 Erreichen der Schallgeschwindigkeit mit Quarkus

Der Heilige Gral der Entwicklerproduktivität?

Anfang dieses Jahres ist RedHat in die Arena eingetreten, in der Spring Boot und Micronaut bereits gegeneinander kämpften, um das beliebteste Full-Stack framework für die Erstellung von „microservice“ und „serverless“ Apps zu werden. Was genau unterscheidet Quarkus von seinen Mitbewerbern?

In diesem Artikel lernen Sie die Grundlagen zum erstellen von „microservice“ Apps mit Quarkus kennen, indem Sie Code aus die Spring PetClinic konvertieren in eine „cloud-native“ Quarkus App mit Hibernate Panache, RestEasy und GraalVM.

von Erik-Berndt Scheper

Was ist Quarkus?

„Container-first“ und „cloud-native“

Quarkus ist ein „Kubernetes native“ Java-Framework, das Java zu einer führenden Plattform machen soll in der neuen Welt von „serverless“ Apps, „microservices“, containers und cloud. Dabei nutzt Quarkus Oracle GraalVM um „native“ Apps zu erstellen.

Erzielen Sie unglaublich schnelle Startzeiten in der Größenordnung von Millisekunden, sowie eine geringe Speichernutzung für Ihre Apps. Diese Eigenschaften ermöglichen die automatische „scale up“ und „scale down“ fur microservices sowie Function-as-a-Service (FaaS) Apps.

„Imperative“ und „reactive“

Obwohl Java Entwickler schnell ein Cloud-natives, ereignisgesteuertes, asynchrones und reaktives Modell einführen, um Geschäftsanforderungen zu erfüllen und reaktionsschnelle Apps zu erstellen, kennen die meisten Java-Entwickler das imperative Programmiermodell viel besser und möchten das imperative Modell nutzen, um eine neue Plattform, sowie Quarkus, einzuführen.

Darum unterstützt Quarkus sowohl imperative als auch reactive Programmier­paradigmen für „microservices“. Quarkus hat Unterstützung von [MicroProfile 2.2] (<http://download.eclipse.org/microprofile/microprofile-3.0/microprofile-spec-3.0.html#microprofile2.2>), die [Reactive Streams Operators Spezifikation] (<https://github.com/eclipse/microprofile-reactive-streams-operators>) und sogar [Reactive Messaging] (<https://github.com/eclipse/microprofile-reactive-messaging>), um mit Apache Kafka zusammen zu arbeiten.

Optimiert für Entwickler Freude

Die Vision hinter Quarkus ist es, mehr als nur Produktivität anzustreben: Die Nutzung sollte Spaß machen! Deshalb hat das Team dahinter so viel Aufmerksamkeit darauf verwendet, dass Live-Codierung, „extensions“ und „unified configuration“ einfach funktionieren!

* Im Entwicklungsmodus, den Sie mit mvn compile quarkus:dev starten können, unterstützt Quarkus Live-Codierung, indem geänderte Dateien transparent kompiliert werden, wenn eine HTTP-Anfrage eingeht.
* Das „extension“-system soll dazu beitragen, ein lebendiges Ökosystem rund um Quarkus zu schaffen. Extensions, die im Grunde nichts anderes als Projektabhängigkeiten sind, konfigurieren, booten und integrieren ein Framework oder eine Technologie in eine Quarkus App. Dazu stellen sie GraalVM die richtigen Informationen zur Verfügung, damit Ihre App „native“ kompiliert werden kann.
* Eine einzige Konfigurationsdatei („application.properties“) genügt, um Quarkus sowie alle „extensions“ zu konfigurieren. Um die Größe dieser Datei zu verringern, sollte jeder extension sinnvolle Standardkonfigurationseigenschaften bereitstellen.

Wie teste ich meine Apps mit Quarkus?

Bevor wir uns mit dem Code befassen, ist es klug um einen ersten Blick zu werfen auf den Testansatz von Quarkus. Mit Quarkus können Sie Tests in zwei verschiedenen Modi ausführen: JVM und „native“.

