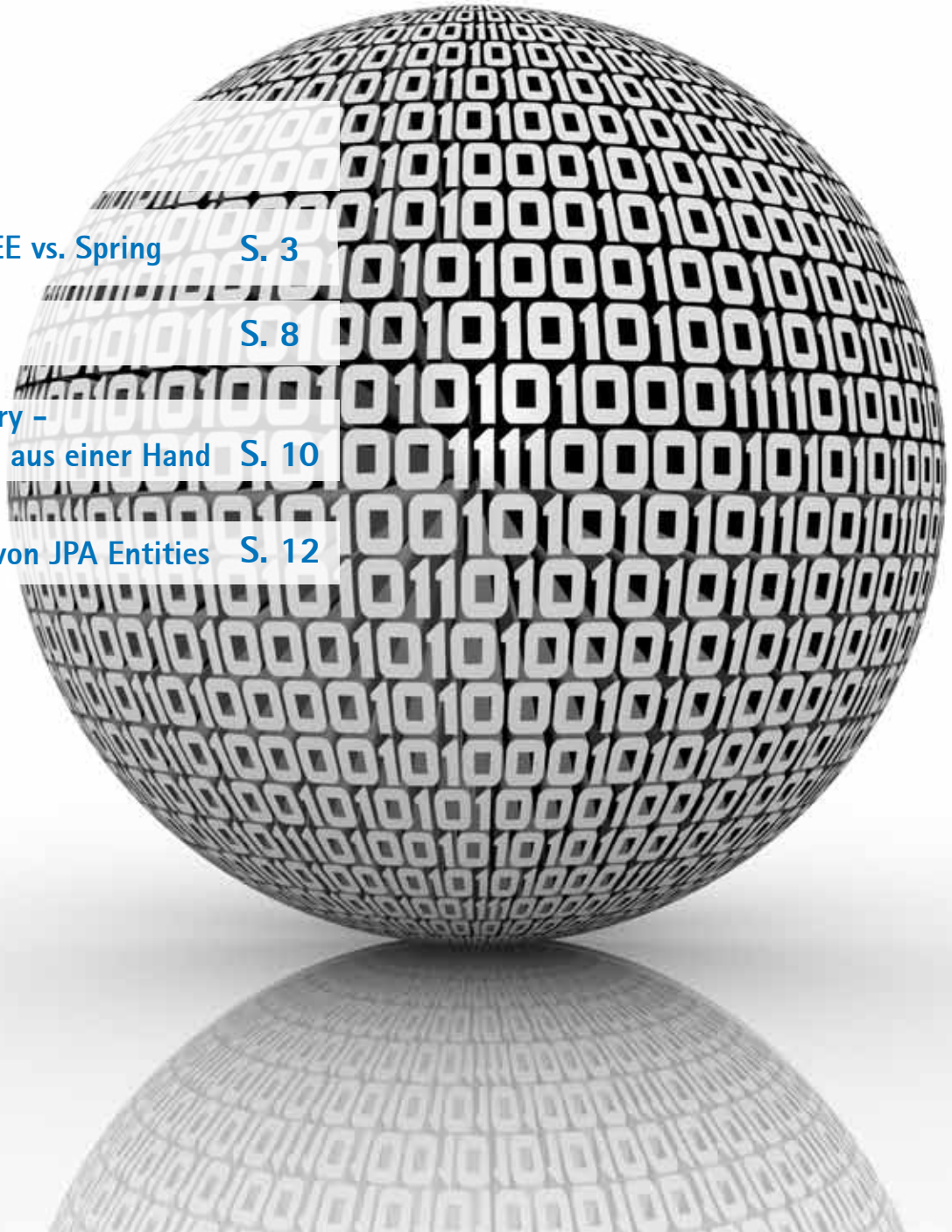
Ausgabe Herbst 2020

www.gedoplan.de

GEDOPLAN aktuell

In dieser Ausgabe:



Liebe Leserin, lieber Leser,

in einem Gastbeitrag stellt Daniel Krämer, anderscore GmbH, die alles entscheidende Frage: "Mit welcher Java Technologie bauen wir unsere Services?" und vergleicht Jakarta EE und Spring unter dem Schwerpunkt MIcroservices.
Es gibt viel Neues bei GEDOPLAN IT Training. Wir stellen Ihnen unsere beiden neuen Dozenten, Anke Tatus und Markus Pauer vor.
Und wir berichten von einer echten Success Story. Über mehrere Jahre haben wir Hapag-Lloyd bei der Migration einer unternehmensweiten Großapplikation auf eine moderne Java EE-Architektur in allen Phasen unterstützt. Dazu gehörte u. a. die Durchführung der Schulungen der über 150 Anwendungsentwickler.
Unsere Webseite haben wir neu strukturiert und vor allem die Suche nach dem für Sie passenden Kurs erleichtert.
Dirk Weil, GEDOPLAN GmbH, beschäftigt sich mit der "Identität und Gleichheit von JPA Entities" und gibt wichtige Tipps für die Implementierung.

Viel Spaß beim Lesen!

