

# Ateliers

## Git, Maven, Nexus, Sonar, Jenkins

### Pré-requis :

Poste développeur avec accès réseau Internet libre  
Linux (Recommandé) ou Windows 10

### Pré-installation de :

- Git
- JDK17+
- Docker
- VSCode
- 

### Images docker utilisées :

- gitlab/gitlab-ce:latest
- sonatype/nexus3
- sonarqube:latest

## Table des matières

Atelier 1 : Git (Optionnel).....	3
1.1. Installation et configuration.....	3
1.2. Initialisation d'un dépôt, enregistrement de modifications.....	3
1.3. Fusion et rebase de branches locales.....	3
1.4 Branches distantes.....	4
Ateliers 2 : Maven.....	5
2.1 Installation de Maven et Projet simple.....	5
2.1.1 Installation maven.....	5
2.1.2 Personnalisation settings.xml.....	5
2.1.3 Création d'un projet simple.....	5
2.2 : Dépendances.....	6
2.3 : Multi-modules Configuration cycle de vie, profiles.....	6
2.3.1 Projet multi-modules.....	6
2.3.2 Adaptation du build.....	7
2.2.3 Création d'un profil.....	8
Ateliers 3 : Release, Nexus.....	9
3.1 Pré-requis : Installation Nexus.....	9
3.2 Configuration projet, utilisation Nexus en miroir.....	9
3.3 Déploiement de snapshots/releases.....	9
3.4 Plugin Release de Maven.....	9
Ateliers 4 : SonarQube.....	11
4.1 Installation.....	11
4.2 Scanner Maven.....	11
4.3 Exclusions.....	12

4.4 Création d'un profil qualité.....	12
4.5 Création d'une porte qualité.....	12
Ateliers 5 : Jenkins.....	13
5.1 Prise en main de Jenkins Legacy.....	13
5.1.1 Installation.....	13
5.1.2 Configuration générale et outils.....	13
5.1.3 : Création d'un job Maven.....	14
5.2 : Pipelines Jenkins.....	14
5.2.1 Plugins.....	14
5.2.2 Jenkinsfile déclaratif.....	14
5.3 Docker.....	15
5.3.1 Utilisation d'un agent docker.....	15
5.3.2 Construction et push d'une image.....	15
5.4 Kubernetes.....	15
5.4.1 Installation Jenkins dans le cluster.....	15
5.4.2 Pipeline avec un agent Kubernetes.....	16

# Atelier 1 : Git (Optionnel)

## **1.1. Installation et configuration**

### Installation du binaire

En fonction de votre système d'exploitation, installer le binaire Git. Pour les postes Windows utiliser ***<http://msysgit.github.io>***

### Configuration

En utilisant la commande git config définir dans l'endroit approprié les variables suivantes :

- *user.name*
- *user.email*
- *core.editor*
- *merge.tool*

Visualiser vos changements avec :

*git config --global --edit*

Essayer la commande

*git help -w config*

## **1.2. Initialisation d'un dépôt, enregistrement de modifications**

### Initialisation d'un dépôt

Récupérer les fichiers fournis et les décompresser dans un répertoire de travail.

Initialiser le dépôt, ajouter les fichiers et effectuer un premier commit

### Ajout et modification de fichiers

Ajouter un fichier de votre choix dans l'arborescence et modifier un fichier existant.

Avant d'indexer les modifications, exécuter la commande *git status* et *git diff*

Indexer les fichiers et effectuer la commande *git diff --cached*

Valider vos modifications avec *git commit -v* et éditer le message de commit

Visualiser l'historique avec *gitk*

## **1.3. Fusion et rebase de branches locales**

### Création de Tag

Sur le projet, créer un tag 1.0 puis visualiser ses informations avec *gitk*

#### Création de branche thématique

Créer une nouvelle branche nommée **dev**

La visualiser et basculer la copie locale vers la nouvelle branche

Faire évoluer, en modifiant des fichiers

#### Basculement sur branche master et correction

Basculer sur la branche master, créer une branche **hotfix** et effectuer quelques commits sur les mêmes fichier que précédemment en ayant modifié les mêmes lignes

Quand la branche est prête, fusionner avec **master**, puis supprimer la branche **hotfix**

#### Fusion avec Résolution de conflits

Fusionner la branche **dev** avec la branche **master** et utiliser l'outil graphique pour la résolution de conflits

