Creation d’une toolchain custom

Table des matières

[Création de l’environnement de travail sur le poste local 3](#_Toc529286001)

[Initialisation de l’environnement de travail sur le poste local 3](#_Toc529286002)

[Création du code source 3](#_Toc529286003)

[Création d'un projet sur GitHub 7](#_Toc529286004)

[Clone sur le poste/serveur local 7](#_Toc529286005)

[Préparation de l’environnement de build 8](#_Toc529286006)

[(Optionnel) tester le fonctionnement du code dans un Docker 9](#_Toc529286007)

[Initialisation d’une instance de tests 9](#_Toc529286008)

[Compilation du projet 9](#_Toc529286009)

[Préparation de l’environnement Docker 9](#_Toc529286010)

[Installation de Jenkins et paramétrage de GitHub et Maven 11](#_Toc529286011)

[Installation de Jenkins 11](#_Toc529286012)

[Création d’un token d’authentification dans GitHub 12](#_Toc529286013)

[Configuration du Webhook GitHub dans Jenkins 13](#_Toc529286014)

[Création du repo Nexus 14](#_Toc529286015)

[Installation de Nexus 14](#_Toc529286016)

[Création de repository 14](#_Toc529286017)

[Compilation et publication via Jenkins 16](#_Toc529286018)

[Tests Sonar 17](#_Toc529286019)

[Installation de Sonar 17](#_Toc529286020)

[Exécution des tests via Jenkins 17](#_Toc529286021)

[dependency-check (OWASP) 18](#_Toc529286022)

[Installation du plugin dans Jenkins 18](#_Toc529286023)

[Installation du plugin dans Sonar 18](#_Toc529286024)

[Exécution des tests via Jenkins 19](#_Toc529286025)

[Tests unitaires 20](#_Toc529286026)

[Configuration des tests unitaires 20](#_Toc529286027)

[Exécution des tests via Jenkins 20](#_Toc529286028)

[Affichage des tests unitaires dans Sonar 20](#_Toc529286029)

[Build des images Docker par Jenkins 22](#_Toc529286030)

[Création d'une registry Docker 23](#_Toc529286031)

[Création de la registry en http 23](#_Toc529286032)

[Création de la registry en https (requis pour Kubernetes) 23](#_Toc529286033)

[Gestion des images Docker et de la registry 24](#_Toc529286034)

[Déploiement dans Kubernetes 26](#_Toc529286035)

[Installation de Rancher et Kubernetes 26](#_Toc529286036)

[Installation manuelle d’un master Kubernetes et d’un Node séparé 27](#_Toc529286037)

[Quelques commandes du CNI Weave 29](#_Toc529286038)

[Diverses commandes Kubernetes 29](#_Toc529286039)

[Installation d’un Ingress Controller Kubernetes-Nginx 29](#_Toc529286040)

[Création d’une configuration de déploiement Kubernetes 29](#_Toc529286041)

[Stage Jenkins de déploiement dans Kubernetes 31](#_Toc529286042)

[Spécifier une registry Docker 32](#_Toc529286043)

[Installation Kubernetes via conjure-up sur Ubuntu 32](#_Toc529286044)

# Création de l’environnement de travail sur le poste local

## Initialisation de l’environnement de travail sur le poste local

1. Créer le répertoire du projet sur le disque local (exemple : « *C:\API-javaSpringboot »* sous Windows)
2. Créer la structure de répertoires suivantes :
   * API-javaSpringboot
     + src
       - main
         * java

hello

* + - * test
        + java

hello

## Création du code source

Créer les 4 fichiers suivants utilisés pour le code source :

* API-javaSpringboot/pom.xml
* API-javaSpringboot/src/main/java/hello/Greeting.java
* API-javaSpringboot/src/main/java/hello/GreetingController.java
* API-javaSpringboot/src/main/java/hello/Application.java
* API-javaSpringboot/src/test/java/hello/ApplicationTests.java

API-javaSpringboot/**pom.xml** :

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.springframework</groupId>

<artifactId>API-javaSpringboot</artifactId>

<version>0.1.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.5.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.jayway.jsonpath</groupId>

<artifactId>json-path</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<properties>

<java.version>1.8</java.version>

</properties>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-releases</id>

<url>https://repo.spring.io/libs-release</url>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>spring-releases</id>

<url>https://repo.spring.io/libs-release</url>

</pluginRepository>

</pluginRepositories>

</project>

API-javaSpringboot/src/main/java/hello/**Greeting.java** :

package hello;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

API-javaSpringboot/src/main/java/hello/**GreetingController.java** :

package hello;

import java.util.concurrent.atomic.AtomicLong;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class GreetingController {

private static final String template = "Salut, %s!";

private final AtomicLong counter = new AtomicLong();

@RequestMapping("/greeting")

public Greeting greeting(@RequestParam(value="name", defaultValue="World") String name) {

return new Greeting(counter.incrementAndGet(),

String.format(template, name));

}

}

API-javaSpringboot/src/main/java/hello/**Application.java** :

package hello;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

API-javaSpringboot/src/test/java/hello/**ApplicationTests.java**

package hello;

import java.util.Map;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.boot.test.web.client.TestRestTemplate;

import org.springframework.boot.web.server.LocalServerPort;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.test.context.TestPropertySource;

import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.BDDAssertions.then;

/\*\*

\* Basic integration tests for service demo application.

