Creation d’une toolchain provisionnant un conteneur Java dans un cluster FarGate

Table des matières

[Création de l’environnement de travail sur le poste local 2](#_Toc253733)

[Initialisation de l’environnement de travail sur le poste local 2](#_Toc253734)

[Création du code source 2](#_Toc253735)

[Création d'un projet sur GitHub 9](#_Toc253736)

[Clone sur le poste/serveur local 9](#_Toc253737)

[(Optionnel) tester le fonctionnement du code dans un Docker 11](#_Toc253738)

[Initialisation d’une instance de tests 11](#_Toc253739)

[Compilation du projet 12](#_Toc253740)

[Préparation de l’environnement Docker 12](#_Toc253741)

[Enregistrement de l'image Docker dans la registry ECS 14](#_Toc253742)

[Récupération de l'image Docker dans la registry ECS 14](#_Toc253743)

[Création du pipeline 16](#_Toc253744)

[Fichiers de build utilisés par CodeBuild 16](#_Toc253745)

[Templates CloudFormation 17](#_Toc253746)

[Déploiement du pipeline et de la registry ECR 17](#_Toc253747)

[Déploiement du cluster Fargate, son service et la TaskDefinition 26](#_Toc253748)

[Utilisation des 2 template Cloudformation pour créer l'ensemble 33](#_Toc253749)

[Instanciation des templates 33](#_Toc253750)

[Vérification du bon fonctionnement 33](#_Toc253751)

[Création d’une image Docker Custom pour CodeBuild 35](#_Toc253752)

# Introduction

L'objectif est de créer une toolchain serverless permettant de faire des tests unitaires, test OWASP et tests de vulnérabilité, tout en gardant l'historique dans Sonar si besoin.

Sonar est installé dans un filesystem hébergé par EFS. A chaque build de l'application, ce filesystem est monté par 'image Docker utilisée par CodeBuild afin d'utiliser les paramètres de Sonar et préserver l'historique. Ce filesystem peut également être monté sur une EC2 afin de modifier les configurations de Sonar et consulter l'historique des tests.

## Fonctionnement de l'ensemble

### Déploiement de la toolchain

* **Pré-requis** :
  + Le **filesystem EFS** doit déjà avoir été créé **avec le répertoire "/opt/sonar"** (cf modop dans les chapitres plus bas). Car pas de possibilité d'importer des données directement depuis S3 vers EFS via une Lambda (montage EFS impossible).
  + Le **repository** de l'image Docker de build customisée pour CodeBuild doit exister ainsi que **l'image de build** en question (cf modop dans les chapitres plus bas).
* Un seul template permet de créer toutes les ressources réseau et la toolchain. Afin de monter un volume EFS sur l'image Docker de build utilisée par CodeBuild, il faut déclarer un VPC, subnet privé et Security Group dans la configuration de CodeBuild. CodeBuild n'accepte de récupérer les sources auprès que GitHub que depuis un subnet privé via une NAT Gateway (utilisation d'une Internet Gateway impossible).
* Une fois le template déployé, le filesystem EFS doit être modifié pour être dans le même VPC et le même subnet privé que CodeBuild.

### Exécution de la toolchain

* *Note : le premier build échoue en attendant que l'utilisateur configure EFS.*
* Les sources sont récupérées via GitHub.
* L'image de build customisée est exécutée par CodeBuild et le cache Maven est récupéré depuis S3 (pas au premier lancement : cache vide).(toutes les étapes sont détaillées dans le buildspec):
  + Dans la phase de **pre-build**, le filesystem NFS est monté et Sonar est démarré en arrière plan (requière 1 minute pour se lancer).
  + Un script vérifie si l'image Docker de base de l'application existe dans son repository. A la première exécution ce n'est pas le cas : elle est donc recréée à partie du dockerfile "dockerfile\_basis" directement dans la registry locale. C'est fait en tâche de fond. Durant les builds suivant l'image sera dispo dans le repo, elle sera donc 'pullée' en tâche de fond.
  + Dans la phase de **build**, Maven compile le code, puis exécute les tests unitaires et les tests OWASP. Au premier build c'est très long car il télécharge toutes les dépendances.
  + *Les étapes de démarrage de Sonar, création (ou récupération) de l'image de base et compilation/tests de maven sont exécutés en parallèle.*
  + Une fois les tests terminés et Sonar démarré, les tests Sonar sont exécutés.
  + Dans la phase de **Post-build**, l'image Docker est buildée à partir de l'image de base (reconstruire ou téléchargée) afin de gagner du temps (les 9/10ème des layer étant déjà prêts). L'image est en plus taguée 'latest' afin d'être utilisée comme image de base pour le prochain build. Les 2 images sont alors poussées dans le Repo ECR de l'application.
  + Enfin, toutes les dépendances et plugins Maven téléchargés sont uploadés dans S3 afin d'être récupérés au prochain build et réduire largement le temps de build.

