Supersonisch Java-geweld met Quarkus

Zelf aan de slag met dit framework.

Eén van de hedendaagse focuspunten van het Java ecosysteem is het optimaliseren van het gebruik hiervan in cloud omgevingen. Vaak betaal je op basis van je gebruikte CPU en RAM. Het is dan een goed idee om consumptie hiervan te minimaliseren. Maar wie Java een beetje kent, weet dat CPU en RAM door de JVM uitbundig geconsumeerd worden. Gelukkig is de Java community oplossingen aan het zoeken, met name op het gebied van de enterprise Java frameworks. Hier komt het gloednieuwe Quarkus framework van RedHat in beeld. Wat Quarkus allemaal kan en hoe je zelf aan de slag kan gaan met dit framework, dat lees je in dit artikel.

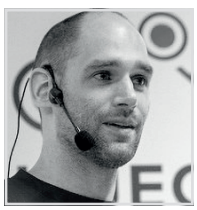
Voordat we in de details van Quarkus (referentie 1) duiken, is het belangrijk om te weten waar dit framework zijn kracht vandaan haalt. Het is eigenlijk 'gewoon' een Java-gebaseerd enterprise framework dat draait op een JVM. Echter, om daadwerkelijk CPU en RAM consumptie te optimaliseren, dan wel te minimaliseren, is het aan te raden om Quarkus te runnen op Oracle's GraalVM. Mocht je nog niet bekend zijn met GraalVM (referentie 2), dan is het raadzaam om het vorige nummer van Java Magazine (2019-02) er nog even op na te slaan. Ons artikel focust zich op Quarkus waardoor we niet verder ingaan op GraalVM. Maar weet dat Quarkus deze VM gebruikt om er het leeuwendeel van zijn efficiëntie uit te halen.

Om CPU en RAM consumptie te optimaliseren, moet je Java enterprise framework, zoals Quarkus, hier wel op voorbereid zijn. Dit is iets wat in frameworks, zoals Jakarta EE gebaseerde applicatieservers en Spring (Boot), nog niet mogelijk is/was vanwege het toepassen van dynamische class loading en reflection. In de Spring Boot community zijn ze bijvoorbeeld binnen het Sprung Fu incubatieproject hard aan het werk om Spring Boot met GraalVM te kunnen draaien. Andere Java gebaseerde enterprise frameworks, zoals Micronaut en Helidon, zijn vanaf het begin al ontworpen om met een declaratieve programmeerstijl deze problemen te voorkomen.

Wat is er dan zo speciaal aan Quarkus als GraalVM het werk verzet? Quarkus is namelijk een framework dat voortkomt uit de Jakarta EE community, wat net als de eerder genoemde frameworks is toegespitst op het bouwen van enterprise applicaties en microservices in een cloud omgeving. Om het draaien van een Jakarta EE compatible applicatieserver op GraalVM mogelijk te maken, heeft RedHat dit voor elkaar gekregen door met Quarkus elke vorm van dependency injection en reflection (runtime dynamica) te elimineren. Dit is ideaal in een cloud, zeker bij het toepassen van een containerized setup met Docker. De applicatie en bijbehorende server zijn hierdoor volledig geoptimaliseerd voor die Docker image Linux distributie. Deze image zal na het bouwen niet meer veranderen en op elke omgeving als zodanig gedraaid worden. Daarmee zijn alle genoemde frameworks toegespitst op GraalVM en het compileren naar native images, omdat AOT (Ahead Of Time) compilatie mogelijk is geworden.

Mocht je al bekend zijn met het bouwen van Jakarta EE gebaseerde enterprise applicaties, dan heb je geluk. Het mooie van Quarkus is namelijk dat je gebruik kunt blijven maken van de Jakarta EE en MicroProfile annotaties van bijvoorbeeld JAX-RS en CDI die je al kent. Als ontwikkelaar is er geen extra learning curve voor het bouwen van microservices in het Jakarta EE ecosysteem en het draait

Ivo Woltring is Software Architect en Codesmith bij Ordina JTech en houdt zich graag bezig met nieuwe ontwikkelingen in de software wereld.



Edwin Derks is Software Architect en CodeSmith bij Ordina JTech en heeft naast Java een passie voor cloud-driven development en serverless architecture.

op GraalVM om optimale performance en resource consumption te realiseren. Hoe gaaf is dat?!