\*

\* @author Dave Syer

\*/

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

@TestPropertySource(properties = {"management.port=0"})

public class ApplicationTests {

@LocalServerPort

private int port;

@Value("${local.management.port:${local.server.port}}")

private int mgt;

@Autowired

private TestRestTemplate testRestTemplate;

@Test

public void shouldReturn200WhenSendingRequestToController() throws Exception {

@SuppressWarnings("rawtypes")

ResponseEntity<Map> entity = this.testRestTemplate.getForEntity(

"http://localhost:" + this.port + "/greeting", Map.class);

then(entity.getStatusCode()).isEqualTo(HttpStatus.OK);

}

}

## Création d'un projet sur GitHub

a) Se connecter sur GitHub et créer un nouveau Repository :

https://github.com/API-javaSpringboot

b) Dans la fenêtre de création d'un nouveau repository :

Par exemple : API-javaSpringboot

c) Copier le lien https du repository :

Par exemple : https://github.com/maddoudou22/API-javaSpringboot.git

## Clone sur le poste/serveur local

1. Installer Git Bash sous Windows et l’exécuter :
   1. Se placer dans le répertoire du projet (exemple : « *C:\API-javaSpringboot »*).
   2. Exécuter les commandes suivantes :
      * $ git init
      * $ git config --global user.email [you@example.com](mailto:you@example.com)
      * $ git config --global push.default simple
      * $ git config --global credential.helper wincred

*Note : cette commande permet d’éviter de renseigner renseigner le username et mot de passe du compte GitHub chaque push.*

* + - $ git clone <https://github.com/maddoudou22/API-javaSpringboot.git>
    - $ git add \*
    - $ git commit -m "First push !"
    - $ git push --set-upstream https://github.com/maddoudou22/API-javaSpringboot.git master

*Note : pour les push suivants : $ git push https://github.com/maddoudou22/API-javaSpringboot.git*

1. Simplifier le push de code dans GitHub :
   1. Créer un fichier API-javaSpringboot/**Git Push.bat**:

echo off

SET nom\_Projet\_Git=API-javaSpringboot

SET URL\_Projet\_Git=https://github.com/maddoudou22/API-javaSpringboot.git

SET Repertoire\_Local\_Projet=C:\API-javaSpringboot

cd %Repertoire\_Local\_Projet%

echo "Git Add ..."

git add -A

echo "Git Commit ..."

git commit -m "latest push from windows"

echo "Git Push ..."

git push %URL\_Projet\_Git%

*Note : à chaque modification d’un fichier du code source, exécuter ce script afin de pousser l’ensemble des modifications dans GitHub.*

## Préparation de l’environnement de build

*Note : provisionner une instance EC2 qui acceuillera la toolchain Devops (provisionner une EC2 t3.large de préférence pour éviter les problème de mémoire de JVM lors du premier Build)*

1. Installer la version 8 de Java .

$ apt-get install openjdk-8-jdk

1. Sélectionner la version de Java à utiliser par défaut :

$ update-alternatives --config java

1. Récupérer le chemin de la JRE voulue (ex : « */usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java »*) et ajouter la ligne suivante à la fin du fichier *"/etc/environment"* :

$ JAVA\_HOME="/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java"

1. Sourcer le fichier *"/etc/environment"* :

$ source /etc/environment

1. Installer Maven :

$ apt-get install maven

# (Optionnel) tester le fonctionnement du code dans un Docker

## Initialisation d’une instance de tests

1. Provisionner une image EC2 Ubuntu dans AWS
2. Changer la timezone de l’instance :

$ apt-get update

$ dpkg-reconfigure tzdata

## Compilation du projet

1. Se placer dans le répertoire contenant le POM du projet (exemple : *« ~/API-javaSpringboot »*).
2. Compiler le projet :

$ mvn clean package

1. Vérifier l’absence d’erreurs et la présence de l’archive .jar dans le dossier target (exemple : *« ~/API-javaSpringboot/target/API-javaSpringboot-0.1.0.jar »*).
2. Tester le fonctionnement de l’application : exécuter l'application via Spring Boot :

$ java -jar target/API-javaSpringboot-0.1.0.jar

1. Tester avec l'URL suivante dans un navigateur :

http://<IP publique instance EC2>:8080/greeting

La réponse doit être :

{"id":1,"content":"Hello, World!"}

1. Tester avec l'URL suivante :

http://<IP publique instance EC2>:8080/greeting?name=User

La réponse doit être :

{"id":2,"content":"Hello, User!"}

## Préparation de l’environnement Docker

1. Installer Docker :

$ apt-get install docker.io

1. Créer un répertoire de déploiement (exemple : « ~/docker-tests »).
2. Se placer dans ce répertoire et créer le dockerfile *"dockerfile-exec-jar"* avec le contenu suivant :

FROM java:8

WORKDIR /root/docker-tests

ADD API-javaSpringboot-0.1.0.jar API-javaSpringboot-0.1.0.jar

EXPOSE 8080

CMD java -jar API-javaSpringboot-0.1.0.jar

*(****Attention*** *: laisser une ligne vide à la fin du fichier)*

1. Créer l’image Docker :

$ docker build -f dockerfile-exec-jar -t Image-API-javaSpringboot /root/docker-tests/

1. Vérifier la bonne création de l'image :

$ docker images

Doit retourner la liste des images, dont l’une doit être « Image-API-javaSpringboot ».