Commiter et tagger la branche master v1.1

#### Reprendre un tag

Reprendre le tag 1.0. Dans quel état est le HEAD

#### Rebase

Refaire la même opération que précédemment mais en utilisant la commande *rebase*

## **1.4 Branches distantes**

Démarrer un serveur gitlab :

```
docker run --detach \
--hostname gitlab.formation.org \
--publish 443:443 --publish 80:80 --publish 22:22 \
--name gitlab \
--volume /srv/gitlab/config:/etc/gitlab \
--volume /srv/gitlab/logs:/var/log/gitlab \
--volume /srv/gitlab/data:/var/opt/gitlab \
gitlab/gitlab-ce:latest
```

Créer un utilisateur puis un projet

Recréer une branche locale dans votre repository, y faire un commit.

Pousser toutes les branches vers le dépôt distant et assurer vous que les branches distantes sont configurées afin qu'elles soient de branches de suivi des branches locales

# Ateliers 2 : Maven

## 2.1 Installation de Maven et Projet simple

### Objectif

Ce premier projet permet de créer le plus petit projet Java (Hello world) avec Maven. Il permet de se familiariser avec le fichier pom, les commandes Maven principales et la notion de repository local. Un site de documentation est également généré.

### 2.1.1 Installation maven

Télécharger une distribution de Maven et la décompresser dans un répertoire de travail (c:\Formation\Maven, /home/<user>/Maven)

Positionner les variables d'environnement suivantes :

M2\_HOME

M2=\$M2\_HOME/bin

MAVEN\_OPTS (par exemple " -Xms256m -Xmx512m ")

PATH=\$M2:\$PATH

Vérifier que JAVA\_HOME pointe vers la bonne version de Java

Lancer `mvn -version` pour vérifier le bon fonctionnement

### 2.1.2 Personnalisation settings.xml

Nous voulons configurer l'emplacement du repository local.

Créer un fichier `${user.home}/.m2/settings.xml` à partir de `$M2_HOME/conf/settings.xml`

Renseigner la balise `<localRepository/>` à la valeur voulue (c:\Formations\maven\repository /home/<user>/Maven/repository )

Ajouter dans `<pluginGroups>`, la ligne suivante :

```
<pluginGroup>fr.jcgay.maven.plugins</pluginGroup>
```

### 2.1.3 Création d'un projet simple

1. Créer un projet dénommé «hello» en utilisant le plugin **archetype** et en indiquant des valeurs appropriées pour `groupId` et `packageName`.

Par exemple :

```
mvn archetype:generate -DgroupId=org.formatation -DartifactId=hello
-DarchetypeArtifactId=maven-archetype-quickstart -
DarchetypeVersion=1.4 -DinteractiveMode=false
```

2. Observer la structure du répertoire créé, ainsi que les fichiers de configuration et ceux du repository local.  
Quelle est la version du plugin `maven-install` utilisée ?  
Quel code source a été généré ?
3. Éditer le fichier `pom.xml` pour ajouter les informations suivantes :

- Utiliser la propriété **project.name** dans l'URL
- Ajouter des informations sur le type de licence, un développeur
- Utiliser le plugin d'aide pour afficher le POM effectif utilisé par maven et vérifier la résolution de la propriété project.url

```
mvn help:effective-pom
```

4. Exécuter **mvn install** une première fois et observer les fichiers générés dans le répertoire projet ainsi que dans le dépôt local.
5. Exécuter le programme généré avec la commande :  

```
java -cp target/hello-1.0-SNAPSHOT.jar org.formation.App
```
6. Générer la documentation du projet avec **mvn site**, observer les objectifs exécutés puis parcourir le site web généré

## 2.2 : Dépendances

### Objectif

Ce TP permet d'aborder les points suivants :

- Gestion des dépendances
- Retrouver une dépendance

Récupérer le projet fourni et comprendre les relations d'arborescence des POM.

Où sont définies les versions ?

Quelle est la version de driver PostgreSQL ?

Tenter une compilation avec :

```
./mvnw compile
```

Retrouver la dépendance manquante, faut-il indiquer une version ?