Konventionell erweitern die Testklassen im „native“ Modus die Tests im JVM-Modus und werden in einem Docker-Container unter Verwendung der von GraalVM erstellten „native“ App ausgeführt. Der Vorteil der Wiederverwendung derselben Testklasse für JVM- und native Tests besteht darin, dass wir direkt zu Beginn eines Projekts Tests schreiben können.

Es hat sich als nützlich erwiesen, mit [HTTPie] (<http://www.httpie.org>) die Integrität neuer REST-Services zu überprüfen, obwohl Sie auch „curl“ oder „wget“ verwenden können, wenn Sie sich damit besser auskennen. HTTPie („http“) ist ein mächtiges Kommandozeilenprogramm mit JSON-Unterstützung, Plugins und vielem mehr.

Wie kann Ich mit Quarkus anfangen?

Zum Zeitpunkt des Schreibens, August 2019, war die neueste Version von Quarkus "0.21.1". Aus dem Versionsnummernschema können Sie ableiten, dass Quarkus derzeit als "Beta"-Qualität eingestuft wird. Zu diesem Zeitpunkt wird jede neue Quarkus-Version wahrscheinlich ihre Abhängigkeiten und Bibliotheken auf die neuesten Versionen aktualisieren. Daher habe ich für diesen Artikel die neuesten verfügbaren Versionen von Java, Maven und GraalVM verwendet: GraalVM Community Edition Version 19.2.0, AdoptOpenJDK 8u222 und Maven 3.6.1.

Erstellen einer Quarkus App von der Spring PetClinic Demo-App

Um ein Gefühl für Quarkus zu bekommen, konvertieren wir den Code aus der [Spring PetClinic Demo-App] (<https://github.com/spring-projects/spring-petclinic>), um eine Cloud-native Quarks App zu erstellen. Beginnen wir mit der Ausführung der Spring PetClinic App.

Listing 1

git clone https://github.com/spring-projects/spring-petclinic.git

cd spring-petclinic

mvn clean package

mvn spring-boot:run

Ende

Lassen wir uns abschließend sicherstellen, dass der Spring PetClinic App ordnungsgemäß funktioniert, indem Sie entweder „http://localhost:8080“ in Ihrem bevorzugten Browser öffnen oder den REST-Service mit HTTPie anrufen:

Listing 2

http :8080/vets

Ende

Wie zu erwarten, wird eine JSON-ähnliche Liste aller Tierärzte in der PetClinic-App zurückgegeben. Erstellen wir diesen REST-Service in Quarkus neu.

1. Erstellung einer Quarkus Skelett App

Ähnlich wie bei Spring-Boot und Micronaut können wir das Quarkus Maven-Plugin verwenden, um unsere erste „Hallo Welt“ App mit Quarkus zu erstellen und auszuführen.

Listing 3

mvn io.quarkus:quarkus-maven-plugin:0.21.1:create \

-DprojectGroupId=com.github.acme \

-DprojectArtifactId=quarkus-petclinic \

-DclassName="com.github.acme.quarkus.petclinic.web.resource.VetResource" \

-Dpath="/vets"

cd quarkus-petclinic

./mvnw compile quarkus:dev

Ende

Wenn wir den resultierenden Dienst testen, indem wir „http :8080/vets“ anrufen, wird Hallo im Klartext zurückgegeben. Nicht wirklich aufregend, aber was können Sie sonst noch von einer „Hallo Welt“ App erwarten?

1. Implementierung des Domainmodells

JPA, der De-facto-Standard in Object Relational Mapping, wird in Quarkus mithilfe von Hibernate ORM vollständig unterstützt. Um Hibernate zu konfigurieren, benötigen wir auch eine Datenquelle, um die Verbindungen zu einer Datenbank zu erhalten. In Quarkus ist Agroal die Standardimplementierung für Datenquellen und connection-pools. Wir können Unterstützung für Hibernate. Agroal und JDBC hinzufügen, indem wir die Quarkus extensions zu unserem Projekt hinzufügen:

Listing 4

./mvnw quarkus:add-extension -Dextensions="agroal, hibernate-orm, jdbc-h2, jdbc-mariadb"

Ende

Da diese Aktion die Maven pom.xml ändert, müssen wir den Quarkus dev-build stoppen. Gleichzeitig können wir das Modell (Entity classes) aus der Spring PetClinic in unsere neu erstellte Quarkus PetClinic kopieren.

