GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

OKT / 2020

JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis

#JAVAPRO

JAVA JETZT SO SCHNELL WIE C !

DIE ZUKUNFT VON JAVA

GRAALVM - QUARKUS - MICROSTREAM

Mit freundlicher Unterstützung unserer Partner.

JAVAPRO PARTNER NETWORK

Die JAVAPRO wird von den Mitgliedern des JAVAPRO Partner Network finanziert und aktiv unterstützt. Dadurch sind wir in der Lage, redaktionell unabhängig zu arbeiten und die JAVAPRO kostenlos für die gesamte Java-Community zu produzieren sowie die Java Konferenz JCON 2020 zu veranstalten.

GOLD PARTNER

ORACLE

Fast Lane

Google Cloud

MicroStream

eclipse

XDEV

SILBER PARTNER

RAPID clipse

JFrog

TimoCom

trivago

consol Wir unternehmen IT

BRONZE PARTNER

viadee IT-Unternehmensberatung

raytion

GEBIT Solutions

X 2019 DEVCON

4DSoft

Michael Page

ETECTURE

Werden auch Sie Mitglied des JAVAPRO Partner Network und unterstützen Sie die JAVAPRO - Das kostenlose Fachmagazin für die Java Community.

www.javapro.io/partner

JAVAPRO

Impressum

JAVAPRO

Verlag:
JAVAPRO
Mergenthalerallee 73-75
65760 Eschborn
Telefon: +49 (0)800 - 528 277 6

E-Mail: info@javapro.io
Website: http://www.javapro.io

Chefredakteur:
Markus Kett (V.i.S.d.P.)

Redaktion:
info@javapro.io

Gestaltung, Layout, Produktion:
Impuls Mediengruppe GmbH
Im Gewerbepark 29
92681 Erbendorf

Preis: kostenfrei

Illustrationen:
Pixabay (Public Domain)

Erscheinungsweise: Vier mal jährlich

Gründungsjahr: 2017

#JAVAPRO #Editorial

COVID-19 ist auch an der JAVAPRO nicht spurlos vorübergegangen. Wir mussten die Produktion kurzzeitig einstellen. Aber ab sofort sind wir wieder für euch da – sogar mit verstärktem Team, ungebrochen großer Motivation und enormen Spaß an der Arbeit an diesem tollen Magazin. Ab sofort wird wieder alle drei Monate eine neue JAVAPRO Ausgabe erscheinen.

Mit dieser Ausgabe wollen wir die derzeit wohl brennendste Frage in Bezug auf Java beantworten: „Wird Java gerade von anderen Technologien überholt, soll man auch weiterhin auf Java setzen, welche Innovationen sind von Java in naher Zukunft zu erwarten?"

Seit mehr als zwanzig Jahren ist Java die führende Programmiersprache und Anwendungsplattform. Doch jetzt stehen wir an der Schwelle einer neuen Software-Epoche: Cloud-Native, Serverless-Computing, Big-Data, Künstliche Intelligenz, Virtual-Reality, Internet-of-Things und Automotive sind keine Science-Fiction mehr, sondern mittlerweile alltägliche Anforderungen in Unternehmen und Startups. Je schneller solche Systeme arbeiten, zum Beispiel KI- und Machine-Learning-Algorithmen, je schneller Massendaten und komplexe Datenstrukturen verarbeitet werden können, desto leistungsfähiger und effizienter werden die Systeme und machen bestimmte Anwendungsfälle wie selbstfahrende Autos überhaupt erst möglich, desto weniger Rechenleistung ist erforderlich um solche Anwendungen zu betreiben und desto weniger Infrastrukturkosten fallen letztendlich an. Immer mehr Unternehmen verlagern immer mehr Anwendungen in die Cloud und merken jetzt, dass sie die Übersicht verlieren und ihre Kosten explodieren. Spätestens in der Post-Corona Zeit müssen Unternehmen ihre Kosten drastisch senken und achten mehr als je zuvor auf effiziente, ressourcenschonendere Systeme. Bei Neuentwicklungen und Migrationen in die Cloud spielen zudem Entwicklungszeit und Time-to-Market eine wichtige Rolle.

