Florian Nari

Veille Technologique - Quarkus

Résumé

Introduction

Aujourd'hui, avec l'avènement du Cloud, la hausse du prix de l'électricité, et le changement climatique, les performances sont de plus en plus importantes. Java a été développé au début des années 90, sans vraiment prendre en compte ces problématiques. Aujourd'hui, Java garde une image de plateforme très lourde, lente, et qui demande beaucoup de ressources.

GraalVM AOT (développé par Oracle, comme Java) veut régler ce problème en compilant l'application en langage machine. En effet, si Java est considéré comme lent, c'est parce que le code est exécuté au sein d'une « JVM » (quasiment comme un langage interprété). GraalVM AOT promet ainsi un démarrage plus rapide, une empreinte mémoire allégée, et des performances améliorées.

En Java, le principal Framework est Spring. Il est utilisé notamment pour l'injection de dépendances, pour les API REST, et pour la persistance des données. Malheureusement, une application Spring peut-être très longue à démarrer (environ une minute pour les applications que je développe au travail par exemple). En plus, Spring n'est pas compatible avec GraalVM AOT (à part à titre expérimental).

Le nouveau Framework Quarkus a pour but de permettre une vitesse de démarrage accélérée, et une compatibilité avec GraalVM AOT. Quarkus est développé par Red Hat, une entreprise très active dans l'univers du logiciel libre à travers des distributions linux (Red Hat Enterprise Linux, CentOS, Fedora), des serveurs d'applications Java (WildFly/JBoss), et la conteneurisation (OpenShift, CoreOS/Container Linux).

Ainsi, en passant d'une application Spring classique à une application Quarkus/GraalVM AOT, on devrait gagner en temps de démarrage, en vitesse d'exécution, et en utilisation de la mémoire.

Processus de veille

Pour vérifier ça, j'ai développé une application Spring complète (avec API REST, injection de dépendances, et persistance des données) et son équivalent avec Quarkus. Puis j'ai utilisé Apache JMeter pour faire des tests de performance. (Les durées sont en seconde)

Plateforme	Framework	Démarrage	Plan de test
JVM HotSpot	Spring	4.9	305
	Quarkus	1.7	79
GraalVM EE 22.1 AOT	Quarkus	0.0	74
GraalVM CE 22.2 AOT Pré-release*	Spring Native	0.2	327

Ensuite, au vu du faible écart de performance entre GraalVM AOT et la JVM HotSpot, j'ai développé une application simple avec beaucoup de calculs. (Les durées sont aussi en secondes)

JVM HotSpot	57
GraalVM EE 22.1 AOT	61

Là encore, l'écart de performances n'est pas flagrant.

Ainsi, j'en ai déduit qu'actuellement, sous Windows, une application compilée avec GraalVM AOT est plus rapide à démarrer qu'une application Java classique, mais n'est pas plus rapide à l'exécution. Mais Quarkus en lui-même est plus rapide (à démarrer et à s'exécuter) que Spring.

Table des matières

Florian Nari

Introduction

Avant de commencer, je précise que le dépôt git associé à ce rapport se trouve ici https://github.com/floriannari/veille-technologique-Quarkus.

Quarkus est un Framework java conçu pour fonctionner avec la JVM, mais aussi avec GraalVM AOT.

La JVM HotSpot

Habituellement, les applications écrites en Java sont « compilées » sous forme de bytecode Java. Ce bytecode est ensuite interprété par la JVM HotSpot (machine virtuelle java).

Cela permet à Java d'être multiplateforme. C'est une des plus grandes forces de Java. D'où le slogan « Write once, run anywhere ».

Mais en contrepartie, les performances sont considérées comme moins bonnes par rapport à des langages réellement compilés en code machine (comme le C/C++).

GraalVM

GraalVM est une alternative à la JVM HotSpot.

GraalVM supporte plusieurs langages de programmation (Java, JavaScript, Ruby, R, Python, WebAssembly…). Et permet au code écrit dans ses différents langages de communiquer ensemble dans le même espace mémoire.

Mais la fonctionnalité de GraalVM qui nous intéresse ici, c'est la compilation en code machine (GraalVM AOT). Cela permet de s'affranchir de la JVM. Et donc théoriquement de gagner en performances.

Framework

Un Framework est un « cadre » dans lequel on va développer notre application.

