

Créer un namespace pour y déployer Jenkins

kubectl create namespace devops-tools

3.3.2 Installation Jenkins dans le cluster

Utiliser les manifeste Kubernetes fournis

Création d'un rôle ***jenkins-admin*** et d'un compte service associé

kubectl apply -f serviceAccount.yaml

Récupérer le nom du nœud :

kubectl get nodes

Editer le fichier volume.yaml pour indiquer le nom du nœud à la ligne 32.

Création d'un volume pour le stockage de Jenkins (Classe de Stockage, Persistent Volume et Persistent Volume Claim):

kubectl apply -f volume.yaml

Déploiement d'une image Jenkins (Image lts, port 8080 et 50000 exposés, Montage de volume)

kubectl apply -f deployment.yaml

Vérifier le statut du déploiement :

kubectl get deployments -n devops-tools

Exposition de Jenkins en tant que Service de type NodePort (Port 8080→ 32000, 50000 → 32001)

kubectl apply -f service.yaml

Pour accéder à jenkins, trouver l'IP du cluster via :

jsonpath="{.items[0].status.addresses[0].address}"

kubectl get nodes -n devops-tools -o jsonpath=$jsonpath

Accéder ensuite à ***http://<node-ip>:32000***

Pour déverrouiller Jenkins, accéder au log du pod jenkins pour voir le mot de passe de déverrouillage

kubectl -n devops-tools get po

kubectl logs -f <jenkins-pod>

Continuer la phase de configuration en installant les plugins suggérés et en créant un admin.
Vous pouvez également installer les plugins que vous voulez, peut-être ***Blue Ocean***

3.3.3 Installation et configuration du plugin Kubernetes

Installer le plugin ***Kubernetes***

Puis ajouter un cloud de type Kubernetes dans l'interface d'administration de Jenkins, lui donner le nom KubLocal par exemple.

Tester la connexion au cluster. Jenkins étant déjà dans le cluster les champs de la première partie du formulaire peuvent être vide

Trouver ensuite l'URL Jenkins, c'est l'URL utilisée par le pod agent pour contacter le contrôleur Jenkins. On peut la voir avec :

kubectl -n devops-tools get service

Renseigner l'URL jenkins : http://<cluster-ip>:8080

Sauvegarder.

Créer ensuite un podTemplate nommé ***jdk17-agent*** qui inclut un container nommé ***openjdk-17*** associé à l'image ***openjdk:17-alpine***

3.3.4 Pipeline avec un agent Kubernetes

Pousser le projet sur une URL publique (*github.com* ou *gitlab.com*)

Créer un Multi-branch pipeline pointant sur le dépôt public

Créer une branche ***kubernetes***

Utiliser le template installé par administrateur pour exécuter la première phase de la pipeline.

Pour la seconde phase, , récupérer le fichier ***kubernetesPod.yaml*** fourni et le visualiser.

Utiliser ce fichier yaml dans la pipeline Jenkins du projet multi-modules

Pour une meilleure compréhension, vous pouvez exécuter dans un terminal :

kubectl get pods -n jenkins -w

Ateliers 4 : Plugins CI/CD

4.1 Tests et analyse statique

Récupérer les sources fournis

Configurer le plugin ***JUnit*** afin qu'il tolère un certain nombre de tests échoués

Installer le plugin ***SonarScanner***

Déclarer le serveur SonarQube dans ***Administration Jenkins → Système***

Définir également l'outil SonarScanner ***Administration Jenkins → Configuration des outils***

Modifier la pipeline afin que l'analyse qualité s'effectue par le plugin, conditionner l'état du build à la porte qualité SonarQube

Installer le plugin ***OWASP Dependency-Check*** et le configurer pour publier les rapports

4.2 Ansible

Installer le plugin ***Ansible***.

Exécuter des commandes AdHoc

Exécuter un playbook

4.3 Zulip

Installation Zulip Server

Récupérer le fichier docker-compose fourni

```
docker compose up -d
```

Puis créer une organisation

```
docker-compose exec -u zulip zulip \
    "/home/zulip/deployments/current/manage.py generate_realm_creation_link"
```

Accèder au serveur zulip et créer une organisation et un compte.

Créer ensuite un channel ***jenkins-notification***

Créer un bot ***jenkins-bot*** et noter son API key

Abonner le bot au channel jenkins-notification

Certificat self-signed

Récupérer le certificat self-signed du serveur zulip :

```
openssl s_client -showcerts -connect localhost.localdomain:443 < /dev/null | openssl x509 -outform PEM > zulip_cert.pem
```

Créer un truststore pour jenkins :

```
keytool -importcert -file zulip_cert.pem -keystore jenkins_truststore.jks -alias zulip-cert
```

Modifiier le script de démarrage de Jenkins pour ajouter les 2 options suivantes :

```
-Djavax.net.ssl.trustStore=/path/to/jenkins_truststore.jks
-Djavax.net.ssl.trustStorePassword=your_password
```

Redémarrer Jenkins

Installer le plugin Zulip dans Jenkins et dans la configuration system indiquer :

Zulip Server : https://localhost.localdomain

Zulip email : l'email de jenkins-bot

API Key : Son API

Dans le free-style project ajouter :

- Un étape envoyant une notification
- Une étape de post-build envoyant une notification zulip

Éventuellement, faire échouer le build puis le refaire réussir