La JVM et Docker, vers une symbiose parfaite

Speakers : David Delabasse (Oracle)
Format : Conférence
Date : 19 avril 2019

Les conteneurs sont utilisés par plus de la moitié des développeurs (sondage StackOverflow).
David rappelle la différence entre un conteneur et une VM qui a besoin d'un hyperviseur.
Les **Kata containeurs** ne sont pas des conteneurs, mais des VM dont le temps de démarrage est optimisé.

Côté JVM, on a plusieurs outils autour des containers : docker-maven-plugin, Jib, Testcontainers. Il y'a également des frameworks comme Quarkus.

David poursuit par une discussion autour de la latence et du comportement de la JVM dans un container.

Latence

Container : empilement de layer. Lorsqu'on démarre un container, il faut charger le container du registry : plus l'image est grande, plus cela demande du temps. Optimisation possible avec un registry co-localisé ou utilisation d'un cache.
La taille des images doit être le plus petit possible.
Généralement, une appli Java a 3 layers :

1. OS
2. Runtime Java
3. Code applicatif

Comment réduire ces layers ?
On peut commencer par faire attention aux dépendances transitives embarquées dans l'application.
Essayer de stocker tout ce qui est statique (ex : dépendances/JAR) dans un layer spécifique. Elles évoluent très peu et pourront être réutilisées du cache Docker. Valable également pour les archis CDS.

Démo avec OpenJDK

Dockerfile:
FROM openjdk:11
WORKDIR .test
COPY HelloWorld.java .
RUN javac HelloWorld.java
CMD java HelloWorld

Quelques commandes :
docker build -t test .
docker run test

Taille de l'image : 815Mo pour un HelloWorld

David va essayer de réduire la taille de l'image
Il utilise openjdk:11-slim comme image de base. La taille de l'image passe à 511Mo

Ensuite, pour exécuter une application, on n'a pas besoin d'un JDK.
Passage à un JRE slim avec du multi-stage build

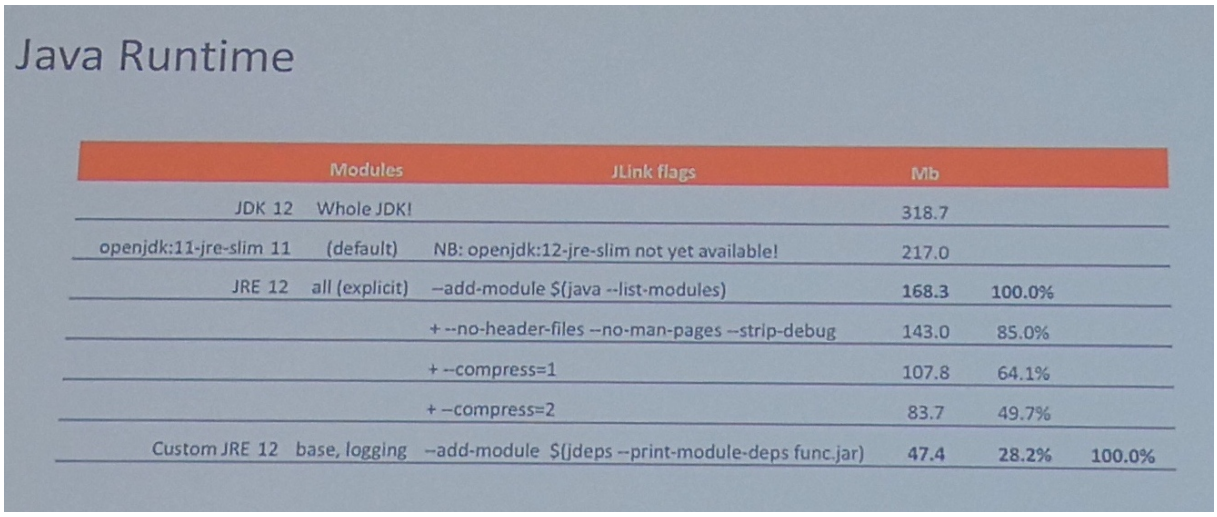
FROM openjdk:11-slim as build
…
FROM openjdk:11-jre
…
Taille de l'image : 273Mo

1ière conclusion : utiliser le JRE à la place du JDK.
La taille est réduite et la surface d'attaque potentielle est réduite.