Ulrich Hake | ulrich.hake@gedoplan.de

Ulrich Hake

Termine

Expertenkreis Java

Thema: **Java on Tracks – Modellbahnsteuerung mit JEE, MicroProfile und self-contained Systems**
Ort: GEDOPLAN GmbH (remote)
Termin: Donnerstag, 03.12.2020 | 18:00 - 19:30 Uhr
Referent: Dirk Weil, GEDOPLAN GmbH

Thema: **Hapag-Lloyd: Vom Monolithen in die verteilte JEE-Welt**
Ort: GEDOPLAN GmbH (remote)
Termin: Donnerstag, 05.11.2020 | 18:00 - 19:30 Uhr
Referent: Jens Seekamp, GEDOPLAN GmbH

Thema: **Thema: Schnell, schneller, Quarkus!**
Ort: GEDOPLAN GmbH (remote)
Termin: Donnerstag, 27.08.2020 | 18:00 - 19:30 Uhr
Referent: Dirk Weil, GEDOPLAN GmbH

Vorträge

Thema: **JEE und Micro – kein Widerspruch!**
Ort: Developer Week (remote)
Termin: Donnerstag, 05.11.2020 | 10:00 - 11:30 Uhr
Referent: Dirk Weil, GEDOPLAN GmbH

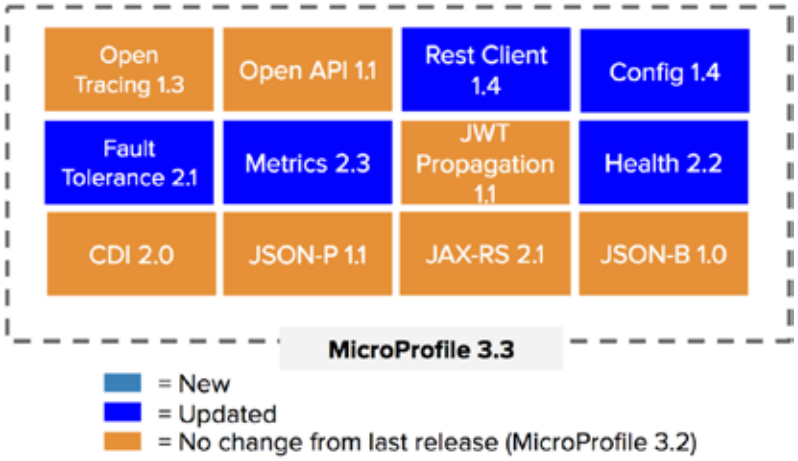
Microservices – Jakarta EE vs. Spring

"Jakarta EE ist einfach viel zu schwergewichtig für Microservices!" Hat Jakarta EE etwa immer noch (zurecht?) den Ruf, schwergewichtig zu sein und somit keine Chance, für diesen Architekturstil überhaupt in Betracht gezogen zu werden? Ist das Spring Framework weiterhin die einzig richtige Antwort auf die Frage: "Mit welcher Java Technologie bauen wir unsere Services?" Einmal mehr stehen sich die ewigen Duellanten Jakarta EE und Spring gegenüber, dieses Mal jedoch mit dem Fokus auf Microservices.

Von Daniel Krämer

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. Durch seine beständige Evolution schafft es Spring, immer wieder neue Entwicklungsansätze in die Welt zu tragen und sich somit den neuen Gegebenheiten unserer sich ständig ändernden Anforderungen anzupassen. Beträchtlich dazu beigetragen hat sicherlich Spring Boot, 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, 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-jar eignet es sich perfekt zur automatischen Skalierung in Container-getriebenen Orchestrierungstools wie z. B. 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. 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 JEE 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, usw. An dieser Stelle sei der Artikel "MicroProfile für Microservices mit Java-EE" aus der JavaPro Ausgabe 2-2019 von Alexander Ryndin empfohlen. (https://thorntail.io/posts/thorntail-community-announcement-on-quarkus/).
Mit diesen Erneuerungen im Schlepptau hat es sich JEE verdient, wieder in den Ring mit Spring zu steigen und sich im Kampf um das bessere Framework für Microservices zu messen.



Vergleichskriterien

Um die beiden Technologien überhaupt miteinander vergleichen zu können, bedarf es allerdings zunächst geeigneter Kriterien. Was aber macht einen klassischen Microservice aus? Einerseits handelt es sich per Definition um ein unabhängig lauffähiges und deploybares Artefakt, das sich mithilfe externer Konfiguration dynamisch an seine jeweilige Laufzeitumgebung (z. B. ein Kubernetes-Cluster) anpasst. 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 nun also beginnen.
Spring Boot sieht sich selbst als Wegbereiter für unabhängige, produktionsreife Anwendungen, die sich "einfach starten" lassen. Diesem Anspruch folgend bestehen die mittels hauseigenem Initializer erstellten Projekte im Wesentlichen nur aus einer startbaren Application-Klasse nebst zugehöriger Konfigurationsdatei:

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

Dank Spring-Boot-Maven-Plugin genügt bereits ein simples mvn spring-boot:run auf der Kommandozeile, um den Microservice mitsamt eingebettetem WebServer (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:

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

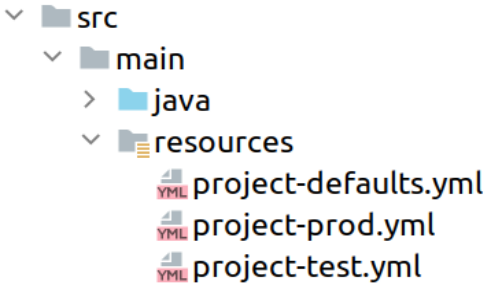
Ins Auge fallen im Vergleich lediglich zwei vorbereitete Dockerfiles, welche der "Kubernetes Native"-Philosophie des Quarkus-Projektes Rechnung tragen. Bevor über das Quarkus-Maven-Plugin auch hier ein eingebetteter WebServer gestartet wird, erfolgt (optional) eine native Compilierung:

```
$ 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. Profile ermöglichen überdies auch die gezielte Aktivierung oder Deaktivierung einzelner Beans:


```
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 einfach folgt:




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

RESTful WebServices werden bei Spring (Boot) als @RestController mit zugehörigen Methoden definiert und zur Laufzeit über einen ComponentScan des ApplicationContextes erfasst:

```
@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));
        }

// ...
```

