Supersonic subatomic Java

Unter den immer populärer werdenden Microframeworks fällt immer wieder Quarkus auf. Es hat sich sich „Freude am Entwickeln“ auf die Fahne geschrieben und verspricht sagenhafte Performancegewinne. Und das, obwohl es eine Sammlung altbekannter Frameworks zu sein scheint. Zeit für eine gründliche Analyse und einen Blick unter die Motorhaube.

Ist Quarkus ein Thema für uns?

Fangen wir mit dem schwierigsten Thema schlechthin an: der Zukunftsprognose. Niemand kann wissen, ob Quarkus die Java-Welt erobern wird. Platzhirsche wie Spring Boot und JavaEE sitzen sehr fest im Sattel, und Quarkus bietet einen ganz ähnlichen Funktionumfang. Wieso sollte es die etablierten Frameworks verdrängen?

Das scheint aber gar nicht das Ziel von Quarkus zu sein. Es gibt keinen vorgezeichneten Migrationspfad von JavaEE oder Spring nach Quarkus. In den meisten Fällen dürfte eine Migration so teuer sein, dass sie den Nutzen bei weitem überschreitet. Quarkus ist vielmehr angetreten, um neue Nischen zu besetzen.

Dieses Kunststück könnte tatsächlich gelingen. Die Welt der Java-Entwickler hat sich geändert. Als Spring Boot – und erst recht JavaEE – entworfen wurden, sah die typische Laufzeitumgebung noch ganz anders aus als heute. Es gab keine Cloud. Dafür war es populär, die UI mit Java zu entwickeln. Bis vor einigen Jahren galten JavaScript, HTML und CSS als schwierig. Ein sehr großer Teil der Java-Entwickler fühlte sich in diesen Bereichen nicht sattelfest. Das machte Java-zentrierte UIs wie JSF, Spring MVC oder GWT populär.

Brave new world

Im Jahre 2021 sieht die Situation ganz anders aus. Die Branche hat sich differenziert. In vielen Firmen hat sich eine Arbeitsteilung etabliert: Java-Entwickler kümmern sich um die Business-Aufgaben im Backend, während das Frontend die Domäne der JavaScript-Entwicker ist.

In der Backend-Entwicklung wiederum findet gerade eine Revolution statt. Der Megatrend „Cloud-Computing“ wirbelt die Welt der Java-Entwickler gründlich durcheinander.

Wie gründlich, zeigt diese – keineswegs vollständige! - Aufzählung:

- Cloud-Computing bedeutet normalerweise, dass Docker zum Alltag gehört.
- Das wiederum bedeutet, dass Java-Programme typischerweise unter Linux laufen.
- Die Application-Server, auf denen mehrere Anwendungen gleichzeitig laufen, wurden durch leichtgewichtigere Sping-Boot-Services abgelöst.
- Noch leichtgewichtiger geht es mit dem Serverless-Ansatz. Besonders prominent sind hier die Azure Functions und Amazon Lambda.
- Microservices sorgen für eine zunehmende Arbeitsteilung.

Das alles ist sehr spannend und aufregend – und bringt ganz neue Ebenen an Komplexität mit. Der Wunsch nach mehr Einfachheit wird laut.

Der Aufstieg der Microframeworks

Eine ganze Reihe neuer Frameworks springen in die Bresche. Sie bezeichnen sich als „Microframeworks“, weil sie sich auf bestimmte Aspekte der Softwareentwicklung konzentrieren. Während Spring Boot noch ein Full-Stack-Framework war, dass alle Aspekte von der UI über die Security bis hin zur Datenbank unterstützt, liefern Microframeworks nur noch einen Teil dieser

Funktionalität. Bei den Microframeworks, die dem Autor in letzter Zeit aufgefallen sind, sind das just die Funktionalitäten, die für Microservices ohne UI benötigt werden.

Aktuell hat der Autor zahlreiche Microframeworks auf dem Radar: Helidon, Micronaut, Spark, Javalin und Ktor. Und natürlich Quarkus.