# 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:\serverlessToolchainJava »* sous Windows)
2. Créer la structure de répertoires suivantes :
   * serverlessToolchainJava
     + src
       - main
         * java

hello

* + - * test
        + java

hello

## Création du code source

Créer les 4 fichiers suivants utilisés pour le code source :

* serverlessToolchainJava/pom.xml
* serverlessToolchainJava/src/main/java/hello/Greeting.java
* serverlessToolchainJava/src/main/java/hello/GreetingController.java
* serverlessToolchainJava/src/main/java/hello/Application.java
* serverlessToolchainJava/src/test/java/hello/ApplicationTests.java

serverlessToolchainJava/**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>serverlessToolchainJava</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>

<!-- Nexus repositories configuration -->

<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>

<!-- /Nexus repositories configuration -->

<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>

<!-- Unit tests declaration -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<!-- /Unit tests declaration -->

</dependencies>

<properties>

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

<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>

<build>

<plugins>

<plugin>

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

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

</plugin>

<!-- Sonar tests -->

<plugin>

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

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

<version>3.5.0.1254</version>

</plugin>

<!-- /Sonar tests -->

<!-- OWASP Dependecy check -->

<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>

<!-- /OWASP Dependecy check -->

<!-- Unit tests declaration -->

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<configuration>

<useSystemClassLoader>false</useSystemClassLoader>

</configuration>

</plugin>

<!-- Code coverage reporting to Sonar -->

<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>

</executions>

</plugin>

<!-- /Code coverage reporting to Sonar -->

<!-- /Unit tests declaration -->

</plugins>

</build>

<!-- Nexus repositories declaration -->

<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>

<!-- /Nexus repositories declaration -->

</project>

*Test avec un pom.xml simplifié :* ***serverlessToolchainJava/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>serverlessToolchainJava</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>*

*<properties>*

*<maven.compiler.target>1.8</maven.compiler.target>*

*<maven.compiler.target>1.8</maven.compiler.target>*

*<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>*

*</properties>*

*<dependencies>*

*<dependency>*

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

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

*</dependency>*

*</dependencies>*

*<build>*

*<plugins>*

*<plugin>*

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

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

*</plugin>*

*<plugin>*

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

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

*<version>3.1.0</version>*

*<configuration>*

*<createDependencyReducedPom>false</createDependencyReducedPom>*

*</configuration>*

*<executions>*

*<execution>*

*<phase>package</phase>*

*<goals>*

*<goal>shade</goal>*

*</goals>*

*</execution>*

*</executions>*

*</plugin>*

*</plugins>*

*</build>*

*</project>*

serverlessToolchainJava/src/main/java/hello/**Greeting.java** :

package hello;

public class Greeting {

private final long id;

private final String content;

public Greeting(long id, String content) {

this.id = id;

this.content = content;

}

public long getId() {

return id;

}

public String getContent() {

return content;

}

}

serverlessToolchainJava/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));

}

}

serverlessToolchainJava/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);

}

}

serverlessToolchainJava/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/serverlessToolchainJava

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

Par exemple : serverlessToolchainJava

c) Copier le lien https du repository :

Par exemple : https://github.com/maddoudou22/serverlessToolchainJava.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:\*serverlessToolchainJava*»*).
   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/serverlessToolchainJava.git
    - $ git add \*
    - $ git commit -m "First push !"
    - $ git push --set-upstream https://github.com/maddoudou22/serverlessToolchainJava.git master

*Note : pour les push suivants : $ git push https://github.com/maddoudou22/serverlessToolchainJava.git*

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

echo off

SET nom\_Projet\_Git= serverlessToolchainJava

SET URL\_Projet\_Git=https://github.com/maddoudou22/serverlessToolchainJava.git

SET Repertoire\_Local\_Projet=C:\serverlessToolchainJava

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.*

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

## Initialisation d’une instance de tests

1. Créer un rôle qui sera endossé par l'instance EC2 pour pousser/récupérer des conteneurs dans la registry ECR d'AWS :
   1. La policy associée à ce rôle sera la suivante :

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"ecr:\*"

],

"Resource": [

"\*"

]

}

]

}

1. Provisionner une image EC2 Ubuntu dans AWS et endossant le rôle créé précédemment.
2. Changer la timezone de l’instance :

$ apt-get update

$ dpkg-reconfigure tzdata

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

1. Installer la CLI AWS (sera utilisé epour pousser/récupérer des images Docker sur la registry ECR):

$ apt-get install python2.7

$ curl -O https://bootstrap.pypa.io/get-pip.py

$ python2.7 get-pip.py

$ pip install awscli

$ aws --version

*-> Doit retourner le numéro de version de AWS CLI*

## Compilation du projet

1. Recréer l'arborescence du projet sur l'instance EC2 *(ex: ~/serverlessToolchainJava/...)*
2. Se placer dans le répertoire contenant le POM du projet (exemple : *« ~/serverlessToolchainJava »*).
3. Compiler le projet :

$ apt-get install maven

$ mvn clean package

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

$ java -jar target/serverlessToolchainJava-0.1.0-SNAPSHOT.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!"}

## Création du filesystem EFS contenant Sonar

### Provisioning du point de montage EFS

Dans la console AWS EFS, provisionner un point de montage EFS en spécifiant un VPC, subnet et un Security Group identique à l'EC2 qui sera utilisée pour installer Sonar.

### Installation manuelle de Sonar

Dans la console EC2, provisionner une instance EC2 avec une Ubuntu 18.04 (type t3.medium ou t3.large recommandée car Sonar est gourmand).

*Selon le tuto : https://www.alibabacloud.com/blog/easy-guide-to-install-sonarqube-on-ubuntu-16-04\_593923*

1. Installer de quelques packages pre-requis :

$ apt-get install -y software-properties-common nfs-common

1. Installer Sonar :

$ mkdir /opt/sonar

$ cd /opt/sonar

$ wget https://binaries.sonarsource.com/Distribution/sonarqube/sonarqube-7.3.zip

$ unzip sonarqube-7.3.zip

1. Créer l'utilisateur Sonar et lui donner les droit sur le répertoire de Sonar:

$ adduser sonar

$ chown -R sonar:sonar /opt/sonar

1. *Vérifier l'UID et le GUID de l'utilisateur Sonar : il faudra reporter les même dans le buildspec.yml afin d'éviter les problèmes de droit lors du montage avec EFS.*
2. Modifier le script d'exécution de Sonar pour que Sonar soit lancé via l'utilisateur 'Sonar' (problèmes d'exécution avec 'root).