Viele IT-Entscheider und noch mehr jüngere Entwickler haben mittlerweile große Zweifel, ob Java für all diese Anforderungen immer noch die richtige Technologie ist. Seit jeher hängt Java der Ruf nach kompliziert, träge, schwergewichtig und speicherhungrig zu sein und die Weiterentwicklung von Java unterliegt einem aufwändigem, bürokratischem Entscheidungsprozess, wodurch die Community auf längst fällige Innovationen oftmals viele Jahre warten muss. Manche bezeichnen Java sogar schon als Dinosaurier. Derweil scheinen andere Sprachen wie Python, JavaScript sowie Cloud-Native- und Serverless-Ansätze mehr und mehr Java den Rang abzulaufen. Im Silicon Valley, dem Hotspot für Hightech und Innovationen, entwickelt niemand mehr mit Java, hört man dort immer häufiger und das stimmt bedenklich. Dass Oracle selbst vor zwei Jahren seine traditionelle Java-Konferenz JavaOne in eine multitechnologische Konferenz (Oracle CodeOne) umstrukturiert und sich nun auch für konkurrierende Sprachen völlig offen zeigt, erweckt zudem den Eindruck, dass selbst für den Hersteller Java nicht mehr an erster Stelle steht. Entsprechend groß war der Aufschrei in der Java Community. Als Cloud-Provider, der mit aller Macht Marktanteile gewinnen möchte, hat Oracle allerdings auch gar keine Wahl als sich auch für andere Sprachen zu öffnen. Die Gedanken dahinter sind also nachvollziehbar. Die Frage, ob man überhaupt noch auf Java setzen sollte, aber auch.

Mit dieser Ausgabe wollen wir diese brennende Frage beantworten. So viel schon mal vorab: Es gibt richtig gute Nachrichten für uns Java Entwickler: 1. So schnell und einfach sterben Dinosaurier nicht aus. Nur wenn erneut ein großer Meteorit die Erde trifft, könnte es auch für Java brenzlich werden. 2. Der Heilige Graal, der zum ewigen Leben verhelfen soll, ist bereits gefunden! Mit GraalVM und einer neuen Generation von Microservice-Frameworks wie Quarkus, Micronaut, Helidon und MicroStream entsteht gerade ein völlig neuer Java-Cloud-Native Technologie-Stack, der sich perfekt für den Einsatz in der Cloud eignet und Java auf ein völlig neues Level bringt: Java-Anwendungen genauso schnell wie mit C, Anwendungsstart in Millisekunden, minimaler Memory-Footprint und bis zu 1000 Mal schnellere Datenbankzugriffe. Das ist keine Vision, sondern heute schon möglich. In dieser Ausgabe erfahren Sie alles Wichtige über den neuen Java-Cloud-Native Stack. Das gesamte JAVAPRO-Team wünscht euch viel Spaß beim Lesen!

Markus Kett
Chefredakteur
JAVAPRO

Twitter: @MarkusKett
LinkedIn: markuskett

#JAVAPRO #JavaEE #MicroProfile #Spring

Java-EE vs. Spring für Microservices

Hat Jakarta-EE (ehem. Java-EE) immer noch den Ruf schwergewichtig zu sein und hat damit geringe Chancen für eine Microservice-Architektur in Betracht gezogen zu werden? Ist das Spring-Framework die bessere Antwort auf die Frage mit welcher Java-Technologie wir unsere Services bauen? Einmal mehr stehen sich die ewigen Duellanten Jakarta-EE und Spring gegenüber, dieses Mal jedoch mit dem Fokus auf Microservices.