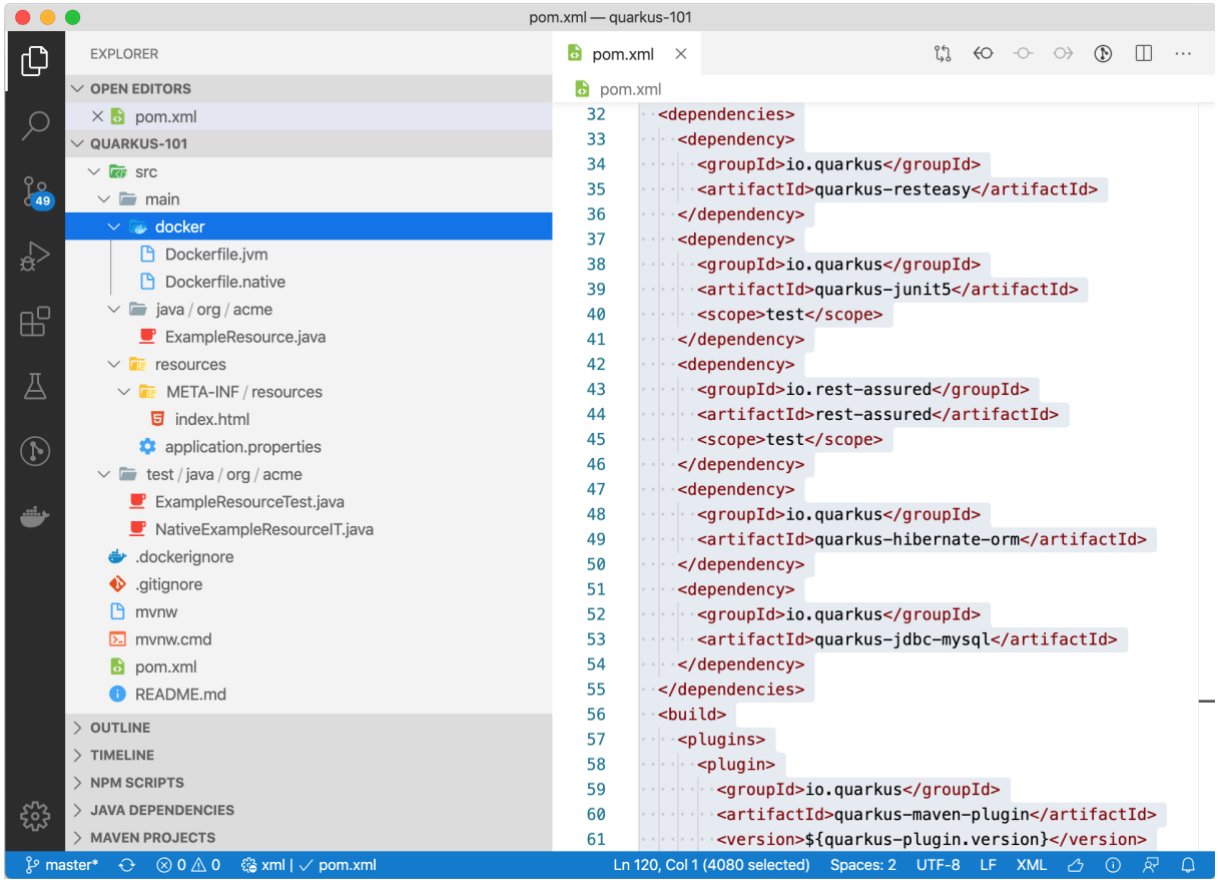
Altbekannte Frameworks neu kombiniert

Eine für die Einarbeitung sehr angenehme Eigenschaft von Quarkus ist, dass es auf den ersten Blick wenig Neuerungen bringt. Im Werkzeugkasten (siehe https://code.quarkus.io/) finden wir unter anderem Hibernate, JAX-RS, CDI, Bean Validation, Flyway und Liquibase sowie Apache Kafka. Viele Java-Entwickler haben mit diesen oder sehr ähnlichen Bibliotheken schon gearbeitet. Das sorgt für einen stressfreien Start.

Es gibt auch Support für Spring. Konkret werden Spring Core, Spring Data, Spring Web, Spring Security und ein Teil von Spring Boot unterstützt. Es handelt sich um einen Kompatibilitätslayer, in der Praxis sind daher Einschränkungen zu erwarten. Die Botschaft ist aber auch hier klar: wer Spring kennt, sollte sich schnell mit Quarkus wohlfühlen.

Weniger bekannt sind die reaktiven Frameworks von Quarkus: Vert.x und Smallrye. Darauf gehen wir später noch ein. Hier ist nur interessant, dass sie auch schon seit Jahren auf dem Markt sind.