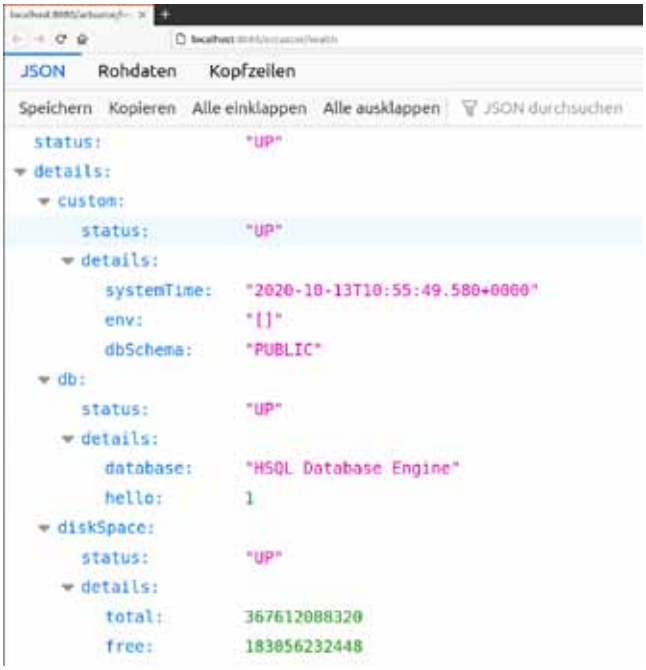
Seit jeher ähneln die verwendeten Annotationen jenen des JAX-RS-Standards, welcher auch MicroProfile und damit Quarkus zugrunde liegt:

```
@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:



Sind diese nicht aussagekräftig genug, lassen sich beliebig komplexe HealthIndicators selbst definieren und als Bean in den Context einbinden:

```
@Component
public class CustomHealthIndicator extends AbstractHealthIndicator {

        @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 HealthCheck zur Verfügung:

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

Implementierungsabhängige Kennzahlen werden hingegen unter /vendor angeboten:
(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:

```
@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) {
   //...
}
```

Mittels @Gauge ist auch eine dynamische Berechnung zum Zeitpunkt des Abrufs möglich:
(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)

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

Zu guter Letzt bietet @Timed von Hause aus sogar bereits ein wenig Arithmetik:
(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 Technologie:

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 AutoConfiguration 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 (z. B. public Attribute in den Entities) einhergeht. 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 (wie z. B. Hibernate oder Netflix OSS) 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 eigene Konfigurationsaufwände, erzeugt aber auch starke Abhängigkeiten, die nicht außer Acht gelassen werden sollten.
Dass (vermeintlich) Totgesagte länger leben, beweist das Trio aus Jakarta EE, MicroProfile und Quarkus wiederum eindrucksvoll. Die meisten enthaltenen Enterprise-Standards (wie etwa CDI und JAX-RS) 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 orientierte Spezifikationen wie etwa Health und Metrics oder auch das hier nicht näher betrachtete Fault Tolerance. 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 JEE als schwergewichtig, langsam und kompliziert galt, gehören damit also endgültig der Vergangenheit an. Wie schon bei klassischen ApplicationServern wirkt sich die Vorauswahl konkreter Implementierungen durch den jeweiligen Anbieter (trotz Extensions) allerdings negativ auf die Flexibilität aus. Manche Entwickler sprechen in diesem Zusammenhang zuweilen von einer Liebes-Hass-Beziehung.

Fazit Organisatorisch:

Aus technischer Sicht ergibt sich also das Bild, dass man mit beiden Technologien wunderbar Microservices entwickeln kann. Man muss daher nicht zwingend von JEE auf Spring oder umgekehrt umsatteln. 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 darauf gelegt 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 welchen 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, welche wir üblicherweise an Microservices stellen, 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 JEE - auf die eigentliche Geschäftslogik konzentrieren. Aber wehe, man hat Anforderungen, welche von den Standards 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, so fühlen sich Entwickler gewöhnlich eher im Spring- oder JEE-Ökosystem daheim. Das Wissen, welches sie sich über Jahre angeeignet haben, 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.

Daniel Krämer [daniel.kraemer@anderscore.com]
Daniel Krämer ist als Software-Entwickler für die anderScore GmbH tätig und begeistert sich für durchdachte Software-Architektur und sauberes Design. In seinen Projekten setzt er sich überwiegend mit Fragestellungen rund um Migration und Integration sowie Java- und Web-Entwicklung auseinander. In Ergänzung zur Arbeit mit dem Code macht es Daniel auch Spaß, seine Kenntnisse und Erfahrungen mit anderen Entwicklern im Rahmen von Vorträgen, Artikeln und Trainings zu teilen.

GEDOPLAN
IT Training

Unsere Trainer

In unserer neuen Rubrik stellen wir Ihnen in loser Reihenfolge unsere Dozenten vor. Sie sollen schließlich wissen, mit wem Sie es zu tun haben.

Alle unsere Dozenten zeichnet aus, dass sie erfahrene Java-Experten sind und ihr Wissen in vielen Projekten mit unterschiedlichen Schwerpunkten erworben haben und auch zukünftig erweitern.

Markus Pauer

Markus Pauer ist als Consultant, Trainer und Entwickler tätig und besitzt eine langjährige Erfahrung im Bereich Java EE. Seine Schwerpunkte liegen aktuell in der Fullstack-Entwicklung sowie in der Integration unterschiedlicher Software-Systeme. Seine Projekterfahrung lässt ihn auf Kenntnisse in Backendtechnologien der Enterprise Java Entwicklung zurückgreifen. Im Bereich der Frontend-Entwicklung besitzt er fundierte Kenntnisse der Webtechnologien Java Server Faces (JSF), Primefaces sowie Angular. Er ist als Trainer für alle Jakarta EE Themen tätig.

Beispielprojekte:

- Entwicklung einer mobilen App zur Steuerung der Handwerkspartner eines Energiedienstleisters

  Konzeption und Realisierung einer Inventuranwendung, bestehend aus einem Frontend auf einem mobilen Datenerfassungsgerät und einem Java-EE-Backend. Das Frontend wurde als Webanwendung im Look&Feel des mobilen Gerätes mit JSF und JavaScript gestaltet. Das Backend wurde auf Basis des JavaEE-Servers JBoss EAP mit Anbindung an die Warenwirtschaft entwickelt.

