01-2016 | Frühjahr | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

Java ist auf dem richtigen Kurs

Funktionale Programmierung
Java 8 und Haskell

Docker
Überblick und Infrastruktur-Deployment

Aktuell
Software-Lizenzrecht für Entwickler

RESTful-Microservices
Dropwizard und JAX-RS

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

4 191978 304903 01

iJUG Verbund

DevCamp 2016

23. - 24. Februar 2016 | Bonn

- ADF Fitness Center
- IoT
- Mobile

devcamp.doag.org

DOAG
Deutsche ORACLE-Anwendergruppe e.V.

Wolfgang Taschner
Chefredakteur Java aktuell

Herzliche Grüße von der JavaOne 2015

Ich nutze die Zeit in San Francisco, um kurz vor Drucklegung der Zeitschrift die neuesten Nachrichten zu übermitteln. Das Wichtigste zuerst: Java ist weiter auf einem guten Kurs. Die Stimmung ist gut; viele Keynote-Speaker greifen das Motto „20 Jahre Java" auf und blicken auf zahlreiche Meilensteine zurück. So auch Sun-Gründer Scott McNealy, der mit seinen „Top Ten Nightmares" sehr humorvoll an fast vergessene Zeiten erinnert.

Zuerst eines der größten Highlights, das im Rahmen der JavaOne eher nur am Rande zur Sprache kommt: Der WebLogic Server 12.2.1 steht als Java-EE-7-Implementierung ab sofort zum Download bereit. Auch die weiteren Neuankündigungen sind eher unspektakulär: Ohne große Überraschungen werden die Roadmaps für Java SE, Java ME und Java EE aufgezeigt und planmäßig die gesteckten Technologie-Ziele eingehalten.

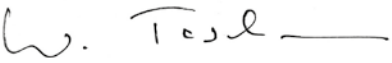
Nachdem die parallel stattfindende Oracle OpenWorld ganz im Zeichen der Cloud steht, gibt es auch für Java einen neuen Java SE Cloud Service. Unter dem Motto „Build, Zip, Deploy" ist dieser die einfachste Art, um Java-SE-basierte Anwendungen beispielsweise unter Tomcat in der Cloud zu nutzen. Bei der Implementierung wird zur Virtualisierung Docker benutzt und zum Tooling der Flight Recorder for Production Use angeboten. Der Service ist skalierbar und es gibt die Funktion „One Click Java Version Update". Man muss sich also nicht um die Pflege der Plattform und um die Security-Fixes kümmern.

Java SE verzeichnet ein großes Wachstum, was belegt, dass das Commitment von Oracle nicht nachlässt, sondern Früchte trägt. Mark Reinhold, Chief Architect Java Platform Group, stellt die wesentlichen Dinge im kommenden JDK 9 (Projekt „Jigsaw") vor – die Eliminierung des Classpath-Hell und die Modularisierung der Java-Plattform. Einen Ausblick auf das, was nach Java 9 kommen soll, gibt Brian Goetz, Chief Language Architect Java Platform Group: Das Valhalla Project bringt Specialized Generics sowie Value Types und im Panama Project wird es ein Foreign-Function-Interface, Data Layout Control sowie Array 2.0 geben.

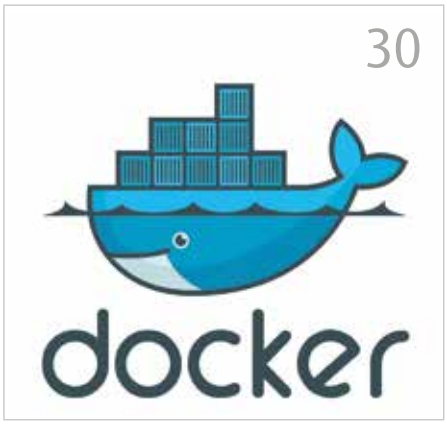
Michael Greene, Vice President and General Manager System Technologies and Optimization, Software and Services Group bei Intel, demonstriert in der Sponsor-Keynote sein Hauptanliegen, eine gute Java-Performance auf der gesamten Intel-Plattform. Neu ist die Unterstützung für Java ME auf Intel Edge Devices sowie der „Intel IoT Developer Kit" für Java. Intel bereitet sich bereits heute auf die Optimierungen kommender Java-Technologien auf der kompletten Intel-Plattform vor.

Der ruhige Verlauf der JavaOne war auch der wichtigste Eindruck für mich. Java ist weiterhin auf einem guten Weg und jeder in der Community kann sich damit entspannt auf seine Entwicklungsprojekte konzentrieren. In diesem Sinne wünsche ich allen viel Erfolg und eine gute Zeit.

Ihr



Die Brownies Collections Library stellt Alternativen bereit



Einsatzmöglichkeiten für Docker

Infrastruktur-Deployment mit Docker

RESTful-Microservices mit Dropwizard

Das Java-Tagebuch

Andreas Badelt, Mitglied der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java – in komprimierter Form und chronologisch geordnet. Der vorliegende Teil widmet sich den Ereignissen im dritten Quartal 2015.

3. Juli 2015

CDI 2.0 - erster Entwurf veröffentlicht

Vielleicht ist es nur ein kleiner Schritt für die Menschheit, aber es ist ein großer Schritt für Java: „Context and Dependency Injection 2.0" liegt jetzt als erster Entwurf zum Review vor. Die großen Themen sind „Modularität" und „Unterstützung für Java SE". Wie inzwischen üblich, arbeitet die Expertengruppe sehr öffentlich; alle relevanten Dokumente – Mails, Blog Posts, die Spezifikation und der Quellcode selbst – sind öffentlich verfügbar.

http://www.cdi-spec.org

8. Juli 2015

Java ME 8 Tutorial Series

Sehr detaillierte Anleitungen für Raspberry-Pi-Bastler für die unterschiedlichen Optionen zur Anbindung von Sensoren und Aktoren bietet ein Tutorial von Jose Cruz. Dies ist bereits der dritte Teil, diesmal über die UART-Schnittstelle. Der erste Teil ging bereits vor einem Jahr online, ich hatte es nur noch nicht erwähnt, was ich hiermit nachhole.

https://community.oracle.com/docs/DOC-917166

14. Juli 2015

Mini-Tutorial zu nicht-blockierenden Web-Anwendungen

Ich bin immer wieder erfreut über die schier unerschöpfliche Zahl von hochwertigen Java-Tutorials im Internet. Gerade die kleinen haben es mir angetan – sie geben einen schnellen Überblick über ein Thema, garniert mit Code-Beispielen. Eines davon ist „Develop Non-Blocking Web Applications in Java", das die Herangehensweise in verschiedenen Standards und Frameworks zeigt, angefangen von Servlet und JAX-RS über Spring MVC bis hin zu vert.x und Play.

https://community.oracle.com/docs/DOC-918126

Nachtrag (25.August): Auch Reza Rahman beschäftigt sich mit dem Thema „Asynchronität" (oder „Reactive") in Java EE 7 und speziell in JAX-RS 2. Dieser Blog-Eintrag ist ebenfalls zur Lektüre empfohlen.

https://blogs.oracle.com/theaquarium/entry/asynchronous_support_in_jax_rs

20. Juli 2015

Jolokia und DevOps

Wer beim letzten JavaLand die „Early Adopters"-Ecke besucht hat, konnte Roland Huss mit seiner JMX-http-Bridge „Jolokia" in Aktion sehen. Wer nicht, kann sich auf Markus Eiseles Blog einen Eindruck darüber verschaffen, wie gut sich das Framework für ein modernes, leichtgewichtiges Monitoring eignet – oder etwas cooler ausgedrückt: „Monitoring DevOps Style".

http://blog.eisele.net/2015/07/monitoring-devops- style-with-wildfly-9.html

20. Juli 2015

WebSphere Liberty ist EE 7 zertifiziert

IBM WebSphere Liberty 8.5 ist für Java EE 7 zertifiziert und schließt damit zu GlassFish 4, (JBoss) WildFly 8, Hitachi Cosminexus und TmaxSoft JEUS auf. Im Gegensatz zu den beiden Letztgenannten ist es sogar für das leichtgewichtigere Web Profile zertifiziert. Wann zieht Oracle mit WebLogic nach? Die ursprüngliche Aussage auf der JavaOne 2014 – Oracle peilt nach der Referenz-Implementierung von GlassFish an, auch den zweiten Oracle-Server wenige Monate nach einem neuen Java-EE-Release hochzuziehen – war überambitioniert. Aber zuletzt hieß es zumindest, dass das entsprechende Release 12.2.1 auf jeden Fall vor Ende 2015 herauskomme.

http://www-01.ibm.com/support/knowledgecenter/SSAW57_8.5.5/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/cwlp_javaee7.html

21. Juli 2015

Warten aufs Christkind – Java 9 Preview

Ben Evans, Co-Chef der London Java Community, beschreibt in einem Artikel auf InfoQ ausführlich zwei großartige Verbesserungen, die wir uns von Java 9 noch versprechen können – neben dem Dauerthema „Modularität": Unterstützung für HTTP/2 und die JShell REPL (Read-Eval-Print-Loop). Dummerweise dauert es noch bis Herbst 2016 – wenn der Zeitplan eingehalten wird. Aber wer schon mehr machen möchte, als nur durch das virtuelle Schlüsselloch zu schauen, für den gibt es ja Adopt OpenJDK sowie das „JDK 9 Outreach"- Programm, das möglichst viele Projekte zum frühzeitigen Testen mit dem JDK 9 animieren soll.

https://wiki.openjdk.java.net/display/Adoption/JDK+9+Outreach

28. Juli 2015

Nochmal: JShell und REPL in Java 9

Ein sieben-Minuten-Video erklärt, wie man die Evaluierungs-Schleife in Java 9 nutzen kann.

https://blogs.oracle.com/java/entry/jshell_and_relp_in_java

5. August 2015

MD5 wird per Default deaktiviert (für Zertifikate)

MD5-Signaturen sind bereits seit dem Jahr 2011 von der IETF als unsicher deklariert

und die Verwendung stärkerer Algorithmen (wie „SHA-*“) wird empfohlen. Mit kommenden Java-Updates bis Ende des Jahres (inklusive Java 9 Early Access) steigt der Druck nun: Per Default sollen MD5-Signaturen dann nicht mehr akzeptiert werden. System-Administratoren, die die existierenden MD5-Zertifikate nicht gegen sicherere austauschen wollen oder können, haben aber immer noch die Möglichkeit, die Property „jdk.certpath.disabledAlgorithms“ anzupassen. Wohlgemerkt: MD5 wird nicht komplett sterben (etwa für generelle Checksummen), es geht nur um Zertifikate.

https://blogs.oracle.com/java-platform-group/entry/strengthening_signatures

12. August 2015

sun.misc.Unsafe soll langsam verschwinden

Nach Mark Reinholds nicht repräsentativer Umfrage gehöre ich wohl zu den wenigen Java-Entwicklern, die nie auf die Idee gekommen sind, sun.misc.Unsafe zu benutzen. Es ist erstaunlich, in wie vielen Projekten die Klasse, die nur zur internen Verwendung im JDK gedacht war, verwendet wird: Akku, Spring, Hibernate, um einige der prominenteren zu nennen. Aber Oracle will die Klasse, die nicht nur dem Namen nach eine große Sicherheitslücke darstellt, langsam verschwinden lassen. Das wird allerdings länger dauern. Im JDK 9 soll sie Modulen nur noch zur Verfügung stehen, wenn die Abhängigkeit explizit deklariert wurde. Darüber hinaus wird sie nicht mehr weiterentwickelt; einzelne, häufig genutzte Funktionalitäten werden Stück für Stück in andere, sicherere APIs ausgelagert.

https://blogs.oracle.com/java/entry/about_sun_misc_unsafe

18. August 2015

Neuer Redirect Scope in Java EE 8

Mit dem MVC-API wird demnächst ein neuer Scope Einzug in Java EE halten: „@RedirectScoped“. Er wird benutzt, um den Kontext zu bewahren zwischen einem ursprünglichen Request, der mit einem Redirect beantwortet wird, und dem dadurch ausgelösten Folge-Request (falls der Redirect denn an den gleichen Server geht). Durch die Integration mit CDI wird kein großer Aufwand betrieben, es ist nur ein neuer CDI Custom Scope.

https://java.net/projects/ozark/sources/sources/show/test/redirectScope

24. August 2015

JSON-B 1.0 – erster Entwurf liegt vor

Der erste Entwurf für die neue JSON-B-Spezifikation (Standard-Dokument und Quellcode / JavaDoc der Referenz-Implementierung) ist jetzt für ein öffentliches Review verfügbar. Die Dokumentation umfasst nur 35 Seiten – genau genommen weniger als 20 relevante Seiten – und nach Aussage der Expertengruppe ist der Entwurf weit fortgeschritten, sodass Neugierige keinen Grund zum Zögern haben sollten.

https://jcp.org/aboutJava/communityprocess/edr/jsr367/index.html

7. September 2015

Oracle trennt sich von etlichen Java-Evangelisten

Die Evangelisten haben jetzt viel Zeit, um ihre Gospels zu schreiben. Wie Heise berichtet, hat sich Oracle auf einen Schlag von mehreren seiner Technik-Experten getrennt, die Java in die Welt tragen sollen. Die Gerüchteküche kochte schnell hoch und zeitweise hieß es sogar, das Team werde komplett aufgelöst. Das konnte zumindest bislang nicht bestätigt werden. Es sieht eher danach aus, als sei es Teil einer generellen Entlassungswelle. Allerdings hat das Team nun seinen bisherigen Leiter Simon Ritter sowie fünf weitere Mitglieder verloren und ist somit auf nur noch neun Personen zusammengeschrumpft. Wenn man sich die Themen anschaut, die von ihnen bedient wurden, könnte man auf die Idee kommen, dass Java EE und GlassFish keine große Rolle mehr spielen. Aber zumindest Ersteres klingt unlogisch und ist wohl Kaffeesatzleserei.

http://www.heise.de/developer/meldung/Oracle-trennt-sich-von-etlichen-Java-Evangelisten-2806645.html

8. September 2015

TIOBE-Index: Java nach Algorithmus-Änderung wieder klar vorn

Neues vom TIOBE-Index, der die Popularität von Programmiersprachen anhand von Treffern in 25 verschiedenen Suchmaschinen messen soll. Um der fortwährenden Kritik am Messverfahren zu begegnen, haben die Herausgeber den Algorithmus verändert; Ausreißer in einzelnen Suchmaschinen werden nun individuell behandelt und entfernt. Vielleicht liegt es daran, dass Java im Vergleich zum Vorjahresmonat einen Satz um fünf Prozentpunkte auf knapp 19,6 Prozent macht und C, das leicht auf 15,6 Prozent verloren hat, wieder von Platz 1 verdrängt. C++ und C# sind mit knapp 7 und knapp 5 Prozent bereits unter „ferner liefen“, aber immer noch weit vor Python, PHP und insbesondere vor JavaScript mit gerade mal 2,3 Prozent (was ich gar nicht so recht glauben kann). Objective-C ist in die Bedeutungslosigkeit abgestürzt (1,4 nach gut 10 Prozent), direkt dahinter liegt nun der Nachfolger Swift, der deutlich zulegen konnte, aber trotzdem nur bei knapp 1,3 Prozent liegt. Egal, ob man den Zahlenspielereien glaubt oder nicht, eine gewisse Selbstbestätigung ist es ja schon für alle Java-Programmiererinnen und -Programmierer, dass sie auf das richtige Pferd gesetzt haben. Was mich aber auch etwas stutzig macht, ist, dass die alternativen JVM-Sprachen wie Scala, Groovy oder Clojure alle mit deutlich unter 1 Prozent weit hinten im Feld liegen. Jetzt möchte ich bitte einen Aufschrei und Meldungen von ganz vielen „Adoption Stories“ hören …

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

8. September 2015

Status von Jigsaw

Mark Reinhold, Chief Architect der Java-Plattform, hat in einem langen Artikel den Stand der Diskussion zum neuen Modul-System Jigsaw beschrieben, das mit Java 9 kommen soll. Soweit scheint sich alles recht intuitiv in die gewohnte Java-Welt einzufügen. Modul-Informationen mit Exporten und Abhängigkeiten werden als „module-info.java“ in einem Package oder einem Jar Root abgelegt und sind sowohl zur Compile-Zeit als auch zur Laufzeit – auch über Reflection – verfügbar. Über die simple Basis hinaus gibt es spezielle Features, etwa für Container. So weit, so gut – der eigentliche Aufwand wird im Zerlegen der Java-SE-Plattform selbst liegen, inklusive Abwärtskompatibilität.

http://openjdk.java.net/projects/jigsaw/spec/sotms
http://marxsoftware.blogspot.de/2015/09/jdk9-state-of-module-system.html

6. Oktober 2015

DZone: JSF und Spring MVC Kopf an Kopf vorn

DZone hat eine Umfrage zur Popularität von Java Web-Frameworks durchgeführt – Kopf

an Kopf vorn lagen JSF und Spring MVC mit jeweils gut 34 Prozent, weit dahinter schon Apache Wicket mit 16,3 Prozent. Das bestätigt das Ergebnis des „Java EE 8 Community Survey", der Anlass zur Aufnahme von MVC in den Standard war – wobei natürlich die Entwicklung mit einem schon lange existierenden De-facto-Standard Spring MVC und einem erst als Entwurf vorliegenden richtigen Standard MVC 1.0 spannend bleibt.

https://dzone.com/articles/poll-what-java-jvm-frameworks-do-you-use

9. Oktober 2018

Verfeinerter Entwurf von MVC 1.0 liegt vor

Gerade haben wir noch davon gesprochen, jetzt liegt der neue Standard MVC 1.0 als „Early Draft Review 2" vor und kann online eingesehen werden (unbedingt auch dem Link zur „Public Project Page" folgen). Wie immer: Die Expertengruppe freut sich über viel und frühes Feedback.

https://www.jcp.org/en/jsr/detail?id=371

Nachtrag

Lücke in Apache-Commons-Bibliothek gefährdet Java-Anwendungen

Wie ZDNet meldet, haben Sicherheitsforscher von Foxglove Security auf eine Lücke in der häufig genutzten Apache-Commons-Bibliothek hingewiesen, die sich zum Einschleusen und Ausführen von Schadcode ausnutzen lässt. Die Schwachstelle beruht auf einer unsicheren Methode, mit der Java Objekte deserialisiert. Sie gefährdet Java-basierte Anwendungen wie JBoss, Jenkins, OpenNMS, WebSphere oder WebLogic.

Wie Foxglove erklärt, ist die Lücke schon seit mehr als neun Monaten bekannt. Weil Java-Anwendungen abhängige Bibliotheken mit jeder Applikation bündeln, statt Shared Libraries zu nutzen, dürfte sich die Schwachstelle schon seit einiger Zeit ausnutzen lassen.

Der Sicherheitsforscher Steve Breen hat einen Exploit für die Schwachstelle entwickelt. Er basiert ihm zufolge auf einer ähnlichen Lücke in Apache Commons im Zusammenhang mit der Deserialisierung von Objekten, die schon im Januar öffentlich gemacht wurde. Basierend auf der Art, wie Java benutzerdefinierten Code beim Deserialisieren von Objekten ausführt, konnte Breen nach eigenen Angaben angepasste Payloads erstellen, um sich Shell-Zugriff zu verschaffen. Das funktioniere auf Maschinen, auf denen JBoss, Jenkins, OpenNMS, WebLogic oder WebSphere laufen oder die Java Remote Method Invocation verwenden.

Apache und Jenkins haben inzwischen reagiert, und Patches angekündigt, um die Schwachstelle zu beseitigen. Jenkins veröffentlichte einen Workaround, der das für den Angriff ausgenutzte Jenkins CLI System deaktiviert. Ein Patch soll bald erscheinen.