Mit anderen Worten: Quarkus bringt viele bekannte Frameworks zusammen. Lassen Sie sich auf https://code.quarkus.io ein Beispielproject generieren. Es wird Ihnen sofort bekannt vorkommen: es ist ein klassisches Maven- oder Gradleprojekt ohne große Überraschungen:



Typischer Aufbau einer Quarkus-Anwendung (Abb. 1)

Die APIs sind also nicht das Alleinstellungsmerkmal von Quarkus. Was ist es stattdessen?

Container First!

Einige Dinge fallen im Screenshot doch auf: der Ordner namens „Docker“, zwei Dateien mit dem Wort „native“ im Namen und das Maven-Plugin von Quarkus. Auf den zweiten Blick fällt noch auf, dass anstelle der Standardbibliotheken Quarkus-Versionen von RESTEasy, Junit, Hibernate usw. verwendet werden.

Quarkus wurde für den Einsatz im Container und in der Cloud entwickelt. Daraus leiten sich verschiedene Optimierungsziele ab. Es sind die Dauerbrenner Performance, Speicherverbrauch und Kosten – jetzt aber mit anderen Schwerpunkten:

- Im Bereich „Performance“ gewinnt die Dauer eines Kaltstarts an Bedeutung. Die „Peak Performance“ ist relevant bei Anwendungen, die mehrere Minuten oder Stunden laufen. Beim Einsatz in der Cloud ist das aber gar nicht mehr die Regel. Um Kosten zu sparen, werden Container schnell herunter- und nur bei Lastspitzen wieder hochgefahren. Besonders augenfällig wird das bei AWS Lambdas und Azure Functions: wenn die Funktion einige Minuten nicht aufgerufen wurde, bedeutet der nächste Aufruf einen Kaltstart.

- Mit „Speicherverbrauch“ ist nicht mehr der Heap-Memory gemeint, sondern der gesamte Speicher, den der Container braucht. Die JVM und das Betriebssystem gehören dazu.

- In der Cloud bestimmen sowohl die Rechenleistung als auch der Speicherbedarf die Kosten. Die Rechenleistung ist dabei bei typischen Enterprise-Anwendungen sekundär: sie können jederzeit weitere Container hochfahren. Damit wird – abhängig vom Preismodell Ihres Cloudproviders - oft der Hauptspeicher zum primären Kostentreiber.

Native Anwendungen mit GraalVM

Quarkus verschiebt einen Teil der Aufgaben vom Anwendungsstart in die Kompilierphase. Das ist nicht zuletzt der GraalVM geschuldet.

GraalVM bietet einen ahead-of-time-Compiler (kurz: AOT-Compiler). Dieser Compiler erzeugt Maschinencode, der von der CPU direkt ausgeführt werden kann. Das hat einige Vorteile und einige Nachteile. Zu den Vorteilen gehört ein dramatisch beschleunigter Anwendungsstart, zu den Nachteilen gehört eine langfristig schlechtere Performance. Und der Verzicht auf Reflection.

Das würde aber den Verzicht auf fast alle populären Java-Frameworks bedeuten. Daher kann die GraalVM zur Compile-Zeit Metadaten erzeugen, mit deren Hilfe die Funktionalität der Reflection-API gerettet werden kann. Der Preis dafür sind erheblich längere Kompilierzeiten.

Ein weiterer Preis ist, dass die fertig kompilierte Anwendung sich zur Laufzeit nicht mehr ändern darf („closed world principle“). In der Praxis ist das keine Einschränkung. Wenn wir eine Anwendung verändern wollen, kompilieren wir sie ohnehin neu. Etwas anders sieht es bei einem traditionellen Application-Server aus. Dort konnten die WAR-Dateien zur Laufzeit installiert werden. Mit einem AOT-Compiler ist so etwas nicht möglich. Die Unterstützung der AOT-Compilers hat auch ein paar Nachteile.

Die Aufgaben der Quarkus Extensions

Quarkus macht aus der Not eine Tugend. Wir hatten oben bereits gesehen, dass wir nicht direkt die gewohnten Frameworks verwenden, sondern speziell für Quarkus angepasste Versionen (die „Quarkus Extensions“). Die Erweiterung besteht darin, zur Kompilierzeit Metadaten zu sammeln und Funktionalitäten vom Anwendungsstart in die Kompilierzeit zu verschieben.

