

2019

Jeu du Takenoko

Yacine Lotfi, Chloé Maccarinelli, Pierre Marion, Alexandre Trujillo

Projet de développement

Master 1 Informatique S2

Table des matières

[I] Echanges Serveur - Joueur 2](#_Toc7642467)

[1/ Les routes implémentées sous forme de web service REST 2](#_Toc7642468)

[2/ lancement de partie 4](#_Toc7642469)

[II] Le découpage 5](#_Toc7642470)

[1/ choix des packages initiaux 5](#_Toc7642471)

[2/ contraintes de couplage 6](#_Toc7642472)

[III] Tests d’intégration 6](#_Toc7642473)

[1/ Les tests avec Cucumber 6](#_Toc7642474)

[2/ L’intégration avec Travis 6](#_Toc7642475)

[IV] Conteneurisation avec Docker 6](#_Toc7642476)

[1/conteneurisation 6](#_Toc7642477)

[2/ intégration avec Travis 9](#_Toc7642478)

# I] Echanges Serveur - Joueur

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

La communication joueur serveur est implémenté selon le schéma ci-dessous :

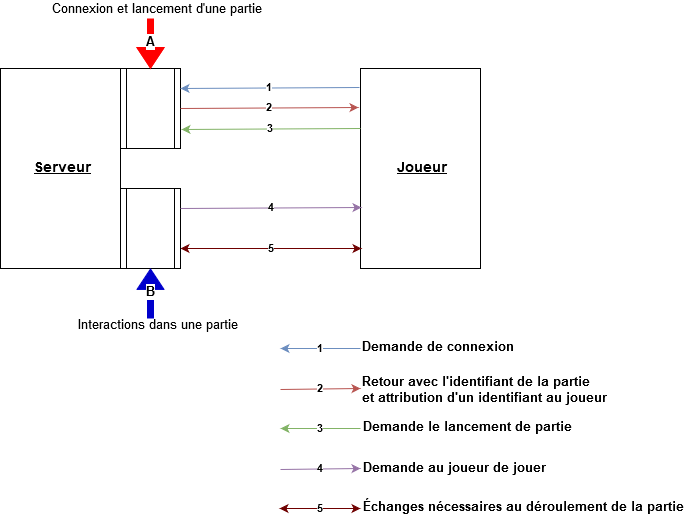


Figure 1 : échanges serveur-joueur

Ci-dessous les différentes méthodes implémentées en REST pour le déroulement de la partie.

(Flèche n°5 sur la Figure 2 ci-dessus)

* **/GetZoneJouee ::** retourne liste de zones jouées (GET)
* **/Piocher ::** retourne la liste des parcelles piochées (GET)
* **/PiocheParcelleIsEmpty ::** retourne pioche parcelle vide (GET)
* **/PandaGetDeplacementsPossible ::** retourne déplacements du panda possible (GET)
* **/PiochePandaIsEmpty ::** retourne pioche panda vide (GET)
* **/JardinierGetDeplacementsPossible ::** retourne déplacements du jardinier possible (GET)
* **/GetFeuilleJoueur ::** retourne la feuille joueur (GET)
* **/FeuilleJoueurGetNbAction ::** retourne le nombre d'action du joueur dans feuille joueur (GET)
* **/FeuilleJoueurGetActionChoisie ::** retourne l'action choisie par le joueur (GET)
* **/FeuilleJoueurGetNbBambouRose ; /FeuilleJoueurGetNbBambouVert ; /FeuilleJoueurGetNbBambouJaune ::** retourne le nombre de bambous en fonction de la couleur (GET)
* **/FeuilleJoueurGetMainObjectif ::** retourne la main d’objectifs du joueur (GET)
* **/PandaGetCoordonnees ; /JardinierGetCoordonnees ::** retourne les coordonnées des entités (GET)
* **/GetListeZonesPosables ::** retourne la liste des zones de parcelles posables (GET)
* **/FeuilleJoueurInitNbAction ::** init le nombre d'action du joueur dans feuille joueur (POST)
* **/ReposeSousLaPioche ::** repose sous la pioche (POST)
* **/FeuilleJoueurSetActionChoisie ::** implémente l'action choisie par le joueur (POST)
* **/FeuilleJoueurDecNbACtion ::** décrémente le nombre d'action du joueur (POST)
* **/DeplacerPanda ::** déplace le panda (POST)
* **/DeplacerJardinier ::** déplace le jardinier (POST)
* **/PoserParcelle ::** pose une parcelle (POST)
* **/PiocherUnObjectif ::** pioche un objectif (POST)
* **/VerifObjectifAccompli ::** vérifie qu'un objectif est accompli (POST)

## 2/ lancement de partie

Détailler la partie A du schéma avec … plusieurs joueurs …..