Autor:

Daniel Krämer ist bereits seit mehreren Jahren als Software Engineer für die anderScore GmbH tätig und begeistert sich für durchdachte Software-Architektur und strukturiertes Design. In seinen Kundenprojekten setzt er sich überwiegend mit individueller Java- und Web-Entwicklung sowie Fragestellungen rund um Migration und Integration (z.B. Microservices) auseinander. Er ist dafür bekannt, auch für komplexe Szenarien effektive und effiziente Strategien zur Test-Automatisierung zu finden. Daniel teilt seine Kenntnisse und Erfahrungen gerne mit anderen Entwicklern. Zu den Inhalten regelmäßig abgehaltener, öffentlicher Trainings zählen Themen wie Microservices, Spring, Apache Wicket und jQuery.

Webseite: https://anderscore.com
GitHub: https://github.com/dkraemer-anderscore
E-Mail: daniel.kraemer@anderscore.com

Maik Wolf ist bei der Kölner anderScore GmbH als Fullstack-Entwickler für Java Enterprise Projekte & Agile Coach im Kundeneinsatz. Durch seine Expertise in den Branchen Logistik, Managed-Hosting, Groß- & Einzelhandel und Dialogmarketing verfügt er über vielseitige und intensive Einsichten in verschiedene Softwarelandschaften und in ganz unterschiedliche Geschäftsanforderungen. Da sein zweites Herz für agile Arbeitsmethoden schlägt, integriert und optimiert er diese in seinen Projekten.

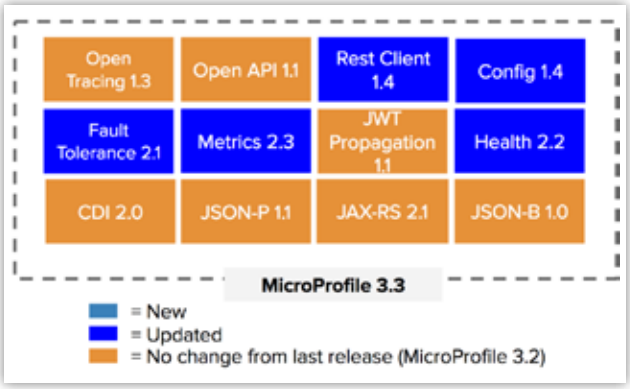
Webseite: https://anderscore.com
Blog: https://maik-wolf.de
Twitter: @da_mwolf
E-Mail: maik.wolf@anderscore.com

Jakarta-EE vs. Spring - schon wieder?

Microservices mit Spring-Boot sind für viele Java-Entwickler zur Norm geworden und werfen einen erheblichen Schatten auf Jakarta-EE, wie Java-EE jetzt heißt. Durch seine beständige Evolution schafft es Spring immer wieder neue Entwicklungsansätze in die Welt zu tragen. Somit passt es sich den neuen Gegebenheiten und unseren sich ständig ändernden Anforderungen an. Spring-Boot hat sicherlich erheblich dazu beigetragen, dessen Grundlagen in der Blütezeit der Microservices entstanden sind. Gerade mit diesem Framework lassen sich schlanke Anwendungen schreiben, die in Self-Contained-JARs ausgeliefert werden. Diese und noch weitere Komponenten aus dem Spring-Ökosystem eignen sich nahezu perfekt, um den Kriterien eines Microservice gerecht zu werden.

Im Schatten steht nun Jakarta-EE mit seinen starren Spezifikationen und dem dazugehörigen, schwergewichtigen Application-Server. Doch genau diesem Nachteil haben sich viele Anbieter solcher Produkte bereits angenommen. Hervorgehoben sei dort die Entwicklung um Thorntail (zu Anfang Wildfly-Swarm). Nach dem Motto Just-enough-application-server konnte man sich bei diesem seine sog. Fractions zusammenstellen. Diese bestehen lediglich aus einzelnen Komponenten von Wildfly sowie Implementierungen anderer Frameworks. Somit war es tatsächlich möglich, einen leichtgewichtigen Application-Server in einer Self-Contained-JAR zu erzeugen. Der Weg zum schlanken Microservice wurde also mit diesen und anderen Frameworks bereits geebnet.