C'est le Framework qui va appeler notre code (à l'inverse d'une bibliothèque qui elle est appelée par notre code).

Le Framework java le plus connu est Spring. Il dispose de nombreuses extensions pour faire des API REST, des pages web, pour persister des données…

Quarkus

Quarkus se veut comme un concurrent de Spring spécialement conçu pour les microservices. En effet, Quarkus a été développé pour être rapide à démarrer et à s'exécuté, et pour être économe en mémoire.

Utilisation de Quarkus

Prérequis

Pour créer et exécuter une application Quarkus, il faut avant tout avoir Java d'installé.
Personnellement, j'utilise Java 17[1]. Pour le développement, j'utilise Eclipse[2], mais n'importe quel IDE fait l'affaire. Quarkus propose une extension[3] pour les principaux IDE (Eclipse, VS Code et IntelliJ), mais je l'ai trouvé assez inutile. (Mais j'imagine qu'au fil des années, ils vont compléter cette extension avec de nouvelles fonctionnalités.)

De préférence, il faut aussi avoir Maven d'installé[4] (on peut faire sans, mais je préfère installer Maven plutôt que d'avoir des scripts à la base de chaque projet).

Pour tester mes applications, il faudra aussi avoir docker d'installé[5].

Installation

Pour créer un projet Quarkus, le plus simple et d'utiliser les lignes de commande. Personnellement je l'ai simplement installé avec Chocolatey (un gestionnaire de paquets sous Windows, c'est un équivalent à APT/RPM/Pacman sur linux). Les méthodes d'installation du CLI Quarkus sont expliquées ici https://quarkus.io/guides/cli-tooling#installing-the-cli.

Pour compiler en une application native, il faudra aussi installer GraalVM et son extension native-image. Les différentes versions de GraalVM seront détaillées plus bas, ainsi que comment les installer.

Utilisation

Créer une application Quarkus

Ce document est accompagné d'une vidéo montrant comment créer une API REST avec Quarkus.

Ensuite, pour créer une application, il faut utiliser la commande

```
quarkus create app [groupId]:[getting-started]
```

Tout est expliqué ici https://quarkus.io/guides/getting-started.

Une fois que le projet Maven est créé, on peut l'importer avec son IDE préféré.

Il faut savoir qu'avec Quarkus, il n'y a pas de « main » comme avec une application classique (même avec Spring). On n'a pas non plus besoin de lui spécifier où trouver les entités à persister, les contrôleurs, les services… (contrairement à Spring).

Pour exécuter l'application pendant qu'on développe, on peut soit utiliser la commande « quarkus dev », soit créer une « run configuration depuis Eclipse »[6] (c'est plus lourd à mettre en place que pour les applications Spring où il suffit d'appuyer sur un bouton si on veut lancer l'application depuis Eclipse. Je considère que ça fait partie du manque de maturité de Quarkus.)

---

[1] https://adoptium.net/
[2] https://www.eclipse.org/downloads/
[3] https://quarkus.io/guides/ide-tooling
[4] https://maven.apache.org/install.html
[5] https://docs.docker.com/desktop/install/windows-install/
[6] https://quarkus.io/blog/eclipse-got-quarkused/#run-application

Mise à part ça, tout se passe comme avec Spring (mais les annotations sont différentes).

Sur le dépôt git, on peut voir les deux applications que j'ai fait, une avec Spring, et une autre avec Quarkus. On peut voir qu'au final, elles sont assez similaires. (À noter que pour la persistance j'ai utilisé le Pattern « active record »[7], mais j'aurais très bien pu utiliser le pattern « repository »[8] comme sur Spring).

Quarkus propose aussi des guides avec des exemples concernant l'injection de dépendances[9], la persistance[10], les API REST[11] et les tests[12].
(Pour voir la liste complète de tous les guides, c'est ici https://quarkus.io/guides/)

Migrer une application Quarkus depuis Spring

Pour cette section, je me suis basé sur la page Quarkus for Spring Developers[13], et sur les guides concernant la compatibilité avec Spring[14].

Le moyen le plus simple pour convertir une application Spring en application Quarkus est d'utilisée les extensions Quarkus pour Spring.