Peter Palaga skizziert in seinem Vortrag[1], wie das aussieht: statt Beans bei jedem Anwendungsstart per Classpath-Scan zu suchen und per Reflection zu erzeugen, werden sie gleich bei Kompilieren in eine statische Hash-Table geschrieben. Damit sind sie sofort verfügbar und können schnell erzeugt werden. Je nach Scope können die Instanzen dieser Beans auch gleich erzeugt werden.

[1] Peter Palaga, „Quarkus from Inside“, https://ppalaga.github.io/presentations/191106-w-jax-quarkus/index.html#/build-time-cdi-2

Er beschreibt auch noch weitere Tricks, mit denen Quarkus den Start der Anwendung optimiert. Die statische Codeanalyse während der Kompilierzeit ermöglicht eine Art „Tree Shaking“. Klassen, die von der Anwendung nicht benötigt werden, werden eliminiert. Das spart den Aufwand, sie zu laden und sie zu initialisieren. Das macht bei Spring Boot und erst recht bei einem Application-Server einen erheblichen Teil der Startzeit aus.

GraalVM

Ein besonders raffinierter Trick wird nicht von Quarkus, sondern von der GraalVM implementiert. Sie führt statische Initializer bereits zur Kompilierzeit aus, nimmt einen Snapshot des so entstandenen Heap und kopiert ihn in das Executable.[2] Wenn die Anwendung später gestartet wird, brauchen die statischen Initializer also gar nicht mehr ausgeführt werden. Das Ergebnis ist ja schon da.

In praktisch allen Präsentationen wird gezeigt, dass der Speicherverbrauch einer nativen, mit dem AOT-Compiler erzeugten Anwendung 80, 90% niedriger ist als der Speicherverbrauch einer traditionellen Anwendung mit dem JIT-Compiler. Das dürfte auch so stimmen – gilt in diesem Ausmaß aber nur für relativ kleine Anwendungen. Bytecode ist erheblich kompakter als Assemblercode. Je größer Ihre Anwendung ist, desto geringer wird der Vorteil des AOT-Compilers, und irgendwann drehen sich die Verhältnisse auch um.

Ähnlich wird es bei den Startzeiten sein. Bei den meisten Benchmarks startet die Anwendung dank des AOT-Compilers rund tausendmal schneller als mit dem JIT-Compiler. Wenn Sie aber eigene Programmlogik bei Programmstart ausführen, wird das Ergebnis weniger überzeugend sein. Der Grundsatz des AOT-Compilers lautet, dass Sie ein wenig „Peak Performance“ gegen schnelle Verfügbarkeit eintauschen. Das sollten Sie beim Anwendungsdesign berücksichtigen.

Für die meisten Microservices – wenn sie denn wirklich „micro“ sind! – ist das Resumée aber tatsächlich: deutlich weniger „Memory Footprint“, dramatisch reduzierte Startzeiten, aber dafür wird Ihr Programm niemals die Performance des JIT-Compilers erreichen. Konkrete Zahlen zu nennen ist schwierig – das kommt sehr auf die jeweilige Aufgabe an – in den meisten Fällen scheint der JIT-Compiler aber rund zwei bis dreimal schneller zu sein.

Ein Randaspekt sei an dieser Stelle auch erwähnt: Quarkus macht die Arbeit mit GraalVM einfach. Wir hatten weiter oben im Screenshot schon gesehen, dass bei einem Quarkus-Projekt standardmäßig die Dateien zum Generieren eines nativen Executables generiert werden. Wenn Sie diese Datei selber erstellen wollen, artet das laut Peter Palaga[3] schnell in eine schier endlose Liste von Parametern aus.

Konkrete Zahlen

Mein Demoprojekt auf GitHub[4] gehört zugegebenermaßen zu den besonders kleinen Lambda-Services – zeigt aber gerade die Unterschiede zwischen dem AOT-Compiler und dem JIT-Compiler besonders deutlich.