$ vi /opt/sonar/sonarqube-7.6/bin/linux-x86-64/sonar.sh

*- Décommenter la ligne "#RUN\_AS\_USER" pour la remplacer par "RUN\_AS\_USER=sonar"*

1. Démarrer Sonar et vérifier qu'il se lance bien en se connectant à la console de Sonar : (http://<IP publique instance EC2>:9000)

$ /opt/sonar/sonarqube-7.6/bin/linux-x86-64/sonar.sh start

## Préparation de l’environnement Docker

1. Installer Docker :

$ apt-get install docker.io

1. Créer un répertoire de déploiement (exemple : « ~/docker-tests »).
2. Copier l'archive JAR dans ce nouveau répertoire docker-test et s'y placer

$ cp target/serverlessToolchainJava-0.1.0-SNAPSHOT.jar ~/docker-test

$ cd ~/docker-test

1. Se placer dans ce répertoire et créer le dockerfile *"dockerfile-exec-jar"* avec le contenu suivant :

FROM java:8

WORKDIR /root/docker-tests

ADD serverlessToolchainJava-0.1.0-SNAPSHOT.jar serverlessToolchainJava-0.1.0-SNAPSHOT.jar

EXPOSE 8080

CMD java -jar serverlessToolchainJava-0.1.0-SNAPSHOT.jar

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

1. Créer l’image Docker :

$ docker build -f dockerfile-exec-jar -t serverlesstoolchainjava /root/docker-tests/

1. Vérifier la bonne création de l'image :

$ docker images

Doit retourner la liste des images, dont l’une doit être « serverlesstoolchainjava ».

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

$ docker run -d -p 8088:8080 serverlesstoolchainjava

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>*

## Enregistrement de l'image Docker dans la registry ECS

1. Vérifier que l'instance EC2 à partir de laquelle on pousse l'image endosse un rôle lui permettant de pousser des images dans ECR.
2. Faire une demande de login auprès d'ECR via la commande suivante :

$ aws ecr get-login --no-include-email

Cette commande renvoie une commande de loggin. Faire un copier/coller et l'exécuter (commande de type *"docker login -u AWS -p password https://aws\_account\_id.dkr.ecr.region.amazonaws.com")*

*Essayer tout simplement avec la commande suivante :*

$(aws ecr get-login --no-include-email --region $AWS\_DEFAULT\_REGION)

1. Dans la console AWS, créer un repository en spécifiant ' [serverlesstoolchainjava](https://eu-west-1.console.aws.amazon.com/ecr/repositories/serverlesstoolchainjava/?region=eu-west-1)' comme nom de repository. Récupérer l'URI du repository *(<Account ID>.dkr.ecr.<region>.amazonaws.com/serverlesstoolchainjava*)
2. Récupérer l'ID de l'image Docker buildée précédemment avec la commande 'docker images'

$ docker images

1. Ajouter un tag à une image existante (image ID = 518a41981a6a, dans la future registry registry ECR, nom de l'image = serverlesstoolchainjava, tag = 0.1.0)

$ docker tag 518a41981a6a <AccountID>.dkr.ecr.<region>.amazonaws.com/serverlesstoolchainjava:0.1.0

1. Pousser l'image du conteneur créé dans le repository ECR :

$ docker push <AccountID>.dkr.ecr.<region>.amazonaws.com/serverlesstoolchainjava:0.1.0

1. hj

## Récupération de l'image Docker dans la registry ECS

1. Répéter les étapes 1) et 2) du chapitre "Enregistrement de l'image Docker dans la registry ECS" pour se logger auprès d'ECR.
2. Récupérer l'image serverlesstoolchainjava avec le tag 0.1.0 depuis le repository ECR :

$ docker pull <AccountID>.dkr.ecr.<region>.amazonaws.com/serverlesstoolchainjava:0.1.0

# Création du pipeline

## Fichiers de build utilisés par CodeBuild

1. A la racine du répertoire du projet, créer le fichier "buildspec.yml" qui sera utilisé par CodeBuild pour exécuter les commande requises pour la compilation, la création du conteneur et son enregistrement dans la registry ECR (CodeBuild lui-même déploie un conteneur puis exécute les commandes spécifiées dans ce fichier).

Ce fichier ne contient que des variables d'environnement transmises par CodeBuild : il n'a jamais besoin d'être modifié quelque soit la version du projet utilisée ou même l'instance de projet.

serverlessToolchainJava/**buildspec.yml** :

version: 0.2

phases:

pre\_build:

commands:

- echo Pre-Build started on `date`

- apt-get update -y

- apt-get install docker.io -y

- docker -v

- echo Logging in to Amazon ECR...

- $(aws ecr get-login --no-include-email --region $AWS\_DEFAULT\_REGION)

build:

commands:

- echo Build started on `date`

- pip install --upgrade awscli

- mvn clean package

- ls .

- ls target/

- echo Get project name and version from the pom.xml...