Effectuer ensuite

```
./mvnw install
```

## 2.3 : Multi-modules Configuration cycle de vie, profiles

### Objectif

Ce TP permet d'aborder les points suivants :

- Les projets multi-modules
- Configuration des plugins, Attachement de plugins
- Les profiles

### 2.3.1 Projet multi-modules

Récupérer les 2 projets fournis

- **multi-module-library**

- **multi-module-application** qui dépend de *multi-module-library*

Organiser le projet en projet multi-module et définir le POM du projet parent. Faire en sorte que le maximum soit mutualiser dans le projet parent

### 2.3.2 Adaptation du build

#### Visualisation du cycle de vie

Afin de pouvoir utiliser un plugin non-standard, ajouter les lignes suivantes dans votre fichier *settings.xml*

```
<pluginGroups>
  <pluginGroup>fr.jcgay.maven.plugins</pluginGroup>
</pluginGroups>
```

Utiliser ensuite le plugin via

```
./mvnw buildplan:list
```

Est-ce que le cycle par défaut a été adapté ?

#### Configuration du traitement des ressources

Dans le projet application, nous aimerions disposer d'un nouveau répertoire de ressources : *src/main/docker*

Récupérer le répertoire **docker** fourni et le mettre dans *src/main*

Configurer ensuite le plugin **resource** afin que ce nouveau répertoire soit traité de la même façon que *src/main/resources*

Pour tester, faites une compilation dans le projet **application**

#### Ajout de la couverture de test dans la phase de build

Nous voulons insérer l'exécution de l'outil **jacoco** permettant d'évaluer la couverture des tests lors de la phase de test.

Retrouver les coordonnées du plugin et Consulter sa documentation

Pour que la couverture de test soit effective il faut en général exécuter 2 objectifs du plugin :

- **prepare-agent**
- **report**

Exécuter la commande maven permettant de générer les tests

### 2.2.3 Création d'un profil

Configurer un profil «**prod**» dans le projet **application** qui :

- Enlève la dépendance transitive *spring-boot-starter-tomcat*
- Ajoute la dépendance *spring-boot-starter-undertow*
- Exécuter le goal *build-info* du plugin *spring-boot-maven-plugin*.

Construire l'application et visualiser les changements.



# Ateliers 3 : Release, Nexus

## Objectif

Se familiariser avec les plugin SCM, Release et Deploy. Nous repartons du TP précédent.

## Étapes

1. Déploiement dans un dépôt privé géré par Nexus
2. Mise au point du process de distribution de release dans un serveur Nexus

### **3.1 Pré-requis : Installation Nexus**

Installation Nexus Manager :

- Utiliser une distribution relative à votre OS
- Utiliser docker :

```
docker run -d -p 8081:8081 --name nexus -v nexus-data:/nexus-data sonatype/nexus3
```

Visualiser la configuration par défaut de Nexus et les différents dépôts disponibles

S'authentifier avec un compte administrateur

### **3.2 Configuration projet, utilisation Nexus en miroir**

Configurer votre installation de Maven afin de pouvoir utiliser Nexus comme gestionnaire et comme proxy de Maven Central

Effectuer un ***mvn install*** sur le projet précédent et observer les artefacts récupérés par Nexus

### **3.3 Déploiement de snapshots/releases**

Dans le *pom.xml* adéquat, ajouter une balise ***<distributionManagement>*** faisant référence à ce serveur.

Tester votre configuration avec :

```
mvn deploy
```

Vérifier le déploiement des snapshots et de releases

### **3.4 Plugin Release de Maven**

Définir un dépôt distant sur ***gitlab*** ou ***github***

Y pousser le projet

Configurer le SCM dans votre *pom.xml*

Utiliser le plugin release pour effectuer une release complète

# Ateliers 4 : SonarQube

## Objectifs

- Installer SonarQube et découvrir son interface
- Adapter la configuration à un projet
- Intégrer Sonar avec Maven

## 4.1 Installation

Télécharger le distribution, l'installer et démarrer le service ou utiliser une image docker :

```
$ docker run -d --name sonarqube -e  
SONAR_ES_BOOTSTRAP_CHECKS_DISABLE=true -p 9000:9000  
sonarqube:latest
```

Se connecter à Sonarqube <http://localhost:9000>

Se connecter avec admin/admin et définir un nouveau mot de passe

Aller dans le menu **My Account** → **Security** et définir un token :

- analysis
- Global Analysis Token
- No expiration

Générer le jeton et le sauvegarder

## 4.2 Scanner Maven

Récupérer le projet **product-service** fourni, le copier dans un répertoire de travail et initialiser un dépôt Git local, effectuer un premier commit

Tester l'intégration avec la configuration par défaut du serveur SonarQube en exécutant le plugin **sonar:sonar**

```
./mvnw clean test sonar:sonar -Dsonar.token=<token>
```

Configurer le plugin *jacoco* pour obtenir des chiffres sur la couverture de test.