Quarkus Microservices

Voordat we zelf aan de slag kunnen gaan, willen we toelichten welke functionaliteit Quarkus precies biedt. Het framework biedt namelijk een groeiende lijst van componenten aan die je tot je beschikking hebt. Neem voor het complete overzicht eens een kijkje op de Quarkus Extensions pagina (referentie 3). Wat daar bijvoorbeeld al te vinden is, zijn de SmallRye (referentie 4) implementaties voor Eclipse MicroProfile en Hibernate voor JPA en vele anderen. Het is best indrukwekkend hoe snel zoveel componenten al beschikbaar zijn gekomen. Dit zijn doorgaans de componenten die standaard nodig zijn voor het bouwen van enterprise software, waarmee je nu al met Quarkus aan de slag kan voor productierijpe microservices. Quarkus evolueert dan ook erg snel op dit moment en adopteert razendsnel bepaalde componenten als er behoefte aan is vanuit de community. Het is op dit moment dus niet zozeer de vraag wanneer een component beschikbaar komt, maar meer hoe snel.

Mocht je behoefte hebben aan een component dat nog niet beschikbaar is in Quarkus, geef dit dan gerust aan via het open source project op Github om dit in gang te zetten.

Zelf aan de slag

Zelf microservices bouwen met Quarkus is niet moeilijker dan met andere frameworks. De getting-started (referentie 5) van Quarkus biedt meer dan voldoende informatie om aan de slag te kunnen. Je hebt alleen een minimale versie van Maven (3.5.3+) nodig en een JDK voor Java 8+ met HotSpot JVM. Als je aan de slag wilt met native images, dan heb je GraalVM (op moment van schrijven is versie 1.0.0-rc16 ondersteund) nodig. Je kunt deze natuurlijk ook als HotSpot JVM gebruiken met zijn impliciete optimalisaties, zoals de Java-gebaseerd JIT compiler.

Getting Started

Een mooie plek om te starten is het "getting-started" project dat beschikbaar is gesteld op de Github repo van Quarkus (referentie 6). In deze voorbeelden gebruiken wij zelf een Docker image met GraalVM 1.0.0-rc16 erop en maven 3.6.1 (referentie 7).

```
docker run                    \
  -it                         \
  --rm                        \
  -v "$(pwd):/project"        \
  -v "${HOME}/.m2:/root/.m2"  \
  -p 8080:8080                \
  ivonet/graalvm:1.0.0-rc16   \
  mvn compile quarkus:dev
```

De eerste keer zal dit net zo lang duren als een conventionele maven build, omdat alle dependencies opgehaald moeten worden. Daarna monitort Quarkus het project op wijzigingen. Dit is mogelijk doordat we Maven opdragen om de **quarkus:dev lifecycle** optie te gebruiken om de applicatie te starten. Deze heeft als voordeel dat je applicatie nu runt, terwijl elke nieuwe build van je applicatie de bytecode van dit proces aanpast. Hierdoor zijn je aanpassingen direct zichtbaar zonder dat je je applicatie opnieuw hoeft te starten.

ZELF MICRO-SERVICES BOUWEN MET QUARKUS IS NIET MOEI-LIJKER DAN MET ANDERE FRAMEWORKS

```
INFO  [io.quarkus] (executor-thread-5) Quarkus 0.15.0 started in 0.660s. Listening on: http://0.0.0.0:8080
INFO  [io.quarkus] (executor-thread-5) Installed features: [cdi, resteasy]
INFO  [io.qua.dev] (executor-thread-5) Hot replace total time: 2.008s
```

Als je dit adres in de browser overneemt:

```
http://localhost:8080/hello/greeting/QuarkusRules
```

krijg je deze output: Hello QuarkusRules

De afgebeelde opstarttijd van 0.66 seconden is een fantastisch resultaat in vergelijking met de conventionele tijd die we gewend zijn van een build cyclus. Je hebt niet eens meer de tijd om tussen builds in even koffie te gaan halen. Je kan dus lekker in de groove blijven.