Vorwürfe äußerte Jeff Gehlbach von OpenNMS, der via Twitter erklärte, Breen hätte die betroffenen Projekte zunächst über die Zero-Day-Lücke informieren müssen, bevor er sie öffentlich machte. Justin Kennedy von Foxglove antwortete darauf, dass man die Schwachstelle nicht als Zero-Day-Lücke betrachtet habe. Apache Commons selbst hat seinen 3.2.x-Zweig um einen vorgeschlagenen Patch erweitert, sodass sich die Serialisierung der anfälligen InvokerTransformer-Klasse per Flag standardmäßig abschalten lässt.

http://www.zdnet.de/88251371/luecke-in-apache-commons-bibliothek-gefaehrdet-java-anwendungen

Andreas Badelt
Leiter der DOAG SIG Java

Andreas Badelt ist Senior Technology Architect bei Infosys Limited. Daneben organisiert er seit 2001 ehrenamtlich die Special Interest Group (SIG) Development sowie die SIG Java der DOAG Deutsche ORACLE-Anwendergruppe e.V. Daneben war er von 2001 bis 2015 ehrenamtlich in der Development Community und ist seit 2015 in der neugegründeten Java Community der DOAG Deutsche ORACLE-Anwendergruppe e.V. aktiv.

Software-Lizenzrecht für Entwickler

Antje Kilián, Heinrich & Reuter Solutions GmbH

Die einen betrachten Lizenzbestimmungen als Ausdruck von Community-Zusammenhalt, für die anderen sind sie ein unbekanntes Übel, dem leider viel zu häufig mit Ignoranz begegnet wird. Diese Fehleinschätzung kommt derzeit einige große Konzerne teuer zu stehen. Dieser Artikel gibt Antworten auf einige grundlegende Fragen.

Wenn ich im Laden ein Buch kaufe, dann darf ich es sofort lesen. Ich kann es verleihen, ich kann Sachen darin markieren oder die Seiten herausreißen. Bei Software ist das anders. Nur weil ich eine CD in der Hand halte, auf der sich die Kopie eines Programms befindet, darf ich dieses noch längst nicht benutzen und schon gar nicht verleihen.

Ein Lizenzvertrag bestimmt im Grunde die Regeln, unter denen eine Software verwendet werden kann. Darf man sie verändern? Darf ich sie sowohl privat als auch kommerziell einsetzen? All das ist Bestandteil eines Lizenzvertrags. Diese Verträge können individuell geschlossen werden, zum Beispiel bei einer speziell entwickelten Software für den Geschäftsgebrauch, oder sie können allgemein für alle Nutzer formuliert werden. In der letzten und häufigeren Variante sind die Lizenzen im Grunde „AGB", denn das ist der Begriff für vorformulierte Verträge, die man in unveränderter Form für mehrere Rechtsgeschäfte mit verschiedenen Personen anwendet. Als solche unterliegen sie auch den „Rechtsvorschriften für AGB", die zum Schutz der Verbraucher gedacht sind. Damit ist zum Beispiel ausgeschlossen, dass eine Lizenz eine Regelung enthält, die für den Nutzer nicht vorhersehbar war.

Lade ich nun eine Software auf eine Plattform hoch, biete ich jedem, der Zugang dazu hat, an, einen Vertrag mit mir abzuschließen. Das ist ähnlich wie bei einem Getränkeautomaten. Dieser ist für jeden, der vorbeikommt, ein ständiges Angebot des Eigentümers. Werfe ich eine Münze in den Automaten, stimme ich den Bedingungen des Eigentümers durch meine Handlungen zu, der Jurist nennt das „konkludent". Gleiches gilt für Software. Wenn ich diese benutze, stimme ich damit den Vertragsbedingungen zu, ob ich sie gelesen habe oder nicht. Lade ich eine Software mit Copyleft-Lizenz herunter, begründet das noch keinen Vertragsschluss. Erst wenn ich sie benutze, verändere oder an andere weitergebe, binde ich mich an die Bedingungen der Lizenz.

Gerade fiel mit „Copyleft" ein wichtiger Begriff, mit dem sich jeder Entwickler auseinandersetzen sollte. In Deutschland herrscht das nicht übertragbare Urheberrecht. Erschafft der Entwickler eines Unternehmens eine Software, so ist immer er der Urheber und nie das Unternehmen. Als Land der Dichter und Denker haben wir uns ein Rechtssystem geschaffen, dass vor allem die geistige Beziehung eines Künstlers, Autors oder eben Entwicklers zu seinem Werk schützt. Es entsteht auch ohne Anmeldung und hilft dem Erschaffer, die Unversehrtheit seines Werkes zu schützen. Auf der anderen Seite gibt es das amerikanische „Copyright". Im Gegensatz zu unserem Urheberrecht, das ausschließlich natürliche Personen innehaben können, kann auch eine juristische Person, wie ein Unternehmen, Copyright auf etwas haben.

„Copyleft" ist kein echter juristischer Begriff sondern lediglich ein Wortspiel, das auf die Unterschiedlichkeit von Copyleft und Copyright hinweisen soll. Während Letzteres das Ziel hat, die Rechte anderer am Werk einzuschränken, hat Ersteres das Ziel, die Einschränkung der Rechte für die Allgemeinheit aufzuheben und das Werk allen beliebig zugänglich zu machen. Copyleft-Lizenzen wie die GNU General Public License (GPL) gewähren also allen Entwicklern das Recht, diese Software zu verwenden und zu verbreiten. Sie sind aber auch viral. Das bedeutet, sie „infizieren" den auf ihrer Basis entstandenen Code und verlangen auch diesem eine Copyleft-Lizenz ab, genau wie allen Programmen, die wiederum auf diesem Code basieren.

Lizenzformen im internationalen Vergleich

Im Wesentlichen unterscheiden wir zwischen fünf Grund-Lizenzformen. Die Bekannteste ist die „proprietäre Software", was übersetzt so viel bedeutet wie „im Eigentum befindlich". Das ist zum Beispiel die im Auftrag eines Kunden entwickelte Software. Diese ist in der Regel „Closed Source" und stark einschränkend. Sie erlaubt den Nutzern normalerweise nur, damit zu arbeiten. Der Code ist normalerwei-

se unveränderlich und auch die Weitergabe ist nicht ohne Weiteres möglich.

Damit in Verbindung stehen Freeware und Shareware. Auch hier darf die Software nicht verändert, aber im Vergleich zur proprietären Software an andere Nutzer weitergegeben werden. Bei der Freeware entstehen dafür keine Kosten, bei der Shareware in der Regel erst nach einer gewissen Testphase.

Die Public Domain ist im amerikanischen Rechtssystem angesiedelt. Hier gibt der Autor keine Lizenz an und drückt damit einen völligen Verzicht auf die – ihm normalerweise zustehenden – Rechte aus. Hier gibt es keine Regeln und jeder kann mit dem Code machen, was er möchte. Leider ist Public Domain in Deutschland kein gültiges Rechtskonstrukt. Hierzulande gilt: Werden Abweichungen vom Urheberrecht nicht explizit benannt, dann gibt es auch keine. Dieser Widerspruch kann zu massiven Problemen führen, da auf den gängigen Code-Sharing-Plattformen häufig Code ohne Lizenz anzutreffen ist. Verwendet der arglose deutsche Entwickler diese, verstößt er im Grunde gegen das Urheberrecht. Um diesem Problem vorzubeugen, arbeiten beispielsweise die Betreiber der Plattform „GitHub" an Möglichkeiten, die Vergabe von Lizenzen in Zukunft zu vereinfachen. Die letzte Gruppe ist gleichzeitig auch die vielseitigste. Es sind die Open-Source-Lizenzen und die freie Software.

Wie offen ist Open Source?

Die Begriffe „freie Software" und „Open Source" werden fälschlicherweise oft als Synonyme verstanden. Vor allem der ideelle Gedanke ist es, der beide voneinander trennt. Zuerst gab es die „Free Software"-Initiative. Diese gründete sich mit einem wichtigen sozialen, politischen und ethischen Anliegen. Sie wollte dafür sorgen, dass durch die Zusammenarbeit vieler Entwickler an einer Software und damit durch die Bündelung ihrer Kompetenzen das bestmögliche Ergebnis entsteht. Es sollte dagegen angegangen werden, dass große Unternehmen aus rein finanziellen Interessen ihren Code geheim halten und somit verhindern, dass andere daraus lernen oder etwas noch Besseres herstellen.

Leider bedeutet das Wort „free" im Englischen nicht nur „frei" sondern auch „kostenlos". Daraus bildete sich ein ernstes Image-Problem für die „free Software", denn dieser Zusatz ließ zunächst einige vor dem Gedanken zurückschrecken und hemmte die Verbreitung.

Aus der „free Software" heraus bildete sich deshalb schließlich die „Open Source"-Initiative. Bei dieser Bewegung war weniger die ideelle Überzeugung entscheidend, sondern der Fokus lag überwiegend auf den technischen Vorteilen eines offenen Codes. Die Definition schaffte mehr Möglichkeiten, Open-Source-Code auch für proprietäre Software zu nutzen und damit den Code für die Allgemeinheit zu schließen.

Daraus entstand die Unterscheidung, dass zwar alle „freie Software" auch „Open Source" ist, aber nicht alles, was „Open Source" ist, wird auch von der Freien-Software-Initiative anerkannt. Diese akzeptiert keine Lizenzen, die neben der viralen Verbreitung noch zusätzliche Einschränkungen abverlangen, etwa die Nennung der Autoren.

Welche Lizenzen es gibt und was es zu beachten gilt

Beginnen wir gleich mit der gleichermaßen bekannten und gefürchteten GPL-Lizenz, der „General Public License". Sie ist die einzige wirklich verbreitete Copyleft-Lizenz, nach der Entwickler Ausschau halten sollten. Laut GitHub ist die GPLv2 mit knapp 13 Prozent die dritthäufigste Lizenz auf ihrer Plattform (siehe *„https://github.com/blog/1964-open-source-license-usage-on-github-com"*). Kurz gesagt gibt die GPL jedem Entwickler das Recht, den unter ihr lizenzierten Code zu verwenden, zu bearbeiten und zu verbreiten. Die Software muss nicht unbedingt kostenlos sein, aber der Code bleibt unter allen Umständen für jeden Interessierten offen zugänglich, denn jedes Werk, das auf ihm basiert, muss ebenfalls unter der GPL lizenziert werden. Von dieser Regelung kann nur dann abgesehen werden, wenn der verwendete Teil für das Gesamtwerk unerheblich ist. Wann dies allerdings der Fall ist, wurde leider nicht genau definiert und ist daher im Zweifelsfall eine Auslegungssache des Gerichts.

Die Lizenz selbst gibt an, es müsse sich um „vernünftig betrachtet selbstständige und in sich selbst getrennte Werke" handeln. Dies wäre zum Beispiel gegeben, wenn lediglich ein Tool verwendet wurde, das unter GPL steht, oder das Programm auch dann in gleichem Maße funktionieren würde, wenn man den GPL-Code komplett weglässt. Im Zweifelsfall gilt allerdings die Faustformel: Wenn man sich nicht ganz sicher ist, dass es unabhängige Werke sind, muss der entstehende Code auch unter GPL-Lizenz stehen.

Die Einhaltung dieser Lizenz sollte sehr ernst genommen werden, denn inzwischen werden ganze Hackathons (siehe *„https://fsfe.org/news/2013/news-20130626-01.de.html"*) organisiert, um Verstöße gegen diese Lizenz aufzudecken. Besteht ein berechtigter Verdacht auf einen Verstoß, kann die Herausgabe des Codes jeder kommerziellen Software von einem Gericht zur Überprüfung gefordert werden.

Eine weitere Besonderheit besteht darin, dass GPL-Code nicht mit jedem anderen Code kompatibel ist. Wie bereits erwähnt, dürfen einer GPL-Lizenz keine zusätzlichen Einschränkungen hinzugefügt werden, weder auf das Original noch auf abgeleitete Werke. Nun kommt noch hinzu, dass es verschiedene Versionen der GPL und anderer gängiger Lizenztypen gibt und nicht alle Versionen miteinander kombinierbar sind. Um etwas Klarheit zu schaffen, bietet die Freie-Software-Initiative eine Liste der kompatiblen Software an (siehe *„http://www.gnu.org/licenses/license-list.html"*). Nicht kompatibler Code darf nicht im gleichen Projekt mit GPL-Code verwendet und unter gemeinsame Lizenz gestellt werden.

Wer die GPL kennt, hat eventuell auch schon von der „LGPL" und der „AGPL" gehört. Die Lesser General Public License (LGPL) ist eine abgeschwächte Form der GPL und hauptsächlich für Bibliotheken gedacht. Die Affero General Public License (AGPL) ist hingegen noch restriktiver als die GPL. Sie wurde geschaffen, um die Lücke zu schließen, nach der die derzeitige GPL „Software as a Service" noch nicht als Weiterverbreitung klassifiziert. Kommt die AGPL zum Einsatz, müssen auch die Hoster von Open-Source-Software den Code der angebotenen Software offenlegen.

Neben den Copyleft-Lizenzen existieren auch einige sehr bekannte Copyright-Lizenzen. Die bekanntesten sind die MIT-Lizenz, die BSD-Lizenz, die MS-PL- und die Apache-Lizenz.

Man hört immer wieder, dass die Berkeley-Software-Distribution-Lizenz (BSD) die am wenigsten restriktive Lizenz sei. Dies ist nur bedingt richtig. Zwar trifft das auf die aktuelle Version der Lizenz zu, dennoch gab es eine erste Version, die die Verpflichtung enthielt, dass jedes entstehende Werk den Zusatz „Dieses Produkt enthält von der Universität von Kalifornien, Berkeley, und Beitragleistenden entwickelte Software" enthalten musste. Nicht nur, dass dieser Satz die Lizenz inkompatibel mit der GPL machte, er inspirierte einige Entwickler auch dazu, ihren eigenen Satz hinzuzufügen, was wiederum darin resultierte, dass eine im Jahr 1970 erschienene Version von NETBSD ganze 75 Werbesätze enthielt, die Entwickler von darauf basierenden Werken zu übernehmen hatten.

Inzwischen wurde diese Klausel entfernt, aber dennoch wird zuweilen Software der alten Lizenzierung angeboten, weshalb Entwickler vor der Verwendung prüfen sollten, mit welcher Variante sie es zu tun haben.

Die Apache-v.2-Lizenz trifft man zwar seltener auf den gängigen Plattformen, aber alle Apache-Software-Projekte stehen unter dieser Lizenz inklusive des Apache HTTP Server. Diese Lizenz verdient besondere Erwähnung, da sie die erste mit der sogenannten „Patenrechtsklausel" war. Diese besagt, sollte jemand eine aus diesem Code resultierende Software patentieren lassen, muss er allen anderen Entwicklern erlauben, die Software weiterhin zu verwenden ohne Angst vor einer Patentrechtsklage. Auch diese Klausel war nicht kompatibel mit der GPL v.1 und v.2. Erst die GPL v.3 erlaubt die gleichzeitige Verwendung beider Lizenzen.

Die Microsoft Public License (MS-PL) gewinnt derzeit zunehmend an Beliebtheit. Die Besonderheit dieser Lizenz ist die Aufforderung, die Trademarks der Mitschöpfer zu benennen und nicht als eigene auszugeben. Zudem enthält sie ebenfalls die Patentrechtsklausel und ist damit inkompatibel mit den GPL-Lizenzen v.1 und v.2. Die Lizenz enthält außerdem ein schwaches Copyleft. Der Code entstehender Werke kann nur dann frei gewählt werden, wenn das Gesamtergebnis in kompilierter Form weitergegeben wird. Bei einer direkten Weitergabe ist zwingend die MS-PL-Lizenz beizubehalten.

Abschließend ist noch die MIT-Lizenz zu nennen, die nicht ohne Grund die derzeit am weitesten verbreitete ist. Diese Lizenz ist wirklich frei und verlangt den Entwicklern lediglich ab, den Lizenztyp zu benennen und den Text weiterzugeben. Sie ist daher auch kompatibel mit jeglicher anderen Lizenz.

Fazit und Best Practice

Ziel dieses Artikels ist es nicht, einen universalen Leitfaden für den Umgang mit Lizenzen zu geben, denn das ist schlicht unmöglich. Es ist dringend anzuraten, bei größeren kommerziellen Projekten die Überprüfung von einem Experten durchführen zu lassen. In der Regel enthalten schließlich fast alle Verträge zur Software-Entwicklung die Klausel, dass der Entwickler für die Einhaltung der Lizenzrechte garantiert, und nicht selten kann eine Zuwiderhandlung zu Schadensersatzforderungen im fünfstelligen Bereich und zum Rückruf aller verbreiteten Kopien führen.

Für die eigenen und kleineren Projekte kann aber Entwarnung gegeben werden, denn mit einigen einfachen Spielregeln hat man hier nur wenig zu befürchten. Zunächst sollte jede Software ein Dokument enthalten, dass sowohl die eigene Lizenz angibt als auch eine Liste aller verwendeter Fremdsoftware mit deren Lizenz und, falls bekannt, dem Autor. Das wird zwar nicht von jeder Lizenz verlangt, schadet im Zweifel allerdings nicht. Zu jeder Lizenz gehört der englischsprachige Originaltext. Es wurde diskutiert, ob eine Verlinkung genüge, das Landgericht München (2007 AZ 7 O 5245/07) hat dies jedoch abgelehnt und fordert stets die Nennung des gesamten Textes.Möchte man GPL-lizenzierten Code verwenden, so heißt es genau zu überprüfen, ob alle anderen Teile kompatibel sind, und natürlich auch die Lizenz des entstehenden Werkes entsprechend zu wählen.

Für die eigene Software gibt es meiner Meinung nach eine einfache Faustformel: Möchte man die Freiheit seines Codes aus ethischer Überzeugung heraus bewahren, dann wählt man „AGPL" oder „GPL". Möchte man einfach nur seinen Code mit anderen teilen und den Mitentwicklern das Leben ein bisschen leichter machen, auch wenn am Ende ein anderer Geld damit verdient, dann wähle man „MIT". Auf keinen Fall jedoch sollte man den Code ohne jegliche Lizenz veröffentlichen, denn das nimmt allen Beteiligten den Spaß.

Abschließend lässt sich sagen, dass jeder Entwickler selbst entscheiden muss, wie ernst er es mit den Lizenzen halten möchte. Das IT-Recht hat Mühe, mit der schnellen Entwicklung des Internets Schritt zu halten und nur wenige haben wirklich mit ernsthaften Konsequenzen zu rechnen. Es gibt derzeit allerdings eine stetig ansteigende Bewegung in diesem Thema, die sich zwar zunächst darauf beschränkt, Exempel an großen Konzernen zu statuieren (etwa Skype Technologies SA vs. Gpl-violations.org LG München 2007), aber bald sicher auch kommerzielle Projekte kleiner Unternehmen erreicht und auch hier empfindliche Strafen einfordert für ein Vergehen, das mit überschaubarem Aufwand hätte vermieden werden können.

Antje Kilián
ak@xamllab.net

Antje Kilián hat einen Masterabschluss in IT-Recht an der Leibnitz Universität Hannover sowie der Strath clyde University Glasgow erworben und arbeitet seitdem bei der Heinrich & Reuter Solutions GmbH in Dresden als Justiziarin. Mit ihrem juristischen Fachwissen unterstützt sie das Team bei der Umsetzung von Software-Projekten für Kunden verschiedenster Größenordnung. Seit einigen Jahren hält sie Fachvorträge zum Thema „IT-Recht für Entwickler" und ist Sprecherin auf verschiedenen Veranstaltungen der Branche.

