

2019

Jeu du Takenoko

Yacine Lotfi, Chloé Maccarinelli, Pierre Marion, Alexandre Trujillo

Projet de développement

Master 1 Informatique S2



Table des matières

[0] Introduction 2](#_Toc8559781)

[I] Le découpage 2](#_Toc8559782)

[1/ contraintes de couplage 2](#_Toc8559783)

[2/ choix des packages initiaux 2](#_Toc8559784)

[II] Echanges Serveur - Joueur 4](#_Toc8559785)

[1/ Les routes implémentées sous forme de web service REST 4](#_Toc8559786)

[2/ lancement de partie 7](#_Toc8559787)

[III] Tests d’intégration 9](#_Toc8559788)

[1/ les tests avec Cucumber 9](#_Toc8559789)

[2/ l’intégration avec Travis 11](#_Toc8559790)

[IV] Conteneurisation avec Docker 12](#_Toc8559791)

[1/conteneurisation 12](#_Toc8559792)

[2/ intégration avec Travis 15](#_Toc8559793)

[V] Conclusion 17](#_Toc8559794)

# I] Introduction

L’objectif est de poursuivre l’apprentissage du Génie Logiciel commencé au 1er semestre, en reprenant notre version la plus stable du Takenoko.

Dans cette partie, nous devons nous baser sur une architecture REST et porter notre application sur des Web Services.

Pour cela nous avons respecté les différents points suivants :

* Séparer le joueur et le moteur de jeu (ici serveur), c’est-à-dire qu’il n’y a plus aucuns liens physiques entre eux (partie II])
* Implémenter des routes REST pour la communication serveur-joueur (partie III])
* Effectuer des tests d’intégration, avec Cucumber, et donc tester notre application à un niveau utilisateur contrairement aux tests unitaires JUnit vus au 1er semestre (partie IV])
* Permettre une intégration continue avec Travis CI (partie IV])
* Permettre un déploiement de notre application sous Docker (partie V])

# II] Le découpage

## 1/ contraintes de couplage

Au début de notre réflexion nous voulions créer seulement deux sous-projets maven « serveur » et « joueur ». Or les deux projets étaient fortement couplés. Ainsi nous avons décidé de créer le projet « commun » de manière à totalement dissocier « serveur » et « joueur ».

## 2/ choix des packages initiaux



Figure 2 : Architecture 1

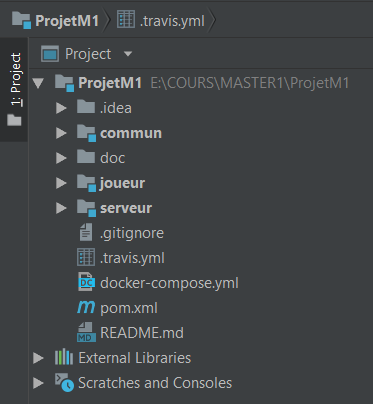
Avant le découpage nous avions l’architecture de la *Figure 2 : Architecture 1* ci-dessus. Après réflexion nous avons répertorié toutes les classes et packages qui étaient dépendant les uns des autres, ainsi nous avons pu déjà mettre en place un sous-projet « commun » qui ne contient pas de classe Application.java.

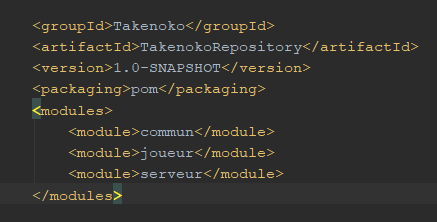
Pour les autres sous-projets, « serveur » et « joueur », nous avons décidé de les implémenter tous les deux comme des serveurs. Ainsi la communication entre le joueur et le serveur peut s’effectuer dans les deux sens comme décrit dans la partie I].

Au niveau des classes, le joueur ne contient que la classe IAPanda (le joueur), et les configurations nécessaires à sa fonction de serveur.

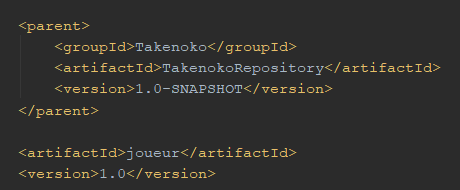
Le serveur quant à lui contient toutes les autres classes relatives au moteur de jeu, et les configurations nécessaires à sa fonction de serveur.

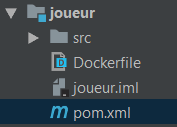
Le « serveur », « joueur » et « commun » sont trois sous projets de notre projet mère ProjetM1 comme ci-dessous.

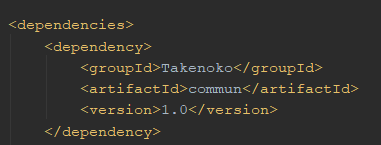


Pour que les sous-projets soient implémentés comme des projets maven à part entière on doit leur ajouter leur propre pom.xml et configurer celui du projet mère de manière qu’il reconnaisse ces modules comme lui appartenant (voir l’image ci-dessous).