Da die Domänenklassen Spring-spezifischen Code verwenden, müssen wir etwas mehr als die Import- und Paketnamen korrigieren. Glücklicherweise können wir von Lambda-Methoden in Java 8 profitieren, was diese Änderungen fast trivial macht.

Nachdem die Kompilierungsfehler behoben wurden, schlägt die Ausführung von mvn compile quarkus:dev leider noch nicht.

ERROR [io.qua.dev.DevModeMain] Failed to start quarkus: java.lang.RuntimeException: io.quarkus.builder.BuildException: Build failure: Build failed due to errors

[error]: Build step io.quarkus.hibernate.orm.deployment.HibernateOrmProcessor#build threw an exception: io.quarkus.deployment.configuration.ConfigurationError: Hibernate extension cannot guess the dialect as no JDBC driver is specified by 'quarkus.datasource.driver'

Wenn Sie sich die Fehlermeldung ansehen, ist der Grund, warum Quarkus nicht startet, offensichtlich. Wir haben noch keine Datenbank konfiguriert.

1. Konfiguration der Datenbanken in Quarkus

Derzeit bietet Quarkus extensions für die folgenden Datenbanken an:

* H2
* PostgreSQL
* MariaDB (und MySQL)
* Microsoft SQL server

Um die Konsistenz mit Spring PetClinic zu gewährleisten, brauchen wir H2 In-Memory-Datenbanken für die Entwicklung und MariaDB (MySQL) für die Produktion. Ähnlich wie bei Spring und Micronaut können Sie Quarkus-[Konfigurationsprofile] (<https://quarkus.io/guides/application-configuration-guide#configuration-profiles>) verwenden, um zwischen verschiedenen Laufzeitumgebungen zu unterscheiden. Voreingestellt versteht Quarkus Entwickler- („dev“) und Produktionsprofile („prof“)

Im Moment müssen wir in der Konfigurationsdatei zumindest H2 konfigurieren:

Listing 4. Kofigurationsdatei: src/main/resources/application.properties

%dev.quarkus.datasource.url=jdbc:h2:mem:default

%dev.quarkus.datasource.driver=org.h2.Driver

%dev.quarkus.datasource.username=username-default

%dev.quarkus.datasource.min-size=3

%dev.quarkus.datasource.max-size=13

%dev.quarkus.hibernate-orm.database.generation=drop-and-create

%dev.quarkus.hibernate-orm.sql-load-script=db/hsqldb/data-and-schema.sql

%prod.quarkus.hibernate-orm.database.generation=none

%prod.quarkus.datasource.driver=org.mariadb.jdbc.Driver

Ende

Jetzt müssen wir nur noch die SQL-Datei für HSQLDB aus der Spring Petclinic in unsere neue Quarkus PetClinic kopieren. Wenn wir jetzt versuchen, unsere Quarkus PetClinic App erneut mit mvn compile quarkus:dev auszuführen, sollte es fehlerfrei starten. Wir können testen, dass der Aufruf von „http :8080/vets“ immer noch „Hallo“ im Klartext zurückgibt. Beheben wir das, indem wir die Klasse „VetResource“ implementieren!

1. Implementierung der VetResource-Klasse

Um die Klasse "VetResource" ähnlich wie Spring PetClinic zu implementieren, werden wir das Repository-Pattern verwenden und die Klasse "VetRepository" mit der Quarkus extension „Hibernate Panache“ erneut implementieren. Hibernate Panache ist eine Quarkus extension, die das Schreiben von Entitäten trivial und unterhaltsam machen soll.

Sie müssen nur zulassen, dass Ihre Entitäten die Klasse "PanacheEntity" erweitern, die privaten @Column Felder in öffentliche Felder ändern und dann kann man alle Get- und Setter, automatisch generierten IDs usw. entfernen.