Mit Java 8 und der traditionellen JVM dauert der erste Start spürbar lang:

```
> ./5-invoke-java-function.sh
Duration: 367.99 ms Billed Duration: 400 ms Memory Size: 128 MB Max Memory
Used: 97 MB Init Duration: 1112.91 ms
```

Dafür ist der zweite und alle folgenden Funktionsaufrufe rund 1000 mal schneller – so lange, bis der Lambda-Service nach einer Pause wieder herunterfährt:

```
> ./5-invoke-java-function.sh
```

[2] Peter Palaga, „Quarkus from Inside“, https://ppalaga.github.io/presentations/191106-w-jax-quarkus/index.html#/static-init-1

[3] Peter Palaga, „Quarkus from Inside“, https://ppalaga.github.io/presentations/191106-w-jax-quarkus/index.html#/5/9

[4] Demoproject auf GitHub, https://github.com/stephanrauh/BeyondJava.net-Articles/tree/master/simple-quarkus-lambda-test

```
Duration: 0.60 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory
Used: 97 MB
```

Wenn wir den Lambda-Service dagegen nativ kompilieren, dauert der Aufruf immer ungefähr eine Millisekunde:

```
./9-invoke-native-function.sh
Duration: 1.04 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory
Used: 52 MB
```

Das ist auch in diesem einfachen Beispiel – in dem es kaum etwas zu optimieren gibt – nur halb so schnell wie die Peak-Performance des JIT-Compilers. Aber: wir können uns darauf verlassen. Bei jedem meiner Versuche lag die Performance im Bereich zwischen 0,9 Millisekunden bis 1,1 Millisekunden.

Wann lohnen sich langsame Microservices?

Traditionell ist die CPU-Performance das Maß der Dinge. Speicher galt lange Zeit als billige Ressource. In der Cloud sieht das oft anders aus. Hier ist der Speicherverbrauch ein wesentlicher Kostentreiber. Der geringere Memory Footprint einer nativen Anwendung erlaubt eine höhere Packungsdichte. Der Speicher, der vorher für einen einzigen Application Server gereicht hat, reicht jetzt für eine ganze Schar von Containern. Wenn sie diese alle gleichzeitig starten – also horizontal skalieren – ist der Durchsatz größer als der Durchsatz des einen Application Servers. Die gute Performance des JIT-Compilers kann das nicht ausgleichen.

Preismodelle ändern sich im Laufe der Zeit, deswegen nenne ich hier keine konkreten Zahlen. Die Rechnung bleibt Ihnen überlassen. Meine Botschaft ist nur: denken Sie bei der Rechnung an die Möglichkeit der horizontalen Skalierung.

Predictable Performance

Manchmal kommt es auch gar nicht so sehr auf die Spitzengeschwindigkeit an. In Echtzeitsystemen ist das entscheidende Kriterium, dass das System garantiert in einer bestimmten Zeitspanne reagiert. Stellen Sie sich vor, die Motorsteuerung Ihres Autos würde mit Java betrieben. Der JIT-Compiler sorgt dafür, dass Ihre CPU reichlich Leistungsreserven hat. Mit Ausnahme des Kaltstarts. Java nimmt erst im Laufe der Zeit Fahrt auf. Für Ihren Motor bedeutet das jede Menge Fehlzündungen. Womöglich sogar jedes Mal, wenn die Start-Stopp-Automatik an der Ampel den Motor ausschaltet.

Der AOT-Compiler sorgt für eine vorhersagbare Performance. Sie mag etwas langsamer sein – aber vor allem ist sie konstant. Ihre Motorsteuerung hat nicht mehr so viel Leistungsreserven, aber darauf kommt es gar nicht an. Entscheidend ist, dass sie keine Fehlzündungen mehr erzeugt.

Quarkus ohne GraalVM

Von den Tricks von Quarkus beim Kompilieren profitieren Sie auch ohne GraalVM. Das merken Sie bei der täglichen Arbeit als Entwickler. Die Anwendungen starten auch mit dem klassischen JIT-Compiler deutlich schneller. Wie schnell genau, hängt von Ihrer Anwendung ab, bei den relativ kleinen Testprogammen des Autors war es meistens Faktor zwei bis drei.

Beim Entwickeln profitieren Sie auch von einem zweiten Feature von Quarkus: dem „Hot Module Reloading“. Sie starten Quarkus im Entwicklungsmodus. Dazu öffnen Sie ein Terminal und geben auf der Kommandozeile den Befehl **„`./mvnw compile quarkus:dev`“** ein. Anschließend programmieren Sie genauso wie z.B. bei Angular: sie editieren Quellcode, beim Speichern wird automatisch kompiliert, und wenn Sie im Browser auf „refresh“ klicken, wird der neue Code ausgeführt.