Ci-dessous un exemple d’implémentation du pom.xml de notre sous-projet « joueur » (même méthode pour serveur et commun).







Comme pour le projet-mère, qui doit connaître ses sous-projets, on doit ici préciser le projet mère mais également le projet « commun » puisque qu’il est le projet contenant les classes communes à « serveur » et « joueur » : « serveur » et « joueur » sont bien dissociés.

# III] Echanges Serveur - Joueur

## 1/ Les routes implémentées sous forme de web service REST

La communication joueur serveur est implémentée selon la figure ci-dessous :

Une image contenant texte, carte

Description générée avec un niveau de confiance très élevé

Figure 1 : échanges serveur-joueur

SERVEUR

* Etapes 1 et 2 de la Figure 2 : échanges serveur-joueur :

Tout d’abord, nous permettons la connexion du joueur au serveur via l’url :

{ipHostServeur}:8080/{Pseudo}/connect

Celle-ci renvoie un tableau de deux entiers :

* le 1er: l’identifiant de la partie
* le 2ème: l’identifiant du joueur voulant entrer dans la partie
* Etape 5 de la Figure 3 : échanges serveur-joueur :

Les informations du jeu sont obtenues via l’url :

{ipHostServeur}:8080/{identifiantPartie}/{OPERATION}

* **Opérations en GET :**

|  |  |
| --- | --- |
| **Opérations** | **Actions** |
| GetZoneJouee | Retourne la liste de zones jouées |
| Piocher | Retourne la liste des parcelles piochées |
| PiocheParcelleIsEmpty | Retourne la pioche parcelle vide |
| PandaGetDeplacementsPossible | Retourne les déplacements du panda possibles |
| PiochePandaIsEmpty | Retourne la pioche panda vide |
| JardinierGetDeplacementsPossible | Retourne les déplacements du jardinier possibles |
| PandaGetCoordonnees | Retourne les coordonnées du panda |
| JardinierGetCoordonnees | Retourne les coordonnées du jardinier |
| GetListeZonesPosables | Retourne la liste des zones de parcelles passables |

* **Opérations en POST :**

|  |  |  |
| --- | --- | --- |
| **Opérations** | **Actions** | **Objet dans le corps de la requête** |
| launch | Lance la partie | Aucun |
| ReposeSousLaPioche | Repose les parcelles non piocher sous la pioche | ArrayList<Parcelle> |

Les actions du jeu sont obtenues via l’url :

{ipHostServeur}:8080/{identifiantPartie}/{identifiantJoueur}/{OPERATION}

* **Opérations en GET :**

|  |  |
| --- | --- |
| **Opérations** | **Actions** |
| FeuilleJoueurGetNbAction | Retourne le nombre d'actions du joueur dans feuille joueur |
| FeuilleJoueurGetActionChoisie | Retourne l'action choisie par le joueur |
| FeuilleJoueurGetNbBambouRose | Retourne le nombre de bambous rose en possession |
| FeuilleJoueurGetNbBambouVert | Retourne le nombre de bambous vert en possession |
| FeuilleJoueurGetNbBambouJaune | Retourne le nombre de bambous jaune en possession |
| FeuilleJoueurGetMainObjectif | Retourne la main d’objectifs du joueur |

* **Opérations en POST :**

|  |  |  |
| --- | --- | --- |
| **Opérations** | **Actions** | **Objet dans le body de la requête** |
| **FeuilleJoueurInitNbAction** | Init le nombre d'action du joueur dans feuille joueur | Aucun |
| **FeuilleJoueurSetActionChoisie** | Init le nombre d'action du joueur dans feuille joueur | Int (0 piocher parcelle /1 piocher objectif / 2 déplacer panda / 3 déplacer jardinier) |
| **FeuilleJoueurDecNbACtion** | Décrémente le nombre d'actions du joueur | Aucun |
| **DeplacerPanda** | Déplace le panda | Une coordonnée |
| **DeplacerJardinier** | Déplace le jardinier | Une coordonnée |
| **PoserParcelle** | Pose une parcelle | Une parcelle |
| **PiocherUnObjectif** | Pioche un objectif | Int (0 parcelle /1 jardinier / 2 pandas) |
| **VerifObjectifAccompli** | Vérifie qu'un objectif est accompli | Aucun |

## 2/ lancement de partie

JOUEUR

* Etape 3 de la *Figure 4 : échanges serveur-joueur :*

Pour :

* Créer un nouveau joueur
* Connecter ce joueur au serveur
* Demander de lancer la partie correspondante à l’identifiant ce nouveau joueur

On utilise l’url suivante :