  Aufgabenschwerpunkt: Konzeption, Design, Implementierung, Dokumentation, Test

  Technologien: Spring Boot, Angular, Netbeans, Visual Studio Code, Docker, Maven, Git

- Neuentwicklung eines B2B-Kundenportals in der Automatenindustrie

  Das bestehende B2B-Portal auf Basis von Oracle Portal sollte durch eine Portallösung mit Hilfe von Liferay Portal abgelöst werden.
  Dazu wurde das Kundenportal vollständig neu entwickelt, da die alte Portaltechnologie nicht mehr weiterverwendet werden konnte. Die Oberflächen-Portlets wurden mit Java Server Faces (JSF) und der Liferay-Faces-Bridge in das neue Portal integriert. Für das Backend des Portals kam eine Microservice-Architektur zum Einsatz, um eine Isolation der einzelnen Funktionsbereiche im Betrieb zu gewährleisten.

  Die bestehende Anbindung an das SAP-System wurde in diesem Zusammenhang modernisiert und optimiert. Die Kommunikation zwischen den Systemen erfolgte über einen Connector, der die Nachrichten zwischen den beiden Systemen asynchron verarbeiten konnte.

  Aufgabenschwerpunkt: Beratung und Entwicklung

  Technologie: Java EE, Liferay Portal, Java Server Faces (JSF), Context and Dependency Injection (CDI), Java Persistence API (JPA), Subversion

Markus Pauer hält im Februar einen JEE Intensivkurs:

JEE 7/8 Intensivkurs
Grundlagen von Jakarta EE / Java EE kompakt aufbereitet für WildFly, OpenLiberty, Payara und KumuluzEE

Termin: 22.02. - 26.02.2021
Ort: Berlin.
Seminarpreis: EUR 2.280,-- statt EUR 2.680,--.

Bitte bei der Buchung einfach ein "GEDOPLAN aktuell" eingeben in das Bemerkungsfeld.

Markus Pauer[markus.pauer@gedoplan.de]

Anke Tatus

Anke Tatus ist als Consultant, Entwicklerin und seit Neuestem auch als Trainerin tätig.

Für viele Jahre arbeitete sie für Kunden in der Stahlbranche und entwickelte dort diverse Lösungen zumeist in Client-Server-Architekturen (Stichpunkte hierzu: Application-Server Wildfly / JBoss, Java Swing GUIs / Schnittstellen zu SAP und produktionsnahen Prozessen).

Erste Trainererfahrungen im Bereich Java SE und Wildfly Administration.

30 Jahre Erfahrung in der IT, seit 5 Jahren bei GEDOPLAN.

Beispielprojekte:

- Neu- und Weiterentwicklung kaufmännischer und produktionsunterstützender Anwendungen in der Stahlindustrie

  Aufbau von Benutzeroberflächen, SAP-Schnittstellen, Ablösung von Alt-Systemen

  Aufgabenschwerpunkt: Beratung, Planung und Umsetzung der Software-Entwicklung, Anwender–Betreuung

  Technologien: Java EE, JBoss, Wildfly, JCO, MySQL

- Planung, Entwicklung und Betreuung von Java-Anwendungen in der metallverarbeitenden Industrie

  Produktionsplanungssystem eines Schweiß-Elektroden-Herstellers
  Prozessplanungs- und –steuerungssystem einer Feuerverzinkungsanlage
  Prozessleitsystem mehrerer Kalt- und Warmwalzwerke

  Aufgaben-Schwerpunkte: GUI-Entwicklung, Schnittstellen zu SAP und zu Produktionsrechnern, DB-Abfragen

Anke Tatus hält im Dezember einen Java Grundlagen-Kurs:

Java Grundlagen (Standard Edition)
Einführung in die objektorientierte Programmierung mit Java SE 11

Termin: 08.12. - 11.12.2020
Ort: Remote.
Seminarpreis: EUR 1.680,-- statt EUR 1.880,--.

Bitte bei der Buchung einfach ein "GEDOPLAN aktuell" eingeben in das Bemerkungsfeld.

Anke Tatus[anke.tatus@gedoplan.de]

Webseite mit neuer Struktur

Wir haben die Webseite überarbeitet und unsere beiden Kernbereiche Schulungen sowie Beratung und Softwareentwicklung unter einer Domain gedoplan.de zusammengeführt.

Die Kurse sind nun in Kategorien eingeteilt und der User kann nach bestimmten Techniken suchen.

Neu ist unser Duo-Training für 2 Kollegen aus einem Team. Der Gedanke dabei ist, zielorientiert beide auf den gleichen Stand zu bringen und individuell auf interne Fragen eingehen zu können und das ohne weitere Teilnehmer.

Wenn der Teilnehmer nur ein bestimmtes Kapitel lernen will, ohne eine ganze Schulung zu besuchen, dann sind unsere neuen "Java-Shortcuts" genau das Richtige. Ein "Java-Shortcut" dauert max 3,5 Stunden und behandelt ein kurzes Thema oder Kapitel.

Hapag-Lloyd Success Story – Beratung und Ausbildung aus einer Hand

Das Projekt

In einem über mehrere Jahre dauernden Projekt hat GEDOPLAN GmbH die Container-Reederei Hapag-Lloyd erfolgreich bei der Migration einer unternehmensweit eingesetzten Großapplikation auf eine moderne Java EE-Architektur unterstützt. Kennzeichnend für das Projekt war eine schrittweise Transformation des Systems von einer Pilotanwendung über die Implementierung von Teilanwendungen. Dies ermöglichte eine ständige Evaluation der gewählten Techniken und der Software-Architektur.
GEDOPLAN begleitete dabei alle Phasen des Prozesses bis hin zur Organisation und Durchführung der Schulungen der über 150 Anwendungsentwickler.

