Quarkus : pourquoi et comment faire une appli Java Cloud Native avec Graal VM

Speakers : Emmanuel Bernard (RedHat), Clément Escoffier (RedHat)
Format : Université
Date : 17 avril 2019
Slides et démo : https://github.com/cescoffier/quarkus-deep-dive

Pourquoi Quarkus ?

Le Quark est une particule subatomique.
RedHat croit beaucoup aux conteneurs. Ils ont investi dans Kubernetes et OpenShift. Depuis des années, ils essaient d'optimiser les temps de démarrage des JVM, et ceci à plusieurs niveaux : équipe cgroup, équipe MiddleWare, software Java.
Lorsqu'ils ont découvert GraalVM, ils ont eu pensé qu'un paradigme allait changer.

Cloud Native

Définition d'Emmanuel du Cloud Native :

- Pateforme Cloud
    - Choix architecturaux structurants : les applications sont bâties pour pouvoir être redémarrée sur un autre serveur …
    - Scalabilité virtuellement infinie. La scalabilité peut augmenter automatiquement en fonction de la demande.
    - Accès instantané à la ressource
    - Les ressources sont fluctuantes : elle peut ne pas être là et l'application doit savoir gérer ce cas
- Termes corrélés
    - Microservices
    - 12-factor app

Comment peut-on être Cloud Native ?

- L'environnement fournit la configuration. La configuration est externalisée
- Etat partagé par les N instances scalables (équivalent à la session)
- Cache distribué

- Les microservices doivent avoir des cycles de vie séparés. Les entrées / sorties sont définies et sert d'interface

Monolith, Microservices & functions

Ordre de grandeur : 1 monolith = 20 microservices = 200 fonctions

Par rapport à un microservice, une fonction est appelée : elle n'écoute pas sur un port. C'est l'infra qui écoute et appelle la fonction. Une fonction peut scaler à 0 (un microservice reste à 1).
Les microservices apportent de l'agilité : on peut modifier un microservice (ex : migration SGBD relationnel vers du NoSQL) sans avoir à redéployer les autres.

Conséquence de l'explosion des nœuds : difficulté de faire communiquer tous ces microservices. Cela devient très complexe et pose de nombreuses problématiques.

Définition de Réactif d'après le dictionnaire Oxford: on réagit à une situation qu'on ne contrôle pas.

Les applications distribuées communiquent en HTTP. Lorsqu'un microservice est tombé, les autres microservices partent en time-out. Les circuit breaker (ex : Hystrix) essaient de résoudre ce problème par du code applicatif.
Lorsque le circuit breaker détecte une indisponibilité, l'utilisateur va réessayer (touche F5) : cela va engorger encore davantage le système. Car derrière le circuit breaker il y'a des threads.

Nécessité d'avoir des brokers (Kafka, MQ …) fournis par le Cloud provider pour avoir des systèmes tolérants aux pannes et élastiques. L'utilisateur est prévenu que sa demande a été prise en compte. On construit un système réactif en utilisant des messages asynchrones : élasticité, résilient et responsif.

Java et ses limites

La JVM a été pensée au début du web pour servir des requêtes HTTP. Le débit (throughput) était plus important que le CPU et la mémoire. A cette époque, on avait un gros serveur multi-threadés et multi-core. On déployait toutes les applications sur le même serveur (la même JVM).
Avec les conteneurs, on a désormais plein d'instances de la même JVM.
Or, la JVM est lente à démarrer :

- Beaucoup de classes à charger : interdépendances entre les classes du JDK (le JDK était vite chargé entièrement). Anecdote de mémoire : MySQL tire JavaFX pour récupérer le mot de passe utilisateur.
- Vérification de bytecode
- De l'interprété au démarrage, phase de chauffage puis réutilisation du code natif

Surcout mémoire :

- Beaucoup de classes à charger

- Dans OpenJDK HotSpot : métadonnées sur la classe (dépendances, code en cache, statistiques d'usage …)

Pour des microservices qui ont peu de données, ce surcout de mémoire est visible.