- apt-get install libxml2-utils -y

- PACKAGE\_VERSION=$(xmllint --xpath "//\*[local-name()='project']/\*[local-name()='version']/text()" pom.xml)

- APPLICATION\_NAME=$(xmllint --xpath "//\*[local-name()='project']/\*[local-name()='artifactId']/text()" pom.xml)

- cp target/$APPLICATION\_NAME-$PACKAGE\_VERSION.jar .

post\_build:

commands:

- echo Post-Build started on `date`

- echo Building the Docker image...

- docker build --build-arg PACKAGE\_VERSION=$PACKAGE\_VERSION --build-arg APPLICATION\_NAME=$APPLICATION\_NAME -t $IMAGE\_REPO\_NAME:$PACKAGE\_VERSION .

- docker tag $IMAGE\_REPO\_NAME:$PACKAGE\_VERSION $AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/$IMAGE\_REPO\_NAME:$PACKAGE\_VERSION

- echo Pushing the Docker image...

- docker push $AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/$IMAGE\_REPO\_NAME:$PACKAGE\_VERSION

- echo Writing image definitions file...

- printf '[{"name":"%s","imageUri":"%s"}]' $IMAGE\_REPO\_NAME $AWS\_ACCOUNT\_ID.dkr.ecr.$AWS\_REGION.amazonaws.com/$IMAGE\_REPO\_NAME:$PACKAGE\_VERSION > imagedefinitions.json

artifacts:

files: imagedefinitions.json

1. A la racine du répertoire du projet, créer le fichier "dockerfile" qui spécifie les modalité de construction du conteneur Docker.

**Attention : c'est dans ce fichier que le port d'exposition du conteneur est spécifié. Il doit être identique au numéro de port spécifié dans la TaskDefinition utilisée par le service du Cluster Fargate** (cf chapitres suivants).

serverlessToolchainJava/**dockerfile** :

FROM java:8

ARG PACKAGE\_VERSION

ARG APPLICATION\_NAME

ENV PACKAGE\_VERSION ${PACKAGE\_VERSION}

ENV APPLICATION\_NAME ${APPLICATION\_NAME}

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

EXPOSE **8880**

CMD ["sh", "-c", "java -jar ${APPLICATION\_NAME}-${PACKAGE\_VERSION}.jar"]

## Templates CloudFormation

### Déploiement du pipeline et de la registry ECR

1. A la racine du répertoire du projet, créer le template CloudFormation "Java-Fargate-Toolchain - Pipeline template.yml" qui déploiera les ressources CodePipeline, CodeBuild et la registry ECR.

**Attention : les variables "**ECRRepositoryName**", "**FargateCluster" et "ServiceName" **doivent avoir les même valeurs que dans le template CloudFormation** "Java-Fargate-Toolchain - FargateCluster template.yml"(cf chapitres suivants).

Le détail des variables de ce fichier est décrit dans le fichier lui-même.

serverlessToolchainJava/**Java-Fargate-Toolchain - Pipeline template.yml** :

AWSTemplateFormatVersion: '2010-09-09'

Description: creation automatisee d'un pipeline DevOps avec un repository ECR et un cluster Fargate

# Note pour la suppression de la stack :

# - Le contenu du bucket doit être supprimé manuellement. CloudFormation ne supprime pas de buckets dont le contenu n'est pas vide.

# - Le contenu du Repository ECR doit être supprimé manuellement. CloudFormation ne supprime pas de Repository ECR dont le contenu n'est pas vide.

# - Dans Code Build, il vaut mieux décocher la case "Allow AWS CodeBuild to modify this service role so it can be used with this build project"

# qui est cochée par Défaut, ça évite des problèmes de modification/décommissionnement de la stack. CloudFormation ne permet pas de décocher cette case lors d'un déploiement

# Il est donc nécessaire de supprimer la policy créée pour CodeBuild manuellement sinon le rôle ne pourra pas être supprimé.

Parameters:

ProjectName:

Description: Nom du projet

Type: String

Default: "Java-Fargate-Toolchain-CFN"

AllowedPattern: '[A-Za-z0-9-]+'

ECRRepositoryName:

Description: Nom du repository cree dans ECR hebergeant les images Docker generee durant l'etape de Post-Build de CodeBuild

Type: String

Default: "javafargatetoolchain"

AllowedPattern: '[a-z]+'

FargateCluster:

Description: Nom du cluster Fargate sur lequel les conteneurs de l'application s'executeront

Type: String

Default: "fargate-cluster-CFN"

AllowedPattern: '[A-Za-z0-9-]+'

ServiceName:

Description: nom du service utilise pour le deploiement des conteneurs

Type: String

Default: "fargate-service-CFN"

AllowedPattern: '[A-Za-z0-9-]+'

bucketS3:

Description: "Bucket contenant les artefacts de l'application pour leur integration et leur deploiement (codes source et imagedefinitions des conteneurs requis)."

Type: 'String'

Default: "javafargatetoolchain"

AllowedPattern: '[a-z0-9]+'

EncryptionKeyARN:

Description: ARN de la cle CMK KMS utilisee pour chiffrer le bucket S3 (imperatif pour l'utilisation avec CodeBuild). Si la cle par defaut "arn:aws:kms:eu-west-1:962109799108:alias/aws/s3" n'existe pas, elle sera creee automatiquement par S3.

Type: String

Default: "arn:aws:kms:eu-west-1:962109799108:alias/aws/s3"

GitHubUser:

Description: Nom d'utilisateur GitHub.

Type: String

Default: "maddoudou22"

GitHubRepository:

Description: Nom du projet declare dans GitHub.

Type: String

Default: "serverlessToolchainJava"

GitHubBranch:

Description: Branch utilisee dans GitHub.