Zu dem neuen Hype um JEE-Microservices hat unter anderem auch das im Frühjahr 2019 erschienene Framework Quarkus beigetragen. Unter dem Slogan „Supersonic-Subatomic-Java" verspricht das von Red Hat gesponserte Open-Source-Projekt „Container-First, Unifies Imperative and Reactive,



Bestandteile von MicroProfile 3.3[3] (Abb. 1)

Developer Joy & Best-of-Breed Libraries and Standards". Quarkus ist somit die konsequente Evolution von Just-enough-application-server hin zu Container-First. Durch seine geringe Startup-Zeit im Millisekundenbereich und der geringen Größe der Self-Contained-JARs, eignet es sich perfekt zur automatischen Skalierung in Container-getriebenen Orchestrierungs-Tools, wie zum Beispiel Kubernetes. Nun werden auch hier sog. Extensions verwendet, die gängige Funktionalitäten zum Betreiben des Microservice mitbringen. Auch dadurch fühlt sich Quarkus an vielen Stellen an wie Thorntail, was daran liegen mag, dass Design und Implementierungsideen von Thorntail 4.x Proof-of-Concept in Quarkus eingeflossen sind[1]. Mit diesen Extensions werden die beworbenen „Best-of-Breed Libraries and Standards" eingebracht. Unter anderem lässt sich hier die Spezifikation für Microservices aus dem Projekt Eclipse-MicroProfile verwenden. MicroProfile seinerseits hat sich der Herausforderung angenommen, einen Standard für die Entwicklung von Microservices auf Basis von Java zu definieren. Wie schon Java-EE ist auch MicroProfile eine Sammlung von Spezifikationen, mit welcher die Entwicklung von Microservices zu einer einfachen Aufgabe werden soll. Die Sammlung beinhaltet bewährte JEE-Technologien wie CDI, JAX-RS, JSON-B & JSON-P, aber auch neue, auf Microservices zugeschnittene Spezifikationen wie Health-Check, Metrics, Config ect. Siehe JAVAPRO 2-2019, „MicroProfile für Microservices mit Java-EE„ von Alexander Ryndin[2]. Mit diesen Erneuerungen kann Java-EE wieder in den Ring mit Spring steigen und sich im Kampf um das bessere Framework für Microservices messen.

Vergleichskriterien

Um die beiden Technologien miteinander vergleichen zu können, bedarf es zunächst geeigneter Kriterien. Was aber macht einen klassischen Microservice aus? Einerseits handelt es sich per Definition um ein unabhängig lauffähiges und deploybares Artefakt. Mithilfe externer Konfiguration passt es sich dynamisch an seine jeweilige Laufzeitumgebung an, wie zum Beispiel einem Kubernetes-Cluster. Da ein Service bekanntlich selten allein kommt, benötigt er weiterhin auch Schnittstellen zur Außenwelt. Aufgrund hoher Interoperabilität werden diese in der Praxis gerne als REST-Endpoints realisiert. Um jederzeit über die korrekte Funktion und Verwendung des Service im Bilde zu sein, werden ferner ein Health-Check sowie möglichst einfach abrufbare Metriken benötigt. Natürlich handelt es sich bei den genannten Kriterien nur um eine subjektive Zusammenstellung, die keinen Anspruch auf Vollständigkeit oder Allgemeingültigkeit erhebt. Dennoch bietet sie eine erste Entscheidungsgrundlage zur Auswahl einer passenden Technologie. Nach Festlegung der Kriterien kann die Gegenüberstellung beginnen.

Spring-Boot sieht sich selbst als Wegbereiter für unabhängige, produktionsreife Anwendungen, die sich einfach starten lassen. Diesem Anspruch folgend bestehen die mittels hauseigenem Initializr erstellten Projekte im Wesentlichen nur aus einer startbaren Application-Klasse nebst zugehöriger Konfigurationsdatei **(Listing 1)**.