{ipHostJoueur}:8081 }/newPlayer

* Etape 4 de la *Figure 5 : échanges serveur-joueur :*

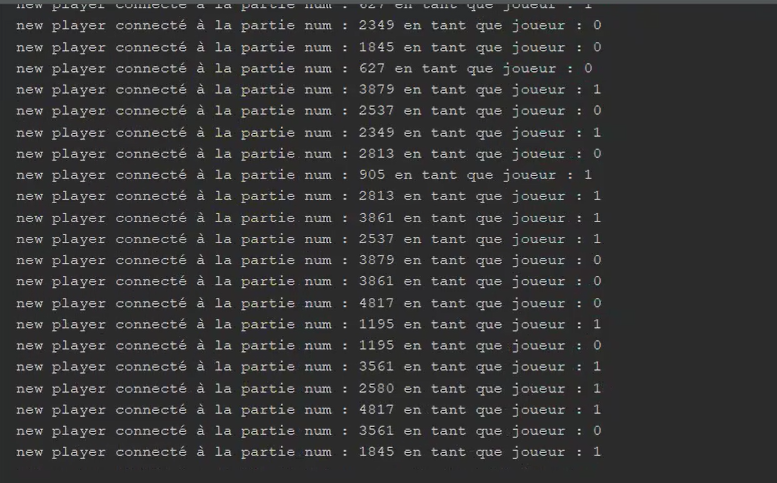
Pour :

* Chercher un joueur dans la liste des joueurs connectés
* Le faire joueur

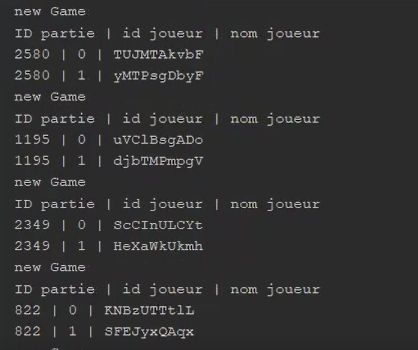
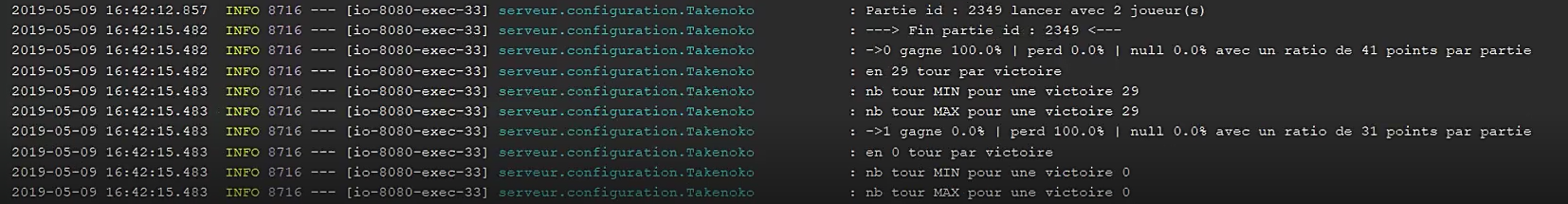
On utilise l’url suivante :

{ipHostJoueur}:8081 }/{identifiantPartie}/{identifiantJoueur}/joue

Ici on lance 15 parties avec 2 joueurs dans chacune.



On voit bien ici deux joueurs connectés à la même partie et celle-ci démarre dans la capture suivante.



# IV] Tests d’intégration

## 1/ les tests avec Cucumber

Dans notre pom.xml on ajoute les dépendances Cucumber nécessaires :



Les tests cucumber vont nous permettre de tester des « scénarios » de jeu, on effectue donc des tests de haut niveau contrairement aux tests unitaires JUnit effectués au premier semestre.

Ces scénarios, comme on le voit dans la *Figure 3 : Tests d’intégrations : Scénarios*, permettent d’assurer le bon fonctionnement des fonctionnalités demandées par l’utilisateur ainsi que les ressources qui en dépendent. Ce n’est donc que via les interactions faites au niveau « interface » que les erreurs internes à l’application pourront être trouvées.