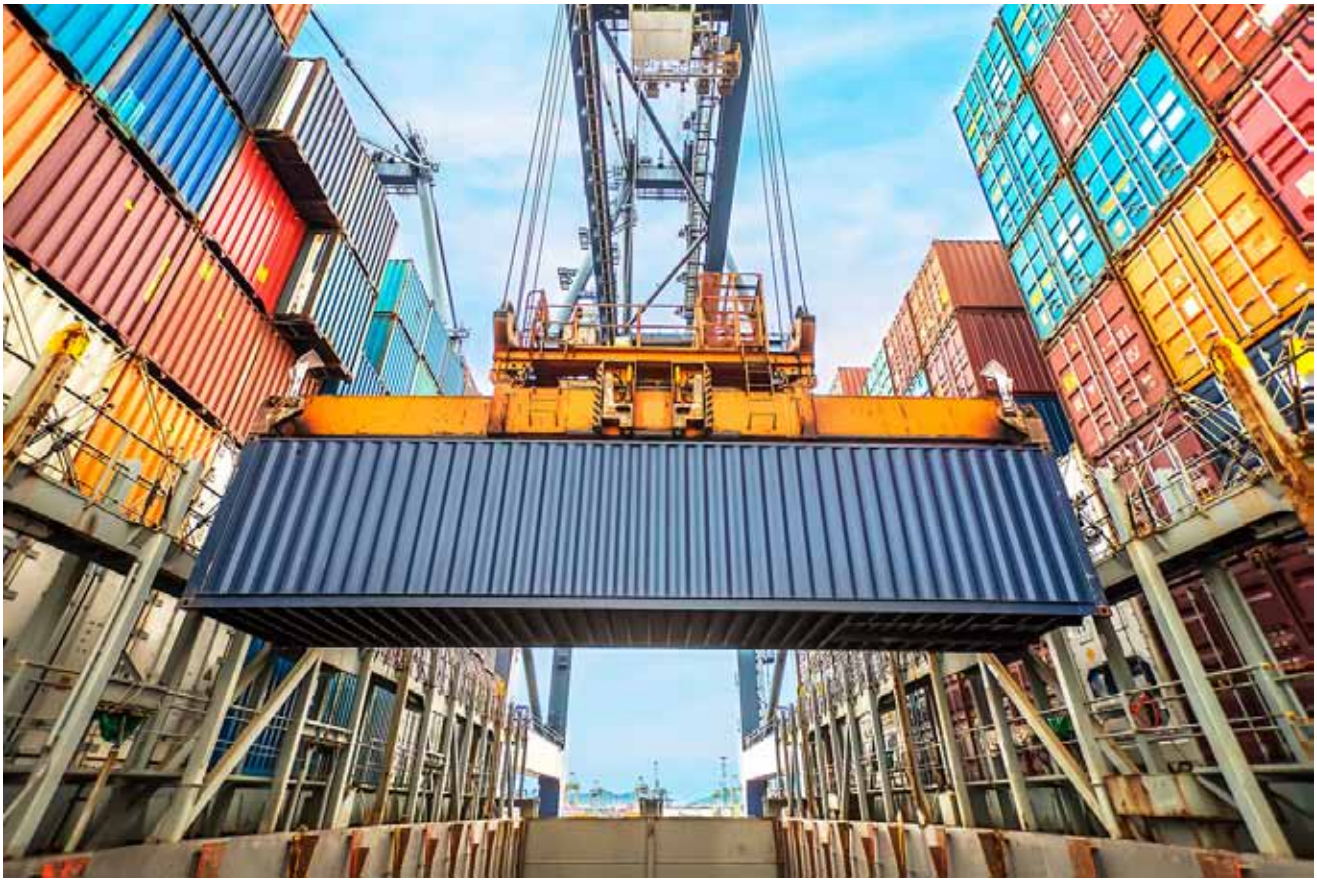
Phasen des Projektes

Die Basis der neuen Software-Architektur stellte eine Pilotanwendung auf Basis der Plattform Java EE dar. Anhand dieser Anwendung sollten Fragen sowohl zur Migration der Großrechner-basierten Anwendung als auch zur künftigen Architektur und der verwendeten Technologien beantwortet werden. Dirk Weil, Geschäftsführer GEDOPLAN GmbH, führte dabei begleitend ein 2-tägiges Review der geplanten Java-EE-Architektur durch.
In der anschließenden Phase wurden erste Teile des neuen Systems implementiert. Fragen zur Anwendungsarchitektur, technischen Machbarkeit und der anwendbaren Konzepte wurden im Rahmen dieses Projektes bearbeitet und beantwortet.

Im Rahmen der Entwicklung einzelner Komponenten wurde schließlich auch die endgültige Zielarchitektur festgelegt:

- Domain-Driven Design (DDD) als grundlegende Vorgehensweise
- Clean Architecture zur Entkopplung von fachlicher Logik und der genutzten Infrastruktur
- Konsequente Modularisierung
- Implementierung einer Referenz-Anwendung zur Evaluierung der Zielarchitektur
- Modellierung des Domänenmodells der Referenz-Anwendung mit UML
- Nutzung der Java-EE-Technologien: CDI, EJB, JPA, JAX-RS, JMS
- Dokumentation der Zielarchitektur inklusive einer Beschreibung der Herleitung und Abwägung der Architektur-Entscheidungen und einem Implementierungsleitfaden für die Java-Entwickler

GEDOPLAN begleitete dabei alle Phasen dieses Teils des Projektes.