Native Image

Het starten kan echter nóg sneller als je diezelfde applicatie bouwt als een native image. Het bouwen van een native image duurt echter significant langer dan een gewone build. Het is daarom een afweging hoe vaak je dit doet. Je kunt deze tijd bijvoorbeeld terugwinnen tijdens de integratietests, waar vaak voor elke test de applicatie gestart en gestopt wordt. Als je veel integratietests hebt, dan kan dit dus door native image te gebruiken vele malen sneller gaan.

Dat is nog niet alles! Een native image consumeert tevens ook veel minder resources,

omdat deze geen dynamische overhead meer genereert. Dit komt je CPU en RAM consumptie ten goede. Er komt overigens ook geen JVM meer aan te pas, omdat je applicatie is gecompileerd naar machinecode voor een bepaalde architectuur. Hierdoor kun je zeer waarschijnlijk geld besparen bij het draaien van je applicatie in cloud omgevingen.

Compileren van de native image:

```
docker run                    \
  -it                         \
  --rm                        \
  -v "$(pwd):/project"        \
  -v "${HOME}/.m2:/root/.m2"  \
  -p 8080:8080                \
  ivonet/graalvm:1.0.0-rc16   \
  mvn package -Pnative
```

Draaien van de native image:

```
docker run                    \
  -it                         \
  --rm                        \
  --name graalvm              \
  -v "$(pwd):/project"        \
  -p 8080:8080                \
  -v "${HOME}/.m2:/root/.m2" \
  ivonet/graalvm:1.0.0-rc16  \
  ./target/getting-started-1.0-
     SNAPSHOT-runner
```

```
./target/getting-started-1.0-SNAPSHOT-runner
INFO  [io.quarkus] (main) Quarkus 0.15.0 started in 0.044s. Listening on: http://0.0.0.0:8080
INFO  [io.quarkus] (main) Installed features: [cdi, resteasy]
```

Je kunt nu de applicatie oproepen met de eerder genoemde URL en dezelfde output verwachten.

Na het compileren van een native image is het belangrijk deze binary ook op dezelfde architectuur te draaien als waarop het gecompileerd is. Dit betekent zowel OS als hardware. Wij hebben gebruik gemaakt van de Oracle CE (Community Edition) build van GraalVM welke op CentOS gebaseerd is. Hierdoor hebben we de native image alleen kunnen draaien op Fedora en CentOS. De kleinste versies van deze Docker images die we konden vinden, zijn > 75MB. De kleinste Alpine image is bijvoorbeeld < 6MB. Echt veel schijfruimte win je dus niet, mocht dit voor jou een reden zijn om naar Quarkus te kijken. Het is ons helaas nog niet gelukt om native images te builden op een Alpine met daarop GraalVM. We misten veel dependencies voor bijvoorbeeld de gcc compiler, waar GraalVM intern gebruik van maakt. Het is dus nog geen silver bullet voor alle situaties.

Tot slot

Ondanks dat de techniek en het concept nog in de kinderschoenen staan, lijkt het erop dat AOT en native images dé trend aan het worden zijn om invulling te geven aan het cloud-native concept voor het bouwen van enterprise software. Merk op dat dit niet alleen geldt voor het Java ecosysteem, maar ook voor verschillende andere programmeertalen en hun frameworks.

GraalVM kan daarin een grote rol spelen vanwege zijn polyglot en runtime optimalisatie mogelijkheden. Het toepassen van Quarkus en GraalVM in je software architectuur is daarom wellicht een mooie kans om sneller te kunnen ontwikkelen en minder operationele kosten te hebben. Wie weet is dit over een paar jaar de de-facto standaard en kunnen we weer aan de slag met de volgende innovatieve iteratie. Wij kijken ernaar uit en genieten tot die tijd van ongekend snelle opstarttijden en lage resource consumption van onze enterprise software. ■

QUARKUS EVOLUEERT ERG SNEL OP DIT MOMENT EN ADOPTEERT RAZENDSNEL BEPAALDE COMPONENTEN BIJ BEHOEFTE VANUIT DE COMMUNITY

REFERENTIES

1. **https://quarkus.io/**
2. **https://www.graalvm.org/**
3. **https://quarkus.io/extensions/**
4. **https://smallrye.io/**
5. **https://quarkus.io/get-started/**
6. **https://github.com/quarkusio/quarkus-quickstarts/tree/master/getting-started**
7. **https://hub.docker.com/r/ivonet/graalvm**