Type: String

Default: "master"

GitHubOAuthToken:

Description: Token Oauth de GitHub. (A generer et recuperer depuis GitHub dans Settings -> Developer Settings -> Personnal Access Tokens)

Type: String

Default: ""

Resources:

#------------------------------------------------------------------------------------------------------------

#- Creation du Bucket S3 hebergeant les artefacts compiles et imagedefinitions.json -------------------------

#------------------------------------------------------------------------------------------------------------

# Creation du bucket lui-meme :

ArtefactsBucket:

Type: AWS::S3::Bucket

Properties:

BucketName: !Ref bucketS3

AccessControl: Private

# Policy associee au bucket :

S3BucketPolicy:

Type: AWS::S3::BucketPolicy

#Si la Policy est créée avant le bucket la création de la stack échoue :

DependsOn: ArtefactsBucket

Properties:

Bucket: !Ref bucketS3

PolicyDocument:

Statement:

-

Action:

- s3:\*

Effect: Allow

Resource:

- !Sub arn:aws:s3:::${bucketS3}

- !Sub arn:aws:s3:::${bucketS3}/\*

Principal:

Service:

- codepipeline.amazonaws.com

- codebuild.amazonaws.com

#------------------------------------------------------------------------------------------------------------

#- Creation du repository ECR hebergeant les images Docker generees par CodeBuild ---------------------------

#------------------------------------------------------------------------------------------------------------

ECRrepository:

Type: AWS::ECR::Repository

Properties:

RepositoryName: !Ref ECRRepositoryName

#------------------------------------------------------------------------------------------------------------

#- CodeBuild ------------------------------------------------------------------------------------------------

#------------------------------------------------------------------------------------------------------------

# Role endosse par CodeBuild pour la creation du projet Codebuild :

BuildProjectRole:

Type: "AWS::IAM::Role"

Properties:

RoleName: !Sub ${ProjectName}-CodeBuildRole

AssumeRolePolicyDocument:

Version: "2012-10-17"

Statement:

#

- Effect: "Allow"

Principal:

Service:

- codebuild.amazonaws.com

Action:

- "sts:AssumeRole"

Path: "/"

Policies:

-

PolicyName: !Sub ${ProjectName}-CodeBuildPolicy

PolicyDocument:

Version: "2012-10-17"

Statement:

-

Effect: Allow

Action:

- s3:PutObject

- s3:GetBucketPolicy

- s3:GetObject

- s3:GetObjectVersion

- s3:ListBucket

Resource:

- !Join ['',['arn:aws:s3:::',!Ref bucketS3, '/\*']]

- !Join ['',['arn:aws:s3:::',!Ref bucketS3]]

-

Effect: Allow

Action:

- kms:\*

Resource: !Ref EncryptionKeyARN

-

Effect: Allow

Action:

- logs:CreateLogGroup

- logs:CreateLogStream

- logs:PutLogEvents

Resource: !Sub "arn:aws:logs:${AWS::Region}:${AWS::AccountId}:log-group:/aws/codebuild/\*"

-

Effect: Allow

Action:

- ecr:GetAuthorizationToken

Resource: "\*"

-

Effect: Allow

Action:

- ecr:BatchCheckLayerAvailability

- ecr:CompleteLayerUpload

- ecr:InitiateLayerUpload

- ecr:PutImage

- ecr:UploadLayerPart

- ecr:GetDownloadUrlForLayer

- ecr:BatchGetImage

- ecr:DescribeImages

- ecr:BatchDeleteImage

- ecr:GetRepositoryPolicy

Resource: !GetAtt ECRrepository.Arn

# Projet Codebuild :

BuildProject:

Type: AWS::CodeBuild::Project

Properties:

Name: !Sub ${ProjectName}-CodeBuild

Description: !Sub Projet CodeBuild cree automatiquement pour le pipeline ${ProjectName}-pipeline

EncryptionKey: !Ref EncryptionKeyARN

ServiceRole: !GetAtt BuildProjectRole.Arn

Artifacts:

Type: CODEPIPELINE

Environment:

Type: linuxContainer

ComputeType: BUILD\_GENERAL1\_SMALL

Image: aws/codebuild/java:openjdk-8

PrivilegedMode: true # Obligatoire pour l'utilisation des commandes Docker

EnvironmentVariables:

- Name: IMAGE\_REPO\_NAME

Value: !Ref ECRRepositoryName

- Name: S3\_KEY

Value: !Ref EncryptionKeyARN

- Name: AWS\_ACCOUNT\_ID

Value: !Ref "AWS::AccountId"

Source:

Type: CODEPIPELINE

TimeoutInMinutes: 10

Tags:

- Key: Name

Value: !Ref ProjectName

#------------------------------------------------------------------------------------------------------------

#- CodePipeline ---------------------------------------------------------------------------------------------

#------------------------------------------------------------------------------------------------------------

# Role endosse par CodePipeline pour la creation du pipeline :

PipelineRole:

DependsOn: S3BucketPolicy

Type: AWS::IAM::Role

Properties:

RoleName: !Sub ${ProjectName}-codepipeline-role

AssumeRolePolicyDocument:

Version: 2012-10-17

Statement:

-

Effect: Allow

Principal:

Service:

- codepipeline.amazonaws.com

Action:

- sts:AssumeRole

Path: /

Policies:

-

PolicyName: !Sub ${ProjectName}-CodePipelinePolicy

PolicyDocument:

Version: "2012-10-17"

Statement:

-

Effect: Allow

Action:

- codepipeline:\*

- iam:ListRoles

- iam:PassRole

- codecommit:GetCommit

- codecommit:UploadArchive

- codebuild:BatchGetBuilds

- codebuild:StartBuild

Resource:

- "\*"

-

Effect: Allow

Action:

- ecs:DescribeServices

- ecs:DescribeTaskDefinition

- ecs:DescribeTasks

- ecs:ListTasks

- ecs:RegisterTaskDefinition

- ecs:UpdateService

Resource:

- "\*"

-

Effect: Allow

Action:

- kms:Decrypt

Resource: !Ref EncryptionKeyARN

-

Effect: Allow

Action:

- s3:PutObject

- s3:GetBucketPolicy

- s3:GetObject

- s3:GetObjectVersion

- s3:ListBucket

Resource:

- !Join ['',['arn:aws:s3:::',!Ref bucketS3, '/\*']]

- !Join ['',['arn:aws:s3:::',!Ref bucketS3]]

# Pipeline

Pipeline:

Type: AWS::CodePipeline::Pipeline

Properties:

RoleArn: !GetAtt PipelineRole.Arn

Name: !Sub ${ProjectName}-pipeline #!Ref AWS::StackName

Stages:

-

Name: Source

Actions:

-

Name: GitHub

ActionTypeId:

Category: Source

Owner: ThirdParty

Version: 1

Provider: GitHub

Configuration:

Owner: !Ref GitHubUser

Repo: !Ref GitHubRepository

Branch: !Ref GitHubBranch

OAuthToken: !Ref GitHubOAuthToken

OutputArtifacts:

- Name: SCCheckoutArtifact

RunOrder: 1

-

Name: Build

Actions:

-

Name: Build

ActionTypeId:

Category: Build

Owner: AWS

Version: 1

Provider: CodeBuild

Configuration:

ProjectName: !Ref BuildProject

RunOrder: 1

InputArtifacts:

- Name: SCCheckoutArtifact

OutputArtifacts:

- Name: BuildOutput

-

Name: Staging #DeployToECS

Actions:

- Name: !Sub ${ProjectName}-Stack

ActionTypeId:

Category: Deploy

Owner: AWS

Version: 1

Provider: ECS

Configuration:

ClusterName: !Ref FargateCluster

ServiceName: !Ref ServiceName

FileName: imagedefinitions.json

InputArtifacts:

- Name: BuildOutput

RunOrder: 1

ArtifactStore:

Type: S3

Location: !Ref bucketS3

EncryptionKey:

Id: !Ref EncryptionKeyARN

Type: KMS

### Déploiement du cluster Fargate, son service et la TaskDefinition

1. A la racine du répertoire du projet, créer le template CloudFormation "Java-Fargate-Toolchain - FargateCluster template.yml" qui déploiera les ressources Cluster Fargate, TaskDefinition et Service ECS.

**Attention : les variables "**ECRRepositoryName**", "**FargateCluster" et "ServiceName" **doivent avoir les même valeurs que dans le template CloudFormation** "Java-Fargate-Toolchain - Pipeline template.yml"(cf chapitre précédent).

Le détail des variables de ce fichier est décrit dans le fichier lui-même.

serverlessToolchainJava/**Java-Fargate-Toolchain - FargateCluster template.yml** :

AWSTemplateFormatVersion: '2010-09-09'

Description: creation automatisee d'un cluster Fargate avec une Task Definition et un service

# Le cluster est cree avec une Task Definition allouant 0,25vCPU et 0,5GB de RAM au conteneur pour (hard-code dans le template pour simplifier

# Note pour la suppression de la stack :

# -

Parameters:

ECRRepositoryName:

Description: Nom du repository cree dans ECR hebergeant les images Docker a deployer

Type: String

Default: "javafargatetoolchain"

AllowedPattern: '[a-z]+'

FargateCluster:

Description: Nom du cluster Fargate sur lequel les conteneurs de l'application s'executeront

Type: String

Default: "fargate-cluster-CFN"

AllowedPattern: '[A-Za-z0-9-]+'

ServiceName:

Description: nom du service utilise pour le deploiement des conteneurs

Type: String

Default: "fargate-service-CFN"

AllowedPattern: '[A-Za-z0-9-]+'

VPC:

Description: VPC dans lequel le Security Group alloue au conteneurs sera cree

Type: AWS::EC2::VPC::Id

Subnet:

Description: Subnet dans lequel le Service execute les conteneurs (doit etre dans le VPC specife ci-dessous)

Type: AWS::EC2::Subnet::Id

# Il est imperatif de specifier une image Docker lors de la creation de la Task Definition. Cependant l'image

# souhaitee n'aura pas encore ete buildee par CodePipeline. En attendant on specifie une image bidon.

# ContainerImage:

# Description: Image utilisee uniquement pour la creation de la TaskDefinition. Sera ecrasee au premier build de l'application par CodeBuild.