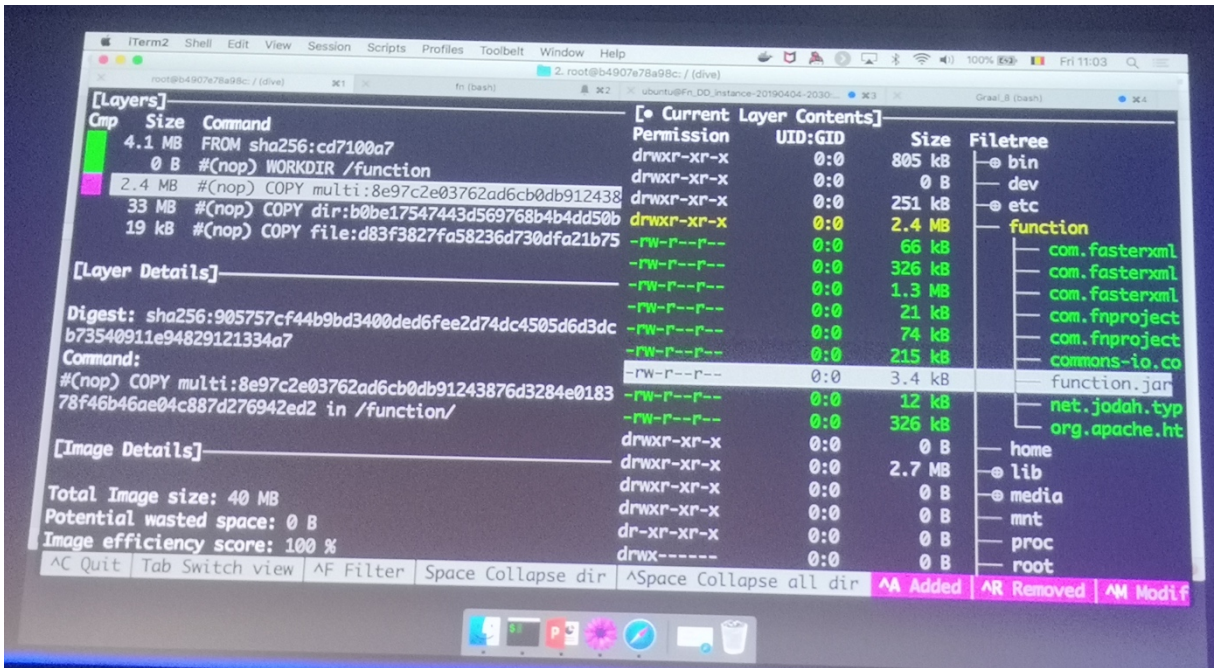
On peut changer d'OS et passer sur Alpine.

On peut ensuite créer son propre JRE (slide).
On descend à 32 Mo (avec compression).
La décompression a un cout de latence au démarrage de l'application.

Démo avec Fn qui est une plateforme FaaS.
Déploiement d'une fonction. Le tooling de Fn utilise Docker pour containeriser la fonction.
L'image pèse 40 Mo.



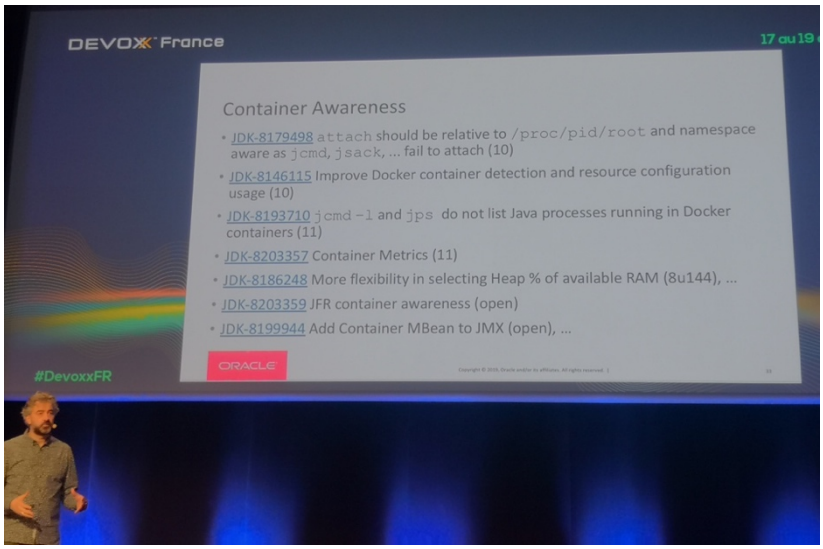
David nous montre l'usage de l'outil Dive pour afficher les layers Docker.