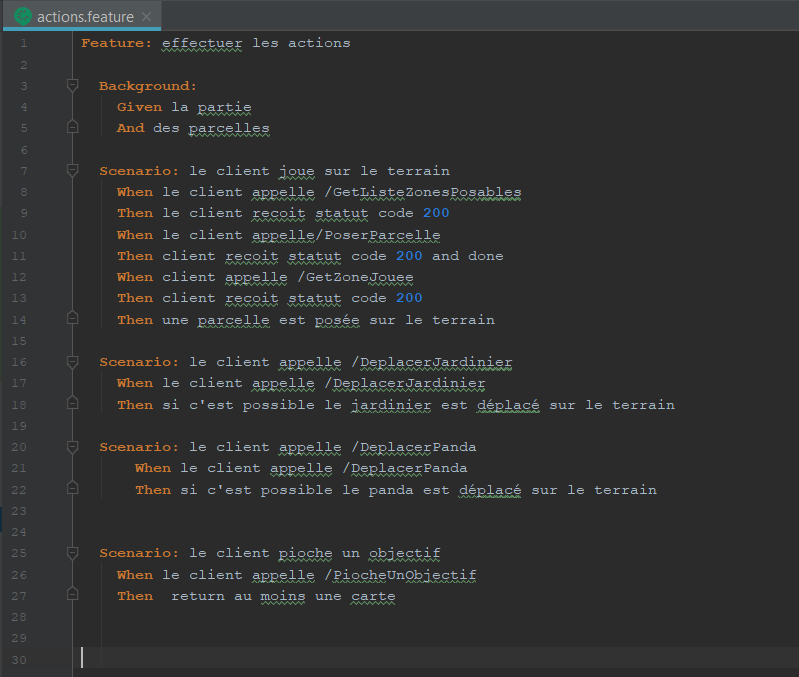


Figure3 : Tests intégrations : Scénarios

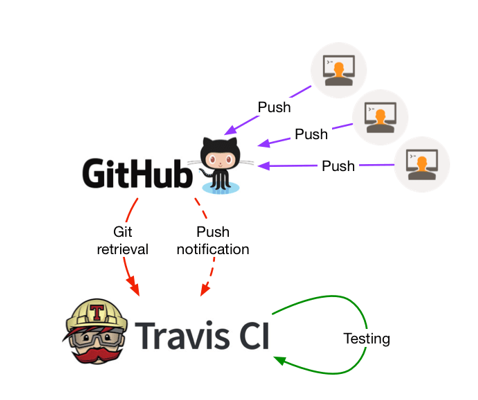
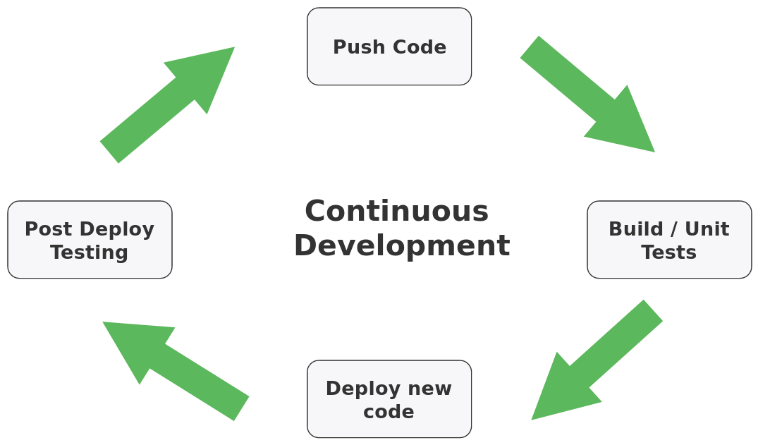
Comme on peut le voir sur la Figure3 : Tests intégrations : Scénarios nous avons utilisé ces tests pour vérifier le bon fonctionnement du jeu mais également, par la même occasion, toutes les routes REST mises en place.

En effet, le **statut code 200,** par exemple, permet de vérifier la réussite d’une requête http.

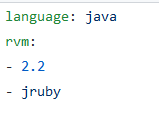
Donc dans nos tests d’intégrations on vérifie d’une part que nos routes sont valides et d’autre part, que les actions de jeu sont également celles attendues.

## 2/ l’intégration avec Travis

A ce stade nous mettons en place une intégration avec Travis CI comme ci-dessous.



Nous créons un fichier **travis.yml** contenant le code ci-dessous :



Travis est un outil de déploiement relié a Github pour que tous les tests soient exécutés à chaque commit sur le dépôt (voir *Figure 4 : Scénarios de Tests IA)*. De plus, lors d’un pull request (voir *Figure 5 : Pull request*), les tests sont également effectués afin savoir si l’acceptation de la branche introduira des confits.