1. Exécuter un conteneur basé sur cette image (exécution en arrière-plan) :

$ docker run -d -p 8088:8080 Image-API-javaSpringboot

1. Vérifier le bon fonctionnement du conteneur :

$ docker ps

Doit retourner la liste des conteneurs en cours d’exécution. Le port 8088 du serveur hôte est mappé sur le port 8080 du conteneur.

1. Tester avec l'URL suivante dans un navigateur :

http://<IP publique instance EC2>:8088/greeting

La réponse doit être :

{"id":1,"content":"Hello, World!"}

*A ajouter dans le pom.xml pour que Maven créé l’archive .jar dans un autre endroit que ‘/target’ :*

*<plugin>*

*<groupId>org.apache.maven.plugins</groupId>*

*<artifactId>maven-jar-plugin</artifactId>*

*<version>2.3.1</version>*

*<configuration>*

*<outputDirectory>${targetRepo}</outputDirectory>*

*</configuration>*

*</plugin>*

# Installation de Jenkins et paramétrage de GitHub et Maven

## Installation de Jenkins

1. Installer Jenkins :

$ wget -q -O - https://pkg.jenkins.io/debian/jenkins-ci.org.key | sudo apt-key add -

$ echo deb http://pkg.jenkins.io/debian-stable binary/ | sudo tee /etc/apt/sources.list.d/jenkins.list

$ sudo apt-get update

$ apt-get install jenkins

1. Remplacer le port d’écoute par défaut de Jenkins (8080) dans le fichier *« /etc/default/jenkins »* -> Le remplacer par exemple par 9090.
2. Démarrer Jenkins :

$ systemctl start Jenkins

$ systemctl status jenkins

1. Récupérer le mot de passe de Jenkins depuis le fichier *« /var/lib/jenkins/secrets/initialAdminPassword »*.
2. Se connecter à l'URL d'admin de Jenkins (http://<IP publique instance EC2>:9090/) et coller le mot de passe.
3. Laisser l’installation des plugins par défaut.
4. Dans la console d'admin web de **Jenkins**, aller dans « Administrer Jenkins », puis « Gestion des plugins ».
5. Dans l’onglet *‘Disponible’*, sélectionner *‘Blue Ocean’* puis cliquer sur le bouton *‘Installer sans redémarrer’*.
6. Dans l’onglet *‘Disponible’*, sélectionner *‘pipeline-utility-steps’* puis cliquer sur le bouton *‘Installer sans redémarrer’*.
7. Créer un fichier '*Jenkinsfile*' à la racine du répertoire de l'application à côté du '*pom.xml'*. Ce fichier contiendra les Stage à exécuter :

API-javaSpringboot/**Jenkinsfile :**

pipeline {

agent any

environment {

package\_version = readMavenPom().getVersion() // Sert à utiliser la version du package définit dans le pom.xml comme variable dans le reste du fichier (pas de harcoding).

applicationName = 'API-javaSpringboot' // Convention : utiliser le même nom que l'artifactId du pom.xml

}

stages {

stage('test') {

steps {

echo 'Test stage ...'

}

}

}

}

## Création d’un token d’authentification dans GitHub

1. Dans le menu général de GitHub, aller dans *‘Settings’*, puis *‘Developer Settings’,* puis *‘Personal access tokens’*
2. Cliquer sur le bouton ‘*Generate New Token’* et saisir le nom « *Jenkins-token* » dans ‘*Token Description*’.
3. Cliquer sur le bouton ‘*Generate New Token’* et saisir le nom « *Jenkins-token* » dans ‘*Token Description*’.
4. Sélectionner les scopes suivants :

* repo (tout)
* admin:repo\_hook (tout)
* gist
* user (tout)

1. Clicker sur "*Generate token*" et noter le token généré (exemple : 2*2ede6581035a0cb521b306bb76e187217e68c76*)

## Configuration du Webhook GitHub dans Jenkins

1. Dans la console d'admin web de **Jenkins**, aller dans « Administrer Jenkins », puis « Configurer le système ».
2. Rechercher le paramètre « URL de Jenkins » et remplacer le port d’écoute par celui modifié précédemment (ex : 9090).