(Listing 1)

```
.
├── src
│   └── main
│       ├── java
│       │   └── com.anderscore.spring
│       │       └── Application.java
│       └── resources
│           └── application.properties
└── pom.xml
```

Dank Spring-Boot-Maven-Plugin genügt ein simples **`mvn spring-boot:run`** auf der Kommandozeile, um den Microservice mitsamt eingebettetem Web-Server (meist ein Tomcat) auf Port 8080 hochzufahren. Nicht viel anders verhält es sich bei Projekten auf Basis von MicroProfile bzw. Quarkus, welche sich ihrerseits wahlweise über den MicroProfile- bzw. Quarkus-Starter oder mittels Kommandozeile unter Verwendung des Quarkus-Maven-Plugins **`mvn quarkus:create`** generieren lassen **(Listing 2)**.

(Listing 2)

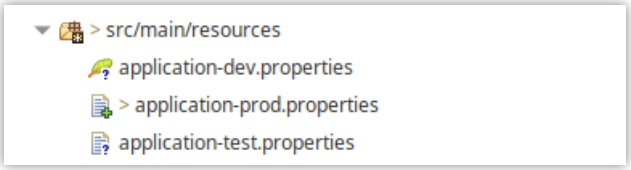
```
.
├── src
│   └── main
│       ├── docker
│       │   ├── Dockerfile.jvm
│       │   └── Dockerfile.native
│       ├── java
│       │   └── com.anderscore.microprofile
│       │       └── RestResource.java
│       └── resources
│           └── application.yml
└── pom.xml
```

Ins Auge fallen im Vergleich lediglich zwei vorbereitete Docker-Files, welche der Kubernetes-Native-Philosophie des Quarkus-Projektes Rechnung tragen. Bevor über das Quarkus-Maven-Plugin auch hier ein eingebetteter Web-Server gestartet wird, erfolgt optional eine native Kompilierung **(Listing 3)**.

(Listing 3)

```
$ mvn package -Pnative && ./target/microprofile-quarkus
$ mvn quarkus:dev
```

Variable Konfigurationswerte lassen sich bei Spring bequem in umgebungsspezifische Properties- oder YAML-Dateien auslagern und an nahezu beliebiger Stelle injizieren (Abb. 2). Profiles ermöglichen überdies auch die gezielte Aktivierung oder Deaktivierung einzelner Beans (Listing 4).

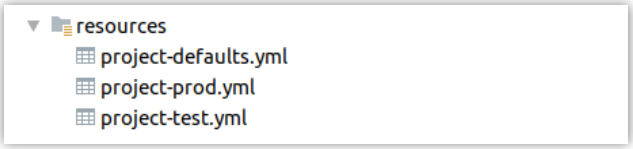


(Abb. 2)

(Listing 4)

```
mvn spring-boot:run -Dspring.profiles.active=test
```

Wenig Anlass zur Überraschung bietet der anschließende Blick auf Quarkus, weil die MicroProfile-Config-Spezifikation diesem bewährten Prinzip folgt:



(Abb. 3)

(Listing 5)

```
$ java -jar myapp-quarkus.jar -Stest
```

RESTful Webservices werden bei Spring (Boot) als **`@RestController`** mit zugehörigen Methoden definiert und zur Laufzeit über einen Component-Scan des Application-Contextes erfasst (Listing 6).

(Listing 6)

```
@RestController
@RequestMapping("/tasks")
public class TaskController {

@Autowired
private TaskRepository taskRepository;

@PostMapping
@ResponseStatus(CREATED)
public void createTask(@RequestBody Task task) {
   taskRepository.save(task);
  }

@GetMapping("/{id}")
public Task findTask(@PathVariable long id) {
   return taskRepository.findById(id)
      .orElseThrow(() -> new NotFoundException(id));
  }

// ...
```

Seit jeher ähneln die verwendeten Annotationen jenen des JAX-RS-Standards, welcher auch MicroProfile und damit Quarkus zugrunde liegt (Listing 7).