# II] Le découpage

## 1/ choix des packages initiaux



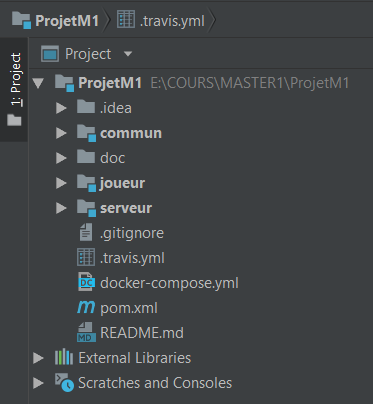
Figure 2 : Architecture 1

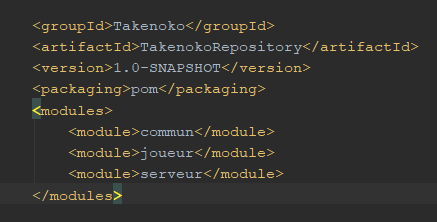
Avant le découpage nous avions l’architecture de la Figure 2 ci-dessus. Après réflexion nous avons répertorié toutes les classes et packages qui étaient dépendant les uns des autres, ainsi nous avons pu déjà mettre en place un sous-projet « commun » qui ne contiendra pas de classe Application.java.

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].

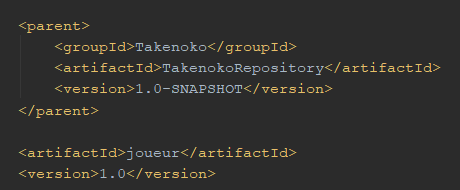
Au niveau des classes, le joueur ne contient que la classe IAPanda (le joueur), et les configurations nécessaires à sa fonction de serveur.

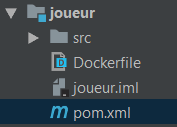
Le serveur quant à lui contient toutes les autres classes relatives au moteur de jeu, et les configurations nécessaires à sa fonction de serveur.

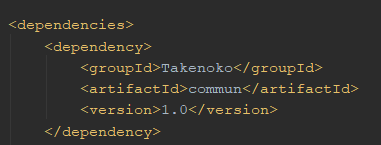
Le « serveur » ,« joueur » et « commun » sont trois sous projets de notre projet mère ProjetM1 comme ci-dessous.

Pour que les sous-projets soient implémentée 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.

## 2/ 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 ».

A détailler ?!??!