Dans le pom.xml, on va simplement remplacer les dépendances Spring par leurs équivalents Quarkus, et ajouter les dépendances « quarkus-spring-X » pour faire le lien. (Ainsi, pour schématiser, un import de « org.springframework.beans.factory.annotation.Autowired » va implicitement référer « javax.inject.Inject » de manière transparente).

Cela fonctionne pour les principales fonctionnalités comme par exemple les injections de dépendance[15] (@autowired), les API web[16] (@RestController) et Spring Data JPA[17] (@JpaRepository).

Mais Spring étant un Framework extrêmement complet, cela ne fonctionne pas avec chacune des fonctionnalités permises par Spring.

À noter qu'il faudra aussi modifier les classes de test et les fichiers de configuration (application.properties/application.yml), et supprimer la classe contenant le main. Dans les faits, j'ai aussi dû modifier un controller. En effet, avec Spring, je n'avais pas besoin de préciser l'annotation « @RequestParam » sur le message en paramètre de « addMessage ».

Ainsi, il suffit de modifier quelques fichiers pour convertir une application Spring basique avec Quarkus (et donc améliorer potentiellement ses performances). Ensuite on peut compiler et exécuter l'application comme n'importe quelle application Quarkus.

---

[7] https://quarkus.io/guides/hibernate-orm-panache#adding-entity-methods
[8] https://quarkus.io/guides/hibernate-orm-panache#defining-your-repository
[9] https://quarkus.io/guides/cdi-reference
[10] https://quarkus.io/guides/hibernate-orm-panache
[11] https://quarkus.io/guides/rest-json
[12] https://quarkus.io/guides/getting-started-testing
[13] https://quarkus.io/blog/quarkus-for-spring-developers/
[14] https://quarkus.io/guides/#compatibility
[15] https://quarkus.io/guides/spring-di
[16] https://quarkus.io/guides/spring-web
[17] https://quarkus.io/guides/spring-data-jpa

Tester mes applications

Il y a 4 applications principales : « Quarkus », « Spring-Native », « Spring-with-Quarkus », « Spring », et « Spring-Native ». Elles font toutes exactement la même chose (mais en utilisant des technologies différentes).

Pour celles-ci, il faudra d'abord avoir configuré la BDD.

Créer la BDD

Pour tester mes applications, il faut exécuter la commande « docker-compose up » depuis la racine de mon dépôt git. Cela créera une base de données PostgreSQL sur le port 5433.

Initialiser la BDD

Maintenant que la base de données est créée, il faut lui donner sa structure. Pour cela, il faut modifier les fichiers de configurations d'une des applications, pour lui dire de créer le schéma de données (par exemple avec Quarkus, il faudra passer « quarkus.hibernate-orm.database.generation » à « create ». Avec Spring ce sera « spring.jpa.hibernate.ddl-auto » qu'il faudra passer à « create ».)

Exécuter les applications

J'ai créé un dossier « executables » dans le dossier de chaque projet. Pour les projets Quarkus, il contient un exécutable, et un dossier contenant un jar. Pour le projet « Spring », il contient un jar, et pour le projet « Spring-Native », il contient un exécutable.

Construire une image native

https://quarkus.io/guides/building-native-image

Qu'est-ce que GraalVM ?

Comme évoque tout à l'heure, GraalVM est une JVM alternative permettant la compilation anticipée.

D'après Wikipédia[18] « La compilation anticipée permet d'avoir une vue d'ensemble du code, ce qui n'est pas le cas d'une compilation à la volée qui ne peut faire que des optimisations locales. La compilation anticipée permet de diminuer les coûts de traitements des exceptions et les appels aux méthodes et interfaces. »

[18] https://fr.wikipedia.org/wiki/Compilation_anticipée

Versions de GraalVM

GraalVM

Il existe 2 versions de GraalVM.

Compare Editions

	GraalVM Community	GraalVM Enterprise
License	GNU General Public License V2 with the "Classpath" Exception	Oracle Technology Network (OTN) license for dev/test; Commercial license for production deployments
Base JDKs	OpenJDK 11.0.13 and 17.0.1	Oracle JDK 8u311, 11.0.13 and 17.0.1
Support	Community support via public channels	Global 24x7 Enterprise support from Oracle
Speedup vs. OpenJDK on Renaissance Suite	1.04x	1.3x
Native Image performance vs. OpenJDK (OPs per GB/sec)	0.82x	1.34x

GraalVM Community

GraalVM CE[19] est la version libre et gratuite de GraalVM.