*Note : c’est cette URL que GitHub appellera à chaque commit.*

1. Dans la console Web de **GitHub**, aller dans le repository créé et cliquer sur son bouton *'Settings'*, puis sur *'Webhooks'* dans le menu de gauche. Cliquer sur le bouton *'Add Webhook'*.
2. Dans le champ *'Payload URL'*, coller l'URL de Jenkins , suivie de "*github-webhook*" (exemple : http://52.210.141.11:9090/github-webhook/).
3. Dans "*Content type*", choisir "*application/json*", puis cliquer sur le bouton "*Add Webhook*".
4. Dans la console d'admin web de **Jenkins**, créer un nouvel item.
5. Saisir un nom au projet, puis sélectionner « *Pipeline* ». Cliquer sur le bouton « *Ok* ».
6. Dans la page qui apparaît, dans l’onglet ‘*Général’*, cocher la case ‘*GitHub Projet*’ et saisir l’URL du projet (exemple : *« https://github.com/maddoudou22/API-javaSpringboot/ »*).
7. Dans la section « *Build triggers* », cocher « *GitHub hook trigger for GITScm polling* ».
8. Dans la section « *Pipeline* », dans ‘*Definition’* sélectionner *‘Pipeline script from SCM’* :
   1. Dans *‘SCM’*, Sélectionner *‘Git’*. Dans ‘Repositories’ :
      1. Dans ‘Repository URL’, saisir l’URL du repo GitHub (exemple : [*https://github.com/maddoudou22/API-javaSpringboot*](https://github.com/maddoudou22/API-javaSpringboot)).
      2. Dans ‘*Credentials’*, laisser ‘*aucun’* et cliquer sur le bouton ‘*Ajouter’*. Dans ‘*Type’* choisir ‘*Secret text’* et coller le token généré par GitHub (exemple : 2*2ede6581035a0cb521b306bb76e187217e68c76*).
9. Cliquer sur le bouton « *Sauver* ».
10. Dans le menu Jenkins du pipeline créé, cliquer sur le bouton *‘Lancer un build’* pour tester la configuration.
11. Tester le Webhook : modifier une partie du code source sur le poste de travail et exécuter le batch « *Git Push.bat* ».

Dans le projet créé, Jenkins doit détecter le build dans l’historique des builds.

## Création du repo Nexus

### Installation de Nexus

1. Sur l’instance EC2, créer un répertoire Nexus (exemple : « */var/lib/nexus »*).
2. Se placer dans ce répertoire et récupérer la dernière version d’une archive Nexus depuis un repo externe :

$ wget http://www.sonatype.org/downloads/nexus-latest-bundle.zip

1. Dézipper l’archive :

$ apt-get install zip

$ unzip nexus-latest-bundle.zip

*Note : un ‘Application Directory’ est créé dans ‘/var/lib/nexus/nexus-<version>’. Un ‘Data Directory’ est créé dans ‘/va/lib/nexus/sonatype-work/nexus/’.*

1. *Afin de démarrer Nexus en tant que root : non recommandé, à ne faire que dans le cadre d’un POC :*
   1. Ajouter la ligne RUN\_AS\_USER=root à la fin du fichier *« /etc/environment »*
   2. Sourcer le fichier d’environnement :

$ source /etc/environment

* 1. Exporter la variable d’environnement :

$ export RUN\_AS\_USER=root

1. Démarrer Nexus :

$ /var/lib/nexus/nexus-2.14.10-01/bin/nexus start

1. Accéder à la console de Nexus :

http://<IP publique instance EC2>:8081/nexus/

1. Se logger via les credentials par défaut *(admin/admin123)*.

### Création de repository

1. Créer un **nouveau repository** de **SNAPSHOTS** :

* Dans le menu de gauche, dans *‘Views/Repositories’* cliquer sur *‘Repositories’*.
* Cliquer sur le bouton ‘*Add’* puis ‘*Hosted repository’*.
* Saisir un *Repository ID* (exemple : *« API-javaSpringboot-snapshots »*) et un *Repository Name* (exemple : *« API-javaSpringboot snapshot repository »* ).
* Laisser Maven 2 en provider et sélectionner *‘Snapshot’* pour le *‘Repository Policy’*.
* Cliquer sur ‘Save’.
* Aller dans l’onglet ‘*Summary’* de ce repo, puis copier l’URL du repo (exemple : http://<IP instance EC2>:8081/nexus/content/repositories/API-javaSpringboot-snapshots)

1. Créer un **nouveau repository** de **RELEASE** :

* Dans le menu de gauche, dans *‘Views/Repositories’* cliquer sur *‘Repositories’*.
* Cliquer sur le bouton ‘*Add’* puis ‘*Hosted repository’*.
* Saisir un *Repository ID* (exemple : *« API-javaSpringboot-release »*) et un *Repository Name* (exemple : *« API-javaSpringboot release repository »* ).
* Laisser Maven 2 en provider et sélectionner *‘Release'* pour le *‘Repository Policy’*.
* Cliquer sur ‘Save’.
* Aller dans l’onglet ‘*Summary’* de ce repo, puis copier l’URL du repo (exemple : http://<IP instance EC2>:8081/nexus/content/repositories/API-javaSpringboot-release)

1. Dans le fichier ‘pom.xml’ de l’application, ajouter la section <distributionManagement> pour indiquer à Maven comment utiliser les repos *(exemple créé à partir des exemples ci-dessus)* :

API-javaSpringboot/**pom.xml** :

[...]

<distributionManagement>

<snapshotRepository>

<id>API-javaSpringboot-snapshots</id>

<name>API-javaSpringboot snapshot repository</name>

<url>http://localhost:8081/nexus/content/repositories/API-javaSpringboot-snapshots</url>

</snapshotRepository>

<repository>

<id>API-javaSpringboot-release</id>

<name>API-javaSpringboot release repository</name>

<url>http://localhost:8081/nexus/content/repositories/API-javaSpringboot-release</url>

</repository>

</distributionManagement>

[...]