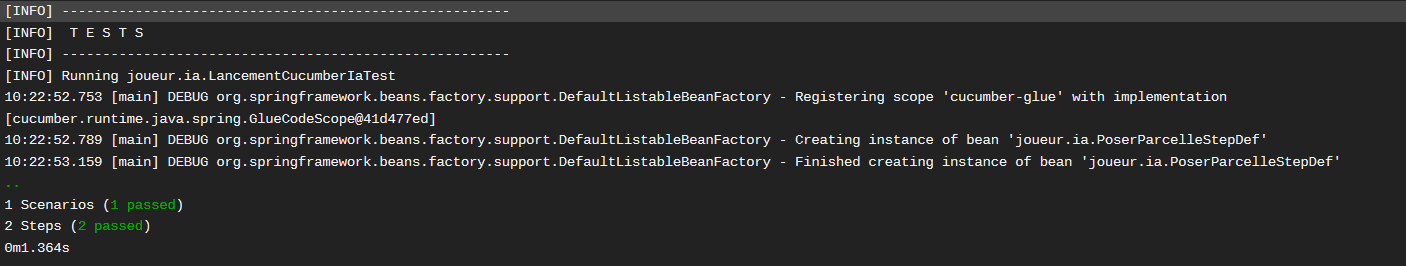


Figure 4 : Scénarios de Tests IA

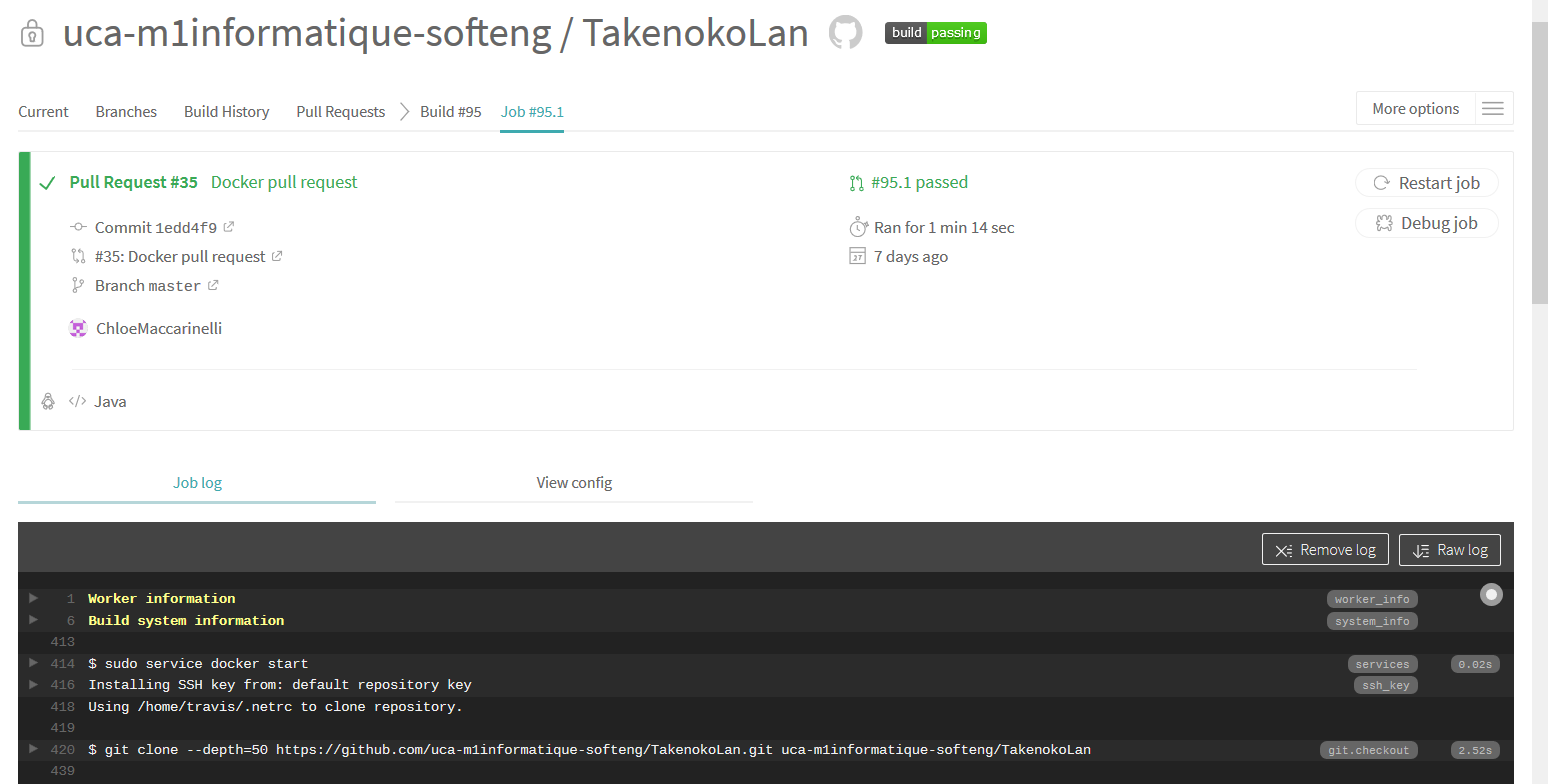
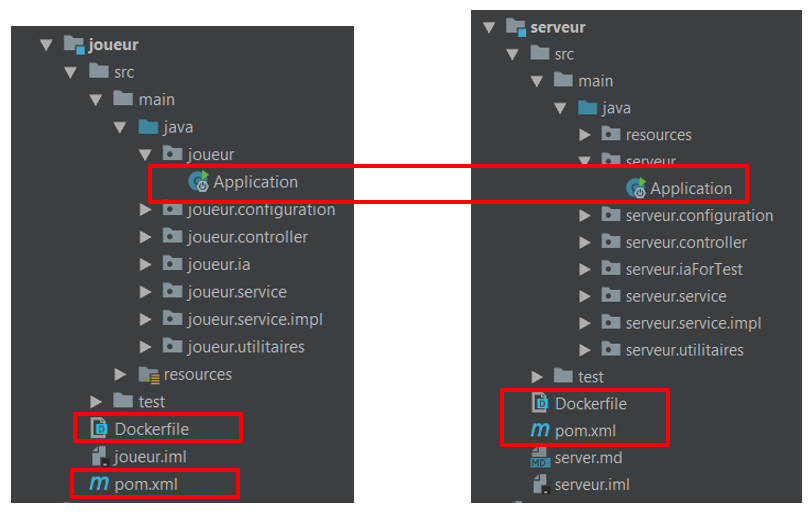