GraalVM Entreprise

GraalVM EE[20] est la version payante de GraalVM.

(Cette version est payante si on l'utilise en production, mais on peut très bien la télécharger gratuitement pour un usage personnel, ou pour une simple démonstration.)

Mandrel

Mandrel[21] est un fork de GraalVM CE spécialement conçu pour Quarkus. Il diffère de GraalVM CE par l'exclusion de fonctionnalités inutiles pour Quarkus (comme la prise en charge des autres langages de programmation), et par l'inclusion de l'extension « native image ».
Il n'est pas compatible Windows/MacOs (hors Docker).

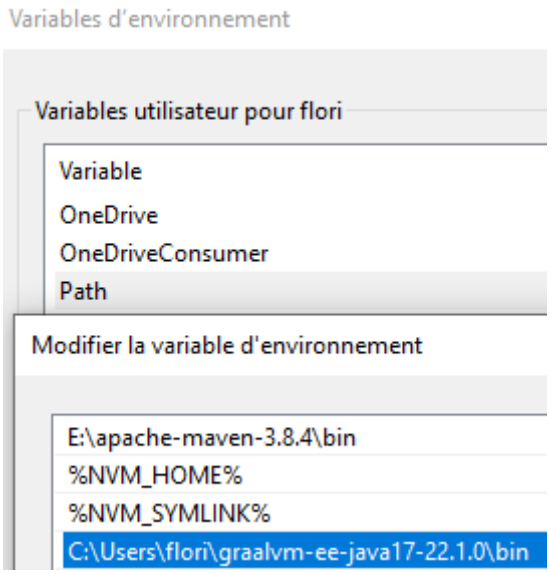
[19] https://www.graalvm.org/downloads/
[20] https://www.oracle.com/downloads/graalvm-downloads.html
[21] https://github.com/graalvm/mandrel

Installation de GraalVM sur Windows

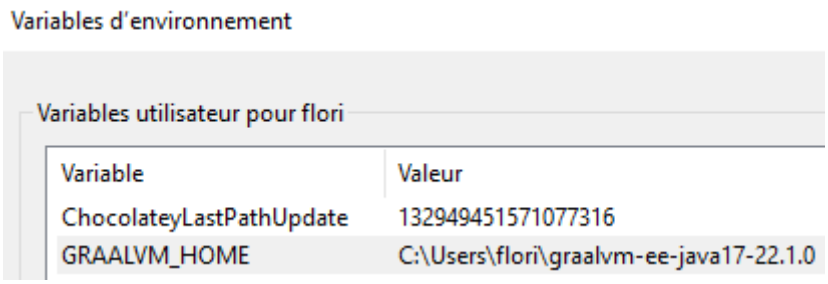
Il faut d'abord télécharger et extraire le fichier zip dans le dossier d'installation.

> Pour la version CE -> https://github.com/graalvm/graalvm-ce-builds/releases/
>
> Pour la version EE -> https://www.oracle.com/downloads/graalvm-downloads.html (il faudra créer un compte oracle si ce n'est pas déjà fait.)

Il faut ensuite ajouter le dossier « /bin » à la variable d'environnement PATH.



Pour Quarkus, il faudra aussi ajouter une variable d'environnement « GRAALVM_HOME » contenant le dossier d'installation.



Installer native image

Maintenant, on doit normalement pouvoir exécuter la commande

> « gu install native-image »

pour installer « native image » (la fonctionnalité de GraalVM qui nous intéresse).

Installer Visual Studio 2017 Visual C++ Build Tools

Sous Windows, pour pouvoir compiler nativement en C, Quarkus demande d'installer Visual Studio C++ Build Tools https://aka.ms/vs/15/release/vs_buildtools.exe

Construire l'image native

Ensuite, il suffit d'exécuter une commande

Pour linux

> « quarkus build –native »

Pour Windows

> « cmd /c 'call "C:\Program Files (x86)\Microsoft Visual Studio\2017\BuildTools\VC\Auxiliary\Build\vcvars64.bat" && mvn package -Pnative -DskipTests' »

C'est beaucoup plus long qu'une « compilation » classique en Java (environ 4 minutes, contre moins d'une minute pour générer un Jar).

Benchmark

Une fois les applications développées, il a fallu tester leurs performances.