Schulung der Mitarbeiter

Parallel zu der beginnenden Realisierung von ersten Teilen des neuen Informations-Systems entwickelte GEDOPLAN in Zusammenarbeit mit Hapag-Lloyd ein durchgängiges Schulungskonzept für die Mitarbeiter und begann mit der Organisation und Durchführung der speziell auf die Anforderungen der neuen Software konzipierten Schulungsmaßnahmen. Dies wurde u. a. dadurch sichergestellt, dass GEDOPLAN-Mitarbeiter in Personalunion sowohl am vorausgehenden Evaluierungs-Projekt beteiligt waren als auch die meisten Seminare durchführten. So zeichneten sich die Seminare auch dadurch aus, dass aktuelle Probleme beispielhaft gelöst werden konnten und das Gros der praktischen Übungen direkten Projektbezug hatte.
Zunächst nahmen Hapag-Lloyd-Mitarbeiter an offenen Seminaren teil. Anschließend führten wir einige Inhouse-Seminare durch. Ende 2017 beauftragte Hapag-Lloyd GEDOPLAN mit der Entwicklung eines Konzeptes für umfassende projektbegleitende Schulungen.
Der von GEDOPLAN entwickelte Schulungsplan für alle Entwickler umfasste schließlich folgende Schulungsmaßnahmen:

- auf die Bedürfnisse von Hapag-Lloyd angepasste Standardschulungen in Java SE und Java EE
- individuell ausgearbeitete 8-tägige spezifische Architekturschulung über die nach Java zu migrierende unternehmensweite Großapplikation (auch in Englisch)
- maßgeschneiderte Firmenschulung im Bereich Tooling, Continuous Integration mit Maven, GIT, JUnit, Jenkins, Artifactory
- unsere Standardseminare „SQL Masterclass - Einsatz von SQL als Applikationssprache" und „Java Persistence API für DBAs- Nutzung und Konfiguration von JPA aus Sicht der Datenbankadministration"

Insgesamt führte GEDOPLAN 175 Schulungstage zum Standardpaket Java SE + Java EE und 160 Schulungstage zum Backend für die mehr als 150 Projektmitarbeiter durch. Und es geht weiter in 2020. Auch in diesem Jahr werden weiterhin Kurse zu den genannten Themen durchgeführt, wegen der Corona-Krise auch remote!

Identität und Gleichheit von JPA Entities

Entities haben ein oder mehrere Id-Attribute für die Abbildung auf den Primary Key in der Datenbank. Wie alle Klassen haben sie zudem die Methoden equals und hashCode, die das Konzept der Gleichheit von Objekten definieren. Zwischen diesen beiden Sichten auf Gleichheit und Identität gibt es Zusammenhänge, die bei der Implementierung beachtet werden müssen.

Von Dirk Weil

JPA-Entitäten enthalten ein Id-Feld (oder -Eigenschaft), das mit @Id versehen ist. Im Folgenden nehme ich ein einzelnes Feld an, aber auch Compound Ids sind möglich.

Entitätsobjekte sind Geschäftsobjekte, daher ist es i. A. nötig, dass wir für Entitätsklassen equals und hashCode implementieren. Ich möchte hier darauf hinweisen, dass es für unseren Geschäftscode sehr unwahrscheinlich ist, eine dieser Methoden direkt aufzurufen, da wir normalerweise keine expliziten Vergleiche von Entitätsobjekten in unserem Geschäftscode benötigen, aber wir verwenden Collections, die ihrerseits equals und hashcode zum Nachschlagen oder Hinzufügen von Einträgen aufrufen.

Wir haben mehrere Erwartungen bezüglich der Semantik von equals und der Collection-Operationen, besonders in Bezug auf Entity-Objekte:

1) Objekte, die auf demselben Datenbankeintrag basieren, sollten in Bezug auf equals gleich sein, und Objekte aus unterschiedlichen DB-Zeilen sollten unterschiedlich sein:

   a) einschließlich Objekten aus verschiedenen Entity-Managern,
   b) einschließlich Detached Objects,
   c) einschließlich neuer (transienter) Objekte,
   d) auch wenn einige Nicht-Id-Attribute geändert wurden (sie werden in "ihren" DB-Datensätzen landen!).

2) Hashcode-basierte Collections (z. B. "HashSet") sollten sich freundlich verhalten, d. h.

   a) das Hinzufügen mehrerer Objekte sollte möglich sein - auch für neue (transiente) Objekte,
   b) die Collection sollte intakt bleiben, selbst wenn in ihr enthaltene Objekte in der Datenbank persistiert werden.

Wenn die Entität eine fachliche Id hat, d. h. dass das Id-Attribut eine geschäftliche Bedeutung hat und z. B. vom Konstruktor festgelegt wird, können die Erwartungen leicht erfüllt werden, indem genau das Id-Attribut in equals und hashCode verwendet wird.

Wenn Sie jedoch alle Felder anstatt nur die Id vergleichen möchten, bricht Ihre Implementierung die Erwartung 1.d. Wenn Sie sich entschließen, equals und hashCode überhaupt nicht zu implementieren und stattdessen die von Object abgeleiteten Methoden zu verwenden, bricht Ihre Implementierung die Erwartungen 1.a. und 1.b.

Als Zwischenfazit implementieren Sie also equals und hashCode in Ihren Entitätsklassen und stützen sie nur auf das / die Id-Attribut(e):

```
@Entity
public class Foo {
  @Id
  private String id;
  ...

  public Foo(String id, ...) {
    this.id = id;
    ...
  }

  public int hashCode() {
    return (this.id == null) ? 0 : this.id.hashCode();
  }

  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    Foo other = (Foo) obj;
    if (this.id == null) {
      return other.id == null;
    }
    return this.id.equals(other.id);
  }
```

Die Sache wird komplizierter, wenn Sie generierte Ids verwenden möchten. JPA unterstützt Sie dabei mit der Annotation @GeneratedValue, die zusätzlich zu @Id auf das Id-Attribut gesetzt werden muss. Der Generator arbeitet für ganzzahlige Zahlenfelder und es ist am besten, auf primitive Typen zu verzichten, um den zusätzlichen Wert null für nicht gesetzte IDs zu erhalten. Sie würden also Integer, Long oder BigInteger verwenden, je nachdem, wie viele Einträge Sie erwarten:

```
@Entity
public class Foo {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;
```

Die Probleme mit generierten Ids in Bezug auf Identität und Gleichheit rühren daher, dass der Entitymanager die Ids spät setzt – vielleicht sogar erst beim Commit der einfügenden Transaktion.