Figure 5 : Pull request

# V] Conteneurisation avec Docker

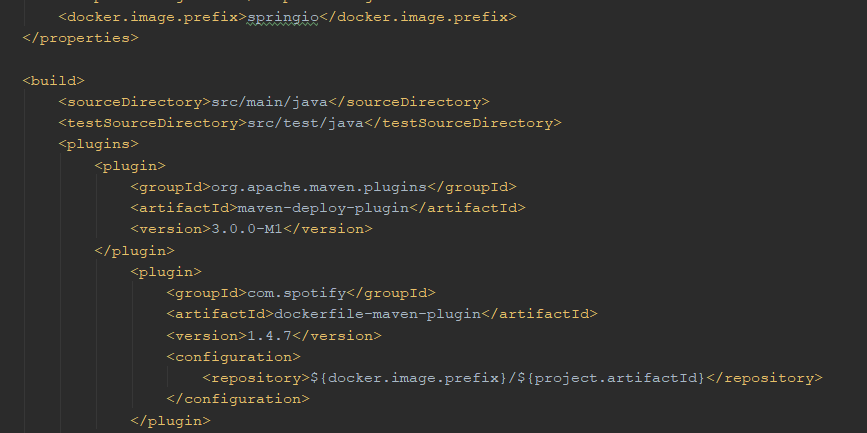
## 1/conteneurisation

La première étape de la conteneurisation est de créer une image du projet. Après découpage du projet comme expliqué dans la partie II], on se retrouve avec deux sous projets exécutables, c’est-à-dire contenant un **Application.java**.

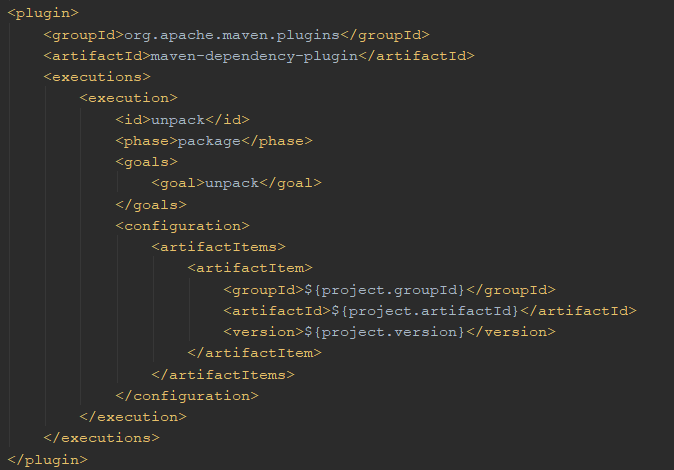


Afin de créer une image il faut donc dans chacun des projets créer un fichier Dockerfile et modifier les dépendances de notre pom.xml.

**1ère étape : Les dépendances**

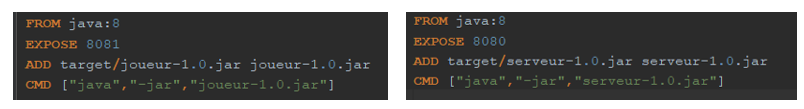


Ici on spécifie que le répertoire avec le nom de l'image, se terminera ici sous le nom de springio/serveur ou springio/joueur (artifactId du projet).



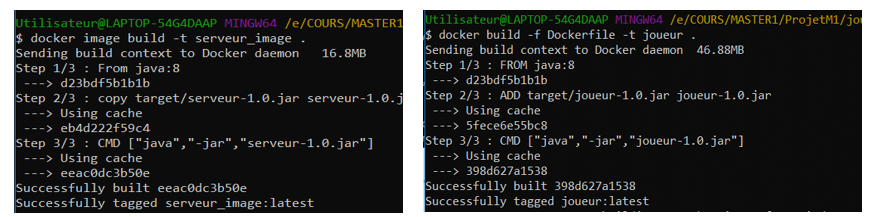
Ici, on s’assure que le .jar est bien décompressé avant la création de l’image docker.