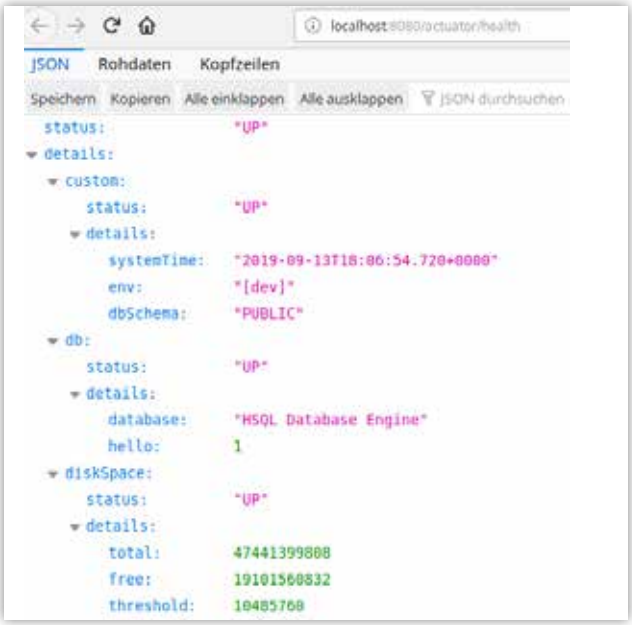
(Listing 7)

```
@Path("/tasks")
@Produces(MediaType.APPLICATION_JSON)
public interface TaskResource {

    @POST
    @Path("/{id}")
    void createTask(Task task);

    @GET
    @Path("/{id}")
    Task findTask(
            @PathParam("id") Long id
    );
// ...
```

Um zur Laufzeit mehr über seinen Microservice zu erfahren, stellt Spring das Actuator-Projekt bereit. Dieses lässt sich leicht als Dependency über einen Starter einbinden und stellt daraufhin eine Vielzahl nützlicher Endpunkte über REST und JMX bereit. Ein Blick auf den Health-Endpunkt genügt bereits, um wichtige Lebensfunktionen zu messen (Abb. 4).



(Abb. 4)

Sind diese nicht aussagekräftig genug, lassen sich beliebig komplexe **`HealthIndicators`** selbst definieren und als Bean in den Context einbinden (Listing 8).

(Listing 8)

```
@Component
public class CustomHealthIndicator extends AbstractHealthIndi-
cator {
```

```
    @Autowired
    private Environment environment;

    @Autowired
    private DataSource dataSource;

     @Override
     protected void doHealthCheck(Builder builder) throws Exception {
       builder.up()
              .withDetail("systemTime", new Date())
              .withDetail("env",
                    Arrays.toString(environment.getActiveProfiles()))
              .withDetail("dbSchema",
                    dataSource.getConnection().getSchema());
      }
}
```

MicroProfile verfügt mit der Health-Spezifikation über ein vergleichbares Konzept, unterscheidet jedoch grundsätzlich zwischen Liveness (Service läuft) und Readiness (Service ist bereit). Entsprechende Endpunkte stellen auch hier alle relevanten Informationen für einen Health-Check zur Verfügung (Listing 9). Implementierungsabhängige Kennzahlen werden hingegen unter **/vendor** angeboten (Listing 10).

(Listing 9)

```
{
"gc.total;name=G1 Young Generation1": 15,
"gc.total;name=G1 Old Generation1": 0,
"cpu.systemLoadAverage": 1.42,
[...]
}
```

(Listing 10)

```
{
"bufferPool.usedMemory;name=mapped1": 0,
"bufferPool.usedMemory;name=direct1": 368640,
"memoryPool.usage.max;name=CodeHeap 'profiled nmethods'1":
16832896,
[...]
}
```

Ferner bietet auch MicroProfile die Möglichkeit, selbst definierte Metriken zu erheben und unter einem eigenen Endpoint (relativ zu **/application**) zur Verfügung zu stellen. Sollen etwa die Aufrufe einer bestimmten fachlichen oder technischen Methode gezählt werden, lässt sich dies leicht mithilfe der **@Counted** Annotation bewerkstelligen (Listing 11). Mittels **@Gauge** ist auch eine dynamische Berechnung zum Zeitpunkt des Abrufs möglich (Listing 12).