Le Class Data Sharing (CDS) est présent depuis Java 5.
Il permet de réduire l'empreinte mémoire entre JVM en partageant les méta-données.
La représentation mémoire des méta-données est persistée sur disque.

java -Xshare:on
java -Xshare:dump -> création d'un fichier de méta-donnnées
Javac HelloWorld.java
Constat : 111ms sans CDS à 77ms avec CDS
CDS est à utiliser. Mais cela augmente la taille de l'image (+20Mo). Le bénéfice vaut le coup.

Dans le JDK Oracle 9, on peut utiliser CDS au niveau de l'application. Open Sourcé en OpenJDK 11.
Le JEP 350 vise à créer l'archive à la fin de l'exécution de l'application. On l'aura peut-être en Java 13 ou 14.



Pour aller plus loin, on peut utiliser GraalVM. C'est une machine virtuelle open source polyglotte développée par Oracle. Elle permet de générer un binaire à partir de code Java.

Javac HelloWorld.java
native-image HelloWorld => generation d'un binaire en 20 sec

Avec Docker, l'idée est de faire un maximum de choses pendant le build pour améliorer les temps de démarrage.
L'exécutable pèse 8 Mo. L'exécution prend 3 ms. Il n'y a plus de Java Runtime séparé. Substrate VM l'a remplacé. Le GC n'est pas le même.
Pour autant, GraalVM a des limitations et des inconvénients (ex : InvokeDynamic non supporté). Actuellement, GraalVM ne supporte que Java 8.

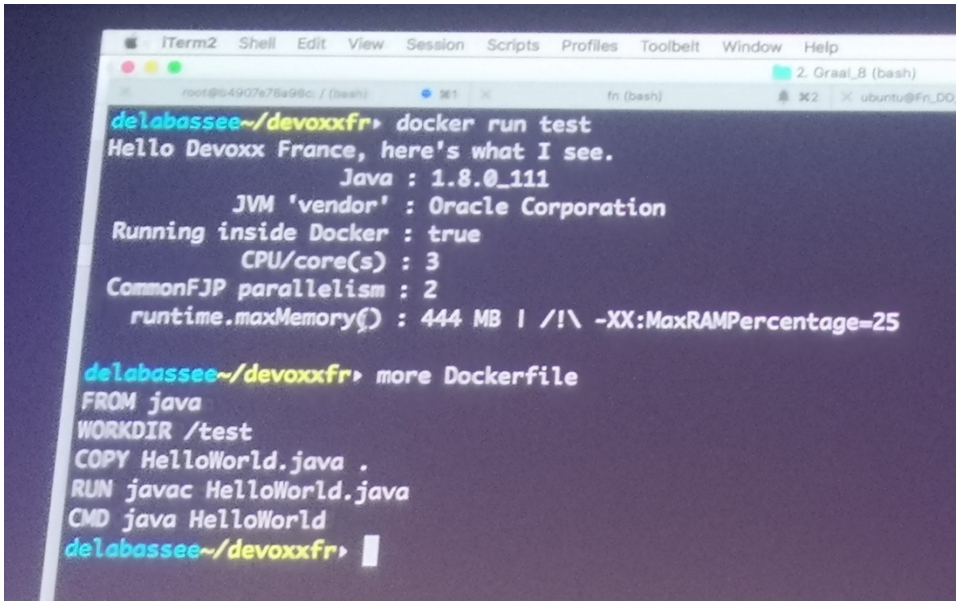
Comportement de la JVM lorsqu'elle tourne dans Docker

Depuis Java 8, l'intégration de Docker dans la JVM a été constamment améliorée.
La JVM comporte 659 flags.
Ergonomics permet à la JVM de s'auto-tuner. Elle regarde la plateforme et le nombre de cœurs.
Quel GC doit-on utiliser lorsque le container est éphémère ? Possibilité d'utiliser Epsilon – No-Op Garbage Collector : ne désalloue rien. Serial GC n'a pas un gros surcout. Préférer Serial GC à Epsilon.