Funktionale Programmierung mit Java 8 und Haskell

Nicole Rauch, Software-Entwicklung und Entwicklungs-Coaching

Funktionale Programmierung hat zu guter Letzt auch in Java Einzug gehalten. Aber ist das wirklich funktionale Programmierung? Wie sieht es aus, wenn man Java 8 mit einer althergebrachten funktionalen Programmiersprache wie Haskell vergleicht? Was kann der aus der OO-Entwicklung kommende Java-Programmierer von funktionalen Sprachen lernen und in seinen Alltag integrieren? Der Artikel geht diesen Fragen nach. Nebenher werden grundlegende Aspekte und Besonderheiten der funktionalen Programmierung vorgestellt.

Funktionale Programmierung ist derzeit in aller Munde. Doch weit weniger bekannt ist, dass viele Firmen funktionale Programmierung tatsächlich einsetzen, und das oft schon seit geraumer Zeit. Wirft man zum Beispiel einen Blick auf die Liste von Firmen, die Haskell kommerziell einsetzen *(siehe „https://wiki.haskell.org/Haskell_in_industry")*, entdeckt man durchaus den einen oder anderen bekannten Namen. Was bringt Firmen dazu, auf eine so ungewöhnliche Programmiersprache zu setzen? Um dieser Frage auf den Grund zu gehen, sehen wir uns einmal genauer an, was „funktionale Programmierung" eigentlich bedeutet.

Die Kernaspekte funktionaler Programmierung

Ein wichtiger Aspekt ist „Immutability" – einer Variablen darf also nur einmal ein Wert zugewiesen werden. Viele funktionale Sprachen setzen diesen Aspekt konsequent um, andere machen es zumindest schwierig, einmal gesetzte Werte später zu verändern. In Java gibt es keine direkte Unterstützung für „Immutability", es ist jedoch einfach, zumindest den einen Teil der Klassen eines Systems unveränderlich zu gestalten: Man kann gekapselte Attribute privat machen, sie nur im Konstruktor zuweisen und sämtliche Methoden so implementieren, dass sie nur lesend auf diese Attribute zugreifen. Vorteile bietet diese Vorgehensweise insbesondere bei der Parallelisierung: Concurrency-Probleme durch das quasigleichzeitige Modifizieren derselben Variablen aus verschiedenen Threads heraus sind damit unmöglich. *Listing 1* zeigt ein Beispiel.

```
class Point {
  private int x, y;
  public Point (int x, int y) {
    this.x = x;
    this.y = y;
  }
  // im restlichen Code werden x
und y nur noch gelesen
}
```

Listing 1

Seiteneffekt-Freiheit

Als „Seiteneffekt" bezeichnet man alles, was den Ablauf eines Programms oder die Außenwelt verändert, ohne von einer Funktion zurückgegeben worden zu sein. Wichtige Beispiele dafür sind Ein-/Ausgabe, Exceptions, Logging, Abhängigkeit von (externen) Konfigurationen, Veränderungen des Zustands sowie Nicht-Determinismus, etwa durch die Verwendung eines Zufallszahlen-Generators.

In funktionalen Sprachen wird die Seiteneffekt-freie Programmlogik üblicherweise von Seiteneffekt-behaftetem Code getrennt. In manchen Sprachen ist es sogar aus der Typ-Signatur einer Funktion ersichtlich, ob diese einen Seiteneffekt produziert, oder anders ausgedrückt: Normale Funktionen dürfen in diesen Sprachen gar keine Seiteneffekte haben.

Von derartigen Einschränkungen ist eine Sprache wie Java weit entfernt.

Trotzdem kann es sinnvoll sein, sich gewisse Codierungsregeln aufzuerlegen und die Bereiche, die Seiteneffekt-behaftet sind, von der Seiteneffekt-freien Kernlogik zu trennen. Dies kann beispielsweise innerhalb einer Klasse dadurch erfolgen, dass man Seiteneffekt-freie (und damit sehr einfach testbare) Methoden implementiert, die wiederum von Seiteneffekt-behafteten Methoden derselben Klasse aufgerufen werden.

Es ist sogar unschädlich, diese Seiteneffekt-freien Methoden öffentlich zu machen, da sie ja keine Effekte auf die Klasse haben können, in der sie implementiert sind. In manchen Projekten wird sogar dazu übergegangen, diese Methoden statisch zu machen, um die Seiteneffekt-Freiheit noch deutlicher zu machen. *Listing 2* zeigt ein Beispiel für eine derartige Trennung.

Funktionen sind „first order citizens“

Ein weiterer wichtiger Aspekt ist die Tatsache, dass Funktionen „first order citizens“ sind, mit Funktionen kann man also dasselbe machen wie mit Zahlen oder Strings. Konkret heißt das:

- Eine Funktion kann einer Variablen zugewiesen werden
- Eine Funktion kann als Argument an eine Funktion übergeben werden
- Eine Funktion kann von einer Funktion zurückgegeben werden

In Java 8 kann man statische Methoden und Objektmethoden sowie Lambda-Ausdrücke einer Variablen zuweisen. Java bietet sogenannte „Funktions-Interfaces“ als Typen an (ein Beispiel dafür ist „IntBinaryOperator“), man kann jedoch auch ein eigenes Funktions-Interface definieren – die einzige Voraussetzung ist, dass es nur eine Methoden-Deklaration enthält, die natürlich vom Typ her zur zugewiesenen Funktion passen muss *(siehe Listing 3)*. In Haskell gibt es keine Objekte, daher fällt das Codebeispiel entsprechend einfacher aus *(siehe Listing 4)*.

Currying

Dem aufmerksamen Leser wird auffallen, dass das Aufrufen einer Funktion mit einem Parameter, die eine Funktion mit einem Parameter zurückgibt, in Java deutlich anders aussieht als das Aufru-

```
class TrennungVonSeiteneffekten
{
  public void mitSeiteneffekt(){
    String initialerWert = Sys-
tem.console().readLine();
    String ergebnis = ohneSeiten
effekt(initialerWert);
    System.out.println(„Das Re-
sultat: “ + ergebnis);
  }
  public static String
ohneSeiteneffekt(String ini-
tialerWert){
    return /* Ergebnis der Funk-
tion */ ;
  }
}
```

Listing 2

```
// Zuweisung einer statischen Methode:
class Example1 {
  static int staticTimes (int x, int y) { return x * y; }
}
IntBinaryOperator times = Example1::staticTimes;
times.applyAsInt(3, 5);      // liefert 15

// Zuweisung einer Objektmethode:
class Example2 {
  int times (int x, int y) { return x * y; }
}
Example2 example2 = new Example2();
IntBinaryOperator times = example2::times;
times.applyAsInt(3, 5);      // liefert 15

// Zuweisung einer Lambda-Expression:
IntBinaryOperator times = (x, y) -> x * y;
times.applyAsInt(3, 5);      // liefert 15
// Dasselbe mit eigenem Funktionsinterface:
interface TimesFunction { int eval(int x, int y); }
TimesFunction times = (x, y) -> x * y;
times.eval(3, 5);    // liefert 15

// Funktion an Funktion übergeben:
class Example3 {
  static int apply(IntUnaryOperator func, int arg) {
    return func.applyAsInt(arg); }
}
Example3.apply(x -> 3 * x, 5);      // liefert 15

// Funktion aus Funktion zurückgeben:
interface FunctionFunction { IntUnaryOperator eval(int x); }
FunctionFunction times = x -> { return y -> x * y; };
times.eval(3).applyAsInt(5); // liefert 15
```

Listing 3

```
-- Zuweisung einer Funktion:
times x y = x * y
timesVar = times
timesVar 3 5 -- liefert 15
-- Zuweisung eines Lambda-Ausdrucks:
times = (\ x y -> x * y)
times 3 5 -- liefert 15

-- Funktion an Funktion übergeben:
apply func arg = func arg
apply (\ x -> 3 * x) 5       -- liefert 15

-- Funktion aus Funktion zurückgeben:
times x = (\y -> x * y)
times 3 5 -- liefert 15
```

Listing 4

fen einer Funktion mit zwei Parametern; in Haskell ist der Aufruf dagegen genau gleich. Dies liegt daran, dass viele funktionale Sprachen nur Funktionen mit einem Parameter kennen.

Definiert man eine Funktion, die mehrere Parameter zu haben scheint, so wird dies intern in eine Funktion umgewandelt, die einen Parameter nimmt und eine Funktion zurückliefert, die wiederum einen Parameter nimmt etc. so lange, bis alle Parameter versorgt sind. Dies nennt man „Currying".

Was zunächst nur wie ein technisches Detail klingt, hat durchaus einen praktischen Nutzen: Man kann eine Funktion durch Anwenden auf einen Teil ihrer Argumente spezialisieren und diese spezialisierte Funktion im weiteren Programmverlauf beliebig oft aufrufen – der Aufwand für die Spezialisierung muss hierbei nur einmal geleistet werden. Später wird es noch ein Beispiel hierzu geben.

Wichtige Bibliotheks-Funktionen

Die Funktion „filter" nimmt eine Funktion „f" sowie eine Collection „C" und liefert eine Collection mit denjenigen Elementen von „C", für die „f" den Wert „true" zurückliefert. *Listing 5* zeigt, wie dies in Java unter Verwendung der Stream-Bibliothek aussieht, *Listing 6* in Haskell.

Die Funktion „map" nimmt eine Funktion „f" sowie eine Collection „C" und liefert eine Collection, die die Ergebnisse des Anwendens der Funktion „f" auf jedes Element von „C" enthält. *Listing 7* zeigt ein Codebeispiel in Java, *Listing 8* dasselbe in Haskell.

Die allgemeinste und damit flexibelste und mächtigste der hier vorgestellten Bibliotheks-Funktionen ist „reduce", in Haskell unter dem Namen „foldl" bekannt. Sie nimmt eine Funktion „f", einen Startwert (auch „Akkumulator" genannt) und eine Collection „C". Zunächst wird „f" mit dem Startwert und dem ersten Element von „C" aufgerufen. Der zurückgegebene Wert wird zusammen mit dem nächsten Element von „C" an „f" übergeben. Dies geschieht so lange, bis alle Elemente von „C" verarbeitet sind.

```
Arrays.asList(1, 2, 3,
4).stream()
    .filter(x -> x % 2 ==
0).toArray();    // liefert
[2,4]
```

Listing 5

Das Ergebnis der letzten Berechnung ist gleichzeitig das Ergebnis des gesamten „reduce"-Ausdrucks. Diese Verarbeitungskette ist in *Abbildung 1* anhand eines Beispiels anschaulich dargestellt. *Listing 9* zeigt ein Beispiel in Java, *Listing 10* eines in Haskell.

Eine einfache Berechnung

Bisher haben wir die Kernbestandteile funktionaler Programmierung kennengelernt. Nun wollen wir sie in einem praktischen Beispiel zur Anwendung bringen.

Angenommen, wir möchten die Summe der Quadrate der Zahlen von 1 bis 10 berechnen *(siehe Abbildung 2)*. In traditionellem Java-Code würde man dies vermutlich etwa so wie in *Listing 11* berechnen. An dieser Stelle verlassen wir kurz die Welt der funktionalen Programmierung und erinnern uns an gute Praktiken objektorientierter Programmierung. Relevante Stichworte sind hier „Clean Code" und das „Single Responsibility Principle". Letzteres besagt, dass jede Klasse, jede Methode und jede Codezeile genau eine Verantwortlichkeit haben sollte. Schauen wir uns also nochmal unsere obige Imple-

```
filter (\x -> x `mod` 2 == 0)
[1,2,3,4] -- liefert [2,4]
```

Listing 6

```
Arrays.asList(1, 2, 3,
4).stream().map(x -> x +
5).toArray();
  // liefert [6,7,8,9]
```

Listing 7

```
map (\x -> x + 5) [1,2,3,4]  //
liefert [6,7,8,9]
```

Listing 8

```
Arrays.asList(2, 3, 4,
5).stream().reduce(1, (x, y) ->
x*y);
  // liefert 120
```

Listing 9

```
foldl (*) 1 [2,3,4,5]-- liefert
120
```

Listing 10

$$sum = \sum_{i=1}^{10} i^2$$

Abbildung 2: Die Summe der Quadrate der Zahlen von 1 bis 10

```
int sum = 0;
for(int i = 1; i <= 10; i++) {
  sum = sum + i * i;
}
```

Listing 11

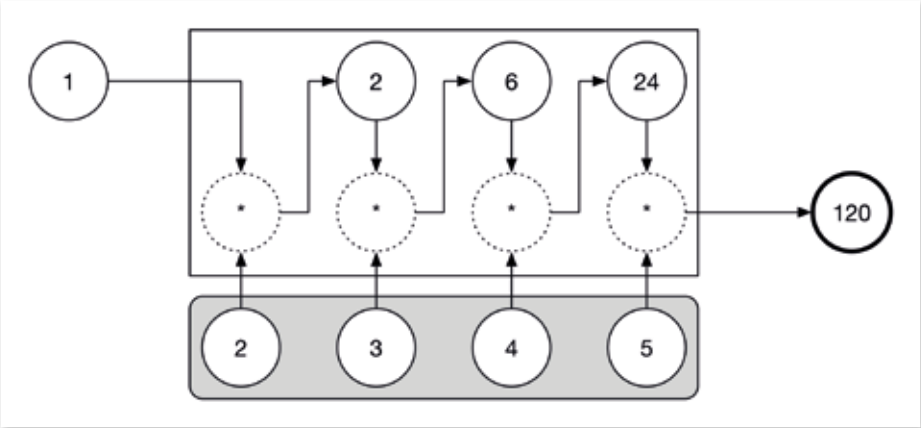


Abbildung 1: Ein Aufruf von „reduce" mit der Multiplikations-Funktion, dem Startwert 1 und der Collection [2, 3, 4, 5]

```
// Erzeugen der Zahlenfolge von 1 bis 10:
IntStream sequence = IntStream.rangeClosed(1, 10);
// Quadrieren einer Zahl:
IntUnaryOperator square = x -> x*x;
// Berechnen der Quadratzahl jeder Zahl in der Folge:
IntStream squaredSequence = sequence.map(square);
// Addieren zweier Zahlen:
IntBinaryOperator add = (x,y) -> x+y;
// Aufsummieren der Quadratzahlen:
Integer sum = squaredSequence.reduce(0, add);
```

Listing 12

```
IntStream.rangeClosed(1, 10).map(x -> x*x)
    .reduce(0, (x,y) -> x+y);       // liefert 385
```

Listing 13

```
foldl (+) 0 (map (\x -> x*x) [1..10])       -- liefert 385
```

Listing 14

mentierung an. In diesen knapp drei Zeilen Java-Code sind so einige Verantwortlichkeiten versteckt:

- Erzeugen der Zahlenfolge von 1 bis 10
- Quadrieren einer Zahl
- Berechnen der Quadratzahl jeder Zahl in der Folge
- Addieren zweier Zahlen
- Aufsummieren der berechneten Quadratzahlen

Eine der Stärken funktionaler Programmierung ist das Trennen dieser Verantwortlichkeiten. *Listing 12* zeigt, wie das in funktionalem Java aussieht. Zusammengesetzt ergibt sich der Code aus *Listing 13*; in Haskell siehe *Listing 14*.

Hierbei fällt etwas unangenehm auf, dass man den Code quasi von rechts nach links lesen muss: Das Definieren der Liste der Zahlen von 1 bis 10 findet ganz rechts statt, links davon wird „map" zum Quadrieren verwendet und wiederum links davon dient „foldl" zum Aufsummieren der Quadrate. Wem dies zu ungewohnt erscheint, der kann sich einen „forward pipelining"-Operator definieren, der das Ganze umdreht: „(>.>) x f = f x". Hier wird das Funktionsargument quasi von links in die Funktion hineingeschoben, was für eine lesbarere Variante sorgt: „[1..10] >.> map (\x -> x*x) >.> foldl (+) 0".

Hier sehen wir auch ein schönes Beispiel für Currying, denn sowohl „map" als auch „foldl" werden partiell evaluiert („map" mit der Quadrierungsfunktion, „foldl" mit Additionsfunktion und Startwert), damit unser „pipelining"-Operator nur noch die Liste durchzureichen braucht.

Weiterführende Themen

Natürlich besitzen funktionale Programmiersprachen noch weitere Features, die sich nicht direkt in Java 8 finden lassen. „Typ-Inferenz", „Pattern Matching", „Algebraische Datentypen", „Lazy Evaluation" oder auch „Monaden" sind nur einige Beispiele dafür. Wer tiefer in die Thematik einsteigen möchte, kann sowohl on- als auch offline auf ein reichhaltiges Angebot an Literatur zurückgreifen.

Fazit

Aspekte funktionaler Programmierung haben inzwischen Einzug in viele nicht funktionale Programmiersprachen gefunden. Dies erlaubt es, elegante neue Sprachkonzepte wie Lambdas oder diverse funktionale Bibliotheksfunktionen auch in einer Sprache wie Java zu nutzen.

Doch es kommt nicht nur darauf an, durch die Programmiersprache gut unterstützt zu werden. Mindestens genauso hilfreich ist das Anwenden funktionaler Konzepte, selbst wenn es dafür keine entsprechende Sprachunterstützung gibt. Hierzu gehören die konsequente Verwendung von Immutabilität und die klare Isolierung von Seiteneffekten in bestimmte Codebereiche. Ist die überwiegende Menge des Codes Seiteneffekt-frei, werden die Vorteile wie größere Verständlichkeit, leichtere Nachvollziehbarkeit, bessere Testbarkeit oder einfachere Parallelisierbarkeit sofort spürbar.

Nicole Rauch
info@nicole-rauch.de

Nicole Rauch ist freiberufliche Software-Entwicklerin und Software-Entwicklungscoach mit umfangreichem Hintergrund in Compilerbau und formalen Verifikationsmethoden. Zu ihren Schwerpunkten gehören „Specification by Example" und „Domain-Driven"-Design sowie die Sanierung von Legacy-Code-Applikationen. Unabhängig davon ist die funktionale Programmierung ihre heimliche Liebe. Neben ihrer Entwicklertätigkeit wirkte sie an der Ausrichtung mehrerer selbstorganisierter Konferenzen und an der Initiierung der Softwerkskammer mit, einer deutschsprachigen User Community zum Thema „Software Craftsmanship". Nicole Rauch führt ganztägige Workshops zum Thema „Einführung in die funktionale Programmierung" durch. Nähere Informationen unter *„http://www.nicole-rauch.de"*

Java als Integrationslösung in einer gewachsenen Anwendungslandschaft

Claus Straube, Landeshauptstadt München

Die Landeshauptstadt München (LHM) hat im Juli 2015 mit dem Einwohnermeldewesen (EWO) eines ihrer zentralen IT-Systeme ausgetauscht und im Zuge dieses Projekts eine stadtweite Richtlinie für Enterprise Application Integration (EAI) entwickelt. Dieser Artikel zeigt, wie EAI auf Basis von Java bei der LHM umgesetzt wird und wie es mit ihrer Hilfe möglich ist, einen zentralen Anwendungsbaustein in einer stark vernetzten und heterogenen Anwendungslandschaft minimalinvasiv auszutauschen.

Die Landeshauptstadt München (LHM) war eine der ersten deutschen Kommunen, die zur Unterstützung der Verwaltung Computer eingesetzt hat. Bereits Anfang der 1980er-Jahre gab es dort eine IT-Abteilung, die spezielle Fachverfahren entwickelt hat, beispielsweise für die Gewerbestelle, das Bürgerbüro oder die Waffenbehörde.

In den vergangenen fünfunddreißig Jahren hat sich einiges getan. Aus der damaligen IT-Abteilung ist ein Eigenbetrieb („it@M") mit mehr als 700 Mitarbeitern entstanden. Zu den anfangs wenigen Fachanwendungen sind inzwischen mehrere Hundert dazugekommen.

