Rapport du projet de Java EE

Maxence Ahlouche Maxime Arthaud Korantin Auguste
Martin Carton

24 janvier 2013

1 Introduction

Le choix du sujet étant libre, nous avons choisi de faire une plateforme pour se faire s'affronter des « IA » jouant à des jeux tour par tour [1].

Les webmasters du site peuvent ajouter autant de jeux qu'ils le souhaitent (il suffit de programmer une simple classe jouant le rôle d'arbitre en Java, et un morceau de Javascript pour l'affichage).

Les utilisateurs quant à eux peuvent coder leurs « IA » dans n'importe quel langage. Les différentes « IA » et l'arbitre communiquent entre eux en JSON par socket. Les combats peuvent être lancés par les utilisateurs. Pour éviter de surcharger le serveur dans le cas où trop de combats se dérouleraient en même temps, les combats peuvent être lancés sur des machines distantes, qui récupèrent leurs tâches grâces à une queue JMS.

L'interface web permet de s'inscrire, de se connecter, d'uploader des « IA », de les faire combattre contre celles des autres utilisateurs, de regarder les scores, de regarder le déroulement tour par tour d'un combat, etc.

2 Choix des technologies

Dès le début du projet, nous avions décidé d'utiliser quelque chose de plus évolué que ce qui nous a été présenté en cours. Notre choix s'est donc naturellement porté vers Spring. Toutefois, en surfant sur le World Wild Wed, nous avons trouvé un projet tout neuf qui nous a semblé particulièrement intéressant : Spring Boot [2].

Spring Boot est une surcouche de Spring, dont la philosophie est *convention over configuration*. Cette approche convenait parfaitement à nos besoins, car nous n'avions aucune envie de configurer toutes les couches d'un projet JEE une par une. Un autre grand avantage de Spring Boot est qu'il permet de compiler toute l'application, ainsi qu'un serveur Tomcat, JPA et une base de données H2 (si besoin) en un seul fichier war, qu'il nous suffit de lancer avec Java pour avoir un serveur Web complet.

Le principal inconvénient de cette approche est qu'il n'était souvent pas trivial de modifier la configuration par défaut, d'autant plus que ce projet est encore jeune : le peu de documentation disponible était réparti entre plusieurs sites (leur Github, leur site, et leur ancien site plus maintenu), et était souvent incomplète. De plus, comme ce projet n'est pas encore utilisé suffisamment largement, StackOverflow ne nous a (pour une fois) pas été d'une grande aide.

Afin de gérer toutes nos dépendances, nous avons décidé d'utiliser Gradle, qui est une alternative à Maven. Ce choix nous a semblé le meilleur, car la syntaxe du fichier de configuration est bien plus user-friendly (nous avons réussi à faire un site Web fonctionnel en Java EE sans une seule ligne de XML \o/).

1. Le terme « IA » est un peu trompeur, nous entendons par là de simples programmes jouant aux jeux automatiquement.
2. `https://github.com/spring-projects/spring-boot`

Afin de représenter les vues, nous n'avons pas choisi les templates JSP mais un système de templates nommé Thymeleaf. Bien que ce dernier ne soit certainement pas le meilleur qui soit (notamment parce qu'il préfère faire des includes à tout va plutôt que de faire de l'héritage de templates), il nous a permis de découvrir une autre manière de traiter le problème, qu'aucun d'entre nous n'avait vue auparavant, étant plutôt habitués au système de templates de Django (un autre framework Web en Python).

Le HTML de base est du HTML5, couplé à du CSS3. Le design a été entièrement fait main, et n'utilise pas de framework de type bootstrap ou autre.

Concernant le choix du DBMS, notre choix s'est porté sur PostgreSQL, car l'un d'entre nous avait déjà un serveur configuré à disposition sur un serveur personnel.

3 Backend : génération des matchs

L'interface web permet de lancer des matchs. Lors du lancement d'un match, une ligne est insérée dans la table « match » de la base de donnée, et un message contentant l'ID du match est ajouté dans la queue JMS.

Des « workers » jouent le rôle de consommateurs de cette queue. Ils attendent un message, récupèrent les informations relatives au match dans la base de donnée, puis lancent le match. Pour cela, ils décompressent les archives des joueurs (placées dans un dossier qui serait par exemple partagé via NFS), et lancent le script bash *launch* à la racine.

Cette architecture permettrait donc de traîter énormément de matchs en parallèle, même si les matchs mettent du temps à s'exécuter. Le traitement des matchs peut s'effectuer sur autant de machines physiques que l'on souhaite : c'est une solution qui passe extrêmement bien à l'échelle.

Nous avons utilisé le principe du classement Elo [3] provenant des échecs. Le principe est simple : un joueur part avec 1000 points Elo. S'il gagne contre quelqu'un, ses points augmenteront en fonction de son score Elo et de celui de son adversaire. Par exemple, un match entre le premier et le dernier ne changera pratiquement pas les scores Elo. À l'inverse, si le dernier gagne, on pourra observer de grandes variations dans leurs scores.

4 Problématique de sécurité

Actuellement, aucune vérification de sécurité n'est effectuée sur les workers, et il est donc possible de faire n'importe quoi dessus. C'est une piste importante d'amélioration.

Pour y remédier, on pourrait lancer chaque script joueur avec un UID/GID Unix qui lui est propre, ainsi qu'avec un démon surveillant la consommation mémoire/CPU du script. Ainsi, il n'aurait un accès limité au système et ne pourrait pas le saturer. Couplé à de bonnes règles `ulimit` et à un `iptables` bien configuré pour interdire l'accès au réseau à ces programes (via le module `owner`), on devrait atteindre une bonne sécurité.

Il est aussi possible de virtualiser toute l'exécution, ou de la faire dans des sandbox LXC. On peut aussi regarder du côté de Google Native Client...

3. `https://fr.wikipedia.org/wiki/Classement_Elo`