Zunächst verwenden Sie das Id-Feld zum Vergleich in equals und in hashCode weiterhin so, wie Sie es zuvor für fachliche Ids getan haben. Aber wenn Sie das auf genau die oben gezeigte Weise machen,

brechen Sie die Erwartung 2.a, weil equals alle neuen (transienten) Objekte als gleich bewertet, da ihre Ids alle NULL sind.

Sie können dies beheben, indem Sie equals so ändern, dass false zurückgegeben wird, wenn eine der verglichenen Ids noch nicht gesetzt ist.:

```
public boolean equals(Object obj) {
  ...
  if (this.id == null) {
    return false;
  }
  return this.id.equals(other.id);
}
```

Während dies nun die Erwartung 2.a. erfüllt, bricht es die Erwartung 2.b: Wenn Sie einem HashSet einige neue Entitäten hinzufügen und diese danach persistieren, ist die Collection aufgrund der veränderten Hash-Codes beschädigt.

Leider gibt es keine Möglichkeit, dies zu umgehen, wenn generierte IDs verwendet werden! In vielen Anwendungen ist dies allerdings kein Problem, da die Programmsequenz "Mehrere Objekte erstellen", "zu einer hashbasierten Collection hinzufügen", "Objekte persistieren", "Collection verwenden" nicht sehr häufig ist und durch Umstellen der Reihenfolge entschärft werden kann.

Bitte beachten Sie, dass die Verwendung einer hashbasierten Collection möglicherweise nicht direkt sichtbar ist, weil sie bspw. als Relationsattribut in einer anderen Entitätsklasse versteckt ist.

Welche Möglichkeiten haben Sie denn nun, wenn Sie generierte IDs verwenden möchten und Ihre Geschäftslogik unter dem oben diskutierten Hashbasierte-Collection-Problem leidet? Die Lösung ist, eine früh, d. h. im Konstruktor gesetzte Id zu verwenden, die leicht und – im Sinne der Performanz – ohne DB-Zugriff erzeugt werden kann. java.util.UUID bietet die gewünschte Funktionalität an. Diese Standardklasse repräsentiert 128-Bit-Zahlen, die häufig als 35 Zeichen langer Text angegeben werden. java.util.UUID verwendet Zufallszahlen als Basis und bietet eine Verteilung, die Duplikate sehr unwahrscheinlich macht. Es gibt aber auch andere Implementierungen, die z. B. MAC-Adressen und Zufallszahlen verwenden.

```
@Entity
public class Foo {

  @Id
  private String id;

  private String description;

  public Foo(String description) {
    this.id = UUID.randomUUID().toString();
    ...
```

Mit UUIDs picken Sie sich die Rosinen aus dem Kuchen: Sie profitieren von früh gesetzten Ids, die dennoch generierte Werte haben.

Die Größe von UUIDs kann schmerzhaft sein. Sie sind ungefähr viermal so groß wie ein Integer, aber die Speicheranforderungen hängen von Ihrem Datenbankprodukt ab. Denken Sie daran, dass sie Ids sind, somit also jeder Fremdschlüssel in der DB die gleiche Größe hat. Wenn dies ein Problem ist, können Sie auf früh gesetzte UUIDs zurückgreifen, um equals und hashCode zu unterstützen, diese aber nicht mit @Id annotieren. Zusätzlich fügen Sie ein weiteres ganzzahliges Feld als JPA-generierte Id hinzu, d. h. ganzzahliges Attribut, annotiert mit @Id @GeneratedValue. Jetzt enthalten alle Tabellen eine Zahl als Primärschlüssel sowie eine nicht primäre UUID-Spalte, aber alle Fremdschlüssel sind nur (kürzere) Zahlen.
Das Thema Ids in JPA, das auf den ersten Blick einfach erscheint, ist also weit davon entfernt. Beim Entwerfen von Entitätsklassen können Sie das folgende Rezept verwenden:

- Wenn Ihre Entität ein identifizierendes Geschäftsattribut enthält, geben Sie dies als ID an. equals und hashCode nutzen genau dieses Attribut. Alle oben genannten Erwartungen werden erfüllt. (ChemicalElement im Showcase).
- Wenn Sie keine passende Geschäfts-ID finden und ...
  - wenn das oben beschriebene Hashbasierte-Collection-Problem für Ihre Anwendung keine Rolle spielt, verwenden Sie eine mit @Id @GeneratedValue annotierte ID. Lassen Sie equals und hashCode genau dieses Attribut verwenden, aber ändern Sie equals so, dass ungesetzte Ids false als Rückgabewert liefern. Alle oben genannten Erwartungen werden mit Ausnahme von 2b erfüllt. (LabExperiment3NonNullIdEquality im Showcase).
- Wenn Sie früh gesetzte Ids verwenden wollen / müssen, nutzen Sie UUIDs.
  - Wenn die DB-Größe nicht wirklich wichtig ist oder Ihr Modell nur wenige Assoziationen enthält, verwenden Sie die UUIDs direkt als JPA-Ids. Lassen Sie equals und hashCode genau dieses Attribut verwenden. Alle oben genannten Erwartungen werden erfüllt. (LabExperiment4UUIdEquality im Showcase).
  - Wenn Ihnen der Speicherverbrauch von Primär- und Fremdschlüsseln in Ihrer Datenbank wichtig ist, verwenden Sie die UUID nur für equals und hashCode und fügen Sie Ihrer Klasse eine separate mit @Id @GeneratedValue annotierte Id hinzu. Lassen Sie equals und hashCode genau das UUID-Attribut verwenden. Alle oben genannten Erwartungen werden erfüllt. (LabExperiment5AddIdEquality im Showcase).

Unter https://github.com/GEDOPLAN/jpa-equals-demo gibt es ein Showcase, in dem die verschiedenen Optionen demonstriert werden. Dort finden Sie die oben referenzierten Klassen.