Außerhalb der Stadt ist ein Markt für kommunale Software entstanden. Dadurch muss nicht alles selbst programmiert werden, sondern man kann auch Software kaufen. Das hat die Stadt in vielen Fällen auch getan. Entsprechend heterogen stellt sich die aktuelle Anwendungslandschaft dar – sowohl vom Alter als auch von den Technologien durch alle Schichten hindurch. Das wäre grundsätzlich noch kein Problem, wenn nicht die Notwendigkeit bestehen würde, dass diese Anwendungen Daten untereinander austauschen müssten.

Basis-Muster für die Nutzung von EAI bei der LHM

Der Datenaustausch zwischen zwei Fachverfahren wurde in der Vergangenheit in der Regel durch Punkt-zu-Punkt-Verbindungen gelöst. Dazu wurden in den Fachverfahren entsprechende Schnittstellen zum jeweils anderen Fachverfahren implementiert. Mit dieser Vorgehensweise wird der Lebenszyklus der Fachverfahren unweigerlich verbunden. Wird jetzt der Anbieter einer Schnittstelle – in EAI-Terminologie „Provider" genannt *(siehe Abbildung 1, Punkt 1)* – ausgetauscht, so müssen zwangsläufig alle Nutzer, ergo „Consumer" *(siehe Abbildung 1, Punkt 2)*, angepasst werden.

Aufgrund der Altersstruktur der Anwendungslandschaft kommt es bei der LHM relativ häufig zu einem solchen Austausch, durch den in der Folge ein sehr großer Anpassungsaufwand bei den angeschlossenen Fachverfahren entsteht. Man hat deshalb beschlossen, dass Fachverfahren nur noch über eine Integrationskomponente (EAI) miteinander kommunizieren dürfen.

Abbildung 2 zeigt deutlich den Effekt. Während in *Abbildung 1* beide Verfahren auf Protokoll- und Format-Ebene fest miteinander verbunden sind, nutzen hier die Software-Komponenten ihre eigenen Schnittstellen, um miteinander zu kommunizieren – beim Fachverfahren „A" *(Punkt 1)* beispielsweise per Webservice und XML, beim Verfahren „B" (Punkt 2) über einen RPC-Aufruf.

Innerhalb der EAI *(Punkt 3)* findet sowohl die Protokoll-Umwandlung zwischen

Web-Service (WS) und Remote Procedure Call (RPC) statt als auch die Format-Umwandlung „XML zu binär" und zurück. Zusätzlich besteht noch die Möglichkeit, durch ein fachliches Daten-Mapping semantische Unterschiede der Schnittstellen auszugleichen.

Was auf den ersten Blick nur nach einem größeren Aufwand aussieht, bietet auf den zweiten Blick entscheidende Vorteile:

- Ein neu eingeführtes Fachverfahren kann in vielen Fällen so verwendet werden, wie es geliefert wird, da die Systeme in der Regel mit den richtigen fachlichen Schnittstellen ausgestattet sind, die aber oftmals technisch nicht passen (etwa Web-Service vs. Datei-Import).
- Ohne Anpassungswünsche an Schnittstellen wird die Auswahl an Standard-Software-Herstellern größer, da diese oftmals keine kundenspezifische Entwicklungslinie pflegen wollen. Im öffentlichen Bereich reduziert ein größeres Anbieterfeld das Risiko, in der Vergabe ohne Anbieter dazustehen.
- Durch EAI ist es möglich, standardisiert und automatisiert die Schnittstellen zu testen *(siehe später im Artikel)*. Dadurch wird der Testaufwand gesenkt und gleichzeitig die Testqualität erhöht. Man hat sich zusätzlich dazu entschlossen, immer zwei EAI-Artefakte zu erstellen *(siehe Abbildung 3)*. Hier gibt es ein Integrations-Artefakt, das an den Lebenszyklus von Fachverfahren „A" *(Punkt 1)*, und eines, das an den von Fachverfahren „B" *(Punkt 2)* gebunden ist. Die Kommunikation zwischen den beiden Artefakten findet über eine normalisierte Schnittstelle statt, die sich natürlich bei Einführung an der Schnittstelle des Providers orientiert (Punkt 3).

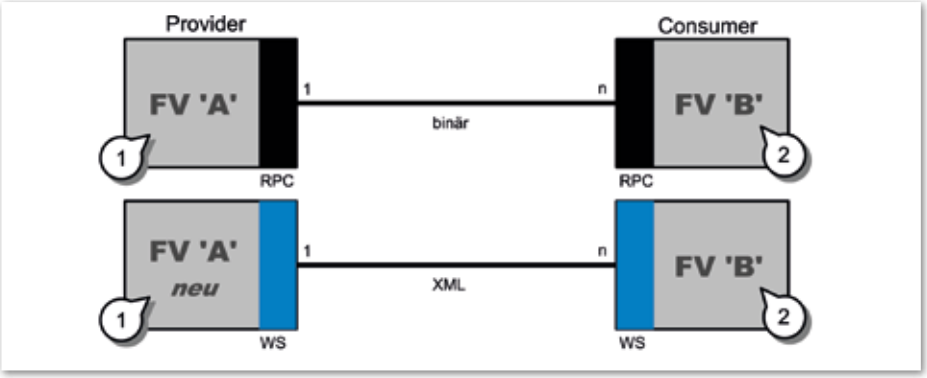


Abbildung 1: Verbindung zwischen Fachverfahren (FV) ohne EAI

Diese Schnittstelle hat ihr eigenes Datenformat und Protokoll. Werden die Artefakte in einem OSGi-Container *(siehe „http://www.osgi.org")* ausgebracht, so wird dies in der Regel OSGi sein, die Kommunikation von Komponenten auf unterschiedlichen Servern (siehe unten) könnte aber auch über Java Messaging (JMS), Rest oder Webservice erfolgen. So entsteht ein fachlich motiviertes API, das das dahinter liegende Fachverfahren für den Anwender völlig transparent macht. Ausschlaggebend für die Entscheidung, die EAI-Komponente zu trennen, waren unter anderem folgende Punkte:

- Anwendungsbezogene Aspekte (Content Filter, Content Enricher, Transformation etc.), die innerhalb der EAI gelöst werden, lassen sich dem Verantwortungsbereich des jeweiligen Fachverfahrens zuordnen. Bei der Landeshauptstadt München gibt es für jede Anwendung Personen, die deren Lifecycle verantworten. Für sie ist es einfacher, wenn es hier eine klare Trennung gibt.

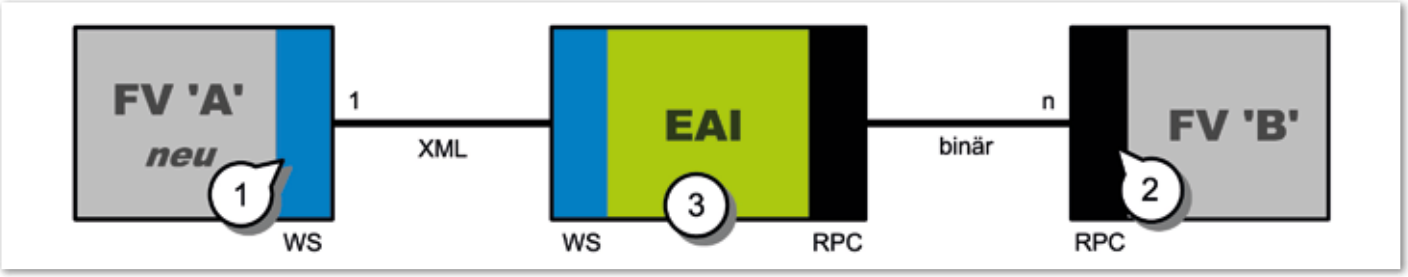


Abbildung 2: Verbindung zwischen Fachverfahren (FV) mit EAI

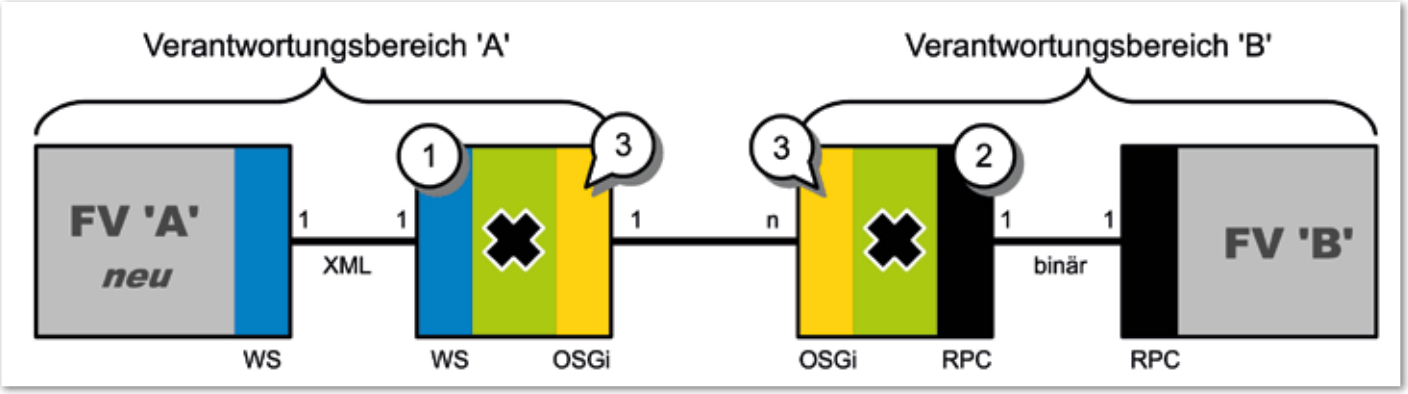


Abbildung 3: Zweistufige EAI zwischen den Fachverfahren

- Bei einer „1:n"-Verbindung können Artefakte auf der Seite der „n" Fachverfahren unabhängig voneinander eingerichtet werden. Dadurch wird die Gefahr von Seiteneffekten minimiert.
- Ist auf einer Seite mit vielen Konsumenten zu rechnen, sind standardisierte Entwicklungsartefakte über Maven-Archetypes *(siehe „https://maven.apache.org")* möglich. Dadurch wird die Umsetzung noch homogener.
- Es entsteht ein fachliches API, das sich unabhängig von einer Software einsetzen lässt.

Testen der Integration

Bei allen Vorteilen, die eine EAI bietet, steigt natürlich die Komplexität der gesamten Anwendungslandschaft. Veränderungen in einem hochintegrierten Verbund von Anwendungen („Anwendungs-Netzwerk") stellen immer ein hohes Risiko dar. Umso wichtiger ist es, die Integration gut zu testen.

Die EAI bietet hier die Möglichkeit, die Kommunikation zwischen zwei Systemen mithilfe von automatisierten Regressionstests (möglichst vollständig) zu testen. Bei der Landeshauptstadt München kommt hierzu ein sogenannter „Testrekorder" zum Einsatz. Dieser zeichnet die Kommunikation zwischen den Fachverfahren (etwa zwischen „B" und „A" in *Abbildung 4*) auf. Hierzu führt ein Tester seine vorab definierten Testfälle in Anwendung „B" (Consumer) durch.

Auf die Anfrage von „B" (Request) erfolgt eine entsprechende Antwort von „A" (Response). An der Schnittstelle werden diese Request- und Response-Nachrichten aufgezeichnet. Der Tester muss nun beurteilen, ob das Ergebnis der Anfrage im Sinne des Testfalls korrekt ist. Wenn ja, dann wird dieser Testfall mit der aufgezeichneten Request- und Response-Nachricht per Apache Subversion *(SVN, siehe „https://subversion.apache.org")* an den Testserver übergeben. Dieser Testserver, der auf Jenkins *(siehe „https://jenkins-ci.org")* basiert, führt nun automatisiert und periodisch alle Testfälle durch.

Bei Änderungen innerhalb des Anwendungs-Netzwerks kann man nun innerhalb von Minuten sehen, welche Teile nicht mehr so wie bisher funktionieren. Wobei ein roter Test nicht unbedingt bedeutet, dass ein Fehler vorliegt. Wenn die Testregeln streng eingestellt sind, dann führt auch eine anders sortierte Ergebnisliste zu einem fehlerhaften Test, obwohl das Ergebnis fachlich völlig korrekt ist und den Konsumenten nicht einschränkt. Der Test sollte also eigentlich grün sein, obwohl das Ergebnis leicht abweicht. Die oben dargestellte Methode befreit einen davon, Hunderte Testfälle per Hand durchzuführen. Es ist allerdings immer noch eine Interpretation der Ergebnisse notwendig, um die richtigen Schlüsse zu ziehen.

Deployment von Integrations-Artefakten

Das Deployment der Artefakte erfolgt nicht immer gleich (zum Beispiel auf einem zentralen EAI-Server), sondern ist von der jeweiligen Fachanwendung abhängig. Bei der LHM gibt es drei Optionen, um Integrations-Artefakte auszubringen:

- Fachanwendung und EAI-Artefakt haben eine gemeinsame Laufzeitumgebung
- Fachanwendung und EAI-Artefakt laufen auf demselben Betriebssystem, haben aber eine getrennte Laufzeitumgebung
- Fachanwendung und EAI-Artefakt haben sowohl eine getrennte Runtime als auch ein getrenntes Betriebssystem

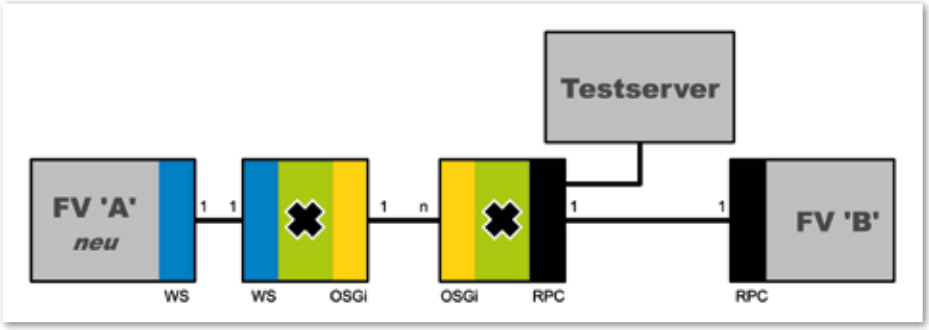


Abbildung 4: Automatisierte Regressionstests mit Testserver

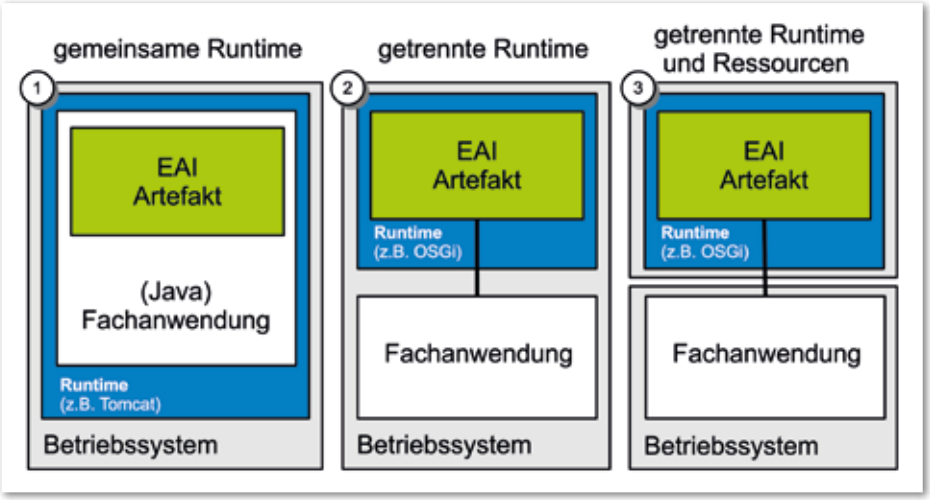


Abbildung 5: Deployment von EAI-Artefakt und Fachverfahren

Abbildung 5 zeigt die Varianten, wobei es sich bei dem grün dargestellten EAI-Artefakt immer um eine der beiden in *Abbildung 3* dargestellten Komponenten (Provider oder Consumer EAI) handelt. Natürlich können beide EAI-Artefakte (Provider und Consumer EAI) in derselben Laufzeitumgebung ausgebracht werden, es ist aber nicht zwingend notwendig.

Variante 1 wird in der Regel bei Eigenentwicklungen in Java verwendet. Das EAI-Artefakt wird dann als eigenes Maven-Modul entwickelt und der Anwendung als Abhängigkeit zur Verfügung gestellt. Die Kommunikation zwischen Fachanwendung und Komponente erfolgt dann beispielsweise über ein Java-API. Die Security in der Kommunikation zwischen EAI und Anwendung wird durch die globale Security der Software abgedeckt. Vorteil dieser Variante ist, dass kein zusätzlicher Server benötigt wird, was Komplexität und Kosten senkt.

Variante 2 wird dann verwendet, wenn die Fachanwendung nicht in Java entwickelt wurde oder es sich um ein Kaufprodukt handelt. Das EAI-Artefakt wird in einer eigenen Runtime ausgebracht, die sich aber immer noch die Ressourcen mit der Fach-

anwendung teilt. Die Security kann hier über eine Absicherung der Schnittstellen zwischen Software und Artefakt erfolgen oder über das Betriebssystem (etwa durch Port-Schließung). Diese Variante bietet dieselben Vorteile wie Variante 1.

Variante 3 wird vor allem bei Kauf-Software verwendet. In vielen Fällen bestehen die Hersteller darauf, dass keine Software neben ihrem Programm installiert ist, die potenziell Seiteneffekte (etwa durch hohen Ressourcen-Verbrauch) auf ihr Verfahren haben kann. Das EAI-Artefakt ist deshalb auf einem eigenen Server installiert. Auf diesem kann unter Umständen auch mehr als ein EAI-Artefakt installiert sein.

Die Security muss hier über eine Absicherung der Schnittstellen zwischen Software oder Artefakt erfolgen, alternativ über eine entsprechende Netzwerk-Segmentierung in Verbindung mit einer Port-Policy. Diese Variante bietet den Vorteil, dass die Ressourcen der beiden Komponenten so voneinander getrennt sind, dass sie sich keinesfalls gegenseitig beeinflussen können.

Monitoring und Logging

Um nach der Installation einen sicheren Betrieb gewährleisten zu können, ist es unabdingbar, dass die Verantwortlichen im Fehlerfall möglichst umfassende Informationen aus dem System erhalten beziehungsweise dort holen können. Die Basis dafür ist ein ineinandergreifendes Monitoring und Logging.

Bei der Landeshauptstadt München ist dies so umgesetzt, dass im Fehlerfall über das Monitoring eine Meldung ausgelöst und – abhängig von der Fehlerkategorie – direkt an den 3rd-Level-Support oder den Betrieb geschickt wird. So lassen sich die Reaktionszeiten bei schweren technischen Fehlern drastisch verkürzen. Wichtig ist allerdings, dass die Fehler entsprechend eingeordnet (Infrastruktur, Technik, Fachlichkeit) werden, um sie nur der richtigen Person zuordnen zu können.

Die Meldung enthält bestimmte Parameter (beispielsweise die Tracing ID, die per Mapped Diagnostic Context (MDC) über Komponentengrenzen weitergegeben wird), um in der über das Logging gesammelten Datenbasis nach weiteren Informationen suchen zu können. So kann zum Beispiel der Weg einer Nachricht über „n" Stationen verfolgt und teilweise auch der fachliche Inhalt (unter Berücksichtigung des Datenschutzes) nachvollzogen werden.