***Note****: lorsque la version dans le pom.xml finira par « -SNAPSHOT », le package .jar généré ira dans le repo de snapshots (exemple : 1.1.0-SNAPSHOT).*

*Lorsque la version dans le pom.xml ne finira pas par « -SNAPSHOT », le package .jar généré ira dans le repo des releases (exemple : 1.1.0).*

1. Copier le template de configuration « settings.xml » de Maven : :

$ cp /usr/share/maven/conf/settings.xml /var/lib/jenkins/.m2/

$ chown jenkins: /var/lib/jenkins/.m2/settings.xml

1. Editer le fichier configuration « settings.xml » pour utiliser le repo Nexus précédemment créé *(exemple créé à partir des exemples ci-dessus) en modifiant la section <server>* :

/var/lib/jenkins/.m2/**settings.xml** :

[...]

<servers>

<server>

<id>API-javaSpringboot-snapshots</id>

<username>admin</username>

<password>admin123</password>

</server>

<server>

<id>API-javaSpringboot-release</id>

<username>admin</username>

<password>admin123</password>

</server>

</servers>

[...]

### Compilation et publication via Jenkins

1. Dans le Jenkinsfile, ajouter 2 stages : un pour la compilation des sources, l'autre pour la publication dans un repo :

API-javaSpringboot/**Jenkinsfile :**

[...]

stage('Build') {

steps {

echo 'Building ...'

// Supprime les fichiers présents dans le répertoire 'target/' du workspace de Jenkins (/var/lib/jenkins/workspace/API-javaSpringboot\_pipeline/), copie les sources depuis GitHub dans ce workspace, puis ne fait que la compilation (pas de déploiement dans Nexus). N'exécute pas non plus les tests unitaires.

sh 'mvn -Dmaven.test.skip=true clean install'

}

}

stage('Publish package') {

steps {

echo 'Publising into the Nexus repository ...'

// Publie le .jar précédemment compilé (sans le recompiler à nouveau) dans le repo de snapshot ou release selon si la version du package définit dans le pom.xml finit par '-SNAPSHOT' ou non.

sh 'mvn jar:jar deploy:deploy'

}

}

[...]

## Tests Sonar

### Installation de Sonar

1. Dans l’instance EC2, télécharger et démarrer une image Docker de SonarQube :

$ docker pull sonarqube

$ docker run -d --name sonarqube -p 9000:9000 -p 9092:9092 sonarqube

1. Accéder à la console de Sonar : http://<IP publique instance EC2>:9000
2. Se logger avec admin/admin, puis dans l’écran de bienvenue :
   1. Générer un token d’authentification avec un nom (exemple : ***2a2cd66cab7710c20b0f91b258c7e94b8ed40228***)
   2. Sélectionner le langage *Java* pour l’analyse

*Sonar affiche la commande à coller dans un nouveau staging Jenkins pour lancer le scan.*

* 1. Cliquer sur le bouton *‘Finish this tutorial’*.

### Exécution des tests via Jenkins

1. Dans le Jenkinsfile, ajouter un stage pour l'exécution des tests :

API-javaSpringboot/**Jenkinsfile :**

stage('Sonar - Code Quality') {

steps {

echo 'Check Code Quality ...'

sh 'mvn sonar:sonar'

}

}

## dependency-check (OWASP)

### Installation du plugin dans Jenkins

1. Dans la console d'administration de Jenkins, installer le plugin *"OWASP Dependency-Check Plugin"* puis redémarrer Jenkins.

### Installation du plugin dans Sonar

1. Dans l’instance EC2, récupérer le plugin *"*dependency-check-sonar-plugin *"* de Steve Sprigett depuis GitHub afin d'afficher le rapport des vulnérabilités OWASP directement dans Sonar :

$ wget https://github.com/stevespringett/dependency-check-sonar-plugin/releases/download/1.1.1/sonar-dependency-check-plugin-1.1.1.jar

1. Le copier dans le conteneur exécutant Sonar, puis redémarrer Sonar :

$ cp sonar-dependency-check-plugin-1.1.1.jar <ID du conteneur Sonar>:/opt/sonarqube/extensions/plugins/

$ docker restart <ID du conteneur Sonar>

1. Dans le pom.xml, ajouter les *'properties'* suivantes :

API-javaSpringboot/**pom.xml** :

[...]

<properties>

[...] <dependency.check.report.dir>${project.build.directory}</dependency.check.report.dir>

<sonar.dependencyCheck.htmlReportPath>${project.build.directory}/dependency-check-report.html</sonar.dependencyCheck.htmlReportPath> </properties>

[...]

1. Dans le pom.xml, ajouter les *'plugins'* suivants :

API-javaSpringboot/**pom.xml** :

[...]

<plugin>

<groupId>org.sonarsource.scanner.maven</groupId>

<artifactId>sonar-maven-plugin</artifactId>

<version>3.5.0.1254</version>

</plugin>

<plugin>

<groupId>org.owasp</groupId>

<artifactId>dependency-check-maven</artifactId>

<version>3.3.2</version>

<configuration>

<failBuildOnCVSS>8</failBuildOnCVSS>

</configuration>

<executions>

<execution>

<goals>

<goal>check</goal>

</goals>

</execution>

</executions>

</plugin>

[...]