Exécuter la commande maven pour voir la couverture de test dans sonar

## 4.3 Exclusions

Nous voulons exclure de l'analyse :

- Toutes les classes contenant «Dto»

Nous voulons exclure de la couverture de test :

- Toutes les classes présentes dans un package `org.formation.model`

Vérifier vos configuration en lançant l'analyse.

Si c'est correct, committer vos changements dans votre repository git

Traiter une ou 2 issues

## 4.4 Création d'un profil qualité

1. Créer un profil en copiant le profil SonarWay
2. Activer toutes les règles Java sauf les règles dépréciées

Relancer une analyse

Reprendre le profil SonarWay

Retrouver l'identifiant de la règle qui interdit de déclarer 2 attributs sur la même ligne.

Désactiver la pour les classes du package *model*

Relancer une analyse, si vous êtes satisfait committer

## 4.5 Création d'une porte qualité

Créer une nouvelle porte qualité à partir de SonarWay

- Baisser le taux de couverture de test
- Ajouter une contrainte au niveau de la documentation
- Ajouter une contrainte au niveau de la dette technique

Relancer une analyse

# Ateliers 5 : Jenkins

## 5.1 Prise en main de Jenkins Legacy

### Objectifs

- Script de démarrage Jenkins
- Configuration
- Job Maven

### 5.1.1 Installation

Récupérer le .war exécutable et le script de démarrage

Démarrer Jenkins, installer les plugins recommandés et définir un administrateur

### 5.1.2 Configuration générale et outils

#### Gestion des plugins

Installer le plugin Maven

#### 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. Enregistrer vos modifications

#### Configuration serveur de mail (Optionnel)

5. Cliquer sur « *Administrer Jenkins → Configurer le système* »

[stageojen@plbformation.com](mailto:stageojen@plbformation.com)

stageojen

<pop3.plbformation.com>

<smtp.plbformation.com>

(port pop 110 et smtp 587)

### 5.1.3 : Création d'un job Maven

#### Création job et test

1. De retour sur la page d'accueil, Activer le lien « Créer un nouveau job »
2. Créer un job Maven nommé **1\_test** effectuant les cibles clean package et publiant les tests
3. Configurer le SCM afin qu'il point vers le dépôt gitlab de multi-module par exemple
4. Indiquer que le build est déclenché à chaque changement dans GIT et que le repository est interrogé toutes les 5 mns. (\*/\* \* \* \* \*)
5. Exécuter manuellement le build et observer les résultats
6. Observer l'archivage automatique et les artefacts archivés, la configuration multi-modules, Assurer vous de la publication des résultats de test

#### Déclenchement de job

1. Modifier un fichier du projet provoquant une erreur dans les tests, committer le changement dans le repository
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

## 5.2 : Pipelines Jenkins

#### Objectifs

- Mettre en place une pipeline Jenkins avec le plugin pipeline et un Jenkinsfile

### 5.2.1 Plugins

Installer le plugins *Blue Ocean*

Dans VsCode, installer les extensions :

- **Jenkins Pipeline Linter Connector**
- **JenkinsFile Support**

Configurer l'extension Jenkins Pipeline Linter Connector :

- **CrumbUrl** : [http://localhost:8082//crumbIssuer/api/xml?xpath=concat\(//crumbRequestField,%22:%22,//crumb\)](http://localhost:8082//crumbIssuer/api/xml?xpath=concat(//crumbRequestField,%22:%22,//crumb))
- **Url** : <http://localhost:8082/pipeline-model-converter/validate>
- **User** : L'utilisateur admin de Jenkins
- **Pass** : Le mot de passe

### 5.2.2 Démarrage

Dans le projet product-service, créer une branche ci, ajouter le fichier Jenkinsfile fourni,

Vérifier que le fichier Jenkinsfile est valide via le Linter :