Apache Camel als Basis der EAI

Die oben beschriebenen Konzepte werden bei der LHM auf Basis von Apache Camel *(siehe „http://camel.apache.org")* umgesetzt. Die Auswahl von Camel als EAI-Framework basiert auf einer aufwändigen Evaluation. Den Ausschlag für Camel haben letztendlich unter anderem folgende Punkte gegeben:

- 100 Prozent Java, ohne XML-Konfiguration
- Leichtgewichtig und modular
- Apache 2.0 License
- Gute Testbarkeit
- Implementierung der Integration Patterns *(siehe „http://www.enterpriseintegrationpatterns.com/patterns/messaging")*
- Aktive und offene Community (die LHM hat bereits eine Komponente eingebracht)

Besonders hervorzuheben ist das „fluent"-API beziehungsweise -DSL, durch die es möglich ist, sehr komplexe Dinge in wenigen Zeilen Java-Code abzubilden. *Listing 1* zeigt ein kleines Code-Beispiel, das rekursiv Dateien einliest und diese dann in einen „Gelesen"-Ordner verschiebt (#1). Die Nachricht wird im Anschluss nach ISO-8859-1 konvertiert (#2), geloggt (#3) und an eine asynchrone In-Memory-Queue weitergeleitet (#4). Die Nachrichten werden auf zehn Threads aufgenommen (#5) und an einen Service zur weiteren Verarbeitung übergeben (#6).

Für all diese Tätigkeiten sind nicht mehr als sechs Zeilen Code (netto) notwendig. Dieser ist zudem sehr gut leserlich und dieselben Dinge werden immer gleich umgesetzt. Lässt man fünf Entwickler eine Datei mit Camel von der Platte lesen, wird der Code immer wie unter #1 aussehen. Würden diese Entwickler das in reinem Java lösen, dann hätte man fünf unterschiedlich umgesetzte Lösungen. Insofern unterstützt Camel durch seine Einfachheit und Vereinheitlichung in der Entwicklung auch die langfristige Wartbarkeit der Integrationsartefakte.

Integration am Beispiel von EWO

Dass man mit Apache Camel nicht nur vergleichsweise kleine Probleme wie oben lösen kann, zeigt das Projekt „EWOM5.0". Aufgabe war es hier, das Meldesystem (EWO) der Stadt München auszutauschen. Als führendes System für Bürgerdaten ist EWO eine wichtige Komponente in der Anwendungslandschaft der LHM. Entsprechend wurden auch mehr als achtzig Systeme (dreißig über die Online-Schnittstelle, fünfzig über Batch-EAI) ermittelt, die Daten aus EWO herauslesen oder hineinschreiben. Die Lese-Operationen sind hier der Normalfall. Diese Anwendungen waren in der Regel, wie in *Abbildung 1* dargestellt, direkt mit dem EWO-System verbunden. Das heißt, bei einem Austausch des Einwohnermeldewesens hätten alle diese Anwendungen entsprechend angepasst werden müssen.

Man hat sich innerhalb des Projekts deshalb dazu entschlossen, zwischen EWO und den Altverfahren eine Fassade zu bauen *(siehe Abbildung 6)*. Diese sollte auf der rechten Seite die alte Schnittstelle implementieren, die angebundenen Verfahren müssten also im besten Falle nichts an ihren Schnittstellen ändern. Auf der linken Seite würde diese EAI-Fassade die Schnittstelle des neuen EWO-Produkts implementieren. In der Mitte sollte ein Mapping von „Blau" *(siehe Abbildung 6, Punkt 2)* nach „Schwarz" *(siehe Abbildung 6, Punkt 1)* erfolgen.

Zusätzlich wurde noch die zeitliche Komponente in der EAI berücksichtigt. Da mehr als achtzig Fachverfahren an EWO angebunden werden mussten, bot es sich an, die Fassade bereits zwischen „EWO alt" und die angebundenen Fachverfahren

```
from("file://in?move=.done&recursive=true") // #1
   .routeId("FILE_IN")
   .convertBodyTo(byte[].class, "iso-8859-1") // #2
   .to("log://file.in.route?level=INFO") // #3
   .to("seda://do"); // #4

from("seda://do?concurrentConsumers=10") // #5
   .routeId("DO_SOMETHING")
   .bean(MyDoService.class); // #6
```

Listing 1

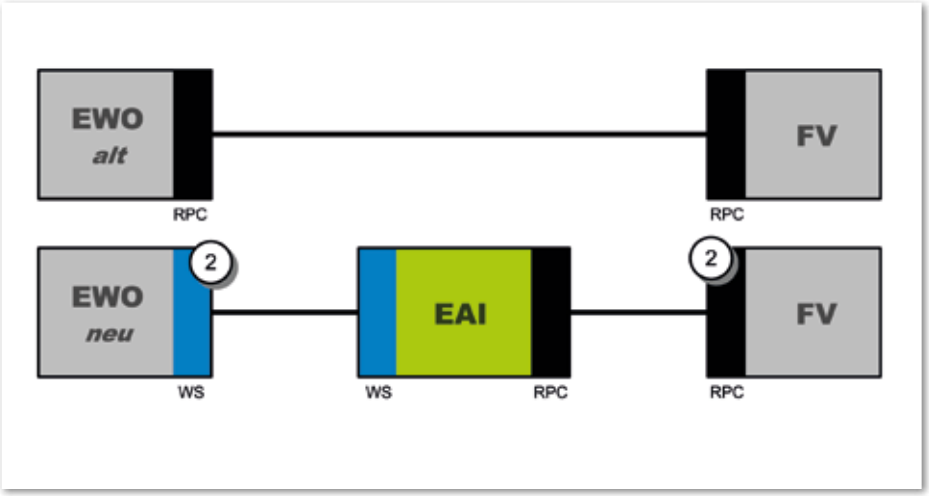


Abbildung 6: Verbindung zwischen EWO und den Fachverfahren (FV)

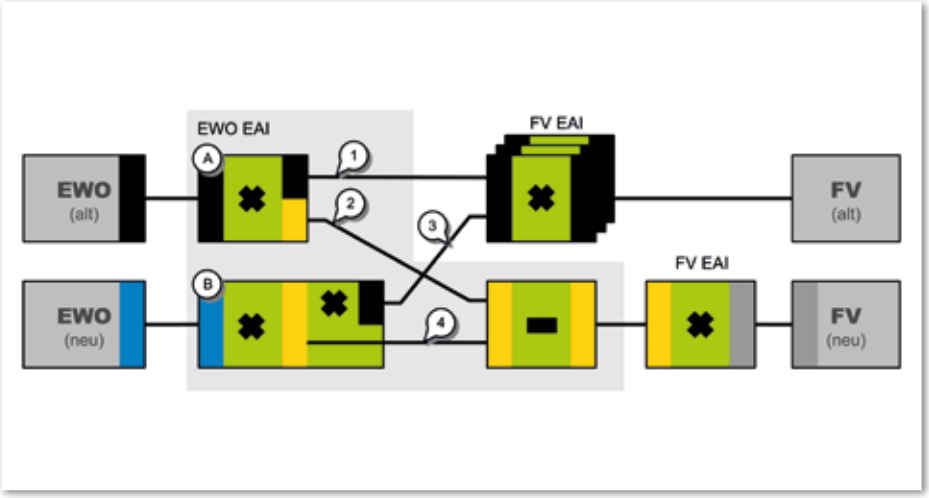


Abbildung 7: Verbindungen zwischen EWO „alt" beziehungsweise „neu" und den Fachverfahren (FV)

zu bauen. Die Hoffnung – die sich auch bestätigte – war, dass am Tag der Inbetriebsetzung des neuen EWO-Verfahrens dann sehr viel weniger zu tun wäre, womit sich an diesem Tag das Risiko auch deutlich verringern würde.

Abbildung 7 zeigt die Umstellungs-Architektur. Weg 1 wurde für alle bisher an EWO angebundenen alten Verfahren implementiert. Auf diesem Weg wurde mehr als ein Jahr vor dem Go-Live des eigentlichen EWOs als erstes Fachverfahren eine Natural-Anwendung an die EAI angebunden. Im Vier-Wochen-Rhythmus wurden dann zwischen vier und sechs Verfahren in unterschiedlichsten Technologien (Natural, Java, PHP etc.) hinzugefügt. Der größte Aufwand war hierbei nicht die Implementierung der EAI-Artefakte, sondern die entsprechenden Testfälle zu definieren und aufzunehmen (siehe oben). Am Ende des Projektes gab es rund tausend fachliche Integrationstestfälle, die täglich gegen beide Systeme liefen und so im Vergleich den Status der Fertigstellung dokumentierten.

Für Anwendungen, die während der Projektzeit – und danach – an EWO angebunden wurden beziehungsweise werden, hat man zusätzlich eine neue Schnittstelle geschaffen (Weg 2, gelbe Schnittstelle). Dies war nötig, da sich die alte Schnittstelle etwas überholt hatte und nicht mehr optimal auf die Bedürfnisse der Fachverfahren gepasst hat. In den nächsten Jahren werden jetzt sukzessive alle Verfahren von der schwarzen auf die gelbe Schnittstelle umgestellt, und zwar immer dann, wenn das Fachverfahren ausgewechselt beziehungsweise im Bereich der EWO-Schnittstelle angepasst wird. Am Tag der Produktivsetzung des neuen EWOs wurde dann vom Weg „1/2" auf Weg „3/4" umgeschaltet. Dank OSGi musste hier lediglich Artefakt „A" durch „B" ersetzt werden. Im Rahmen der Vergleichstests hatte man das täglich geübt, insofern ging dieser Teil sehr einfach über die Bühne.

Fazit

Apache Camel und Java sind perfekt geeignet, um damit ein komplexes und umfangreiches Integrationsvorhaben in einer heterogenen Anwendungslandschaft durchzuführen. Die Technologieauswahl und natürlich auch die Umsetzung der Integration haben maßgeblich dazu beigetragen, dass das EWO-Projekt in Time, in Budget – aber vor allem auch in Quality umgesetzt werden konnte.

Claus Straube
claus.straube@muenchen.de

Claus Straube ist seit dem Jahr 2011 als IT-Architekt für Java-Architektur und Anwendungs-Integration beim internen IT Dienstleister (itM) der Landeshauptstadt München (LHM) angestellt. Zusätzlich berät er freiberuflich Organisationen in Architekturfragen – insbesondere bei Integrationsthemen. Vor seiner Zeit bei der LHM war er als Entwickler und Architekt bei Oracle, Kabel Deutschland und einem Startup beschäftigt.

WildFly-Installationen parametrisieren und mit Git verwalten

Martin Welß

Die Konfiguration eines Application-Servers ist komplex und oft muss in einem Projekt eine ganze Reihe von Installationen verwaltet werden. Die Installationen von Hand abzugleichen, ist mühsam und fehlerträchtig. Daher wird in diesem Artikel am Beispiel von WildFly/JBoss ein Weg vorgestellt, wie man sich mithilfe von Git und konsequenter Parametrisierung der Konfiguration die Arbeit deutlich leichter machen kann.

```
exampleds.jdbc.url: jdbc:postgresql:example
exampleds.user: wildfly
exampleds.password: wildfly
txn.node.identifier: node-test1
jboss.bind.address: localhost
```

Listing 1

Der Application-Server findet sich in einem Projekt an vielen Stellen: auf jedem Rechner der Entwickler, dem Continuous Integration Server, den Test-Systemen und schließlich dem Produktionsserver – also entlang der gesamten DevOps-Pipeline. Hinzu kommt, dass alle Installationen Unterschiede in der Konfiguration aufweisen: Log-in und URL der Datenbank, Kennwort für das SSL-Zertifikat, Mailserver, IP-Adressen und Anbindungen an weitere Backend-Systeme.

Gerade bei den Kennwörtern möchte man vermeiden, dass sie im breit zugänglichen Repository des Versionskontroll-Systems landen. Da dieses, in diesem Fall Git, sowieso schon Teil des Projekts ist und sich somit die Team-Mitglieder damit auskennen, bietet es sich an, es auch für die Verwaltung der Application-Server zu verwenden. Die wichtigsten Operationen werden so zu Einzeilern:

- „git clone" zur Installation
- „git pull" zum Updaten
- „git checkout", um auf einen bestimmten Stand oder Branch zu kommen
- „git diff" oder „git status", um sofort alle lokalen Änderungen zu sehen

Damit man also seine WildFly-Installationen mit Git verwalten kann, muss man zunächst einen Weg finden, um genau die Parameter der Konfiguration in eine Datei auszulagern, die zum konkreten Server gehört und nur dort vorhanden ist (sowie im Backup des Servers). WildFly bietet die Möglichkeit, in den Konfig-Files Java-System-Properties zu verwenden. Diese können gesammelt in einer Datei gespeichert sein, die dann alle Parameter für genau einen konkreten Server enthält (etwa für das Produktionssystem), dort im Home-Verzeichnis des Application-Server-Users liegt und nicht von Git überwacht wird. *Listing 1* zeigt eine exemplarische Properties-Datei „$HOME/system.properties", die wir hier in diesem Beispiel verwenden.

Hier werden die Zugangsdaten zur Datenbank, die Node-Id des Transaktions-Subsystems und die Bind-Address des Servers konfiguriert. Beim Autor liegt diese Datei immer im Home-Verzeichnis desjenigen Users, der WildFly startet, und heißt „system.properties". Auf diese Properties kann nun in den WildFly-Konfig-Files zurückgegriffen werden. *Listing 2* zeigt den entsprechenden Ausschnitt der Datenbank-Konfiguration (siehe *„$JBOSS_HOME/standalone/configuration/standalone-full.xml"*).

Die Properties werden mit einer ähnlichen Syntax wie Shell-Variablen, also in geschweiften Klammern mit vorangestelltem

```
<datasource jndi-name="java:jboss/datasources/ExampleDS" pool-
name="ExampleDS" enabled="true" use-java-context="true">
<connection-url>${exampleds.jdbc.url}</connection-
url>     <driver>postgres</driver>     <security>         <user-
name>${exampleds.user}</user-name>         <password>${exampleds.
password}</password>     </security> </datasource>
```

Listing 2

Dollarzeichen, an den entsprechenden Stellen eingesetzt. Damit WildFly die Properties auch kennt, muss man die Datei beim Start angeben *(siehe Listing 3)*.

Nach diesem kurzen Überblick werden nachfolgend die Schritte aufgezeigt, die notwendig sind, um WildFly mit Git zu verwalten und zu parametrisieren. Das Ergebnis steht als lauffähiges Beispiel auf GitHub in Form von zwei Projekten zur Verfügung.

WildFly mit Git verwalten

Ausgehend von einem neu aus dem Download-Archiv ausgepackten WildFly müssen nun alle durch die Laufzeit veränderlichen Dateien von Git ignoriert werden. Das erreicht man, indem man zunächst im Basis-Verzeichnis der frischen Installation mit „cd $JBOSS_HOME" und „git init; git add --all; git commit -m initial" ein lokales Repository anlegt und den Original-Zustand einchecckt. Danach kann man WildFly am besten mehrmals starten und mit „git status" feststellen, welche Dateien sich geändert haben.

Hier müssen jetzt zwei Fälle unterschieden werden: Es gibt erstens die Dateien, die reine Laufzeit-Informationen enthalten und problemlos ignoriert werden können, also Log-Files oder temporäre Dateien. Dann gibt es allerdings noch die Konfig-Files, die nicht ignoriert werden dürfen; genau die sind das Ziel unserer Parametrisierungs-Bemühungen und dort müssen an den entsprechenden Stellen „system.properties" eingetragen werden. Zu ignorierende Dateien werden dann in der Datei „.gitignore" eingetragen. In diesem Beispiel verwenden wir die Konfiguration „standalone-full"; *Listing 4* zeigt, wie die Datei „$JBOSS_HOME/.gitignore" dann aussieht.

Jetzt fügen wir noch den Postgres-JDBC-Treiber hinzu, um Zugriff auf eine ordentliche Datenbank zu haben. Nach dem Download des JDBC-Treibers von *„https://jdbc.postgresql.org" starten wir „jboss-cli" über „./bin/jboss-cli"*. Danach erscheint das „disconnected"-Prompt. Dort wird der JDBC-Treiber als Modul hinzugefügt *(siehe Listing 5)*.

```
cd $JBOSS_HOME ./bin/standalone.
sh --server-config=standalone-
full.xml -P ~/system.properties
```

Listing 3

Mit „q" kann man „jboss-cli" wieder verlassen. Als Nächstes ist der wichtige Schritt zur Parametrisierung zu gehen und mit einem Editor die Konfigurationsdatei „standalone/configuration/standalone-full.xml" zu bearbeiten, sodass der oben erwähnte Ausschnitt in dem Element „<datasource>" die entsprechenden „system.properties" verwendet.

Damit der Start von WildFly nachfolgend noch fehlerfrei funktioniert, muss ab jetzt ein gültiges Log-in zu einer laufenden Postgres-Datenbank in den „system.properties" eingetragen sein. Bereits jetzt kann man sich mit „git diff" oder „git status" anzeigen lassen, was sich im Vergleich zum letzten Commit verändert hat. Es ist zu empfehlen, eine weitere Stelle in der gleichen Konfigurationsdatei zu parametrisieren *(siehe Listing 6)*.

Dort ergänzt man im Element „<core-environment>" des Transaktions-Subsystems das Attribut „node-identifier", das für jede Installation anders sein sollte, damit es keine Nebeneffekte bei mehreren WildFly-Installationen im selben LAN gibt. Somit ist es ein klarer Kandidat für die Auslagerung in die „system.properties".

Die „jboss.bind.address" ist bereits von Haus aus über ein System-Property einstellbar, man muss es nur setzen. Nach einem Probelauf, um zu prüfen, ob alle Änderungen wie gewünscht funktionieren, lässt sich der erreichte Stand mit Git einchecken *(siehe Listing 7)*.

```
[disconnected /] module add --name=org.postgres --
resources=/home/mw/Downloads/postgresql-9.4-1202.jdbc41.jar --
dependencies=javax.apistandalone/configuration/standalone-
full.xml,javax.transaction.api
```

Listing 5

```
standalone/configuration/standalone_xml_history
standalone/data
standalone/deployments
standalone/log
standalone/tmp
```

Listing 4

```
<subsystem xmlns="urn:jboss
:domain:transactions:3.0">
<core-environment node-
identifier="${txn.node.identi-
fier}">
```

Listing 6

WildFly mit Beispiel-Anwendung

Wer nicht alles selbst eintippen möchte, kann sich von GitHub mit „git clone https://github.com/martin-welss/wildfly-git-install.git wildfly-git-install" den bis hierhin beschriebenen Stand klonen. Allerdings ist ein WildFly ohne installierte Anwendung wenig spannend, daher klont man sich mit „git clone https://github.com/martin-welss/jtrack-ee7.git jtrack-ee7" direkt noch eine kleine JEE-Beispielanwendung dazu.

JTrack verwendet Gradle als Build-System, das muss daher ebenfalls installiert werden. Dazu wird die aktuelle Version von gradle.org heruntergeladen und auf dem lokalen Rechner ausgepackt. Jetzt setzt man als letzten Schritt in der Datei „$HOME/.bashrc" die Environment-Variablen *(siehe Listing 8)*.

Dabei sind die Pfade an die lokalen Gegebenheiten anzupassen. Anschließend kann man WildFly mit „cd $JBOSS_HOME" und „./bin/standalone.sh --server-config=standalone-full.xml -P ~/system.properties" im ersten Terminal starten

```
git add --all git commit -m "add
postgres jdbc driver with system
properties"
```

Listing 7

sowie danach in einem zweiten Terminal mit „cd $HOME/jtrack-ee7" und „gradle loadDB" JTrack bauen und einrichten. Die Anwendung steht nun unter *„http://localhost:8080/jtrack-ee7"* zum Test bereit.

Fortgeschrittene Parametrisierung

In der Praxis ergeben sich noch weitere Anforderungen an eine WildFly-Installation, von denen einige nachfolgend besprochen werden. Beim Debugging für Entwickler kann man sich mit der IDE mit einem WildFly zwecks Remote Debugging verbinden. Dabei ist keinerlei Änderung an der Konfiguration notwendig, sondern es reicht die Option „—debug" beim Start durch „./bin/standalone.sh --debug --server-config=standalone-full.xml -P=$HOME/system.properties".