(Listing 11)

```
@Counted(unit = MetricUnits.NONE,
        name = "tasksCreated",
        absolute = true,
        displayName = "Created items",
        description = "Metrics to show how many times createTask
        method was called.",
        tags = {"tasks=create"}
        )
@POST
@Path("/{id}")
public void createTask(Task task) {
    //...
}
```

(Listing 12)

```
@Inject
@ConfigProperty(name="defaultEstimation")
private Long defaultEstimation;

@Gauge(unit = "Hour", name = "defaultEstimation", absolute =
true)
public Long getDefaultEstimation() {
    return defaultEstimation;
}
```

Zur Bestimmung der (zeitlichen) Lastverteilung kann ferner die Annotation **@Metered** herangezogen werden (Listing 13). Zu guter Letzt bietet **@Timed** von Haus aus sogar ein wenig Arithmetik (Listing 14).

(Listing 13)

```
@Metered(
        name = "findTask",
        unit = MetricUnits.MINUTES,
        description = "Metrics to monitor findTask method.",
        absolute = true
)
@GET
@Path("/{id}")
public Task findTask(
        @PathParam("id") Long id
) {
    ....
};
```

(Listing 14)

```
@Metered(
        name = "findTask",
        unit = MetricUnits.MINUTES,
        description = "Metrics to monitor findTask method.",
        absolute = true
)
@GET
@Path("/{id}")
public Task findTask(
        @PathParam("id") Long id
) {
    ....
};
```

Fazit technologisch:

Welche Technologie eignet sich denn nun besser für die Realisierung eines Microservice? Wie schon so oft präsentiert sich Spring (Boot) einmal mehr als Vorreiter und darf nach einigen Jahren intensiver Erprobung in der Praxis guten Gewissens als absolut produktionstauglich bezeichnet werden. Spätestens seit Einführung der Auto-Configuration hält sich der erforderliche Glue-Code zum Aufsetzen eines Microservice in sehr überschaubaren Grenzen, sodass der Entwickler sich nun endgültig auf die Implementierung der Fachlogik konzentrieren kann. Insgesamt präsentiert sich das Spring-Ökosystem heutzutage als derart mächtig, dass sich schwerlich ein Bereich findet, welcher noch nicht durch ein entsprechendes Projekt nebst Starter abgedeckt wird. Schafft der Einsatz solcher Technologie im konkreten Projekt einen Mehrwert, kann dies bereits ein Entscheidungskriterium sein. An dieser Stelle lässt sich insbesondere Spring-Data heranführen, welches die Implementierung eigener Logik für Datenbankzugriffe nahezu überflüssig macht. Eine vergleichbare Funktionalität bietet MicroProfile aktuell noch nicht an. Losgelöst davon existiert bei Quarkus mit Panache zwar ein ähnliches Konzept, das aber mit einigen Restriktionen einhergeht, wie zum Beispiel Public-Attribute in den Entities. Seit jeher bestand eine Maxime des Spring-Frameworks auch darin das Rad nicht neu zu erfinden, sondern stattdessen die Einbindung anderer, am Markt etablierter Libraries und Frameworks (Hibernate, Netflix OSS etc.) zu vereinfachen. Ist die Integration spezieller Technologien angedacht, sollte deren Unterstützung ebenfalls berücksichtigt werden. Auf der anderen Seite reduziert die Verwendung vordefinierter Starter- und Parent-POMs zwar einen eigenen Konfigurationsaufwand, erzeugt aber auch starke Abhängigkeiten, die nicht außer Acht gelassen werden sollten.

Dass totgesagte länger leben, beweist das Trio aus Jakarta-EE, MicroProfile und Quarkus eindrucksvoll. Die meisten enthaltenen Enterprise-Standards (CDI, JAX-RS, ect.) sind nicht nur hochgradig praxiserprobt, sondern auch vollständig durchspezifiziert. Vielen Entwicklern dürfte ein Großteil eines solchen Frameworks daher allzu bekannt vorkommen, sodass leicht auf bereits gesammeltes Wissen und Know-How zurückgegriffen werden kann. Vervollständigt werden diese Standards durch neue, vollständig am Bedarf moderner Microservices orientierter Spezifikationen wie etwa Health, Metrics oder Fault-Tolerance. Die Einbindung und Konfiguration von Third-Party-Libraries ist nicht mehr erforderlich, weil Implementierungen wie Quarkus diese bereits über entsprechende Extensions mitliefern. Darüber hinaus beschränkt sich die Konfiguration auch bei MicroProfile auf das Wesentliche. Beides vereinfacht das Setup deutlich und kann so zu einer Kostenersparnis führen.