Je voulais tester les performances des API REST, pas uniquement les performances des services. Donc c'est quelque-chose de tout nouveau que je n'avais jamais fait avant.

J'ai d'abord hésité entre Apache JMeter, et la création d'un Framework de test maison.

J'ai pensé qu'il serait plus propre d'utiliser JMeter, et que ça pourrait me resservir plus tard. J'ai donc installé JMeter et regardé quelques tutos.

JMeter

JMeter est un logiciel de tests de performances. Il est libre et est développé par la fondation Apache.

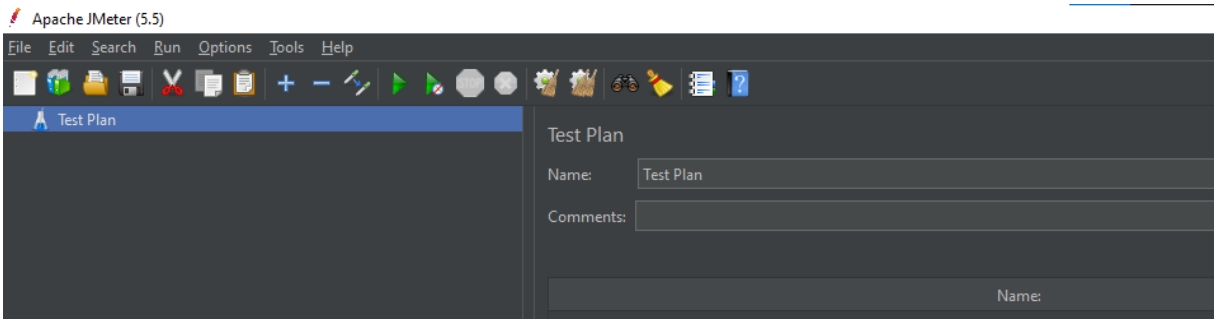
Installation

Pour commencer, il faut télécharger JMeter si ce n'est pas déjà fait.

https://jmeter.apache.org/download_jmeter.cgi#binaries

Ensuite, une fois dézipper, on peut le lancer en exécutant « bin/ApacheJMeter.jar ».

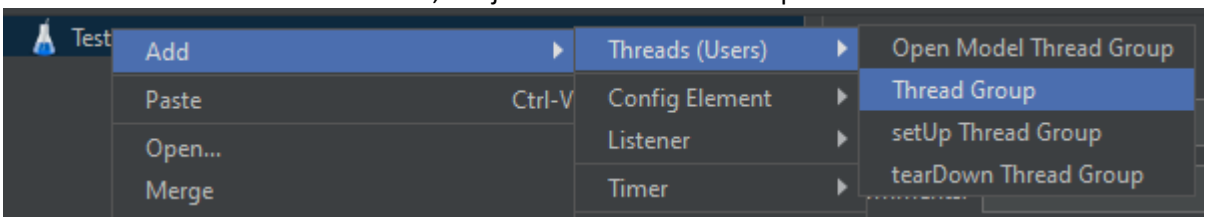
On arrive ainsi sur cette fenêtre.



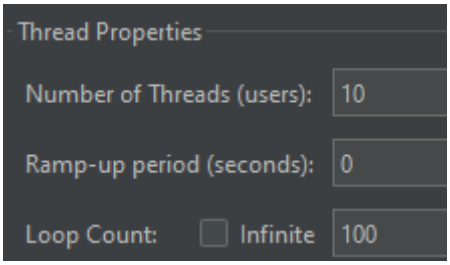
Utilisation

Thread Group

Il faut faire un clic droit sur Test Plan, et ajouter un « Thread Group » :



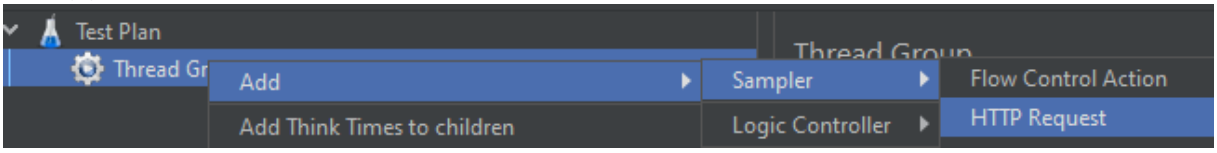
C'est ici qu'on pourra configurer le nombre de Threads, et le nombre de fois que chaque Thread exécutera le scénario.