*View → Commande Palette → Validate Jenkinsfile*

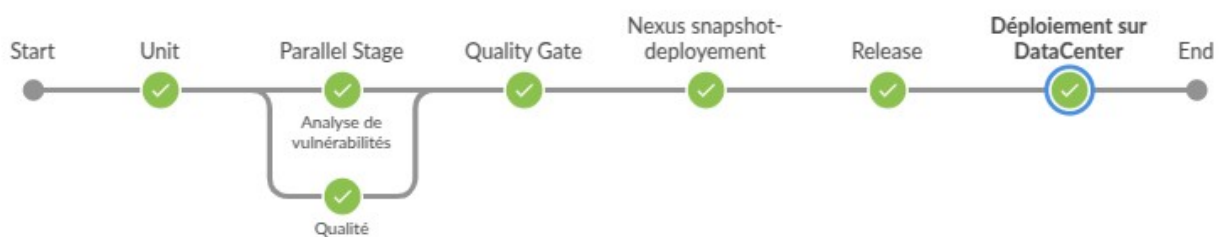
Committer dans la branche **ci** puis pousser sur le dépôt distant

Dans Jenkins, créer un job product-service de type « Multibranche Pipeline »

Configurer :

- L'url du dépôt
- Une période de scan sur 2 minutes

Vérifier l'exécution de la pipeline sur la branche ci, vous devez obtenir ceci dans « Blue Ocean »



### 5.2.2 Phase unit

Dans la phase unit, exécuter les tests unitaires avec la commande suivante :

```
./mvnw -Dmaven.test.failure.ignore clean test
```

Dans la section de post build :

- Publier systématiquement les tests junit
- Si échec envoyer un email sur une adresse

Valider votre Jenkinsfile

Committer, pousser et vérifier que la pipeline s'exécute correctement

### 5.2.2 Phase parallèle

Provisionner un agent pour chaque branche de cette phase parallèle

#### Vulnérabilités (Optionnel)

Dans la phase OWASP utiliser le plugin OWASP permettant de détecter les vulnérabilités :

***org.owasp :dependency-check-maven***

Configurer le plugin et sa version dans le pom.xml dans un profil **owasp**

Vous devez récupérer une clé d'API ici <https://nvd.nist.gov/developers/request-an-api-key>

Et configurer le plugin en ajoutant un bloc d'exécution et de configuration :

```
<executions>
  <execution>
    <goals>
      <goal>check</goal>
    </goals>
  </execution>
</executions>
<configuration>
  <nvdApiKey>${nvd.api.key}</nvdApiKey>
</configuration>
```

Tester en local :

```
./mvnw -Powasp -DskipTests verify
```

La première exécution est longue, vous pouvez en attendant travailler sur l'analyse qualité

Localiser le rapport généré.

Dans le stage Jenkins, ajouter l'appel de la commande Maven et archiver le rapport OWASP

### Analyste qualité

Configurer dans Jenkins le credentials SONAR :

*Administer Jenkins → Credentials → Global → Add Credentials*

Choisir *Secret text*

Copier le token sonar dans *Secret* , lui donner l'id **SONAR\_TOKEN**

Installer le plugin **SonarScanner** puis le configurer

*Administrer Jenkins → System* Indiquer l'url du serveur sonar et le token

Dans le stage d'analyse qualité

Utiliser l'instruction fournie par la plugin withSonarqubeEnv

Dans le bloc, lancer une analyse via Maven

### **5.2.3 Attente de la quality gate**

Dans ce stage, mettre en pause la pipeline pendant une minute (le temps que l'analyse Sonar se fasse) puis faire un appel à **wait4QualiteGate**



### 5.2.4 Déploiement SNAPSHOT

Si nous ne sommes pas sur la branche principale.