Stichwort Ersparnis: Speziell Quarkus setzt auf GraalVM und kann daher nativ kompiliert werden - inklusive Live-Coding im Development-Modus. Im Ergebnis reduzieren sich Startzeit und Round-Trips nach Änderungen drastisch. Jene Tage, in denen Java-EE als schwergewichtig, langsam und kompliziert galt, gehören damit endgültig der Vergangenheit an. Wie schon bei klassischen Application-Servern wirkt sich die Vorauswahl konkreter Implementierungen durch den jeweiligen Anbieter trotz Extensions negativ auf die Flexibilität aus. Manche Entwickler sprechen in diesem Zusammenhang zuweilen von einer Hassliebe.

Aus technischer Sicht ergibt sich also das Bild, dass man mit beiden Technologien wunderbar Microservices entwickeln kann. Man muss daher nicht zwingend von Java-EE auf Spring oder umgekehrt umsatteln.

Fazit organisatorisch:

Viel mehr spielen auch organisatorische Faktoren eine Rolle. Insbesondere ist es ratsam abzuwägen, welche Technologie die Umsetzung individueller Anforderungen besser unterstützt. Betrachten wir zunächst Spring, so kann man dieses flexibel auf die speziellen Anforderungen innerhalb der Anwendung anpassen. Durch seine einfache Integration anderer Libraries und Frameworks kann der Funktionsumfang bei Bedarf schnell erweitert werden. Auf der anderen Seite sollte aber Wert daraufgelegt werden, dass eben diese tatsächlich auch einen Großteil der Anforderungen abdecken. Bindet man stattdessen nach Belieben weitere Projekte ein, kann der eigene Service schnell auch zu einer JAR-Hölle werden. Deshalb gilt es zunächst gründlich zu evaluieren, welche Projekte in welchem Umfang die Umsetzung der eigenen Anforderungen unterstützen. Für diese Aufgabe werden jedoch Mitarbeiter benötigt, die ausreichend Zeit und Wissen mitbringen.

Auf der anderen Seite haben wir das Trio aus Jakarta-EE, MicroProfile und Quarkus, das zu vielen Anforderungen an Microservices schon die passenden Implementierungen durch Libraries mitbringt. Diese haben sich in der Praxis bewährt und wurden von der Community als tauglich befunden. An dieser Stelle wird die Zeit der Evaluation gespart und die Mitarbeiter können sich - gemäß dem ursprünglichen Versprechen von Java-EE - auf die eigentliche Geschäftslogik konzentrieren. Problematisch wird es bei Anforderungen die vom Standard abweichen. Dann erhöht sich der Implementierungsaufwand schnell um ein Vielfaches.

Weiterhin sollte man die Vorlieben der eigenen Mitarbeiter nicht außer Acht lassen. Auch wenn beide Technologien ihre individuellen Vor- und Nachteile besitzen, fühlen sich Entwickler gewöhnlich entweder im Spring- oder im JEE-Ökosystem zu Hause. Das Wissen, welches man sich über Jahre angeeignet hat, sollte weiterhin sinnvoll genutzt und für die Umsetzung von Microservices und deren neuen Herausforderungen herangezogen werden. So muss auch kein Mitarbeiter befürchten, sich unverhofft in einer Umgebung wiederzufinden, mit der er nicht vertraut ist.

Quellen:

1 Thorntail: https://bit.ly/Thorntail
2 JAVAPRO-Artikel: https://bit.ly/JAVAPRO-a2
3 Bestandteile von MicroProfile 3.3: https://bit.ly/microprofile-m3