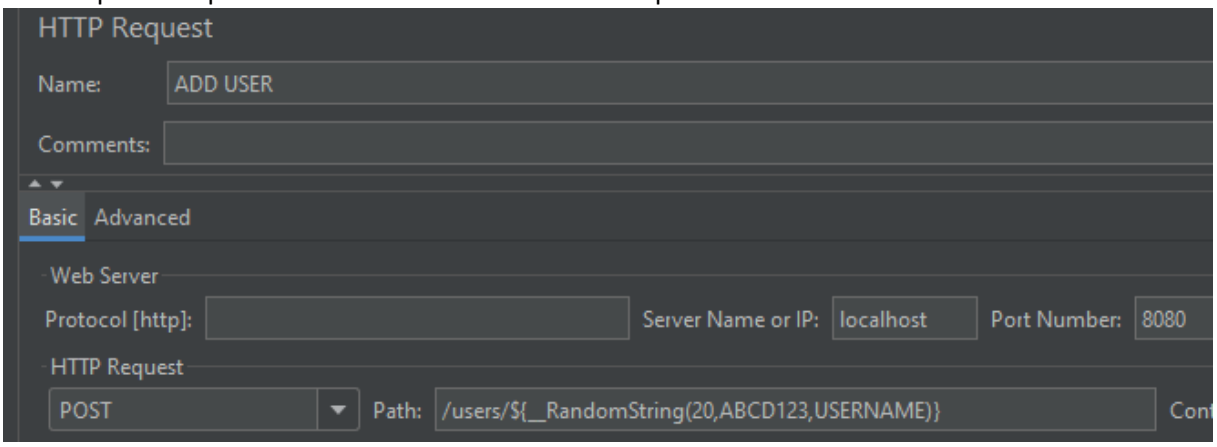
Par exemple ici, 10 Threads effectueront le scénario 100 fois chacun. Soit un total de 1 000 itérations.

Requêtes HTTP

On peut maintenant faire un clic droit sur le thread Group nouvellement créé, et ajouter des requêtes HTTP(S) :



Puis on peut remplir les informations concernant la requête :

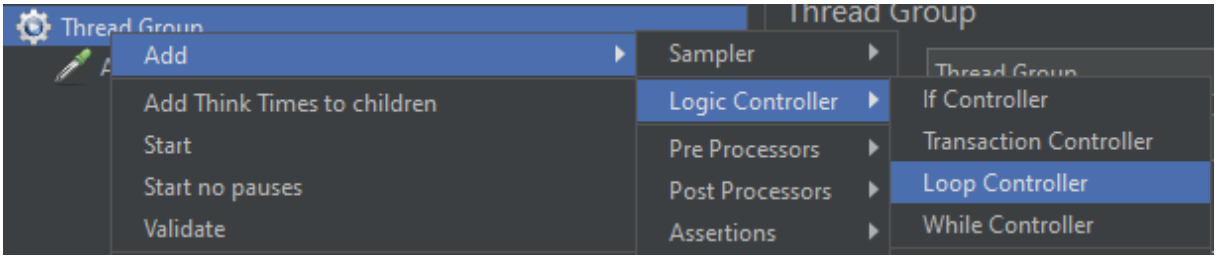


Ici, on envoie une requête « POST » à l'url « localhost:8080/users/{username} ». À noter qu'ici, l'username sera une chaine d'une longueur de 20 caractères composés des caractères « A », « B », « C », « 1 », « 2 », et « 3 ». Il sera ensuite stocké dans une variable locale « USERNAME » (cette variable est indépendante pour chaque thread).

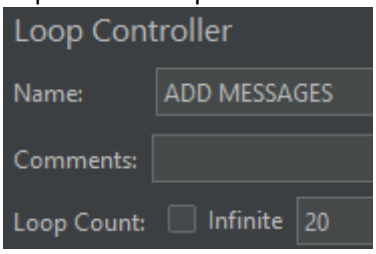
Ainsi, on créé un user avec un username aléatoire.

Boucles

Pour mes tests, je veux envoyer plusieurs messages pour chaque utilisateur, j'ai donc besoin de faire une boucle :



Je peux ensuite préciser combien de fois je veux boucler.