Démo

L'université se poursuit par la démo d'une application construite à l'aide de thorntail :
application JAR exposant un endpoint Rest « Hello World » à l'aide d'un Wildfly embarqué.
L'application ne démarre ni avec 10 Mo de Heap ni avec 28 Mo (OutOfMemoryError).
Il faut précisément 29 Mo pour la démarrer.
En réalité, la JVM consomme 233 Mo.
Via JConsole, on constate que la Non Heap est de 85 Mo.
Bilan : il y'a 233 – 29 – 85, soit 119 Mo occupés par le Resident State Size (RSS).

Pour ces raisons, l'usage de JVM pour les microservices est loin derrière d'autres plateformes comme NodeJS.
Ce qui importe : le nombre de requêtes par seconde par Ko de mémoire consommée.

GraalVM : un nouvel espoir

Avant GraalVM, il y'a une dizaine de projets qui ont essayés de compiler nativement du Java.
Certains ont fait de gros compromis (ex : pas de threads).
GraalVM fait également des compromis, mais sont acceptables pour Quarkus.

GraalVM est une communauté Open Source qui contient plusieurs projets :

- Graal compiler : le compilateur Java vers code natif a été entièrement réécrit en Java. Twitter utilise le Graal compiler dans Java HotSpot VM. Cela marche très bien sur du code Scala.
- Truffle : machine virtuelle universelle permettant de faire des VM polyglottes : Java, Ruby, JavaScript (langage supportant LLVM)
- Substrate VM : VM s'appuyant sur le Graal compiler et fait tourner l'application Java compilée nativement. Elle n'embarque pas de JIT. Elle possède un garbage collector plus simple que ceux existants dans OpenJDK. RedHat contribue sur ce GC.

Substrate VM : le JAR ou les classes Java passent par un compilateur en code natif : 0 et 1.
Pour Quarkus, la cible est le conteneur basé sur du Linux.
Lors du Tree Checking, Substrate VM élimine tout le code mort : les classes inutilisées du JDK jusqu'au champs des classes maison jamais utilisés.

Une 2nde démo s'appuie sur une application affichant un HelloWord depuis un main.
Emmanuel bascule sur GraalVM 1.0.0-rc14.
Commande : *native-image -cp target/ Main*
Sur une vraie application, le temps de compilation est bien supérieure : compter 3 minutes, le temps que le Tree Checking se fasse.
Génération d'un exécutable main.
Commande : *time ./main*

Le code s'exécute 10x plus vite.

Contrainte : il faut donner à GraalVM la liste des ressources à utiliser (ex : application.properties) via une RegEx Java.
La compilation est plus rapide car il y'a un serveur qui tourne et optimise la compilation.
Ressources nécessaires pour GraalVM : ressources statiques (CSS, JS), fichiers de conf …
L'équipe GraalVM travaille sur la possibilité qu'aurait un JAR à indiquer à GraalVM les ressources qu'il embarque.

On tombe sur du ClassNotFoundException lorsqu'on utilise la réflexion. Il faut alors spécifier à GraalVM les classes qu'on utilise par réflexion par l'intermédiaire du fichier reflectconfig.json.
Attention, lorsqu'on précise à GraalVM les classes utilisées par Réflexion, toute l'optimisation sur le code mort est désactivée.

Consommation mémoire : ps s -o rss | grep java
GraalVM : 2 Mo vs application Java : 26 Mo.

Close world asumption

Le côté négatif de GraalVM est que plusieurs mécanismes ne sont pas supportés :

- Dynamic classloading (car pas de class loader : le classe loader est émulé)
- InvokeDynamic & Method handles (problématique pour JRuby)
- Finalizer
- Security Manager
- JVMTI, JMX (travaux en cours pour), native VM interfaces

Certains mécanismes fonctionnent mais avec des limitations :

- Relexion : liste manuelle
- Dynamic proxy : liste manuelle
- JNI : liste manuelle
- Static initializers : les JVM connues appellent le code statique lorsque la classe est initialisée (lazy). Ce comportement est utilisé par les frameworks. Dans GraalVM, tous les blocs statiques sont initialisés à la compilation et sont chargés au démarrage de l'application : initialisation agressive. Ce comportement reste compatible avec les spécs de Java, mais pas avec les frameworks.
- WeakReference, SoftReference … : un remplaçant existe

Des erreurs de jeunesses pourront être corrigées. D'autres ne peuvent pas l'être fondamentalement.
Les lignes de commande deviennent vertigineuse.