### Exécution des tests via Jenkins

1. Dans le Jenkinsfile, ajouter un stage pour l'exécution du test :

API-javaSpringboot/**Jenkinsfile :**

[...]

stage('OWASP - Dependencies check') {

steps {

echo 'Check OWASP dependencies ...'

sh 'mvn dependency-check:check'

}

}

[...]

*Note : ce stage doit être exécuté* ***avant le stage de Sonar****, car c'est Sonar qui récupérera le rapport existant pour l'afficher.*

Le rapport est disponible dans Sonar, dans l'onglet *'More -> Dependency-check'*.

*Note : le plugin ne se sert que du rapport de dépendance '.html' et pas du '.xml'.*

## Tests unitaires

### Configuration des tests unitaires

1. Dans le pom.xml, ajouter la dépendance suivante :

API-javaSpringboot/**pom.xml** :

[...]

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

[...]

### Exécution des tests via Jenkins

1. Dans le Jenkinsfile, ajouter un stage pour l'exécution des tests unitaires (sans recompilation du code, donc le code doit avoir été compilé avant) :

API-javaSpringboot/**Jenkinsfile :**

[...]

stage('Unit test') {

steps {

echo 'Unit testing ...'

sh 'mvn test'

}

}

[...]

### Affichage des tests unitaires dans Sonar

1. Afin d'être affichés dans Sonar, les tests doivent être générés via le plugin JaCoCo. Se loguer dans la console d'administration de Sonar, puis aller dans *Administration -> Marketplace*.
2. Chercher le plugin *'JaCoCo'* et cliquer sur le bouton *'Install'*.
3. Redémarrer Sonar pour activer le plugin.
4. Dans le pom.xml, ajouter le plugin suivant afin d'activer JaCoCo et le rapport dans Sonar :

API-javaSpringboot/**pom.xml** :

[...]

<plugin>

<groupId>org.jacoco</groupId>

<artifactId>jacoco-maven-plugin</artifactId>

<version>0.7.7.201606060606</version>

<executions>

<execution>

<goals>

<goal>prepare-agent</goal>

</goals>

</execution>

<execution>

<id>report</id>

<phase>prepare-package</phase>

<goals>

<goal>report</goal>

</goals>

</execution>

<execution>

<id>jacoco-check</id>

<goals>

<goal>check</goal>

</goals>

<configuration>

<rules>

<rule>

<element>PACKAGE</element>

<limits>

<limit>

<counter>LINE</counter>

<value>COVEREDRATIO</value>

<minimum>0.50</minimum>

</limit>

</limits>

</rule>

</rules>

</configuration>

</execution>

</executions>

</plugin>

[...]

Le *'code coverage'* est maintenant renseigné dans Sonar.

## Build des images Docker par Jenkins

1. Ajouter Jenkins au groupe Docker afin qu’il ait le droit d’utiliser Docker, puis redémarrer Jenkins :

$ usermod -aG docker jenkins

$ service jenkins restart

1. Créer le fichier 'dockerfile' à la racine du projet sur le poste local (sera exporté dans le workspace de Jenkins via GitHub) avec le contenu suivant :

API-javaSpringboot/**dockerfile :**

FROM java:8

ARG PACKAGE\_VERSION

ARG APPLICATION\_NAME

COPY /target/${APPLICATION\_NAME}-${PACKAGE\_VERSION}.jar ${APPLICATION\_NAME}-${PACKAGE\_VERSION}.jar

EXPOSE 8080

CMD java -jar ${APPLICATION\_NAME}-${PACKAGE\_VERSION}.jar

*Note: laisser une ligne vide à la fin du fichier*

1. Dans le Jenkinsfile, déclarer des variables d'environnement pour les metadonnées de l'application + la registry Docker, puis ajouter un stage pour le build de l'image Docker :

API-javaSpringboot/**Jenkinsfile :**

***[...]***

environment {

***[...]***

package\_version = readMavenPom().getVersion() // Sert à utiliser la version du package définit dans le pom.xml comme variable dans le reste du fichier (pas de harcoding).

dockerRegistry = "maddoudou22/api-javaspringboot" // Variable utilisée pour le build d'image de conteneur dans la registry Docker

applicationName = 'API-javaSpringboot' // Convention : utiliser le même nom que l'artifactId du pom.xml

***[...]***

}

stages {

***[...]***

stage('Bake') {

steps {

echo 'Building Docker image ...'

sh 'docker build --rm --build-arg PACKAGE\_VERSION=${package\_version} --build-arg APPLICATION\_NAME=${applicationName} -t ${dockerRegistry}:${package\_version} .'

echo 'Removing dangling Docker image ...'

sh 'docker rmi $(docker images --filter "dangling=true" -q --no-trunc) 2>/dev/null'

}

}

***[...]***

}

## Création d'une registry Docker

### Création de la registry en http

1. Exécuter un conteneur de registry Docker et le démarrer automatiquement :

$ docker run -d -p 5000:5000 -v /mnt/registry:/var/lib/registry --restart=always --name registry registry:2