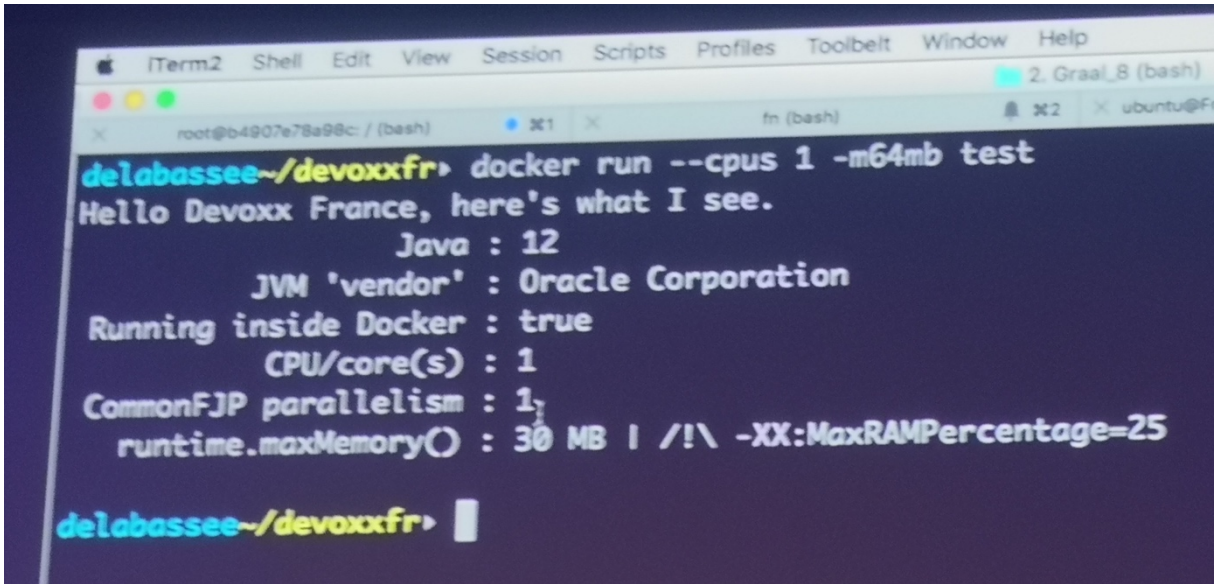
Démo à partir de l'image Docker java : 643 Mo. David déconseille d'utiliser cette image. La JVM est la 1.8.0_111. Vielle image non maintenue. Préférer celle d'OpenJDK.

docker run –cpus 1 test
Permet de limiter le nombre de CPU dans Docker, à condition d'utiliser une JVM récente. Sinon, Docker voir les 3 CPUs du Host. De même pour la mémoire.

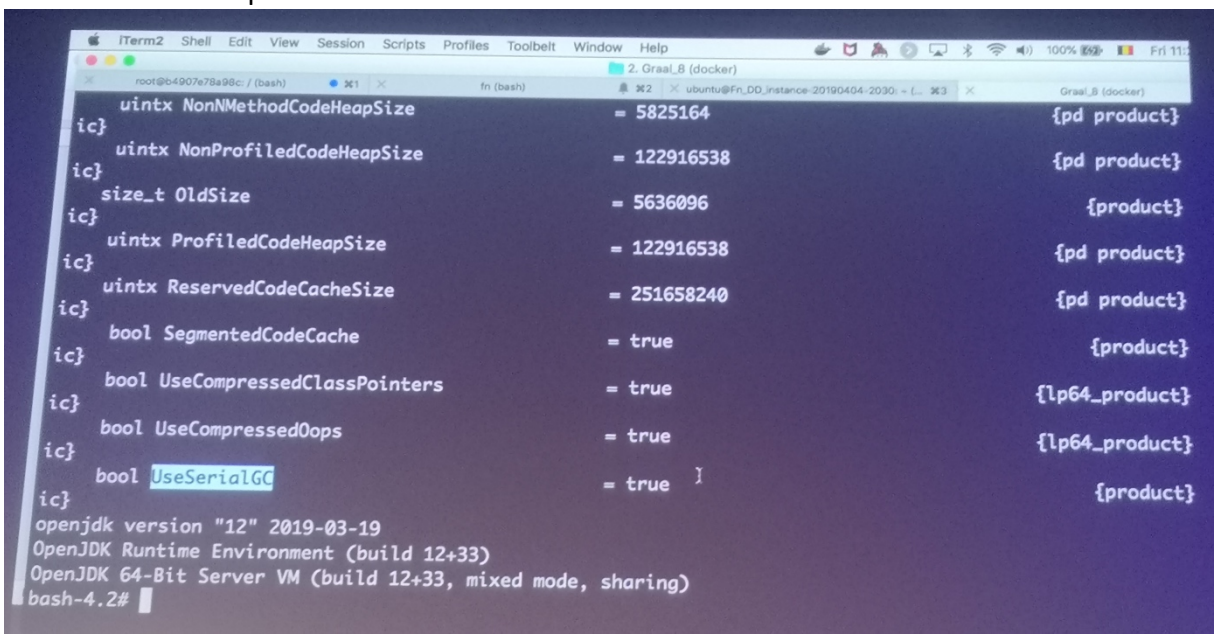
Avec une ancienne version de Java 1.8.0_111 ne supportant pas encore Docker :