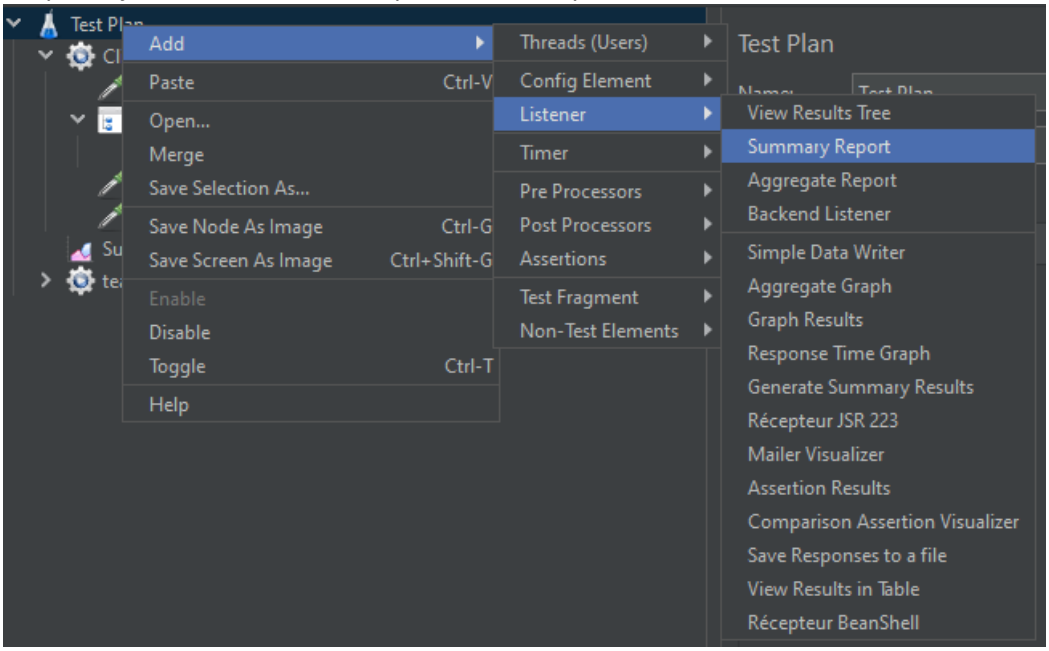
Je peux maintenant faire un clic droit sur ma boucle, et ajouter ma requête HTTP comme vu précédemment.

Rapports

Une fois qu'on a construit notre scénario, on peut l'exécuter en cliquant sur les flèches vertes

mais cela ne nous donne pas beaucoup d'informations.

On peut ajouter des « Listener » pour en savoir plus sur le déroulement des scénarios.



Par exemple avec les Summary Report, on peut connaitre la durée moyenne de chaque requête (en milliseconde).

Label	Average
ADD USER	73
ADD MESSAGE	24
GET MESSAGES	26
GET USERS	10

Résultats

Configuration

Les tests de performance ont été réalisé sur un PC disposant d'un processeur Intel Core i5-4690k disposant de 4 cœurs cadencés à 4.2 GHz, accompagné de 16 Go de mémoire DDR3 cadencée à 1600 MHz. Celui-ci fonctionne sous Windows 10.

Résultats

Durée en seconde avec 10 threads exécutants la boucle 100 fois chacun.

Plateforme	Framework	Démarrage	Plan de test
JVM HotSpot	Spring	4.9	305
	Quarkus	1.7	79
GraalVM EE 22.1 AOT	Quarkus	0.0	74
	Spring with Quarkus	0.0	75
GraalVM CE 22.2 AOT Pré-release*	Spring Native	0.2	327

*Concernant Spring Native, je n'ai pas pu l'utiliser avec GraalVM EE 22.1 (la dernière version), ni avec GraalVM 21.3.2 (la version LTS). Sur les Issues GitHub, j'ai vu que mon problème était connu et sera corrigé avec la version 22.2 (dont la sortie est prévue le 19 juillet 2022).

En attendant j'ai téléchargé la « pré-release » de graalVM CE 22.2 (et cela fonctionne). Mais je n'ai pas pu tester Quarkus avec cette version car Quarkus ne supporte pas encore cette version de GraalVM.