Dirk Weil [dirk.weil@gedoplan.de]
Geschäftsführer der GEDOPLAN GmbH
Fachbuchautor, schreibt Artikel für Fachmagazine, hält Vorträge und leitet Seminare und Workshops zu diversen Java-SE-/EE-Themen.

Beratung und Softwareentwicklung

Individualsoftware passt sich Ihrer Organisation an und nicht umgekehrt. Sie erhalten flexible Software mit der Funktionalität, die Ihr Geschäft benötigt. Wir integrieren das neue System in Ihre IT-Landschaft. Hohe Entwicklungsqualität stellt die Wartbarkeit sicher.

Projekt	Ausgangssituation
Prüfsystem für Statistiken der Krankenversicherer	• Plausibilitätsprüfung hart im Programm implementiert • Programmänderung bei Gesetzesänderungen
Individuelles Betriebsdaten-erfassungssystem (BDE)	• handschriftliche Datenerfassung • aufwendige, fehleranfällige Nacherfassung • keine zeitnahen Daten • keine statistischen Auswertungen
Performance-Optimierung	• Performanceprobleme in kritischen Anwendungen
Kundenportal für eine Versicherung	• Kommunikation mit dem Kunden durch die Sachbearbeiter
Frontend für einen Energieversorger	• manuelle Statistikauswertungen • hoher Aufwand zur Erstellung der Auswertungen

Ansprechpartner
Reinhard Brüggemeyer

Kontakt
reinhard.brueggemeyer@gedoplan.de
0521 – 2 08 89 10

Beratung und Softwareentwicklung
gedoplan.de/beratungundsoftwareentwicklung

JEE-Blog
javaeeblog.wordpress.com/

Expertenkreis Java in Bielefeld
gedoplan.de/java-events

JAVA bietet eine ausgereifte, zukunftssichere Entwicklungsumgebung. Hier finden Sie einen Auszug von unseren Projekten, die wir branchenunabhängig für Mittelständische- sowie Großunternehmen durchgeführt haben.

Lösung	Ergebnis
• komplexe Systemlogik zur Abbildung von Regeln • vom Anwender genutzter Regeleditor • die Logik wird in der Datenbank gespeichert • Simulation von Regeländerungen	• Gesetzesänderungen können ohne Programmanpassungen umgesetzt werden • Auswirkungen werden durch Simulation überprüft • es ist immer nachvollziehbar, wann welche Regel genutzt wurde (Revision)
• robuste User Interfaces • Plausibilitätsprüfungen bei Datenerfassung • Schnittstellen zu den Produktionsplanungssystemen • Analyse- und Auswertungstools	• Einsparungen bei der Datenerfassung • bessere Produktionssteuerung durch zeitnahe Daten • direkte Anbindung der kaufmännischen Systeme
• Optimierung der Konfiguration der Java Persistence API • der Provider OpenJPA wurde durch EclipseLink ersetzt	• Performanceprobleme beseitigt
• neues Frontend mit Vaadin entwickelt • Backend-Services für Datenbankzugriff	• Kunden können ihre Verträge einsehen • Kunden können Schäden direkt melden • flexible Statistiken • weniger Datenerfassungsaufwand
• Angular 2 Frontend • Anbindung Java EE Backend • Restschnittstellen	• zeitnahe Statistiken • aktuelle Daten

GEDOPLAN
IT Training & IT Consulting

GEDOPLAN IT Training Seit 1998 schulen wir Java, viele Seminare auch in englischer Sprache. Unsere Schulungsleiter sind Java-Experten aus der Praxis, die ihr Wissen in Java-Projekten selbst unter Beweis gestellt haben. Unsere Java-Schulungen beinhalten neben den theoretisch notwendigen Grundlagen auch einen entsprechend hohen Praxisanteil. In unseren Seminaren gehen wir auf aktuelle Problemstellungen des Alltags ein. Damit Sie den bestmöglichen Erfolg erzielen, bieten wir zwei Formen an: In offenen Kursen vermitteln unsere Java-Experten ihr Know-how zu unterschiedlichen, definierten Schwerpunkten. Benötigen Sie Expertenwissen für sehr spezielle Java Fragen oder Projekte, führen wir individuelle Gruppen- und Firmenschulungen durch, die wir auf Ihre konkreten Bedürfnisse abstimmen.

GEDOPLAN IT Consulting steht seit vielen Jahren für hochwertiges Consulting in den Java-Technologien. Wir setzen auf offene Standards und Open Source Produkte. Die Java EE Plattform ist unsere Basis für die Entwicklung betrieblicher Anwendungen. Plattformen wie WildFly und Liferay führen schnell und sicher zum Ziel.
Ob Neuentwicklung, Migration nach Java EE oder Codereviews: Wir entwickeln IT-Systeme als Komplettpakete, unterstützen unsere Kunden aber auch gerne vor Ort.

GEDOPLAN
Unternehmensberatung und
EDV-Organisation GmbH
Stieghorster Straße 60
33605 Bielefeld
Fon: + 49 521 / 2 08 89 10
Fax: +49 521 / 2 08 89 45
info@gedoplan.de

Geschäftsstelle Berlin:
GEDOPLAN GmbH
UPPER WEST
Kantstraße 164
10623 Berlin

Postadresse Berlin:
GEDOPLAN GmbH
Kurfürstendamm 11
10719 Berlin

+49 30 / 755 49 188
it-training@gedoplan.de.

Herausgeber: GEDOPLAN Unternehmensberatung und EDV-Organisation GmbH | Stieghorster Straße 60 | 33605 Bielefeld
Verantwortlich: Dipl.-Inform. Dirk Weil | Redaktionsleitung: Ulrich Hake | Fon: 0521 2088910 | E-Mail: info@gedoplan.de
Produktion: networker Medienfabrik GmbH | Nachdruck, auch auszugsweise, nur nach Rücksprache mit der Redaktion.