Kurioserweise handelt es sich um ein sehr altes Feature, das bereits mit Java 1.4 einführt hatte. Seinerzeit hatte IBM einen Application Server entwickelt, der nach meiner Erinnerung ca. fünf bis zehn Minuten zum Starten brauchte. An ein flüssiges Entwickeln war damit nicht zu denken. Als Ausweg wurde das „Hot Code Replacement“ erfunden. Das ermöglicht es der IDE, geänderte Quelltexte in das laufende Programm einzufügen. Der Zustand der Anwendung bleibt dabei erhalten. Bei einer komplexen Anwendung mit UI bleibt es Ihnen dank Hot Code Replacement erspart, sich nach jeder

Programmänderung wieder einloggen zu müssen. Leider funktioniert Hot Code Replacement nicht besonders gut mit Frameworks wie z.B. Spring. Spätestens mit dem Siegeszug der testgetriebenen Entwicklung und der Fokussierung auf Backendentwicklung geriet das Feature in Vergessenheit. Spring Boot startet so schnell, dass das Hot Code Replacement nicht mehr so wichtig ist.

Bei umfangreichen Testsuiten spielt die Startzeit aber doch wieder eine Rolle. Mit Quarkus laufen vor allem die Integrationstests deutlich schneller als bei Spring.

IDEs und Debugging

Der „developer mode" von Quarkus besteht – Stand April 2020 – aus einem Maven-Skript. Das Kommando „`./mvnw compile quarkus:dev`" startet einen Prozess, der die Dateien im Quelltextverzeichnis überwacht und bei jeder Änderung neu kompiliert.

Traditionell arbeiten IDEs wie z.B. Eclipse anders. Im traditionellen Modell kümmert sich die IDE darum, dass die Anwendung kompiliert wird. Das hat einige Auswirkungen.

Eine davon ist relativ überraschend, zumindest aus der Sicht eines Java-Entwicklers: Visual Studio Code ist ein hervorragender Editor für Quarkus. In der JavaScript-Welt – aus der Visual Studio Code kommt – hat es sich ohnehin weitgehend etabliert, den Compiler in einem Terminal-Fenster zu starten. Quarkus überträgt das gleiche Modell in die Java-Welt.

Mit Eclipse können Sie auch mit Quarkus entwickeln. Die Demos für diesen Artikel wurde so entwickelt. Es ist aber ein wenig mühsamer, weil Eclipse standardmäßig kein Terminalfenster mitbringt. IntelliJ bietet ein integriertes Terminal. Trotzdem lohnt es sich, Visual Studio Code in die engere Wahl zu nehmen: Quarkus wird primär von Red Hat vorangetrieben, und bei Red Hat ist aktuell (April 2020) Visual Studio Code sehr populär.

Zum Debugging müssen sie in allen IDEs das „remote Debugging" verwenden. Das ist ein wenig lästig, funktioniert aber tadellos.

Reactive Programming

In der JavaScript-Welt sind in den letzten Jahren reactive Programming und „non-blocking IO" populär geworden. Quarkus überträgt diese Ideen in die Java-Welt – und überlässt Ihnen die Wahl, welche Programmteile Sie traditionell, also imperativ und welche Sie reaktiv implementieren wollen.

Die Idee hinter der reaktiven Programmierung ist, dass Sie das Programm nicht blockieren, wenn Sie auf das Ergebnis einer Datenbankabfrage oder einer REST-Calls warten. In der Zwischenzeit kann das Programm weiterlaufen und sich um andere Aufgaben kümmern. Sobald das Ergebnis da ist, wird eine Call-Back-Funktion aufgerufen. In Java sind das Lambda-Expressions.
Den Unterschied schauen wir uns an einem kleinen Beispiel an. Um die Ideen deutlicher zu machen, präsentiere ich keinen echten Quelltext, sondern stark eingedampften Pseudocode. Echten Code können Sie z.B. auf dem Blog von Niklas Heidloff[5] oder dem interaktiven Tutorial von Katacoda[6] sehen.