Donc si les performances de Spring Native sont aussi mauvaises, c'est peut-être en partie à cause de la version de GraalVM utilisée (GraalVM EE étant annoncé comme 25% plus rapide que GraalVM CE).

Analyse des résultats

On se rend donc bien compte que Quarkus et GraalVM augmentent significativement la vitesse de démarrage. On remarque aussi que Quarkus est plus rapide que Spring.

En revanche, GraalVM AOT ne semble pas améliorer les performances par rapport à la JVM. En voyant ces résultats, je pensais que c'est dû à l'application testée. En effet, dans mon application il y a des API REST, des injections de dépendances, et de la persistance, mais il n'y a pas vraiment de calculs.

Deuxième Benchmark

J'ai donc décidé de faire une autre application pour tester la distance de Levenshtein[22] entre 2 mots. Pour cela j'utilise la librairie Apache Commons Text[23]. (Il s'agit d'un algorithme avec une complexité spatiale et temporelle de « O(n * m) » avec « n » et « m » la longueur des mots à comparer.

Ainsi, là où le benchmark précédent servait surtout à comparer les performances de Quarkus avec Spring, ici on va pouvoir comparer les performances de la GraalVM AOT par rapport à la JVM HotSpot.

Ainsi, voici la durée en seconde de 6*1 000 fois le calcul de la distance de Levenshtein entre 2 chaines de 100 000 caractères.

JVM HotSpot	57
GraalVM EE 22.1 AOT	61

Ainsi, on se rend compte que même pour de lourds calculs, la compilation anticipée ne permet pas de gagner en performances. Je pense que cela est dû à la compilation à la volée (JIT[24]) qui rend le langage java si puissant. Au fur et à mesure que les fonctions sont exécutées, la JVM va les optimiser (en faisant de l'inlining, en les compilant…) tout en prenant compte des caractéristiques de la machine (là où la compilation anticipée doit « deviner » dans quelles conditions l'application sera exécutée).

À ce sujet, dans un article[25] de 2003 mis à jour en 2016 concernant les performances de Java par rapport à C++, J.P.Lewis et Ulrich Neumann (du Laboratoire d'infographie et de technologie immersive de l'Université de Californie du Sud) disaient ceci à propos de la compilation à la volée :

> The JIT compiler knows more than a conventional "pre-compiler", and it may be able to do a better job given the extra information:
>
> - The compiler knows what processor it is running on, and can generate code specifically for that processor. It knows whether (for example) the processor is a PIII or P4, if SSE2 is present, and how big the caches are. A pre-compiler on the other hand has to target the least-common-denominator processor, at least in the case of commercial software.
> - Because the compiler knows which classes are actually loaded and being called, it knows which methods can be de-virtualized and inlined. (Remarkably, modern java compilers also know how to "uncompile" inlined calls in the case where an overriding method is loaded after the JIT compilation happens.)
> - A dynamic compiler may also get the branch prediction hints right more often than a static compiler.

[22] https://fr.wikipedia.org/wiki/Distance_de_Levenshtein
[23] https://commons.apache.org/proper/commons-text/apidocs/org/apache/commons/text/similarity/LevenshteinDistance.html
[24] https://en.wikipedia.org/wiki/Just-in-time_compilation and https://developers.redhat.com/articles/2021/06/23/how-jit-compiler-boosts-java-performance-openjdk
[25] http://scribblethink.org/Computer/javaCbenchmark.html

Conclusion

Ainsi, Quarkus est plus rapide que Spring. Mais sous Windows, la différence de vitesse (en dehors du démarrage) entre une application JVM et une application compilée semble négligeable.

Il faudrait faire des essais supplémentaires sous Linux, et en mesurant la quantité de mémoire utilisée, pour confirmer que la compilation anticipée n'apporte rien (à part une durée de démarrage très réduite). Dans tous les cas, les performances s'amélioreront peut-être à l'avenir, cette technologie n'en est qu'à ses débuts.

En dehors des performances, je trouve que c'est une bonne chose que Spring ait enfin un concurrent. Mais à l'heure actuelle, Quarkus ne me semble pas assez mature pour être utilisé à grande échelle. Il permet déjà pas mal de choses, mais ça reste loin de Spring.

Donc pour l'instant, il ne me semble pas pertinent de re-former les développeurs Spring, et de migrer les applications existantes vers Quarkus.