Und natürlich muss unsere Quarkus App auch noch in der Lage sein, JSON zu konsumieren und zu produzieren. Dazu stehen zwei Erweiterungen zur Verfügung, die die JSON-Bindung unterstützen: Jackson und RESTeasy. Da RESTeasy am ausgereiftesten zu sein scheint, verwenden wir die RESTeasy-JSON-Bindungserweiterung.

Die beide Abhängigkeiten kann man in einer einzigen Anweisung hinzuzufügen:

./mvnw quarkus:add-extension -Dextensions="hibernate-orm-panache, resteasy-jsonb"

Standardmäßig wird eine listAll()-Methode schon von der PanacheRepository Klasse bereitgestellt. Wir werden auch die Methode 'Nach Nachnamen suchen' in unsere VetRepository-Klasse aufnehmen und dabei die von der PanacheRepository-Superklasse bereitgestellte find()-Methode nutzen. Das sieht so aus.

Listing 5. VetRepository

@ApplicationScoped

public class VetRepository implements PanacheRepository<Vet> {

public Vet findByName(String name) {

return find("lastName", name).firstResult();

}

}

Ende

Jetzt können wir die VetResource-Klasse wie folgt implementieren:

Listing 6. VetResource

@Path("/vets")

@Produces(MediaType.APPLICATION\_JSON)

@Consumes(MediaType.APPLICATION\_JSON)

public class VetResource {

@Inject

VetRepository vetRepository;

@GET

public List<Vet> list() {

return vetRepository.listAll();

}

@GET

@Path("/name/{name}")

public Vet findByName(@PathParam("name") String name) {

return vetRepository.findByName(name);

}

}

Ende

Wenn wir jetzt versuchen, unsere Quarkus PetClinic App erneut mit mvn compile quarkus:dev auszuführen und der Aufruf von „http :8080/vets“ testen, erhalten wir die gleiche JSON Antwort wie mit der Spring PetClinic App!

Erstellen einer „native“ App

Das Beste an Quarkus ist die hervorragende Unterstützung für die Erstellung nativer Images. Um das einzurichten, müssen wir die Datei application.properties mit einem Produktionsprofil („prod“) aktualisieren, um MariaDB zu unterstützen, ähnlich wie bei Spring PetClinic. Vorausgesetzt, dass GraalVM bereits installiert ist, können wir die native App so testen und ausführen:

Listing 7. Erstellung der native App

gu install native-image

./mvnw clean package -Pnative

./target/quarkus-petclinic-1.0-SNAPSHOT-runner

Ende

Wenn ich auf meinem Computer (MacBook Pro 2017) das Timing von Spring Boot mit Quarkus „dev“-Modus‘ und Quarkus „native“-Modus mit einander vergleiche, ergebe ich die folgenden Ergebnisse:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Spring boot | Quarkus dev | Quarkus native |
| Anlaufen | 5.654 s | 3.430 s | 0.019 s |
| time http :8080/vets | - real 0m0.270s - user 0m0.215s - sys 0m0.045s | - real 0m0.278s - user 0m0.218s - sys 0m0.047s | - real 0m0.265s - user 0m0.210s - sys 0m0.045s |

Tabelle 1: Vergleichung von Spring Boot mit Quarkus „dev“ und „native“ Modus

Was wir sehen ist, dass der Start unserer nativen Quarkus App viel schneller ist als Spring Boot und Quarkus „dev“ Modus. In diesem Fall dauert die Bearbeitung der Anfrage in allen Varianten ungefähr gleich lange.

Zusammenfassung

Das Konvertieren Ihrer vorhandenen JSON REST-Services von Spring Boot zu Quarkus ist ein Kinderspiel! Was Sie bekommen, ist eine Anwendung mit weniger Code, die viel schneller startet, mit dem zusätzlichen Bonus einer Kubernets-Native-fähigen ausführbaren Datei. Beeindruckendes Zeug!

Quellcode

Der Quellcode ist [hier] (<https://github.com/fbascheper/quarkus-petclinic>) verfügbar. Dort finden Sie auch Testklassen, mit denen Sie die Quarkus App sowohl im JVM- als auch im “native” Modus testen können.