Das erste Beispiel ist JDBC-Code, wie wir ihn schon seit Java 5 kennen:

(Listing 1)

```
// Pseudocode!
public List<Fruit> findAllImperative(Statement client) {
  ResultSet result = client.query("SELECT id, name FROM fruits");

  // hier muss die CPU warten

  return convertResultToArrayList(result);
}
```

5 http://heidloff.net/article/comparing-synchronous-asynchronous-access-postgresql/
6 https://www.katacoda.com/openshift/courses/middleware/middleware-quarkus/reactive-sql

Das Programm hält so lange an, bis die Datenbankabfrage ein Ergebnis zurückliefert. Viel schöner wäre es, wenn wir mehrere Datenbankabfragen parallel abschicken könnten. Reactive Programming macht das möglich:

(Listing 2)

```
public Uni<List<Fruit>> findAllReactive(PgPool client) {
  return client.query("SELECT id, name FROM fruits")
               .map(rowSet -> { convertResultToArrayList(rowSet); });
}
```

In diesem Fall ist der Code, der die Daten verarbeitet, nicht mehr direkt in der nächsten Zeile. Hinter dem Aufruf der Methode **query()** folgt ein Lambda, das asynchron aufgerufen wird, wenn die Daten da sind. In der Zwischenzeit läuft die Anwendung weiter. Deswegen liefert die Methode auch keine Liste. Das Uni<List> ist lediglich ein Versprechen, demnächst eine List zu bekommen.

Die reaktive Programmierung erfordert ein Umdenken. Was passiert zum Beispiel, wenn in der Callback-Methode wiederum eine Datenbankabfrage passiert? Dann würde – um im Bild zu bleiben – ein Versprechen auf ein Versprechen zurückgeliefert. Das erfordert etwas Übung. Quarkus erleichtert den Einstieg, indem es beide Programmierstile zulässt.

Reactive Programming weitergedacht

Im Prinzip könnte der Datenbanktreiber auch einen Datenstrom schicken. Anders ausgedrückt: er könnte bereits anfangen, das Ergebnis der Abfrage zu schicken, wenn er erst einen Teil der Daten gelesen hat. Der Rest der Daten kommt dann später und wird ebenfalls über das Lambda verarbeitet. Während meiner Recherche habe ich keinen Datenbanktreiber gefunden, der dieses Feature bereits beherrscht. Vielleicht kommt das in Zukunft.

Bei vielen anderen Systemen wie z.B. Kafka ist das bereits heute Standard. Als Anwendungsfall können Sie z.B. an Ihr Chat-Programm denken, dass den ganzen Tag im Hintergrund läuft und sofort reagiert, wenn eine Nachricht ankommt. Wenn die Architektur so konzipiert ist, dass die Daten als kontinuierlicher Strom von Ereignissen kommen, ist die asynchrone Verarbeitung in einer Callback-Methode die einfachste Lösung.

Reaktive Programmierung hat noch weitere Vorteile. Wenn die Datenbank sehr groß ist – und wenn der Datenbanktreiber mitspielt – kann die Callbackmethode bereits aufgerufen werden, wenn erst ein Teil der Daten verfügbar ist. Die Callbackmethode wird dann mehrfach aufgerufen – immer dann, wenn weitere Daten ankommen.

Die technische Basis von Quarkus ist reaktiv implementiert. Dadurch integrieren sich die reaktiven APIs nahtlos in Quarkus.

Amazon Lambdas

Auf Quarkus.io[7] findet sich ein Tutorial, das zeigt, wie die Unterstützung von Quarkus aussehen kann. Das Beispielprojekt wird im wesentlichem durch ein *Maven Archetype* erzeugt und enthält einige Skripte, um einen Amazon Lambda-Service mit wenigen Handgriffen zu deployen – oder auch lokal zu testen. Das erleichtert den Einstieg in die Technik deutlich. Vor allem ist es mit den Skripten sehr einfach, ein native kompiliertes Java-Programm als Lambda zu deployen. Auf meinem GitHub-Repository[8] finden Sie die gleiche Anwendung mit Shell-Skripten, die die Arbeit noch etwas weiter vereinfachen.

7 https://quarkus.io/guides/amazon-lambda

8 https://github.com/stephanrauh/BeyondJava.net-Articles/tree/master/simple-quarkus-lambda-test

Der erste Eindruck ist positiv. Der Maven Archetype liefert die Konfiguration von SAM mit, so dass Sie die Lambda-Funktion lokal testen können. Die Shellscripte erlauben sowohl das Deployment einer traditionellen JAR-Datei als auch das Deployment einer nativen Binärdatei.