Quarkus

Leitmotiv Quarkus : « **Supersonic, Subatomic, Java »**

- Supersonic : le démarrage doit être ultra rapide (de l'ordre d'une dizaine de millisecondes). Rend possible l'usage de Cloud Function en Java (ServerLess)
- Subatomic : les exécutables générés sont très petits. Le Dockerfile va être très simple. La consommation mémoire est très basse (20 Mo pour une application d'entreprise). On peut démarrer plusieurs applications sur le même serveur
- Java : nul besoin d'apprendre un nouveau langage. Énorme communauté et plein de projets. Quarkus offre un hot reload super rapide.

Le quarkus-maven-plugin permet de générer un nouveau projet comprenant :
- Un pom.xml : BOM quarkus, quarkus-maven-plugin pour le build, quarkus-junit5
- Une application HelloResource avec endpoint REST

Commande : *mvn compile quarkus:dev*
Utilise Quarkus 0.13.3
D'après Emmanuel, le développement avec Quarkus comporte pas mal de ressemblances avec Play 1 et 2.
Démo : Emmanuel remplace à chaud (hot-replace) la chaine « hello » en « bonjour », ajoute d'une opération en 0,221 secondes puis utilise un fichier properties
A ce stade de la démo, l'application tourne dans une JVM. Il n'y a pas de compilation native.
Quarkus se base sur CDI. Création d'un JavaBean annoté avec *@ApplicationScoped* et d'un *@Inject*

Pour tester : on a une classe annotée avec *@QuakusTest* : l'application est démarrée au début de chaque TU et le premier TU dure 1,241 ms

RedHat a l'ambition de convertir l'écosystème Java en applications natives. Cela demande aux frameworks de changer la manière dont ils fonctionnent, notamment au démarrage.

Au démarrage, un framework comme Hibernate ou Spring :
- Parse les fichiers de configuration
- Scanne le classpath (cherche les beans ou les entity)
- Construit les méta-données utilisées par le framework au runtime
- Prépare la réflexion et crée les proxies
- Démarre, ouvre les IO, les threads …

Quarkus prend un framework comme Hibernate et essaie de faire tout ce travail au build.
Les bénéfices en sont :
- On ne le fait qu'une fois : au build et pas à chaque démarrage
- Toutes les classes de bootstrap ne sont plus nécessaires au runtime
- Gain temps de démarrage et moins de consommation mémoire
- Moins de réflexion

Quarkus fait également au build time :
- Indexation
- Amélioration de bytecode
- Génération des classes qui démarrent les frameworks

Quarkus aide GraalVM :

- Elimination du bytecode au niveau fonctionnel. Exemple : Hibernate sait qu'il va avoir besoin des getter / setter des @Entity
- Demande aux frameworks quelles sont les ressources pour lesquelles il doit être notifié en cas de changement
- Diminue les dépendances : si le cache de 2nd niveau n'est pas utilisé, Hibernate l'indique à Quarkus

GraalVM demande des changements aux frameworks.
Pour le dév, Quarkus est vu comme un plugin Maven ou Gradle et un ensemble d'extensions.

Quarkus tourne sur HotSpot et SubstrateVM.
Quarkus Core s'appuie sur Jandex, Grizmo et Graal SDK.
Comme Dagger, le système d'injection de dépendance ARC est fait au build time. Elle n'est pas faite avec l'annotation processor (comme pour Micronaut), mais après.

Les extensions : bout de code qui fait interagir le framework avec Quarkus au build time.
Extensions existantes : RESTEasy, …
Quarkus génère un JAR. On peut convertir ce JAR en application native avec GraalVM.
Par défaut, Quarkus ne génère pas un fat-jar car dans le monde des conteneurs il faut mieux utiliser un runner et les librairies (JAR).

L'application binaire (Servlert et JAX-RS) prend 20 Mo (sur Mac) et démarre en 0,007 secondes. La requête http est aussitôt prêtre à être exécutée.

Quarkus gère le build cross-platform.
Sur Linux, le binaire fait 19 Mo. Il faut passer par Docker pour exécuter le binaire (Dockerfile fournit)

Une application Quarkus peut tourner sur Kubernetes (démo avec Minikube).

Du Hello World à une TODO application

Implémentation d'un TODO backend : https://www.todobackend.com/
L'API REST a été designée par l'équipe de TODO backend. Permet de comparer plein de technologies. L'équipe TODO fournit l'IHM permettant d'interroger son backend.