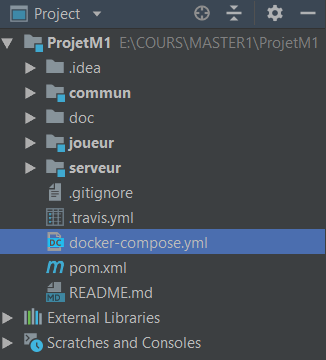
**2ème étape : les Dockerfiles**



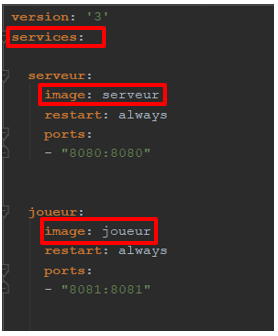
Dans ces Dockerfiles on précise le dossier docker ou l’on souhaite copier notre .jar et on précise comment l’exécuter.

Ci-dessous un test de la création de nos deux images :

docker build -f Dockerfile -t « nom\_image » .

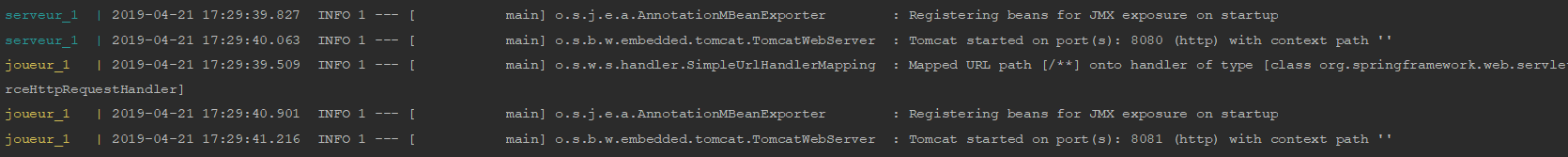
**3ème étape : le docker-compose.yml**

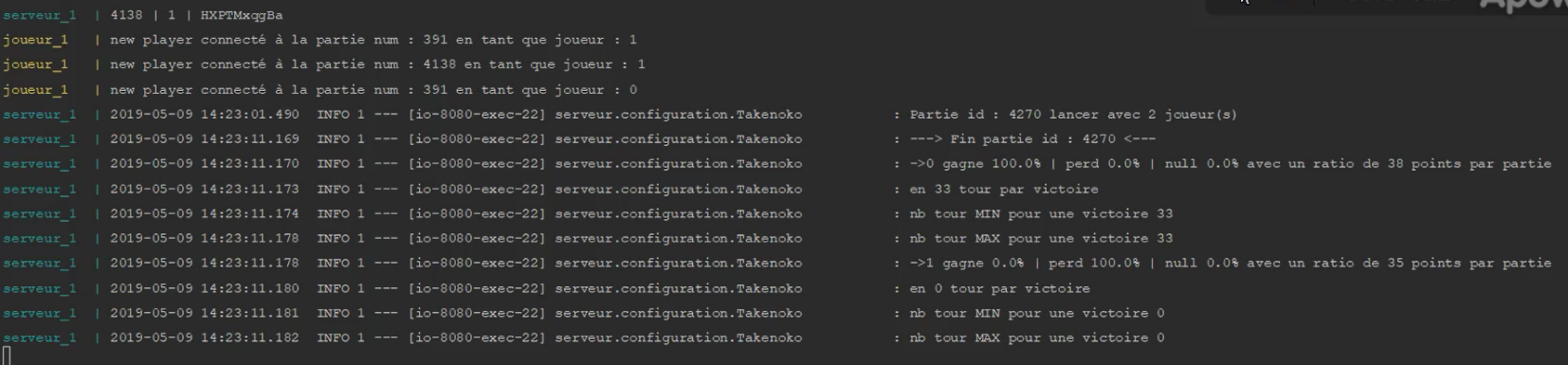
On veut pouvoir lancer nos deux images en même temps. Pour cela on crée, à la racine de notre projet mère un fichier docker-compose.yml, qui sera appelé pour lancer les deux images.



Dans ce fichier, on définit nos deux projets (services) et on fait références aux images créées précédemment. Ce sont-elles qui seront appelées et lancées en même temps.

On lance la commande suivante et on vérifie que les deux images se lancent bien.

 Docker-compose up

Ici on voit bien le serveur se connecter sur le port 8080 et le joueur sur le port 8081. On lance ici 15 parties avec chacune 2 joueurs.

## 2/ intégration avec Travis

À ce stade nous avons l’intégration 1 ci-dessous. Nous voulons l’intégration 2.