# Type: String

# Default: nginx

ContainerPort:

Description: port appele par les clients pour acceder au service

Type: Number

Default: 8080

ContainerCPU:

Description: vCPU alloues au conteneur - 256=0,25vCPU - (attention - verifier qua la valeur soit compatible avec la RAM selectionnee)

Type: Number

Default: 256

AllowedValues:

- 256

- 512

- 1024

- 2048

- 4096

# 256 (.25 vCPU) - Available memory values: 0.5GB, 1GB, 2GB

# 512 (.5 vCPU) - Available memory values: 1GB, 2GB, 3GB, 4GB

# 1024 (1 vCPU) - Available memory values: 2GB, 3GB, 4GB, 5GB, 6GB, 7GB, 8GB

# 2048 (2 vCPU) - Available memory values: Between 4GB and 16GB in 1GB increments

# 4096 (4 vCPU) - Available memory values: Between 8GB and 30GB in 1GB increments

ContainerRAM:

Description: RAM allouee au conteneur - 512=0,5GB - (attention - verifier qua la valeur soit compatible avec le selectionne)

Type: Number

Default: 512

AllowedValues:

- 512

- 1024

- 2048

- 4096

- 8192

# 0.5GB, 1GB, 2GB - Available cpu values: 256 (.25 vCPU)

# 1GB, 2GB, 3GB, 4GB - Available cpu values: 512 (.5 vCPU)

# 2GB, 3GB, 4GB, 5GB, 6GB, 7GB, 8GB - Available cpu values: 1024 (1 vCPU)

# Between 4GB and 16GB in 1GB increments - Available cpu values: 2048 (2 vCPU)

# Between 8GB and 30GB in 1GB increments - Available cpu values: 4096 (4 vCPU)

Resources:

#------------------------------------------------------------------------------------------------------------

#- Creation des elements de securite ------------------------------------------------------------------------

#------------------------------------------------------------------------------------------------------------

# ExecutionRole endosse par ECS

# A role needed by ECS.

# Role that containers in this task can assume. All containers in this task are granted the permissions that are specified in this role.

# There is an optional task execution IAM role that you can specify with Fargate to allow your Fargate tasks to make API calls to Amazon ECR.

ExecutionRole:

Type: AWS::IAM::Role

Properties:

RoleName: !Sub ${ServiceName}-ExecutionRole

AssumeRolePolicyDocument:

Statement:

- Effect: Allow

Principal:

Service: ecs-tasks.amazonaws.com

Action: 'sts:AssumeRole'

ManagedPolicyArns:

- 'arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy'

# TaskRole endosse par ECS

# Role that grants containers in the task permission to call AWS APIs on your behalf.

TaskRole:

Type: AWS::IAM::Role

Properties:

RoleName: !Sub ${ServiceName}-TaskRole

AssumeRolePolicyDocument:

Statement:

- Effect: Allow

Principal:

Service: ecs-tasks.amazonaws.com

Action: 'sts:AssumeRole'

# Security Group associe au conteneur

ContainerSecurityGroup:

Type: AWS::EC2::SecurityGroup

Properties:

GroupDescription: !Join ['', [!Ref ServiceName, ContainerSecurityGroup]]

VpcId: !Ref VPC

SecurityGroupIngress:

- IpProtocol: tcp

FromPort: !Ref ContainerPort

ToPort: !Ref ContainerPort

CidrIp: 0.0.0.0/0

#------------------------------------------------------------------------------------------------------------

#- Creation du Cluster --------------------------------------------------------------------------------------

#------------------------------------------------------------------------------------------------------------

Cluster:

Type: AWS::ECS::Cluster

Properties:

ClusterName: !Ref FargateCluster

#------------------------------------------------------------------------------------------------------------

#- Creation de la Task Definition ---------------------------------------------------------------------------

#------------------------------------------------------------------------------------------------------------

LogGroup:

Type: AWS::Logs::LogGroup

Properties:

LogGroupName: !Join ['', [/ecs/, !Ref ServiceName, TaskDefinition]]

TaskDefinition:

Type: AWS::ECS::TaskDefinition

# S'assurer que le log group est cree avant la TaskDefinition

DependsOn: LogGroup

Properties:

# Nom de la task definition

Family: !Sub ${ServiceName}-TaskDefinition

NetworkMode: awsvpc # Obligatoire pour Fargate

RequiresCompatibilities:

- FARGATE

Cpu: !Ref ContainerCPU

Memory: !Ref ContainerRAM

ExecutionRoleArn: !Ref ExecutionRole

TaskRoleArn: !Ref TaskRole

ContainerDefinitions:

- Name: !Ref ECRRepositoryName

Image: !Ref ECRRepositoryName

PortMappings:

- ContainerPort: !Ref ContainerPort

# Send logs to CloudWatch Logs

LogConfiguration:

LogDriver: awslogs

Options:

awslogs-region: !Ref AWS::Region

awslogs-group: !Ref LogGroup

awslogs-stream-prefix: ecs

#------------------------------------------------------------------------------------------------------------

#- Creation du service --------------------------------------------------------------------------------------

#------------------------------------------------------------------------------------------------------------

Service:

Type: AWS::ECS::Service

Properties:

ServiceName: !Ref ServiceName

Cluster: !Ref FargateCluster

TaskDefinition: !Ref TaskDefinition

DeploymentConfiguration:

MinimumHealthyPercent: 100

MaximumPercent: 200

DesiredCount: 1

LaunchType: FARGATE

NetworkConfiguration:

AwsvpcConfiguration:

AssignPublicIp: ENABLED

Subnets:

- !Ref Subnet

SecurityGroups:

- !Ref ContainerSecurityGroup

#------------------------------------------------------------------------------------------------------------

#- Outputs --------------------------------------------------------------------------------------------------

#------------------------------------------------------------------------------------------------------------

#Outputs:

# Endpoint:

# Description: Endpoint

# Value: !Join ['', ['https://', !Ref DNSRecord]]

# Utilisation des 2 template Cloudformation pour créer l'ensemble

Il y a une dépendance circulaire entre les templates *"Java-Fargate-Toolchain - FargateCluster template.yml"* et *"Java-Fargate-Toolchain - Pipeline template.yml"* :