Effectuer un déploiement d'une version snapshot vers Nexus

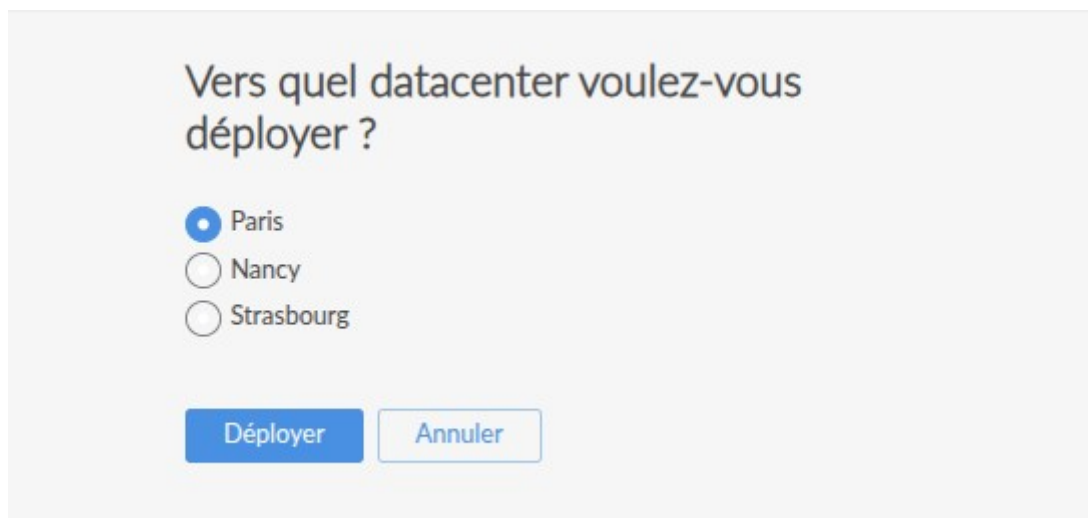
### 5.2.5 Effectuer une release

Si on est la branche principale, effectuer une release.

Attention assurer vous d'avoir un workspace propre avant d'effectuer la commande maven

### 5.2.6 Simuler un déploiement sur une infrastructure

Créer un formulaire demandant à l'utilisateur vers quel Datacenter, il veut déployer



The image shows a web form titled "Vers quel datacenter voulez-vous déployer ?". It contains three radio button options: "Paris" (selected), "Nancy", and "Strasbourg". At the bottom, there are two buttons: "Déployer" (highlighted in blue) and "Annuler" (outlined in blue).

Récupérer le dernier artefact dans nexus la copier sur votre disque en renommant son nom avec le datacenter.

Pour implémenter, vous pouvez utiliser un bloc script qui exécute la logique suivante :

- Se synchroniser avec le dépôt d'origin

```
git fetch origin
```

- Récupérer le dernier tag disponible dans une variable en exécutant

```
git tag --sort=-creatordate | head -n 1
```

- Extraire le n° de version du tag
- Récupérer l'artefact avec wget
- Copier dans le répertoire de déploiement

## 5.3 Docker

### 5.3.1 Utilisation d'un agent docker

Installer le plugin **Docker pipeline**

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

### 5.3.2 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

## 5.4 Kubernetes

Pré-requis :

Compte Github ou Gitlab

### 5.4.1 Installation Jenkins dans le cluster

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

Configurer Helm

```
helm repo add jenkinsci https://charts.jenkins.io
helm repo update
helm search repo jenkinsci
```

Créer un cluster **kind** :

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

Créer un namespace **jenkins**

```
kubectl create namespace jenkins
```

Créer un volume persistant :

```
kubectl apply -f
https://raw.githubusercontent.com/jenkins-infra/jenkins.io/master/content/doc/tutorials/kubernetes/installing-jenkins-on-kubernetes/jenkins-volume.yaml
```

Créer le compte service jenkins

```
kubectl apply -f
https://raw.githubusercontent.com/jenkins-infra/jenkins.io/master/content/doc/tutorials/kubernetes/installing-jenkins-on-kubernetes/jenkins-sa.yaml
```

Visualiser le fichier **jenkins-values.yaml** permettant de personnaliser l'installation helm et exécuter dans le même terminal :

```
chart=jenkinsci/jenkins
```

```
helm install jenkins -n jenkins -f jenkins-values.yaml $chart
```

Récupérer le mot de passe administrateur avec :

```
jsonpath="{.data.jenkins-admin-password}"
secret=$(kubectl get secret -n jenkins jenkins -o
jsonpath=$jsonpath)
echo $(echo $secret | base64 -decode)
```

Récupérer l'URL jenkins avec :

```
jsonpath="{.spec.ports[0].nodePort}"
NODE_PORT=$(kubectl get -n jenkins -o jsonpath=$jsonpath services
jenkins)
jsonpath="{.items[0].status.addresses[0].address}"
NODE_IP=$(kubectl get nodes -n jenkins -o jsonpath=$jsonpath)
echo http://$NODE_IP:$NODE_PORT/login
```

Se logger avec admin et le mot de passe précédent.

### 5.4.2 Pipeline avec un agent Kubernetes

Sur le projet multi-module, récupérer le fichier **kubernetesPod.yaml** et le visualiser.

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

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

Créer un Multi-branch pipeline pointant sur le dépôt public et tester

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

```
kubectl get pods -n jenkins -w
```