Dafür wird GraalVM als Docker-Image heruntergeladen. Der AOT-Compiler braucht also nicht lokal installiert zu werden. Das ist praktisch, bedeutet aber auch, dass es etwas länger dauert. Sie müssen generell Geduld mitbringen: bei meinem „Hello-World“-Programm dauerte das Kompilieren oft zwei, drei Minuten. Bei großen Anwendungen dauert es entsprechend länger.

Die native Kompilierung zeigt am deutlichsten, dass der AWS Lambda-Support noch im „Preview“-Stadium ist. Neben den Kompilierzeiten traten in der Version 1.3.2, die ich bei der Recherche verwendet habe, teilweise kryptische Fehlermeldungen auf. Die langen Laufzeiten liegen allerdings in der Natur der Sache. Die statische Codeanalyse des AOT-Compilers und die Analyse der Reflection dauern einige Zeit. Die Erzeugung der nativen Images wird am besten in einen Jenkins-Job ausgelagert, der asynchron nach dem Einchecken der Quelltexte läuft.

Fazit:

Auf den ersten Blick ist es gar nicht so einfach, zu sagen, was Quarkus ausmacht. Fast alles, was Ihnen in den Tutorials begegnet, ist vertraut. Allenfalls das reaktive Programmieren mit Vert.x und Smallrye mag ungewohnt sein.

Das eigentlich Neue an Quarkus ist, dass es all diese vertrauten Frameworks auf eine neue Weise zusammenbringt. Die Magie liegt in der Build-Pipeline. Quarkus hat den Buildprozess für viele Frameworks so optimiert, dass sie besonders gut mit dem AOT-Compiler von GraalVM harmonieren. Und das wiederum ist ein großer Vorteil in der Cloud, wo Container häufig hoch- und runtergefahren werden.

Auch jenseits der Cloud hat Quarkus ein paar Vorteile. Der Anwendungsstart ist deutlich schneller, und das merken Sie in vielen Situationen. Ihre Unit-Tests laufen schneller, und das gilt insbesondere für Ihre Integrationstests.

Natürlich bringt auch Quarkus ein paar Nachteile mit. Ihre IDE wird auf die Rolle eines Editors reduziert. Der Compiler läuft in einem separaten Fenster, und Debugging ist nur über Remote Debugging möglich.

Für den Arbeitsalltag eines Softwareentwicklers ist aber noch etwas wichtig: der Entwicklermodus von Quarkus bringt das Hot Module Reloading in die Java-Welt. Zugegeben, ähnlich weit waren wir schon vor 18 Jahren, als mit Java 1.4 das Hot Code Replacement veröffentlicht wurde. Aber dieses Feature war den meisten von uns seit vielen Jahren nicht mehr zugänglich.

Ob Quarkus den etablierten Platzhirschen wie JavaEE und Spring Boot den Rang ablaufen wird, ist eine offene Frage. Das ist auch gar nicht das Ziel von Quarkus. Es spricht aber einiges dafür, dass Quarkus im Bereich Cloud-Computing seine Nische finden wird, und dass es uns noch viele Jahre begleiten wird.

Links/Quellen/Verweise:

(siehe Fußnoten)

Kurz-Vita Autor (~300 Zeichen):

Stephan ist Architekt, Entwickler und Trainer mit viel Freude an Presales und Marketing. Er pendelt regelmäßig zwischen der Java-Welt und dem JavaScript-Universum. In der Open-Source-Scene hat er sich einen Namen durch Projekte wie BootsFaces und ngx-extended-pdf-viewer gemacht. Außerdem betreibt er den Blog BeyondJava.net, wo er aktuell eine große Serie über die GraalVM und Quarkus schreibt.

https://www.beyondjava.net
Twitter: @beyondjava / GitHub: stephanrauh
articles@beyondjava.de

Artikel-Umfang:

Subline – 2-4 Sätze

- Überschrift
- Subline – 2-4 Sätze
- Artikel – Umfang 11.000 – 25.000 Zeichen
- Screenshots / Schaubilder / Statistiken / Tabellen etc. nach Bedarf (Bildformat PNG, JPG, TIF, BMP – Auflösung 300dpi) mit entsprechender Beschriftung/Bildunterschrift