# III] Tests d’intégration

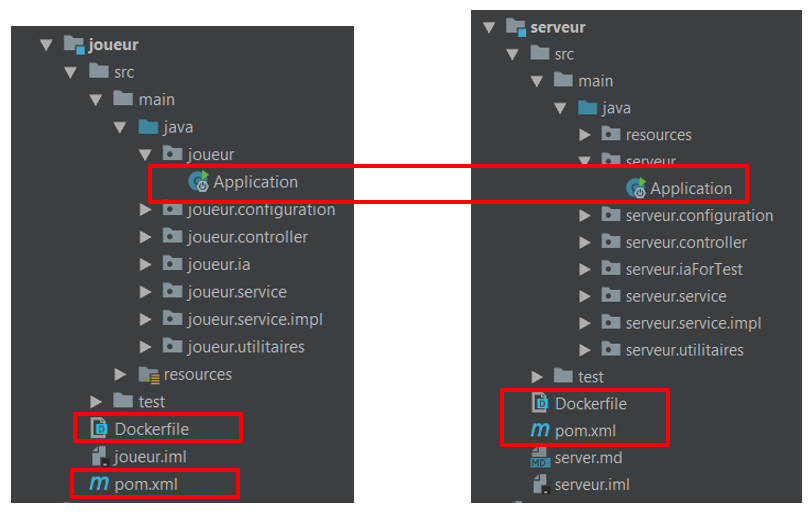
## 1/ les tests avec Cucumber

## 2/ l’intégration avec Travis

# IV] Conteneurisation avec Docker

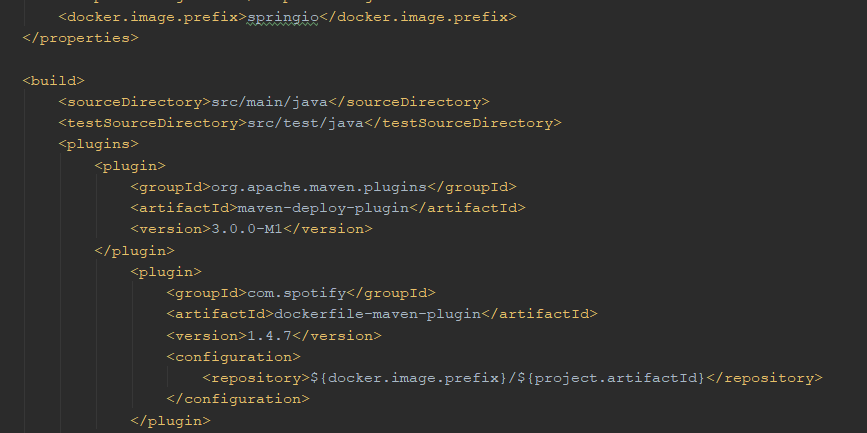
## 1/conteneurisation

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**

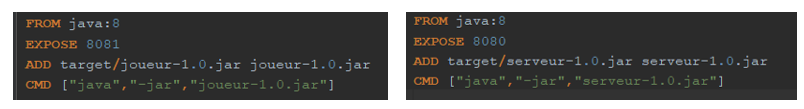


Ici on spécifie que le répertoire avec le nom de l'image, se terminera ici sous le nom de springio/serveur ou springio/joueur (artifactId du projet).



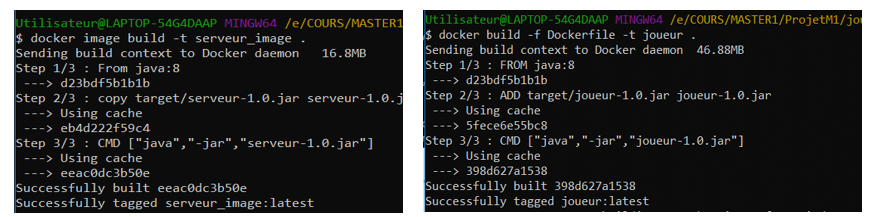
Ici, on s’assure que le .jar est bien décompressé avant la création de l’image docker.

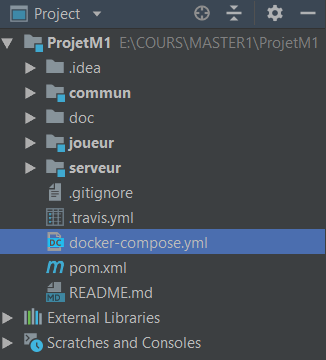
**2ème étape : les Dockerfiles**



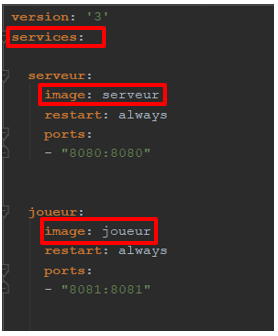
Dans ces Dockerfiles on précise le dossier docker ou l’on souhaite copier notre .jar et on précise comment l’exécuter.

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

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

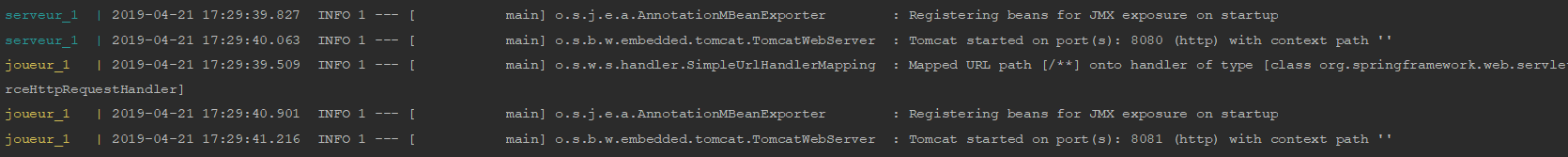
**3ème étape : le docker-compose.yml**

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.

