Migration d'une webapp Tomcat vers Vert.x
(ou Servlet contre Vert.x)

Date : 10 avril 2015
Format : Conférence
Speakers : Florian Boulay, NextPerf

Florian nous livre un retour d'expérience de migration d'une application web utilisant le conteneur de Servlets Tomcat vers la plateforme Vert.x 2.1. Florian insiste sur la version car la version 3 devrait sortir sous peu.

Support de présentation disponible sur SlideShare.

Contexte

Nextperf vend des espaces publicitaires achetées aux enchères au moment de l'affichage d'une page (Real Time Billing)
Beaucoup de trafic : plusieurs milliards de requêtes HTTP par jour.
Partenaires : Google et Yahoo.
Contraintes : 100 ms en prenant en compte le trafic réseau.

Sur Tomcat : temps de réponse serveur de de 4 ms.
Beaucoup d'optimisations : cache, limitation des I/O
Avec Tomcat : 1 connexion TCP = 1 thread.

Volonté de passer sur NIO.

Présentation de Vert.x

Caractéristiques :

- Pushline sur le site de Vert.x : « Vert.x est une plateforme très légère, très performante tournant sur la JVM »
- Polyglote : tous les languages tournant sur la JVM : rubby, scala (avec module supplémentaire)
- Scalable : API utilisant des IO non bloquants et asynchrones (inspiré de Node.JS).
- Evènementielle. Utilise un bus d'évènements
- Rapide, très léger : 4 dépendances
- Vert.x est basé sur Netty

Ce qu'on peut construire avec Vert.x :

- API Rest
- Webapp
- Applications servers
- Frameworks de plus haut niveau

Vert.x arrive relativement nu. Exemple : pas de gestion de cookies.

Comme exemple de code Java, Florian nous montre une classe Server démarrant un serveur http. La classe serveur hérite de Verticle. D'après Florian, le code ressemble à du Node.JS. Vert.x dispose d'une API fluent. Il utilise les lambdas de Java 8pour les callback afin d'améliorer la lisibilité du code.

Avec Vert.x, les développeurs peuvent être confrontés au problème du **callback hell**. Cependant, il existe un module RxJava permettant d'aplatir ce problème.

Dans le monde Vert.x, un **Verticle** est l'équivalent d'une servlet. C'est également le point d'entrée de l'application. Possibilité d'avoir plusieurs Verticle. Packaging possible en modules. Les modules sont les « libs » de Vert.x

Tous les Verticle sont exécutés dans une event loop. Cet event loop dispose de très peu de threads (un thread par core). La programmation d'un Verticle se fait comme une application mono threadée : pas de bloc synchronized ou de variables volatiles. C'est ce qui a plus a Florian.
Règle d'or : ne pas bloquer l'event loop.

Comment intégrer des libs bloquantes ? (ex : envoi de mails, Hibernate …) Et bien il faut les faire s'exécuter dans un **worker verticle**. Un worker verticle dispose d'un pool de threads dédié ne bloquant pas l'event loop.
Cela permet de rendre compatible toutes les librairies existantes bloquantes.

L'**event bus** permet de faire communiquer entre elles les Verticle. Ressemble à un modèle d'Actor (comme Akka). Différents types sont supportés (ex : un String), mais JSON est le

format préconisé car interopérable. Ce bus permet de communiquer avec des Verticles d'autres JVM déployés dans le même cluster Vert.x

Migration de Tomcat vers Vert.x

Tomcat dispose d'un **moteur de template JSP** basé sur l'API Servlet.
Vert.x ne reposant pas sur l'API Servlet, Florian a étudié la possibilité de changer de moteur de templates (Thymleaf et Velocity).
Dans une réponse HTTP, Vert.x ne permet que 2 choses : envoyer une String ou un fichier HTML.
La migration du code HttpServlet est plutôt facile. Les API se ressemblent.

Une complexité de Vert.x réside dans la manière où il utilise les **Classloader**.
Par simplification, Vert.x crée un class loader par instance de Verticle.
Impact : impossibilité d'avoir des données stockées dans un cache.
Solutions possibles : utiliser des shared data, interroger un Verticle local faisant office de cache (requière une sérialisation/désérialisation au sein de la même JVM), utiliser un cache externe possible (Memcache, Redis).
Problème de Classloader remédié dans Vert.x 3. Vert.x 3 apporte également des encodeurs / décodeurs qui permettront de ne pas sérialiser/désérialiser des objets immutables.

Les librairies avec IO bloquantes doivent être isolées dans un worker Verticle.
Certaines disposent d'un module asynchrone. C'est le cas du module asynchrone Mysql Postgresql (codé en Scala par un membre de la communauté Vert.x)
Malheureusement, il n'existe pas (encore ?) de modules asynchrones pour toutes les technos (ex : Cassandra oui mais Mongo non)

Vert.x offre différents niveaux de granularités pour déployer le code : un Verticle peut être déployé / redémarré sans interrompre l'application.
Possibilité de créer un shade-jar. L'application peut être packagée et déployée avec un java –jar (flat jar)

Un Verticle est sans état (pas de sessions web). La migration d'applications statefulls est compliquée.
Vert.x ne fournit pas de mécanismes d'authentification.

Conclusion

- Si c'est une webapp : migration difficile en cas de moteur de template non transposable
- Si il s'agit d'un serveur de services REST : migration envisageable
- Si c'est une application server to server : difficile si beaucoup de champs statiques

Q/A

La migration a été suspendue suite à l'arriver de Vert.x 3 (amélioration du Classloader et du callback hell via RxJava).
Hazelcast est utilisé dans Vert.x 2. Dans Vert.x 3 on peut utiliser également JGroup pour gérer le cluster de Verticles.
Comment définir une Verticle ? par couche ? par cas d'utilisation ? Les accès MongoDB ont été mis dans un Verticle. Les Servlets ont été mis dans un Verticle. La partie batch a été migrée en Verticle.