Pour la persistence, on n'utilise pas du JPA pure mais du JPA **Panache**.
La classe @Entity Todo extends PanacheEntity
Pas de getter/setter
Pas d'ID (Panache le génère)
Panache implémente le pattern Active/Record. Permet de déclarer une requête HQL très compacte.

Code Java permettant de lister de manière ordonnée l'ensemble des TODO :
Todo.listAll(Sort.by("order"))

En mode JVM, l'application démarre en 2,4 secondes (y compris l'exécution de 4 INSERT SQL).

Dans le fichier application.properties, on retrouve de la configuration Hibernate.
Extrait :
quarkus.datasource.url=jdbc :postgresql :rest-crud
quarkus.datasource.min-size=2

Quarkus laisse la possibilité d'utiliser l'EntityManager de JPA sans Panache.
La génération de l'application native est plus longue. La compilation native est utilisée à la fin des tests, une fois tous les tests réalisés.
L'application native pèse sur disque 56 Mo : JVM + Drivers PostgreSQL + Hibernate + JAXRS + Servlet + Panache + RestEasy + Narayana (gestionnaire de transactions). En mémoire, elle prend 26 Mo.
Extensions activées : agroal, cdi, hibernate-orm ; hibernate-validator, jdbc-postresql, narayana-jta, resteasy, resteasy-jsonb.

La compilation a pris du temps car toutes les entités Hibernate ont été scannées.

Reactive all the things

L'objectif de Quakus est de construire un système réactif.
Dans le Reactive Manifesto, sont définis les systèmes Reactive.
Une API comme RxJava 2 ou Reactor permet de construire une application Event Driven. On est dans l'application. Reactive Systems rend réactif l'ensemble des microservices.

Comment une application Quarkus interagit avec Kafkfa ?
Dans la spécification MicroProfile « Reactive Messaging », on retrouve 2 annotations :

- @Outgoing : publie des messages dans Kafka (dans la démo, un message est inséré toutes les 2 secondes)
- @Incoming : permet de lire des messages sur les topics

La spécification s'appuie sur Reactive Streams.

Démo avec un endpoint REST /payment Vert.x qui envoie une valeur sur le channel « internal » puis qui renvoie le tout dans un topic Kafka « transactions ». Dans le code, il n'y a aucune dépendance avec Kafka. On peut donc passer de manière transparente à RabbitMQ ou AMQP (en cours).

Autre démo avec une IHM Server Send Event / Web Socket affichant les transactions.

Les tests

Support de JUnit 4 et 5 :

- @QuarkusTest
- @SubstrateTest

Plusieurs types de tests :

- REST
- Bean : injection
- Infrastructure : CRUD, Kafka
- Tests natifs

@QuarkusTest va démarrer l'application et exécuter les tests.
@TestContainer est utilisé pour démarrer la base de données PostgreSQL.

Pour les tests natifs, nécessité de passer par l'annotation @SubstrateTest : l'application native est buildée et démarrée, puis les tests sont joués.

Conclusion

1. Quarkus rend heureux le développeur (ex : hot reload)
2. Ça boost : temps de démarrage x100
3. Choix de la stack : reactive vs impérative
4. S'appuie sur des frameworks du marché

Questions / Réponses

Quelle est la différence entre les 2 versions de GraalVM : communautaire et entreprise ?
Quarkus fonctionne sur les 2 versions.
La version entreprise de GraalVM ajoute des symboles (sinon utilisation de gdb) et apporte un profiler basé sur un émulateur de JIT.

Peut-on utiliser Quakus avec Spring Boot ?
Quarkus remplace Spring Boot. Il existe une extension Spring DI permettant d'injecter des beans Spring. Une alternative à Spring Data est en cours de dév.

Quarkus sait-il lire des propriétés au runtime (ex : username / password) ?
Oui, il gère les 2 modes : propriété au build time vs runtime

Quel est le lien entre Quakus et le Microprofile ?
Quarkus s'appuie sur les Microprofiles : JPA, JAX-RS. Cependant, Quarkus va un peu plus loin.

Quelles sont les limitations du hot-reload de Quarkus ?
Le hot-reload ne fait pas comme JRebel : le classloader est changé et l'application redémarrée en moins de 200 ms.