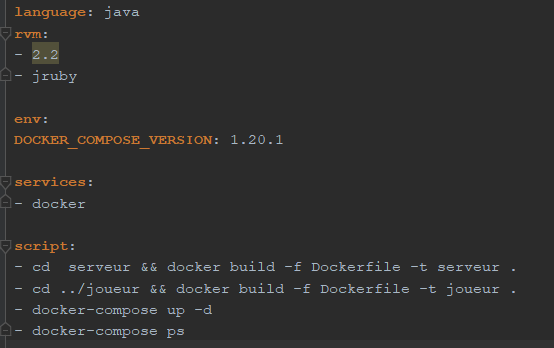
Docker-compose up

## 2/ intégration avec Travis

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

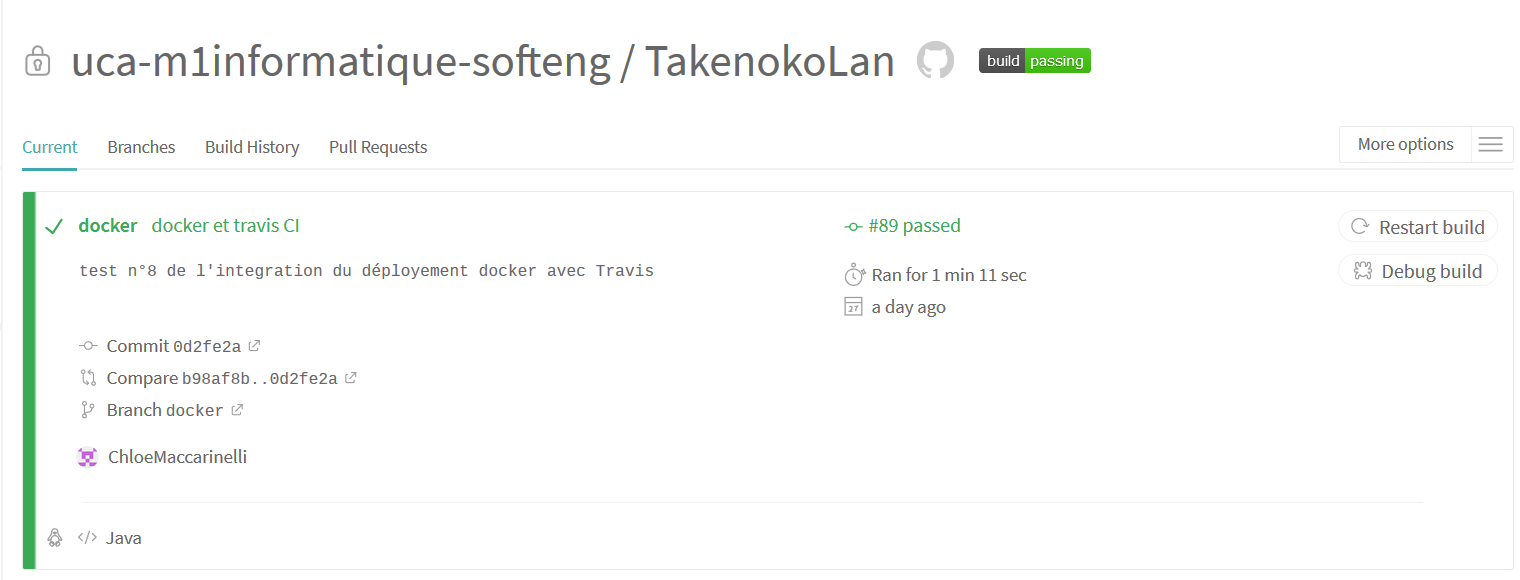


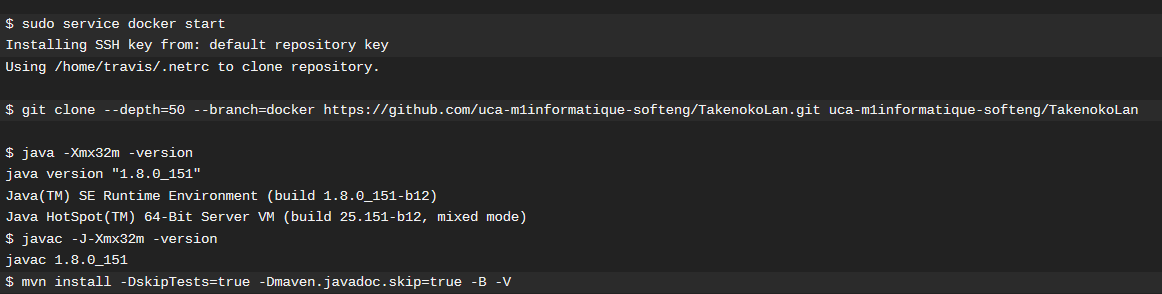
On modifie notre travis.yml afin de communiquer à travis CI les commandes à exécuter :



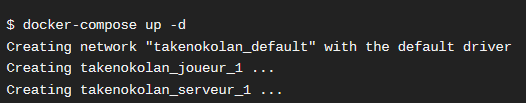
On précise le services docker afin que toutes les dépendances nécessaires soient téléchargées et ensuite travis exécutera le script. Ici on lui demande de créer les images docker et de lancer notre docker-compose.

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 :