David repasse de l'image Orcacle Java 8 à une image OpenJDK 12.
docker run –cpus 1 -m64mb test
Le nombre de CPU 1 et le maxMemory à 30Mo (25% en pratique) est pris en compte.



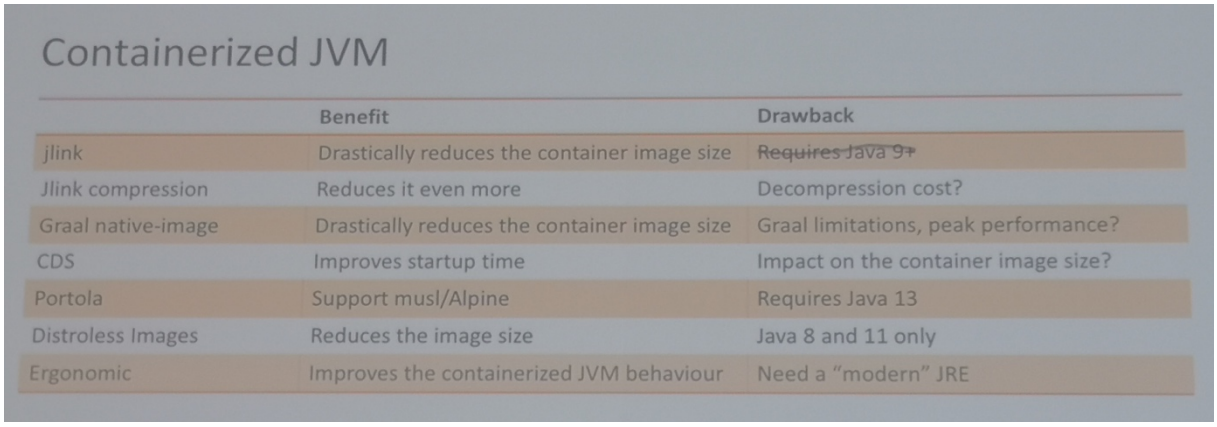
java -X :+PrintFlagsFinal -version
Permet de voir les flags auto-configurés
Le GC SerialGC est préféré à G1.



Conclusion

- Latence : startup time du container et de l'application => nécessite de réduire la taille de l'image. Avec peu d'effort, on arrive à réduire la taille (40 à 60 Mo)
- La JVM est capable de détecter qu'elle tourne dans un conteneur
- Faire attention à l'image de base utilisée : utiliser une version sécurisée. Plusieurs Outils sont à notre disposition : Docker-bench-security, Snyk, Clair, Anchore …

- Utiliser la dernière version de Java. Si on ne peut pas, utiliser une version de Java activement supportée
- Essayer de faire le maximum de chose lors du build
- Réduire au maximum la taille de l'image : utiliser le multi-stage build et Dive



Containerized JVM

	Benefit	Drawback
jlink	Drastically reduces the container image size	~~Requires Java 9+~~
Jlink compression	Reduces it even more	Decompression cost?
Graal native-image	Drastically reduces the container image size	Graal limitations, peak performance?
CDS	Improves startup time	Impact on the container image size?
Portola	Support musl/Alpine	Requires Java 13
Distroless Images	Reduces the image size	Java 8 and 11 only
Ergonomic	Improves the containerized JVM behaviour	Need a "modern" JRE