*Note : cette registry stockera les images Docker dans le répertoire '/var/lib/registry' du conteneur qui sera monté sur le répertoire '/mnt/registry' de l'instance locale.*

### Création de la registry en https (requis pour Kubernetes)

1. Créer un répertoire qui contiendra les certificats de la registry Docker :

$ mkdir -p /mnt/registry/certs

1. Générer un certificat via 'openssl' :

$ openssl req -newkey rsa:4096 -nodes -sha256 -keyout /mnt/registry/certs/domain.key -x509 -days 3650 -out /mnt/registry/certs/domain.crt

Openssl demande alors des informations pour la création du certificat. Toutes peuvent être omises, sauf le *'common name'* avec un nom de domaine ou une IP stable *(pour l'IP, voir la notion de Alternate... pour éviter des erreurs de connexion par les clients par la suite)*.

Country Name (2 letter code) [XX]:  
State or Province Name (full name) []:  
Locality Name (eg, city) [Default City]:  
Organization Name (eg, company) [Default Company Ltd]:  
Organizational Unit Name (eg, section) []:  
Common Name (eg, your name or your server's hostname) []:**devops.maddoudou.click**  
Email Address []:

*La commande a alors généré les fichiers ' domain.crt' et ' domain.key' dans le répertoire /mnt/registry/certs*

1. Exécuter un conteneur de registry Docker et le démarrer automatiquement :

$ docker run -d -p 5000:5000 -v /mnt/registry:/var/lib/registry -v /mnt/registry/certs:/certs -e REGISTRY\_HTTP\_TLS\_CERTIFICATE=/certs/domain.crt -e REGISTRY\_HTTP\_TLS\_KEY=/certs/domain.key --restart=always --name devops-registry registry:2

*Note : cette registry stockera les images Docker dans le répertoire '/var/lib/registry' du conteneur qui sera monté sur le répertoire '/mnt/registry' de l'instance locale.*

1. Copier le certificat dans un répertoire "/etc/docker/certs.d/devops.maddoudou.click:5000/" (même nom que le Common Name du certificat afin de et éviter les erreurs dues au certificat auto-signé), puis redémarrer Docker :

$ mkdir -p /etc/docker/certs.d/devops.maddoudou.click:5000

$ cp /mnt/registry/certs/domain.crt /etc/docker/certs.d/devops.maddoudou.click\:5000/ca.crt

$ /etc/init.d/docker restart

1. Copier le certificat sur l'instance EC2 hébergeant **le master Kubernete et tous les worker nodes** (afin de et éviter les erreurs dues au certificat auto-signé).

*Note : veiller à avoir le certificat de l'instance EC2 hébergeant Kubernetes dans le répertoire '/var/lib/keys/' pour pouvoir faire le scp :*

$ scp -i /var/lib/keys/<certificat instance distante>.pem /mnt/registry/certs/domain.crt ubuntu@<IP instance de Kubernetes>:~/

1. Sur l'instance EC2 hébergeant **le master Kubernete et tous les worker nodes**, créer un répertoire "/etc/docker/certs.d/devops.maddoudou.click:5000/" (même nom que le Common Name du certificat afin de et éviter les erreurs dues au certificat auto-signé) et déplacer le certificat dans ce répertoire :

$ mkdir -p /etc/docker/certs.d/devops.maddoudou.click:5000

$ mv /home/ubuntu/domain.crt /etc/docker/certs.d/devops.maddoudou.click:5000/ca.crt

### Gestion des images Docker et de la registry

***Rappels des images et repositories*** *Docker :*

*Un repository est le nom de l'image sans tag (exemple :* api-javaspringboot*) son nom est forcément en minuscules Il liste toutes les images avec tags qui sont autant d'instances d'images Docker. Un tag est la création d'une nouvelle image à part entière (par exempe, le tag '*api-javaspringboot:v2.1*' fait plusieurs centaines de mégas et est autonome.*

Builder une image 'API-javaSpringboot' dans le repo 'maddoudou' en y ajoutant 2 tags : le numéro de version (v2.1) et le fait que c'est la 'latest'. Le '.' à la fin désigne le répertoire de contexte ( endroit où se trouvent les arrtefacts de build comme le dockerfile).

$ docker build -t local/api-javaspringboot:latest -t local/api-javaspringboot:v2.1 .

Ajouter un tag à une image existante (image ID = *518a41981a6a*, registry dans laquelle l'image devra être pushée : *localhost:5000*, nom de l'image : *API-javaSpringboot*)

$ docker tag 518a41981a6a localhost:5000/api-javaspringboot:latest

Pousser l'image api-javaspringboot:latest dans la registry *localhost:5000* :

$ docker push localhost:5000/api-javaspringboot:latest

Lister tous les repository de la registry (images quelques soient leur tag) de la registry :

$ curl -k -X GET https://localhost:5000/v2/\_catalog

Lister les images du repository 'api-javaspringboot' (image et l'ensemble des tags associés) de la registry :

$ curl -k -X GET https://localhost:5000/v2/api-javaspringboot/tags/list

Supprimer l'image de la registry locale où elle a été buildée :