In der Regel haben die verschiedenen Systeme entlang der DevOps-Pipeline recht unterschiedliche Hardware-Ausstattungen. So haben Entwickler-PCs normalerweise weniger RAM als die Test- und Produktions-Server. WildFly entnimmt diese Optionen der Environment-Variablen „JAVA_OPTS". Sie wird vom WildFly-eigenen Startskript mit Default-Werten initialisiert, wenn sie nicht bereits gesetzt ist. Ihr Wert wird beim Start ausgegeben *(siehe Listing 9)*. Um nun beispielsweise den verfügbaren Speicher zu erhöhen, würde man die Variable „JAVA_OPTS" bereits vor dem Start von WildFly setzen (etwa in der Datei „.bashrc" auf Linux/Mac), und zwar zunächst mit den Default-Werten, die WildFly beim Start ausgibt *(siehe Listing 10)*.

Jetzt kann man die Option „-Xmx" auf „-Xmx1024m" ändern und so den verfügbaren Speicher für die VM auf dem lokalen System verdoppeln. Eine Anpassung von WildFly-Konfig-Files ist nicht erforderlich und das passt bestens in unsere Philosophie, alle lokalen Einstellungen auszulagern. Wenn man die WildFly Web Console verwenden will, muss man mit „./bin/add-user.sh" den Admin User anlegen. Nach Eingabe der erforderlichen Daten wird dieser User eingerichtet und man kann mit „git status" prüfen, welche Änderungen das bewirkt hat *(siehe Listing 11)*.

Das Problem an dieser Stelle ist, dass die User einerseits in der WildFly-Konfiguration liegen und andererseits auf jedem System anders sein können. Da in diesen Dateien auch (verschlüsselte) Kennwort-Informationen liegen, ist zu überlegen, ob man diese Daten mit in das Git-Repository einchecken möchte oder nicht. Wenn es eine gemeinsame User-Basis über die gesamte DevOps-Pipeline gibt und alle User starke Kennwörter verwenden, könnten diese Dateien auch einfach mit eingecheckt werden. Wenn man jedoch nicht möchte, dass verschlüsselte Kennwort-Daten im Git-Repository landen oder es für die verschiedenen Systeme entlang der DevOps-Pipeline keine gemeinsame User-Basis gibt (etwa weil die Entwickler keinen Zugriff auf die Produktionssysteme haben sollen), bieten sich zwei weitere Lösungen an: Man kann einmal den User und die Group Files für Git lokal als unverändert markieren. Dazu werden die betroffenen Dateien mittels „git update-index --assume-unchanged standalone/configuration/mgmt* domain/configuration/mgmt*" für Git als lokal unverändert definiert. Man muss dabei jedoch immer bedenken, dass es sich um eine rein lokale Git-Einstellung handelt und dieser Befehl auf jedem System wiederholt werden muss. Ein erneuter Aufruf von „git status" zeigt dann keine Änderungen mehr an.

Der andere Weg, den Admin User über Kerberos zu verwalten, bietet sich an, wenn in der IT sowieso bereits eine zentrale User- und Rechte-Verwaltung auf Basis von Kerberos im Einsatz ist. Es ist eine ebenso elegante wie mächtige Lösung des Problems, weil sie sich auch gut in das firmenweite IT-Sicherheitskonzept integriert. Seit Version 9 hat WildFly die Möglichkeit, sich mit einer Kerberos-Domain zu verbinden (siehe *„http://darranl.blogspot.co.uk/2014/10/wildfly-9-kerberos-authentication-for.html"*). Es gibt zwei Varianten, Anwendungen einzurichten:

- Wie in früheren Versionen von JBoss üblich, durch das Kopieren des Anwendungs-Archivs in das Deploy-Verzeichnis
- Mithilfe des „jboss-cli"-Deploy-Kommandos

Variante eins hat für unseren Fall den Vorteil, dass die WildFly-Konfiguration unverändert bleibt, aber den Nachteil des etwas umständlichen Mechanismus mit Marker-Dateien, die den Erfolg oder Misserfolg des Vorgangs anzeigen. Variante zwei ist für Skripte zur Automatisierung eleganter, weil es von dem Kommando einen klaren Rückgabewert gibt, der Erfolg oder Misserfolg angibt. Allerdings werden Deployments in die aktuell laufende Konfigurationsdatei eingetragen, in unserem Beispiel würde sich also die Datei „standalone/configuration/standalone-full.xml" ändern. Das ist allerdings auch kein Problem, man muss nur daran denken, dass man vor dem Einchecken von Änderungen alle Anwendungen wieder mittels „jboss-cli undeploy" deinstalliert.

```
export JAVA_HOME=/home/opt/jdk8
export GRADLE_HOME=/home/opt/gradle
export PATH=$JAVA_HOME/bin:$GRADLE_HOME/bin:$PATH export
JBOSS_HOME=$HOME/wildfly-git-install
```

Listing 8

```
JAVA_OPTS:  -server -XX:+UseCompressedOops  -server
-XX:+UseCompressedOops -Xms64m -Xmx512m -XX:MaxPermSize=256m -Djava.
net.preferIPv4Stack=true -Djboss.modules.system.pkgs=org.jboss.byte-
man -Djava.awt.headless=true
```

Listing 9

```
export JAVA_OPTS=-server -XX:+UseCompressedOops  -server
-XX:+UseCompressedOops -Xms64m -Xmx512m -XX:MaxPermSize=256m -Djava.
net.preferIPv4Stack=true -Djboss.modules.system.pkgs=org.jboss.byte-
man -Djava.awt.headless=true
```

Listing 10

```
modified:   domain/configuration/mgmt-groups.properties
modified:   domain/configuration/mgmt-users.properties
modified:   standalone/configuration/mgmt-groups.properties
modified:   standalone/configuration/mgmt-users.properties
```

Listing 11

Fazit

WildFly bietet alle erforderlichen Möglichkeiten, die Konfiguration durch Parametrisierung und Auslagerung in „system. properties" oder Environment-Variablen von lokalen Einstellungen unabhängig zu machen. Dadurch wird es praktikabel, die WildFly-Installationen für die gesamte DevOps-Pipeline in einem zentralen Git-Repository zu verwalten. Das hat eine Reihe von schlagkräftigen Vorteilen gegenüber der Verwaltung von Einzelprojekten:

- Unterschiede in den Konfigurationen können sehr schnell und einfach festgestellt werden. Gerade bei der Fehlersuche ist dieses Feature nicht zu unterschätzen.
- Der Zustand einer Installation ist durch Git immer klar zu erkennen.
- Das Wechseln auf verschiedene Stände beziehungsweise Releases ist mit Git ein Einzeiler. Gerade im Notfall, wenn in der Produktion beispielsweise ein Rollback auf die Vorversion notwendig ist, kann man sich auf Git verlassen.
- Das Experimentieren mit neuen Konfigurationen ist unbeschwert möglich, denn man kann immer wieder auf den Ausgangszustand zurückwechseln. Auch die Verwendung von Branches steht problemlos offen.
- Die Neu-Installation auf einem System ist schnell, zuverlässig und einfach.
- Die meisten Entwickler kennen sich schon mit Git aus.

Diese Vorteile müssen sich vor anderen Tools wie Puppet oder Chef nicht verstecken und können bei der nicht ganz trivialen Verwaltung von WildFly-Clustern ihre ganze Kraft entfalten. Nicht zu vergessen ist allerdings der Umstand, dass auch die Applikationen selbst bei der Parametrisierung mitmachen müssen und ihre systemabhängigen Optionen aus den System-Properties lesen sollten. Und selbst wenn man den Schritt zum zentralen Repository nicht machen möchte, so lohnt sich aus meiner Erfahrung auch einfach die Verwendung eines lokalen Git-Repository zur Verwaltung einer einzelnen Instanz; sie wird durch Git zum Kinderspiel.

Martin Welß
mw@it-focus.de

Martin Welß arbeitet als freiberuflicher Software-Entwickler und Architekt. Seine Schwerpunkte sind auf der einen Seite die Optimierung und Automatisierung der Entwicklungs- und Testprozesse mit modernen Tools wie Git, Gradle, Groovy und Spock und auf der anderen Seite das ganze Spektrum der Java Enterprise Edition 7 mit JBoss/WildFly vom JSF Frontend mit JavaScript über EJB und JMS bis zu JPA mit dem SQL-Backend. Dazu hat er viele Jahre Erfahrung bei der Migration von Alt-Anwendungen auf den jeweils aktuellen Stand der Technik.

Generics, Type Erasure und Fallstricke in der Praxis

Michael Müller

Wer kennt sie nicht, die sogenannten „Generics"? In spitzen Klammern ein Typ angegeben, dann weiß der Compiler, welchen Datentyp eine Variable aufnehmen kann, und sorgt dafür, dass diese keinen unerwarteten Inhalt hat.

Schauen wir uns eine Liste an, die eine Reihe von Integer-Werten aufnehmen soll. *Listing 1* zeigt die nicht typisierte Variante, wie sie in frühen Java-Versionen (bis 1.4) ausschließlich verfügbar war.

Problematisch ist die letzte Zeile. Hier wird der Liste ein String hinzugefügt und dies ist durchaus zulässig, denn die Liste kann potenziell Daten beliebigen Typs aufnehmen.

Wenn nun an anderer Stelle mit diesen Daten gerechnet wird, so kommt es zu einem Laufzeit-Fehler. Mit Version 1.5 hatte seinerzeit Sun die Generics eingeführt um dieses Problem zu lösen. *Listing 2* zeigt nun die typisierte Variante.

```
List integers = new ArrayList();
integers.add(12);
integers.add(20);
integers.add("ten");
```

Listing 1

```
List<Integer> integers = new Ar-
rayList<>();
integers.add(12);
integers.add(20);
integers.add("ten");
```

Listing 2

Der Compiler erkennt nun den falschen Typ in der letzten Zeile und kompiliert den Code erst gar nicht. Mehr noch, moderne Entwicklungsumgebungen wie beispiels-

weise NetBeans weisen bereits im Sourcecode auf den Fehler hin. Heutzutage nutzen wir Generics wie selbstverständlich.

Während andere Hersteller bei der Entwicklung von Programmiersprachen hier und da Brüche in Kauf nehmen, wurde bei Java stets auf eine hohe Kompatibilität geachtet. Im Falle der Generics wurden diese so implementiert, dass der Compiler sie kennt, nicht aber die Java Virtual Machine (JVM). Erreicht wird dies durch eine Typ-Löschung oder „Type Erasure", wie es im Original heißt. Dies bedeutet, der Compiler entfernt die Typ-Information und ersetzt sie, sofern kein speziellerer Typ erzwungen, durch die Angabe von „Object".

Betrachten wir dazu die Klasse „Pair", die ein Schlüssel-Wert-Paar aufnehmen kann *(siehe Listing 3)*. *Listing 4* zeigt, wie das nach der Typ-Löschung aussieht. Netterweise fügt der Compiler aber auch „type casts" ein. Aus *Listing 5* wird so *Listing 6*. Auf diese Weise merken wir als Programmierer nicht, dass der Compiler die Typ-Informationen stibitzt.

Oben hieß es, „Object" sei die Wahl, wenn nichts Spezielleres angegeben ist. Die Spezialisierung lässt sich erreichen, indem wir den generischen Typ von einem anderen ableiten: „public class Pair<K extends Number, V> {...}". In diesem Fall wird für Key nicht „Object", sondern „Number" genutzt.

Type Erasure

Während wir Generics wie selbstverständlich einsetzen, ist das Wissen um Type Erasure schon weniger gestreut. Selbst Entwickler, die es kennen, gehen oft davon aus, dass der Compiler gut vorsorgt und wir in der Praxis immer genau die Typen erwarten können, die wir in den Generics vorgeben. Leider ist das nicht immer so. Häufig werden Typen erst zur Laufzeit gebunden. So versorgt uns beispielsweise Context and Dependency Injection (CDI) zur Laufzeit mit passenden Objekten, die in der Regel kompatibel mit den erwünschten Typen sind.

Oder aber Daten werden interpretiert und dann dynamisch zugewiesen. Zur Laufzeit hat der Compiler allerdings keine Funktion. Hier schützt er uns nicht mehr, wenn unerwartete Daten zugewiesen werden. Dazu ein Beispiel aus der Praxis: In einer Web-Applikation kann der Anwender eine beliebige Anzahl von Artikelnummern angeben. Mithilfe einer „Zufügen"-Schaltfläche kann er eine neue Zeile erzeugen. Nach der Eingabe wird der Artikel aus der Datenbank gesucht und in einer zweiten Spalte angezeigt. Die Artikelnummern werden in einer

```
public class Pair<K, V> {
    private final K _key;
    private final V _value;

    public Pair(K key, V value) {
        _key = key;
        _value = value;
    }

    public K getKey() {
        return _key;
    }

    public V getValue() {
        return _value;
    }
}
```

Listing 3

```
public class Pair {
    private final Object _key;
    private final Object _value;

    public Pair(Object key, Ob-
ject value) {
        _key = key;
        _value = value;
    }

    public Object getKey() {
        return _key;
    }

    public Object getValue() {
        return _value;
    }
}
```

Listing 4

```
Pair<Integer, String> pair = new
Pair<>(10, "ten");
Integer key = pair.getKey();
```

Listing 5

```
Pair pair = new Pair(10, "ten");
Integer key = (Integer) pair.
getKey();
```

Listing 6

```
(01) <?xml version='1.0'
encoding='UTF-8' ?>
(02) <!DOCTYPE html>
(03) <html xmlns="http://www.
w3.org/1999/xhtml"
(04)       xmlns:h="http://
xmlns.jcp.org/jsf/html"
(05)       xmlns:f="http://
xmlns.jcp.org/jsf/core"
(06)       xmlns:c="http://
xmlns.jcp.org/jsp/jstl/core"
(07)       xmlns:jsf="http://
xmlns.jcp.org/jsf">
(08)   <h:head>
(09)     <title>Type Erasure
demo</title>
(10)   </h:head>
(11)   <h:body>
(12)     <h1>Type Erasure demo</
h1>
(13)
(14)     <h:form>
(15)       <table jsf:id="books"
>
(16)         <tbody>
(17)           <c:forEach
items="#{bean.numbers}"
(18)
var="number" varStatus="status">
(19)             <tr>
(20)               <td>
(21)
<h:inputText id="number#{status.
index}" value="#{number}">
(22)
(23)                     <f:ajax
render="@this sum"/>
(24)
</h:inputText>
(25)               </td>
(26)             </tr>
(27)           </c:forEach>
(28)         </tbody>
(29)       </table>
(30)
(31)       <div>
(32)         <h:commandButton
actionListener="#{bean.addNum-
ber}"
(33)
value="Add number">
(34)           <f:ajax
render="books"/>
(35)         </h:commandButton>
(36)       </div>
(37)
(38)       Sum:
(39)       <h:outputText
value="#{bean.sum}" id="sum"/>
(40)     </h:form>
(41)   </h:body>
(42) </html>
```

Listing 7

Liste von Ganzzahlen verwaltet. Schließlich sollen beim Klick auf „Speichern" alle Eingaben (Artikelnummern) gespeichert werden.

Beim Test dieser Applikation kam es beim Speichern jedoch zu einem Laufzeitfehler. Aus irgendeinem Grund waren die Artikelnummern nicht mehr die erwarteten „Integer", sondern „Strings." Ganz offensichtlich wurden hier der Liste Daten eines anderen Typs als des angegebenen untergeschoben. Möglich war dies, weil die JVM aufgrund der Typ-Löschung nicht mehr über die ursprüng-

```
(01) @Named
(02) @ViewScoped
(03) public class Bean implements
Serializable{
(04)   List<Integer> _numbers =
new ArrayList<>();
(05)
(06)   public List<Integer> get-
Numbers() {
(07)     return _numbers;
(08)   }
(09)
(10)   public void addNumber (){
(11)     _numbers.add(null);
(12)   }
(13)
(14)   public int getSum(){
(15)     return _numbers.stream()
(16)                    .filter(i
-> i != null)
(17)                    .re-
duce(0, (a, c) -> a + c);
(18)   }
(19) }
```

Listing 8

```
  public int getSum(){
    int sum = 0;
    for (Integer number : _num-
bers){
      if (number != null){
        sum += number;
      }
    }
    return sum;
  }
```

Listing 9

liche Information verfügte und kein eingeschobenes Cast für den richtigen Typ sorgte. Denn der Compiler fügt zwar, wie oben gezeigt, passende Casts in Zugriffsmethoden ein, schützt aber nicht die Elemente der Liste.

Zur Demonstration des Problems wird hier das Prinzip mit einer abgewandelten und stark vereinfachten Applikation gezeigt: Der Anwender kann auch hier eine beliebige Zahl von Ganzzahlen erfassen. Es wird aber keine weitere Information (wie Artikelname) nachgeschlagen. Vielmehr sollen die Zahlen in dieser Demo addiert werden. Realisiert wird diese Applikation mit JavaServer Faces (JSF, siehe *„http://blog.mueller-bruehl.de/tutorial-web-development“*). *Listing 7* zeigt die Demo-Applikation, bestehend aus einer JSF-Seite sowie einer Named Bean (Java-Klasse).

Die Zahlen werden in einer Liste von Integern vorgehalten. Immer, wenn der Anwender auf „Add number“ klickt, wird der Liste ein neues Element zugefügt (*Listing 7*, Zeilen 32–35, und *Listing 8*, Zeilen 10–12). Die Elemente der Liste werden in einer Tabelle angezeigt, die über die Elemente der Liste iteriert (*Listing 7*, Zeilen 15–29). Unmittelbar nach jeder Eingabe wird ein AJAX-Request gestartet (*Listing 7*, Zeile 23), der die Daten in das Modell überträgt und die Summe ausgibt (*Listing 7*, Zeile 39, und *Listing 8*, Zeilen 14–18). Dabei kommt es zum Laufzeitfehler.

Hier wurden ein wenig Lambda und Streams genutzt. Mancher Entwickler hat sich noch nicht mit diesen Java-8-Features beschäftigen können und mag dies für so etwas wie Voodoo halten. Übersetzt in alte Sprachelemente sieht dies einfach so aus *(siehe Listing 9)*.

Da ein „Integer“ durchaus „null“ sein darf, werden derartige Elemente einfach ignoriert und alle anderen addiert. Dies sieht gut aus. Wo aber verbirgt sich der Laufzeitfehler? Wie gesehen, kann die Liste nach dem Kompilieren Objekte beliebigen Typs aufnehmen. Und in der Tat beinhaltet sie keine „Integer“, sondern „Strings“. Ohne das Wissen um Type Erasure mag der Entwickler an dieser Stelle verzweifeln, hat er doch eine Liste von Integer-Werten deklariert.

Um die Liste tatsächlich mit Integer-Werten zu füllen, müssen wir selbst dafür

```
(01) @FacesConverter(value = "In-
tegerConverter")
(02) public class IntegerConvert-
er implements Converter {
(03)
(04)   @Override
(05)   public Object
getAsObject(FacesContext context,
(06)
UIComponent component, String
value) {
(07)     try {
(08)       return Integer.
parseInt(value);
(09)     } catch (NumberFormatEx-
ception e) {
(10)       return null;
(11)     }
(12)   }
(13)
(14)   @Override
(15)   public String
getAsString(FacesContext context,
(16)
UIComponent component, Object
value) {
(17)     return "" + value;
(18)   }
(19) }
```

Listing 10

sorgen, dass wir zur Laufzeit den richtigen Typ nutzen; der Compiler kann dies nicht prüfen. Im vorliegenden Fall muss man wissen, das JSF erst einmal alles als Strings betrachtet, wenn nicht anders angegeben. Normalerweise erkennt JSF den richtigen Typ, etwa bei einer einfachen Integer-Variablen. Im Falle der Liste wurde jedoch die Typ-Information gelöscht und so nutzt JSF Strings. Zum Glück lässt sich das Problem – einmal erkannt – recht schnell lösen. Wir müssen JSF zwingen, Integer-Werte zu nutzen. In diesem Fall wird dazu ein Integer-Konverter genutzt *(siehe Listing 10)*.