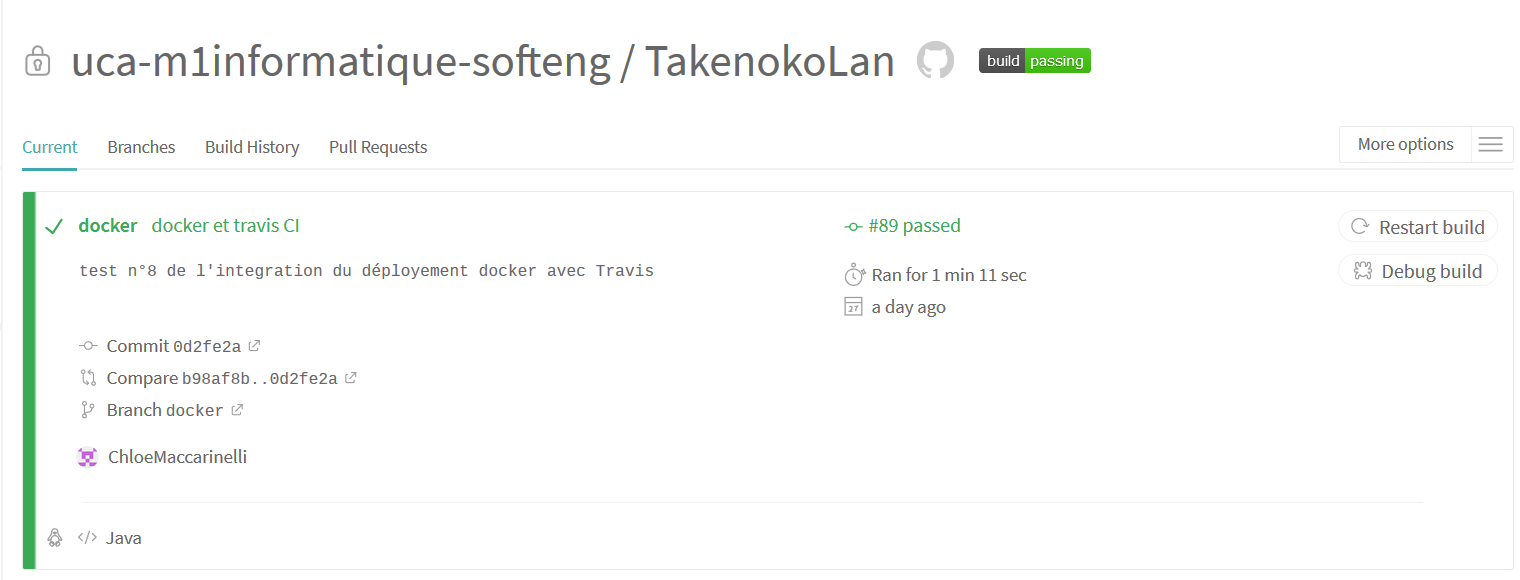


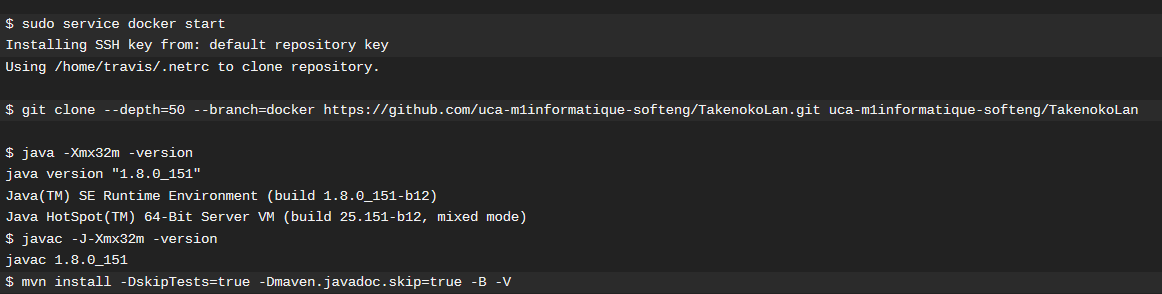
On modifie notre travis.yml afin de communiquer à travis CI les commandes à exécuter :



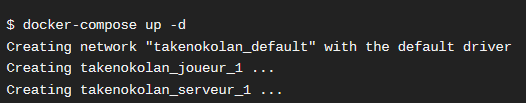
On précise le services docker afin que toutes les dépendances nécessaires soient téléchargées et ensuite travis exécutera le script. Ici on lui demande de créer les images docker et de lancer notre docker-compose.

Ci-dessous l’exécution dans travis CI :

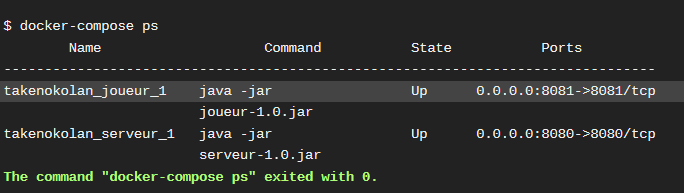




Dans cette première partie du log travis on voit qu’il démarre une machine docker puis récupère le projet sur github et crée les .jar *(mvn install)* afin de pouvoir les utiliser par la suite.

 Ensuite il crée les images docker puis les lances avec notre docker-compose. Les deux images sont alors démarrées.

Comme on peut le vérifier ci-dessous :



# V] Conclusion

????? on met quoi