$ docker rmi -f <ID de l'image>

Récupérer l'image 'api-javaspringboot ' depuis la registry '*localhost:5000'* dans la registry locale :

**Pas encore au pont par Docker ...**

$ docker pull localhost:5000/api-javaspringboot:latest

Supprimer le tag *'api-javaspringboot:latest'* de la registry '*localhost:5000'* avec l'ensemble de ses tags (ici la registry est montée sur l'instace faisont tournet le contenur de la registry dans '/mnt/registry ') :

$ *rm -r* /mnt/registry*/docker/registry/v2/repositories/api-javaspringboot/\_manifests/tags/latest/index/sha256/${hash}*

Vérifier quels répertoires peuvent ensuite être supprimés :

$ docker exec registry bin/registry garbage-collect --dry-run /etc/docker/registry/config.yml

Enfin, supprimer chaque répertoire :

*$ rm -r* /mnt/registry/docker/registry*/v2/repositories/api-javaspringboot/\_manifests/revisions/sha256/${hash}*

### Stage Jenkins de déploiement dans Kubernetes

1. Le serveur Kubernetes étant hébergé sur une instance Rancher séparée**, Jenkins doit exécuter les commandes Kubectl à distance** :
   * Utiliser WinSCP pour copier le certificat ‘.pem’ de la keypair AWS (utilisée pour se connecter à l’instance Rancher) dans le répertoire /var/lib/keys/ de nl’instancé hébergeant Jenkins.
   * **Sur l’instance hébergeant Jenkins** : modifier les droits et le propriétaire de ce certificat afin de l’utilisaetur Jenkins puisse l’utiliser :

$ chmod 400 /var/lib/keys/certificate.pem

$ chown jenkins: /var/lib/keys/certificate.pem

* + **Sur l’instance hébergeant Rancher (donc kubectl)** : copier le fichier de configuration de Kubernetes dans le répertoire de l’utilisateur Ubuntu :

$ mkdir /home/ubuntu/.kube

$ cp /home/root/.kube/config /home/ubuntu/.kube/

* + **Sur l’instance hébergeant Jenkins** : tester le fonctionnement d’une commande ‘kubectl’ depuis une instance Jenkins en tant qu’utilisateur Jenkins :

$ su jenkins

$ ssh -i /var/lib/keys/certificate.pem ubuntu@<kubernetesNodePrivateIP> "kubectl get nodes"

*Doit retourner la liste des nœuds Kubernetes*

1. Dans le Jenkinsfile, déclarer l’IP de l’instance hébergeant Rancher (Kubernetes) en tant que variable d'environnement, puis ajouter un stage pour le déploiement de l'image Docker :

API-javaSpringboot/**Jenkinsfile :**

***[...]***

environment {

***[...]***

kubernetesNodePrivateIP = '172.18.1.189'

***[...]***

}

stages {

***[...]***

stage('Deploy') {

steps {

echo 'Deploying Docker image in Kubernetes ...'

sh 'ssh -i /var/lib/keys/certificate.pem ubuntu@${kubernetesNodePrivateIP} "kubectl get nodes"'

}

}

***[...]***

}

### Spécifier une registry Docker

1. Sur l’instance Rancher, se logger sur Docker avec ses identifiants de DockerHub :

$ docker login

*<saisir les username et mot de passe de DockerHub>*

1. Le fait de se logger a créé un fichier *‘~/.docker/config.json’*. Créer un secret (ici nommé *‘regcred’* qui se pemettra se s’authentifier à une registry privée :

$ kubectl create secret docker-registry regcred --docker-server=<IP du serveur Jenkins hébergeant la registry Docker> --docker-username=<Dockerhub login> --docker-password=<Dockerhub password> --docker-email=<email>

*Doit retourner* ‘secret/regcred created’

*Note: pour supprimer cette commande de l’historique des commandes (vu qu’il y a un mot de passe), utiliser la commande suivante tout de suite après :*

$ history -d $((HISTCMD-1))

s

### Installation Kubernetes via conjure-up sur Ubuntu

1. Sur une instance EC2, installer conjure-up et lancer l’installation de Kubernetes :

*Note : commandes à lancer en tant qu’utilisateur ‘ubuntu’ :*

$ $ sudo snap install conjure-up --classic

$ conjure-up kubernetes

1. Dans l’interface semi-graphique qui s’affiche, sélectionner les choix suivants : *Note : commandes à lancer en tant qu’utilisateur ‘ubuntu’ :*
   * Spell selection : ‘Canonical Distribution of Kubernetes’
   * Add-on selection : aucun (appyer sur la touche ‘entrée’)
   * Choose a Cloud : ‘AWS’
   * New credentials Creations : Saisir les credentials d’un admin AWS
   * Choose a region : eu-west-1
   * Choose a controller : ‘Deploy New Self-Hosted Controller’
   * Network plugin : ‘Flannel’
   * Configure applications : pour chaque composant, ne mettre su’une seule Unit et dans les contraintes ajouter *‘cores=1 mem=1’* (apppuyer sur la touche ‘N’ pour sortir du sous-menu d’un composant.
   * Enfin, appyer sur la touche ‘N’ pour valider le déploiement.

Canonical Distribution of Kubernetes