Ein Konverter hat in JSF die Aufgabe, den als String vorliegenden Eingabewert in den gewünschten Typ zu konvertieren (sowie umgekehrt). Entscheidend ist hier die Rückgabe als Integer (Zeile 8). In *Listing 8* wird der Konverter in Zeile 22 eingefügt: „<f:converter converterId="IntegerConverter"/>“.

Fazit

Generics sind ein relativ altes Feature von Java. Um die JVM kompatibel zu halten, wurde dies so implementiert, dass die Typ-Information beim Kompilieren entfernt wird. Daraus können sich bei dynamischer Zuweisung jedoch Probleme ergeben. Der vorliegende Artikel will helfen, den Blick hierfür zu schärfen.

Michael Müller
michael.mueller@mueller-bruehl.de

Michael Müller verfügt über mehr als dreißig Jahre Erfahrung in Software-Entwicklung, Training und Consulting, davon rund fünfundzwanzig Jahre im Gesundheitswesen. Er leitet den Bereich Softwareentwicklung bei der InEK GmbH. Daneben betätigt er sich als freier Autor und Blogger. Er ist Mitglied der JUG Cologne sowie der JSF-Expert-Group.

High-Performance Lists in Java

Thomas Mauch, www.magicwerk.org

Jede Applikation benötigt Daten. Das Java-Collections-Framework verwaltet diese und zählt damit zu den wichtigsten Teilen des JDK. Es definiert Interfaces und stellt Klassen mit Standard-Implementierungen bereit. Gerade für das am häufigsten verwendete List-Interface fehlt aber eine problemlos einsetzbare Implementierung, denn auch die typischerweise genutzte Klasse „ArrayList" hat Schwächen bei bestimmten Operationen. Die Brownies Collections Library stellt deshalb Alternativen bereit, die in allen Fällen schnell sind und gut skalieren. Außerdem bietet sie mit den Key Collections einen Baukasten, mit dem zur Laufzeit einfach Collections für alle Einsatzgebiete erstellt werden können.

Welche Probleme entstehen beim Einsatz von ArrayList? Wie bereits in den Java Tutorials erklärt wird, kommt typischerweise die Klasse „ArrayList" als generell einsetzbare Implementierung zum Einsatz, da sie meistens schneller ist und weniger Speicher braucht als „LinkedList". Tutorials erwähnen allerdings auch, dass Operationen, die nicht am Ende der Liste durchgeführt werden, langsam sein können.

Dieses Performance-Problem hat seine Ursache in der Art der Datenspeicherung: „ArrayList" speichert die Elemente in einem Java-Array, wobei das erste Element immer an Position „0" gespeichert ist. Damit nun nicht bei jeder „add"-Operation ein neues Array alloziert werden muss, wird es immer schrittweise vergrößert und die (noch) nicht genutzten Slots am Ende der Liste bleiben leer.

Wenn man *Abbildung 1* anschaut, kann man einfach erkennen, weshalb das Einfügen am Ende schnell, am Anfang jedoch langsam ist: Am Ende kann direkt das neue Element gespeichert werden, während am Anfang oder in der Mitte zuerst existierende Elemente verschoben werden müssen, um Platz für das neue Element zu schaffen.

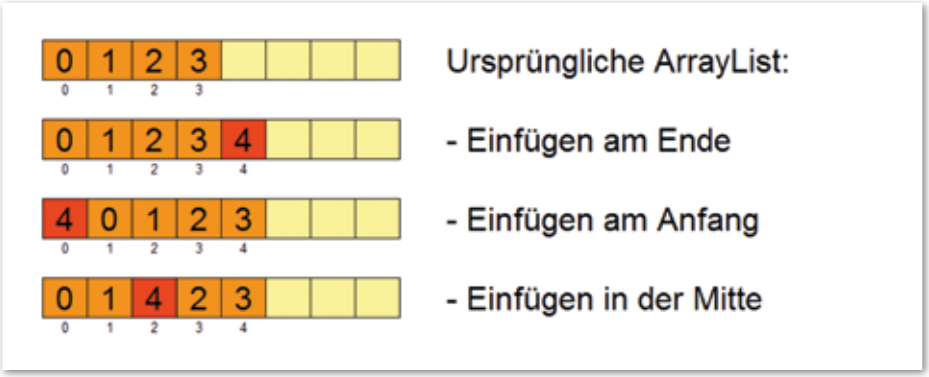


Abbildung 1: Datenspeicherung in „ArrayList"

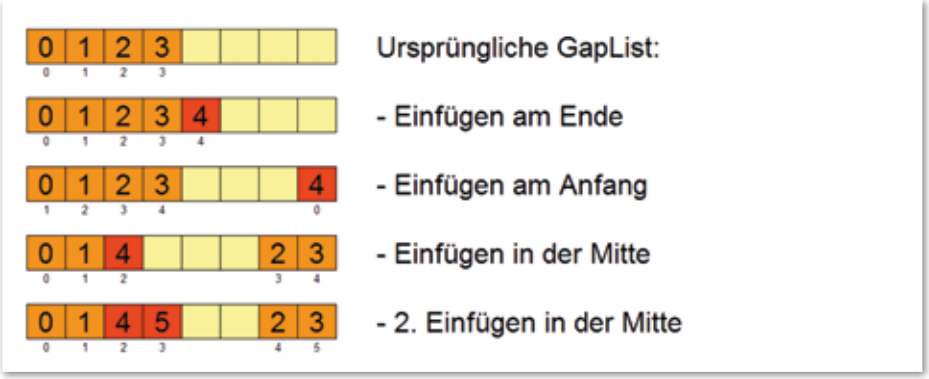


Abbildung 2: Datenspeicherung in GapList

Die Lokalitätseigenschaft

„GapList" optimiert die Datenspeicherung, sodass Operationen an Anfang und Ende der Liste immer schnell sind und an allen anderen Positionen durch Ausnutzen der Lokalitätseigenschaft schnell werden, sobald mehrere Operationen nacheinander in der Nähe stattfinden. „Lokalitätseigenschaft" (englisch: locality of reference) meint, dass in einem gewissen Zeitabschnitt nur auf einen relativ kleinen Bereich der gesamten Datenmenge zugegriffen wird.

Um schnelle Operationen an Anfang und Ende der Liste zu erhalten, gibt es eine einfache Lösung: Anstatt dass wir beim Einfügen am Anfang alle Elemente verschieben, um Platz an Position „0" zu erhalten, belassen wir die Elemente an ihrem Platz und speichern das neue Element am Ende des Arrays. Wir nutzen das Array also als zyklischen Puffer. Für den Zugriff auf die Elemente speichern wir den Offset des ersten Elements im Array und berechnen damit die richtige Position.

Um die Lokalitätseigenschaft der Operationen ausnutzen zu können, erlauben wir ebenfalls eine Lücke (engl.: gap) im Array, das die Elemente speichert. Diese Lücke der nicht benutzten Slots im Array kann an einem beliebigen Ort sein. Sie wird bei Bedarf automatisch erstellt, verschoben oder wieder gelöscht. *Abbildung 2* zeigt, wie die Elemente in GapList gespeichert sind.

Wie man sieht, ist für das Einfügen am Anfang kein Verschieben der Elemente mehr notwendig, sodass diese Operation jetzt gleich schnell wie das Einfügen am Ende ist. Wenn Elemente in der Mitte eingefügt werden müssen, werden nur beim ersten Mal Elemente verschoben, um die Lücke richtig zu platzieren. Bei der zweiten Einfüge-Operation an derselben Position ist dies nicht mehr nötig und damit ist die Operation entsprechend schnell. Dieses Ausnutzen der Lokalitätseigenschaft verbessert die Performance auch, wenn nachfolgende Operationen nicht an der exakt gleichen Position, sondern bloß in der Nähe stattfinden, da dann nur wenige Elemente verschoben werden müssen.

Das Entfernen von Elementen wird analog zum Einfügen behandelt und ist deshalb in „GapList" ebenfalls sehr effizient. „GapList" vereint damit die Stärken von „ArrayList" und „LinkedList", die Implementation ist also so schnell wie „ArrayList" beim Auslesen von Elementen, aber gleichzeitig auch so schnell wie „LinkedList" beim Einfügen oder Entfernen an Anfang oder Ende. Zusätzlich profitieren alle Schreib-Operationen von der Lokalitätseigenschaft der Zugriffe und sind damit typischerweise schneller als bei „ArrayList".

Damit „GapList" auch in bestehenden Anwendungen ohne großen Aufwand eingesetzt werden kann, wurde es als Drop-in-Ersatz für „ArrayList", „LinkedList" und auch „ArrayDeque" konzipiert, es sind also nicht nur die Methoden von „java.util.List" und „java.util.Deque", sondern auch alle anderen „public"-Methoden implementiert. Mit der Implementierung des Deque-Interface bietet „GapList" auch bereits ein mächtigeres API als das spartanische, das ArrayList zur Verfügung stellt und das nicht einmal „getLast()" kennt. Zahlreiche weitere Komfort-Funktionen stellt die Basisklasse „Ilist" zur Verfügung.

Ein weiterer Nachteil der JDK-Implementierungen ist die fehlende Unterstützung primitiver Datentypen. Um die Daten eines primitiven Arrays wie „int[]" zu speichern, muss deshalb eine „List<Integer>" verwendet werden, die ein Vielfaches an Speicher braucht. Die Brownies Collections Library bietet deshalb spezialisierte Implementierungen für alle primitiven Datentypen. Es gibt pro Datentyp jeweils zwei verschiedene Klassen, zum Beispiel folgende für „int":

- „IntGapList" arbeitet direkt mit dem Datentyp „int", kann deshalb aber das Interface „java.util.List" nicht implementieren, das „Object" voraussetzt. Das Interface „IIntList" stellt jedoch alle von „IList" bekannten Funktionen zur Verfügung.
- „IntObjGapList" nutzt intern eine „IntGapList" zur effizienten Datenspeicherung, implementiert allerdings „java.util.List", sodass die Liste auch als „List<Integer>" verwendet werden kann. Die Elemente werden beim Zugriff durch Boxing/Unboxing automatisch in Integer konvertiert.

Mit dem Einsatz einer spezialisierten Version für primitive Datentypen lässt sich viel Speicherplatz sparen. In einer 32-Bit-Umgebung benötigt „IntGapList" nur 25 Prozent des Speichers von „GapList", in einer 64-Bit-Umgebung typischerweise noch weniger.

Die Speicherung großer Datenmengen

Leider löst „GapList" nicht alle Performance-Probleme. Bereits das Hinzufügen am Ende der Liste kann bei einer großen Zahl gespeicherter Elemente langsam werden, wenn das allozierte Array gefüllt ist: In diesem Fall muss ein neues, größeres Array alloziert werden, wohin dann alle Elemente kopiert werden müssen – ein Vorgang, der entsprechend speicher- und zeitintensiv ist.

Auch das Kopieren der ganzen Liste wird eine teure Operation, sobald die Anzahl gespeicherter Elemente entsprechend groß ist. Kopieren ist eine häufige Operation, etwa wenn man über ein API eine Liste zurückgegeben will, die unabhängig von der intern gespeicherten ist.

Die Klasse „BigList" ist deshalb speziell für die Speicherung großer Datenmengen entwickelt worden, wobei „groß" bedeutet, dass sie immer noch in den Hauptspeicher passen. „BigList" speichert die Elemente in Blöcken fixer Größe. Eine Operation zum Einfügen oder Entfernen arbeitet deshalb nur mit den Elementen eines Blocks. Dadurch müssen wenige Daten kopiert werden und die Operation ist entsprechend schnell. Damit die Operationen auf den Blöcken selbst effizient sind, kommt zur Speicherung der Elemente natürlich „GapList" zum Einsatz.

Die Aufteilung der Elemente in die Blöcke erfolgt anhand ihres Index. Für jede Operation auf einer „BigList" muss dann zuerst der Block bestimmt werden, in dem die betroffenen Elemente abgelegt sind. Die Blöcke werden deshalb in einer spezialisierten Baumstruktur verwaltet, die effizienten Zugriff garantiert. Um die Lokalitätseigenschaft ausnutzen zu können, speichert „BigList" bei jedem Zugriff den ermittelten Block, um diesen beim nächsten Zugriff wieder nutzen zu können.

Wenn in einem Block, der schon voll ist, Elemente hinzugefügt werden müssen, wird er in zwei Blöcke gesplittet. Um Speicherplatz zu sparen, werden Blöcke auch wieder zusammengefügt. Dies passiert automatisch, wenn nach einer Lösch-Operation zwei benachbarte Blöcke zusammengeführt werden können.

Zusätzlich wird auf jedem Block ein Reference Counter geführt, der die Verwendung eines „Copy on Write"-Ansatzes erlaubt. Damit sind effiziente Kopier-Operationen auch großer Listen möglich, da die Elemente selbst nicht kopiert werden müssen, sondern nur der Reference Counter anzupassen ist, damit die Blöcke nicht mehr als „privat", sondern als „shared" markiert sind. Bevor eine „BigList" eine Modifikation auf einem „shared"-Block machen kann, muss eine Kopie des Blocks erfolgen. Dazu wird ein neuer Block mit allen Elementen des alten Blocks und Reference Counter „1" erstellt, während der Reference Counter des alten Blocks um „1" verringert wird. Der Reference Counter eines nicht mehr benutzten

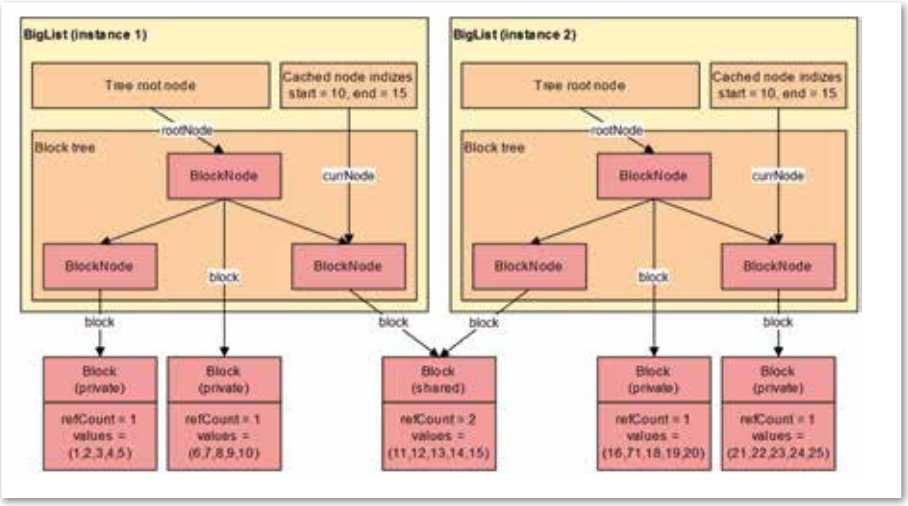


Abbildung 3: Zwei „BigList"-Objekte teilen einen Datenblock

Blocks wird dann spätestens durch den Finalizer der „BigList" vor der Garbage Collection wieder gemindert. *Abbildung 3* zeigt die Details der Implementierung für zwei „BigList"-Instanzen, die einen Block teilen.

Ab welcher Größe sich der Einsatz von „BigList" lohnt, lässt sich leider nicht generell beantworten, da neben der Größe auch die Art der Zugriffe eine Rolle spielt. Grundsätzlich bietet „BigList" aber für jede Größe und alle Operationen eine gute Performance, für umfangreichere Listen meistens die beste.

Key Collections

Eine andere Schwäche des JDK ist die Anzahl der zur Verfügung gestellten Klassen. So kommt es häufig vor, dass man keine Klasse findet, die genau den Anforderungen entspricht. Entweder akzeptiert man diese Lücke, schließt sie mit Programmieraufwand oder zieht Collections aus zusätzlichen Bibliotheken hinzu. Viel besser wäre es doch, wenn man die benötigten Collections selber wie aus einem Baukasten zusammenstellen könnte. Genau diese Funktionalität stellt die Brownies Collections Library mit den Key Collections zur Verfügung.

Die Vielfältigkeit ist durch die Integration der Konzepte von Keys und Constraints möglich. Dazu ein Beispiel: Man möchte die Meta-Informationen zu den Spalten einer Datenbank-Tabelle verwalten, ähnlich wie es beispielsweise „ResultSetMetaData" macht, aber es sollen ebenfalls Änderungen unterstützt werden. Damit ergeben sich folgende Anforderungen:

- Spalten haben eine explizite Ordnung, wie sie beispielsweise für eine Abfrage mit „select *" verwendet wird
- Die Spaltennamen einer Tabelle müssen einzigartig sein, doppelte Namen sind also zu verhindern
- Effizienter Zugriff auf die Spalten-Informationen soll sowohl über die Position der Spalte in der Tabelle als auch über den Spaltennamen möglich sein, analog dazu, wie „ResultSet" den Zugriff auf Daten mit „getObject(int)" und „getObject(String)" erlaubt

Leider bietet das JDK für diese Anforderungen keine optimale Lösung. „List" erlaubt nur effizienten Zugriff über die Position, nicht aber über den Namen, während es bei „Map" gerade umgekehrt ist. Nur wenn man davon ausgehen kann, dass die Liste immer nur wenige Elemente enthält, können wir den Zugriff über den Namen durch Iteration realisieren. Eine Tabelle wird zwar nie eine wirklich riesige Menge von Spalten haben, aber das Iterieren ist bereits bei tausend Einträgen nicht mehr effizient.

Um eine wirklich skalierbare Lösung zu erhalten, muss man die „List" mit einer „Map" synchronisieren. Dies ist keine unmögliche Aufgabe, aber jede Menge Arbeit für eine eigentlich alltägliche Anforderung. Mithilfe der Key Collections hingegen lässt sich eine Datenstruktur mit den gewünschten Eigenschaften durch einen einzigen Aufruf erzeugen *(siehe Listing 1)*. Um die gewünschte Funktionalität zu erreichen, nutzt das Beispiel sowohl Keys als auch Constraints. Diese beiden Konzepte werden nachfolgend vorgestellt.

```
Key1List<Column,String> cols =
  new Key1List.
Builder<Column,String>.
  withKey1Map(Column::getName).
  withPrimaryKey1().build();

class Column {
  String getName() { ... }
  ...
}
```

Listing 1

```
class Table {
  Key1List<Column,String> cols =
...;
  List<Column> getColumns { return
cols; }
}
```

Listing 2

Constraints

Ein Constraint auf einer Collection definiert, welche Bedingungen Elemente erfüllen müssen, damit sie in der Collection enthalten sein können. Beispiele für Constraints sind also, dass Elemente nicht „null" sein oder dass gespeicherte Strings nur Großbuchstaben enthalten dürfen. Zwar haben auch gewisse JDK-Klassen implizite Constraints, aber diese Einschränkungen sind fix und lassen sich weder konfigurieren noch einfach überschreiben.

Constraints sind nicht nur zur Datenhaltung selbst wichtig, sie sind vor allem ein zentraler Baustein, um ein mächtiges API zur Verfügung stellen zu können. Nur mit Constraints ist es möglich, eine Collection, die auch zur internen Speicherung der Daten verwendet wird, direkt via API an den Client weiterzugeben. So lässt sich ein vollständiges und sicheres API mit den Key Collections in einer Zeile realisieren *(siehe Listing 2)*. Ohne solche Unterstützung müsste man die Constraints bei jeder Änderung manuell validieren.