* Le stage "Deploy" du template *"Java-Fargate-Toolchain - Pipeline template.yml"* a besoin du cluster Fargate et du Service pour déployer le conteneur créé par CodeBuild et poussé dans la Registry.
* Dans le template *"Java-Fargate-Toolchain - FargateCluster template.yml"*, la TaskDefinition doit connaître le conteneur Docker à utiliser 'qui ne sera dispo que lors de la première exécution du pipeline.
* Le Service doit être associé à la TaskDefinition pour fonctionner.
* Si le template *"Java-Fargate-Toolchain - FargateCluster template.yml"* est déployé alors que le conteneur n'existe pas, la construction du Service tourne reste constamment en 'Creation in progress' jusqu'à ce que le conteneur soit disponible et que le service puisse exécuter une Task avec succès.

## Instanciation des templates

**Les étapes de déploiement seront donc les suivantes** :

1. Exécution du template *"Java-Fargate-Toolchain - Pipeline template.yml"* via CloudFormation.
   * Le pipeline sera entièrement créé avec succès et le projet sera buildé automatiquement.
   * La phase de déploiement échouera car aucun cluster Fargate n'existera (pas important).
2. Exécution du template *"Java-Fargate-Toolchain - FargateCluster template.yml"* via CloudFormation.
   * Le cluster et la TaskDefinition seront entièrement créés.
   * Le Service sera lui aussi entièrement créé, mais CloudFormation le maintiendra en 'Creation in progress' jusqu'à ce que le service ait exécuté une Task avec succès.
3. Exécution d'une 'Release Change' dans CodePipeline (ou via GitHub) afin de rebuilder le projet.
   * Cette fois, le déploiement se déroulera jusqu'au bout car les ressources TaskDefinition, Cluster et Service seront opérationnelles : une Task sera exécutée.
   * Dans CloudFormation, le service sera finalisé une fois la Task exécutée avec succès. Pas sûr... Néanmoins, l'ensemble fonctionne quand même.

## Vérification du bon fonctionnement

1. Dans la console ECS, aller dans 'Clusters'. Sélectionner le cluster créée, puis sélectionné le service créé. Cliquer sur l'onglet 'Tasks', puis sélectionner la task créée. La Task doit être affichée comme 'RUNNING'. Noter l'adresse IP "Public IP".
2. Dans Postman, exécuter une requête de type POST pointant sur l'URL "http://<Public IP>:8080/greeting".
3. La requête doit retourner le résultat suivant :

{

"id": 1,

"content": "Salut, World!"

}

*Note : il faut parfois du temps entre le moment où la Task est affichée comme 'RUNNING' et le moment où le service est accessible depuis l'extérieur.*

# Création d’une image Docker Custom pour CodeBuild

Il y a une

# docker run -it ubuntu:18:04  
  
  
0) Créer le repo suivant dans ECR : custom-java-8  
1) Créer le fichier dockerfile suivant :  
  
  
FROM ubuntu:18:04  
  
RUN apt-get update  
RUN apt-get install -y tzdata  
RUN ln -fs /usr/share/zoneinfo/Europe/Paris /etc/localtime  
RUN dpkg-reconfigure --frontend noninteractive tzdata  
RUN apt-get install apt-transport-https ca-certificates curl   
software-properties-common sudo -y  
RUN curl -fsSL <https://download.docker.com/linux/ubuntu/gpg> | sudo   
apt-key add -  
RUN add-apt-repository "deb [arch=amd64]   
<https://download.docker.com/linux/ubuntu> bionic stable"  
RUN apt-get update  
RUN apt-cache policy docker-ce  
RUN apt-get install docker-ce -y  
RUN apt-get install openjdk-8-jdk -y  
RUN echo "JAVA\_HOME=\"/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java\""   
 >> /etc/environment  
RUN apt-get install maven -y  
RUN apt-get install python2.7 -y  
RUN curl -O <https://bootstrap.pypa.io/get-pip.py>  
RUN python2.7 get-pip.py  
RUN pip install awscli  
RUN apt-get install libxml2-utils -y  
  
  
  
FROM from\_aws:0.1  
  
RUN apt-get update  
RUN ln -fs /usr/share/zoneinfo/Europe/Paris /etc/localtime  
RUN dpkg-reconfigure --frontend noninteractive tzdata  
  
  
  
2) Exécuter les commandes suivantes (dans le même répertoire que le   
dockerfile) pour créer le conteneur et le pousser dans ECR :  
# docker build -t custom-java-8:0.1 .  
# docker tag custom-java-8:0.1   
962109799108.dkr.ecr.eu-west-1.amazonaws.com/custom-java-8:0.1  
# $(aws ecr get-login --no-include-email --region eu-west-1)  
# docker push 962109799108.dkr.ecr.eu-west-1.amazonaws.com/custom-java-8:0.1  
  
  
3) Créer la permission suivante dans ECR :  
{  
   "Version": "2008-10-17",  
   "Statement": [  
     {  
       "Sid": "CodeBuildAccess",  
       "Effect": "Allow",  
       "Principal": {  
         "Service": "codebuild.amazonaws.com"  
       },  
       "Action": [  
         "ecr:BatchCheckLayerAvailability",  
         "ecr:BatchGetImage",  
         "ecr:GetDownloadUrlForLayer"  
       ]  
     }  
   ]  
}  
  
  
Note : l'instance EC2 sur lesquelles ces commandes sont exécutrées   
doivent avoir un rôle custom 'ECR-fullAccess-Role'