Die Key Collections unterstützen eine Vielzahl von Constraints:

- *withMaxSize*
  Maximal erlaubte Größe der Collection
- *withWindowSize*
  Maximal erlaubte Größe der Collection, bei deren Erreichen Elemente am Anfang automatisch entfernt werden (nur für Listen)
- *withNull*
  Nullwerte sind nicht erlaubt
- *withConstraint*
  Gespeicherte Elemente müssen definierte Bedingung erfüllen
- *withBeforeDelete/withBeforeInsert*
  Definition von Methoden, die vor dem Entfernen/Hinzufügen von Elementen ausgeführt werden

Keys

Ein Key ist ein Wert, der von einer Funktion aus einem in einer Collection gespeicherten Element bestimmt wird. Wie man in *Listing 1* sehen kann, speichern die Key Collections deshalb immer Elemente – als Gegensatz zum JDK-Map-Interface, das Elemente mit externen Keys assoziiert. Die von der Funktion definierten Werte bezeichnet man als „Key Map". Wie auch Einträge in einer Map oder einem Set sollten die als Keys genutzten Werte grundsätzlich „immutable" sein, sie dürfen sich also nach dem Hinzufügen des Elements zur Collection nicht mehr ändern.

Während *Listing 1* eine Collection mit einem definierten Key zeigt, kann es pro Collection auch keine oder zwei Key Maps geben. Auch das Element selbst kann als Key genutzt werden, in diesem Fall sprechen wir von einem „Element Set".

Das Beispiel zeigt auch, dass die Keys einer Collection nicht nur für den effizienten Zugriff auf die Elemente, sondern auch für die Definition von Constraints oder anderen Eigenschaften verwendet werden können. Für jede Key Map oder das Element Set können folgende Eigenschaften festgelegt sein:

- *Null-Werte*
  Erlaubt oder verboten
- *Duplikate*
  Erlaubt oder verboten. Ebenfalls kann spezifiziert werden, dass zwar Null-Werte mehrmals vorkommen können, andere Werte aber einzigartig sein müssen
- *Sortierung*
  Sortiert oder nicht. Wenn eine Key Map sortiert ist, kann ebenfalls festgelegt werden, dass diese Reihenfolge auch für die Elemente selbst verwendet werden soll, wodurch wir eine sortierte Collection erhalten.

Mit Kenntnissen über relationale Datenbanken werden einem diese Konzepte bekannt vorkommen. Im Beispiel ist, wie es der Name der Methode sagt, der Spaltenname als Primary Key der Collection definiert. Analog zum Primary Key in einer Datenbank-Tabelle wird damit garantiert, dass die zu speichernden Elemente korrekt sind und der Zugriff über den Key effizient erfolgen kann.

Die Key Collections implementieren das Collection- oder das List-Interface. Durch die Kombination mit der Anzahl der genutzten Keys entstehen sechs Klassen, die entweder das Collection- oder das List- beziehungsweise Ilist-Interface implementieren:

- KeyCollection<E>
- Key1Collection<E,K1>
- Key2Collection<E,K1,K2>
- KeyList<E>
- Key1List<E,K1>
- Key2List<E,K1,K2>

Mit diesen Klassen und der aufgezeigten Funktionalität lassen sich beispielsweise Collections für jeden gewünschten Anwendungsfall realisieren:

- *SortedList (sortierte Liste)*
  KeyList.withOrderByElem()
- SetList
  KeyList.withPrimaryElem ()
- *BiMap*
  Key2Collection.withKey1Map().withKey-2Map()
- *MultiSet*
  KeyCollection.withElemCount()

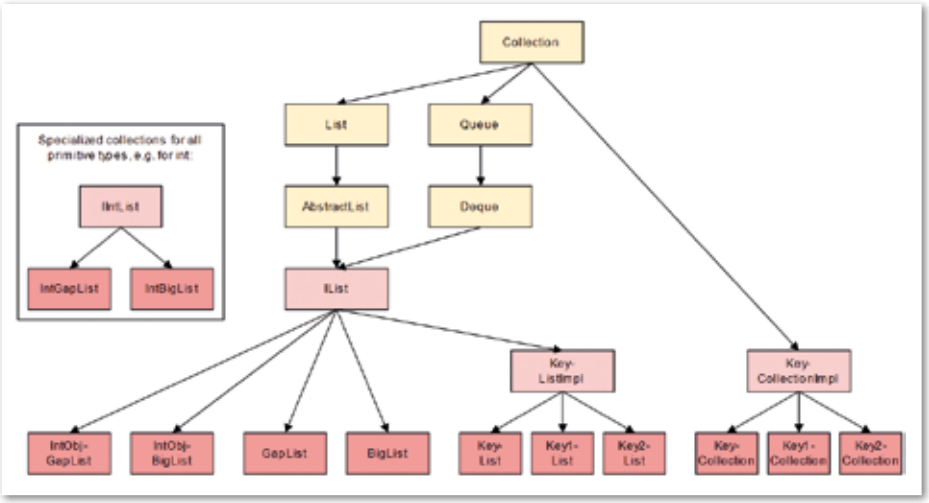


Abbildung 4: Überblick über die Klassen der Brownies Collections Library

Fazit und Ausblick

Das JDK bietet gerade für das meist verwendete Collection-Interface „java.util.List" keine universell einsetzbare Implementierung. Die Brownies Collections Library füllt mit „GapList" und „BigList" diese Lücke. „GapList" bietet eine gute Performance für alle Operationen und kann als Drop-in-Ersatz für „ArrayList", „LinkedList" und „ArrayDeque" verwendet werden.

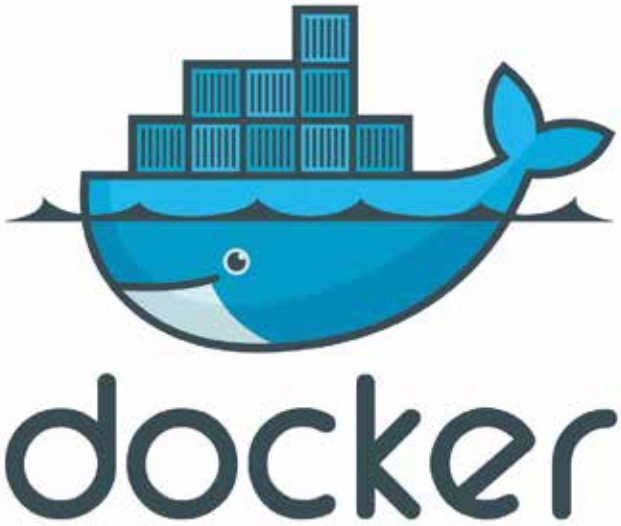
„BigList" skaliert zusätzlich auch für große Datenmengen und garantiert, dass alle Operationen schnell und mit geringem Speicherbedarf ausgeführt werden können. Für beide Klassen gibt es spezialisierte Varianten für primitive Datentypen.

Die Key Collections erweitern die Möglichkeiten der Java Collections. Die einfache Definition von mächtigen Datenstrukturen, der die gewünschte Funktionalität deklarativ zur Laufzeit zugewiesen werden kann, erlaubt ein präzises Abbilden des Datenmodells.

Die Brownies Collections Library ermöglicht damit dem Entwickler, mit geringem Programmier-Aufwand Applikationen zu erstellen, die effizient bezüglich Performance und Speicherplatz-Verbrauch sind und dadurch gut skalieren. *Abbildung 4* zeigt zum Abschluss alle Klassen der Library im Überblick. Weitere Informationen und die Library stehen unter *„http://www.magicwerk.org/collections"*.

Thomas Mauch
thomas.mauch@gmx.net

Thomas Mauch interessiert sich seit der Zeit des Commodore 64 für alle Gebiete der Software-Entwicklung. Er studierte bis zum Jahr 1996 Informatik an der ETH Zürich und hat seither mit verschiedensten Technologien und Programmiersprachen gearbeitet. Der aktuelle Fokus liegt auf Java und Java EE. Zurzeit ist Thomas Mauch bei Swisslog als Software-Architekt tätig, daneben entwickelt er Open-Source-Projekte.



Demystifying Docker

Bernd Fischer, MindApproach GmbH

Die Themen „Docker" und „Container" sind zurzeit auf jeder größeren Konferenz und in (fast) jeder Zeitschrift omnipräsent vertreten. Handelt es sich dabei tatsächlich um eine neue zukunftsträchtige Technologie oder ist es nur alter Wein in neuen Schläuchen? Vor allem: Was bedeutet es für den (Java-)Entwickler? Was kommt auf ihn zu und was verändert sich für ihn?

Dieser Artikel zeigt ausgehend von einer (sehr) einfachen Java-Web-Anwendung einige Einsatzmöglichkeiten für Docker auf (Strategien) und beschreibt Konsequenzen für den Aufbau der Kern-Entwicklungsumgebung, mit anderen Worten: für den Entwickler-Arbeitsplatz. Nach einer kurzen Einführung in das Thema wird eine einfache Java-Web-Anwendung in mehreren Schritten „dockerisiert" und anhand dieses Beispiels einige Einsatzmöglichkeiten für Docker in der Java-Entwicklung diskutiert.

Die Antwort auf die Frage, warum sich ein Java-Entwickler mit dem Thema „Docker" beschäftigen sollte, lässt sich im Rahmen dieses Artikels nicht annähernd beantworten. Um sich jedoch nicht gänzlich um die Antwort zu drücken, zählt der Autor einige grundsätzliche Aspekte auf, die ihn auf diesen Weg führten. Eine ganz wesentliche Einflussgröße war und ist die DevOps-Bewegung, die für ihn als Software-Entwickler durch den Ausspruch von Werner Vogels „You build it you run it" *[1]* auf den Punkt gebracht wird. Der Bogen lässt sich dann weiter über „Continuous Delivery" *[2]* hin zu „Automate Almost Everything" und „Infrastructure as Code" beziehungsweise „Immutable Infrastructure" spannen. Konkret beschäftigte er sich vor allem damit, die Aufgabenstellung, die verschiedenen Arbeits-Umgebungen innerhalb des Entwicklungsprozesses, also die Entwickler-Workspaces und die Test- beziehungsweise QS-Landschaften, so identisch wie sinnvoll möglich mit der Produktivumgebung zu gestalten.

Was ist Docker?

Docker realisiert eine spezielle Form der Virtualisierung, die sich vor allem von anderen bekannten Vertretern wie VirtualBox, VMware, Parallels oder KVM dadurch unterscheidet, dass keine mehr oder weniger vollständige Nachbildung eines Rechners inklusive Hardware, BIOS etc. erfolgt, sondern im Grunde lediglich Prozesse voneinander isoliert sind *(siehe*

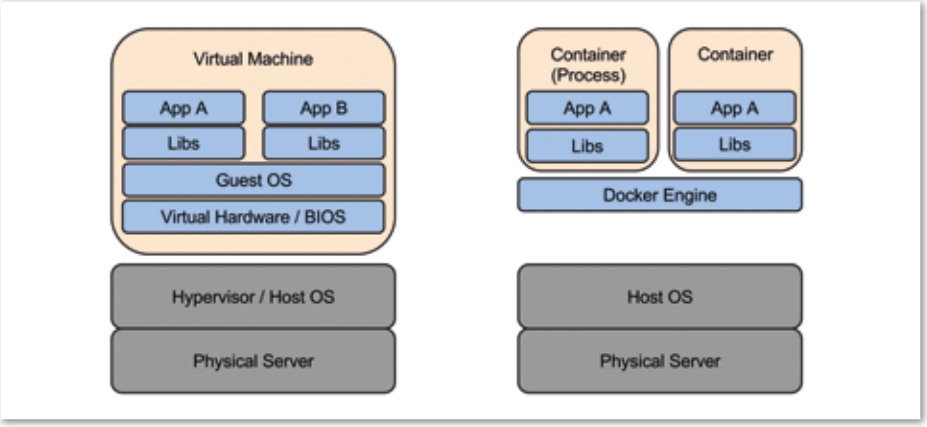


Abbildung 1: Container vs. Virtual Machine (VM)

Abbildung 1). Aus diesem Grund wird diese Form häufig auch „Prozess-Virtualisierung" genannt (siehe auch *[4]*, Kapitel 1.3, Virtualisierungstechniken).

Ein Prozess, auch wenn er in einem Container, wie diese virtuelle Einheit oft auch genannt wird, beheimatet ist, wird im Grunde ganz normal durch das Betriebssystem des Host-Rechners gestartet und ausgeführt. Jedoch kann und wird die Sichtbarkeit von Betriebssystem-Ressourcen – dazu zählen die Prozess-Namen beziehungsweise -IDs, das Filesystem, die Netzwerk-Interfaces, die User etc. – zwischen den auf einem Rechner laufenden Containern untereinander und gegenüber dem Host eingeschränkt beziehungsweise so gesteuert, dass für den einzelnen Container der Eindruck entsteht, er wäre allein in seinem eigenem Universum. Auch wenn sich in diesem Aspekt Container und virtuelle Maschinen im Grundsatz stark ähneln, sind die durch (Voll- oder Para-) Virtualisierungstechnologien errichteten Mauern zwischen VM und Host sowie zwischen den VMs um einiges stärker als bei den Containern.

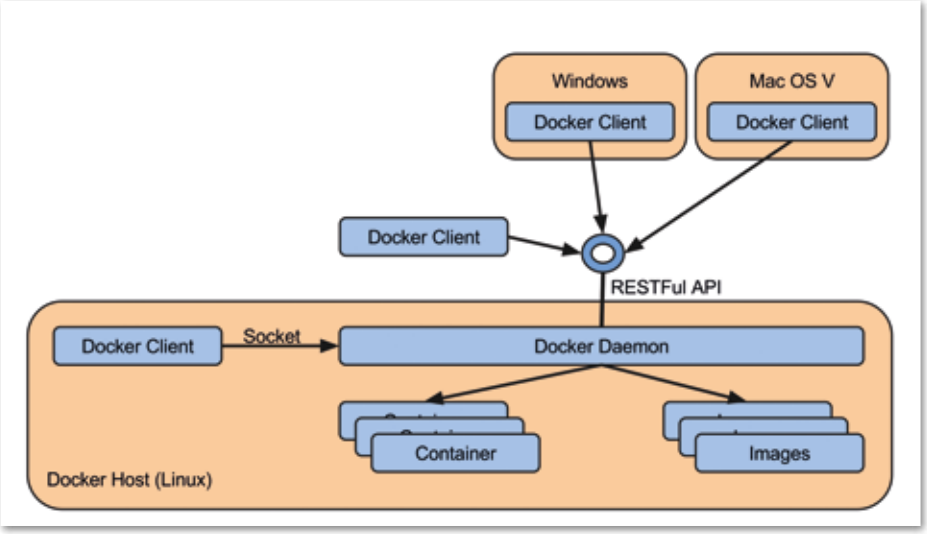


Abbildung 4: Architektur der Docker Engine

Auf der anderen Seite punkten diese durch ihre Leichtgewichtigkeit, die sich vor allem in einem praktisch kaum vorhandenen Overhead hinsichtlich CPU-Leistung und Speicherverbrauch im Vergleich mit außerhalb von Containern gestarteten Prozessen und erst recht im Vergleich zu virtuellen Maschinen zeigt. So lag Mitte September 2015 der Rekord der „Raspberry Pi DockerCon Challenge", bei der es um

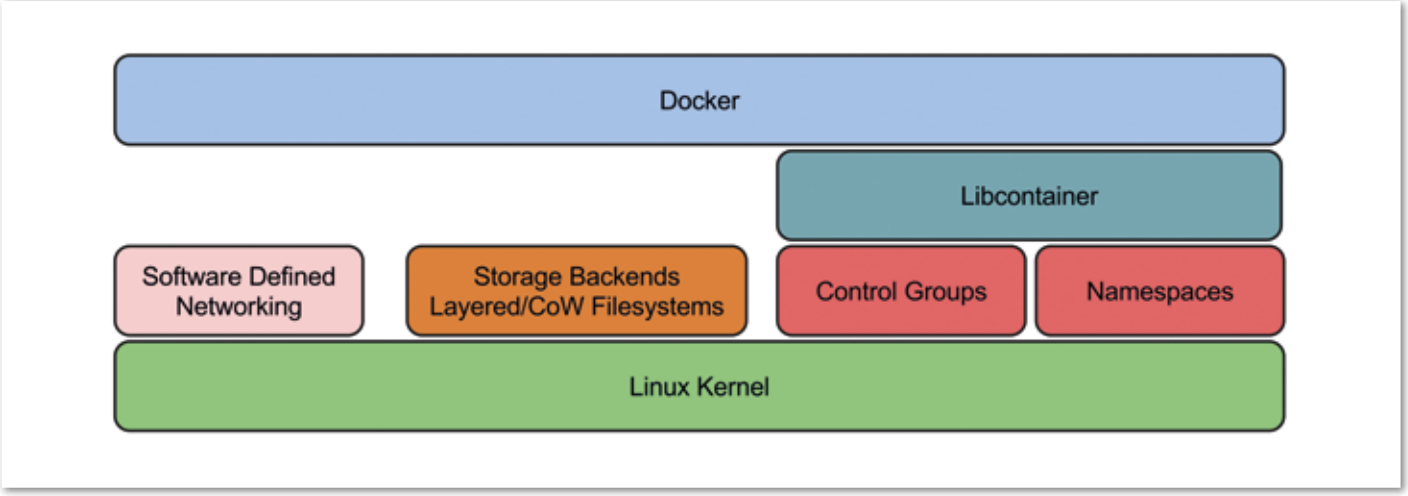


Abbildung 2: Technologien des Linux-Kernels zur Prozess-Isolation

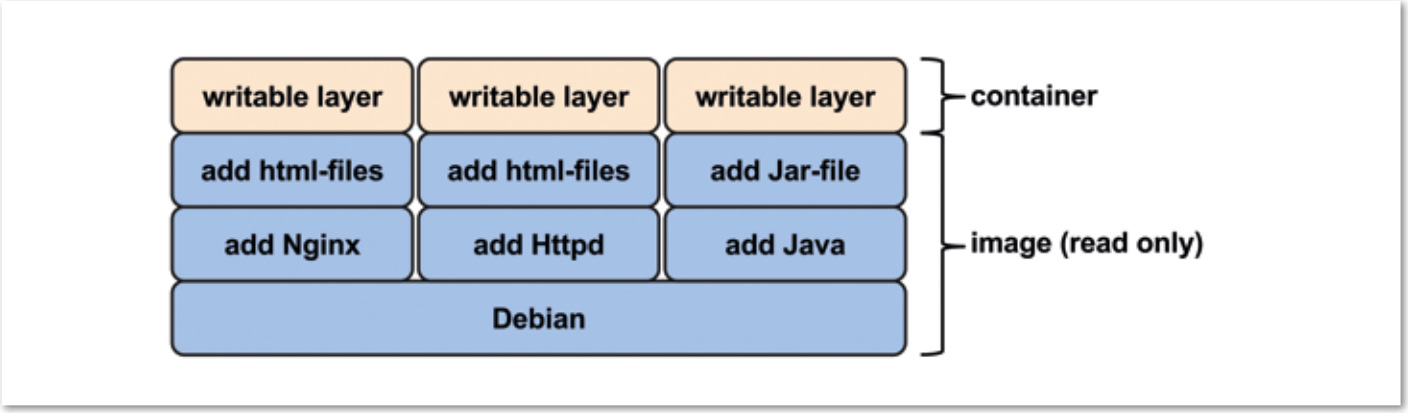


Abbildung 3: Images und Container

die größtmögliche Anzahl gleichzeitig laufender Web-Server auf einem einzelnen Pi 2 geht, bei 2.334.

Docker erweitert die für die eben beschriebene Funktionalität notwendige „Magic", die Bestandteil der Host-Betriebssysteme (im Moment vor allem Linux ab Kernel-Version 3.8, siehe *Abbildung 2*) ist, um eine intelligente Art und Weise der Speicherung und der Verteilung von Anwendungen, die später in einem Container ausgeführt werden sollen.

Statt aus einer oder mehreren monolithischen Dateien bestehen „Images", wie diese im Docker-Jargon genannt werden, aus einzelnen schreibgeschützten („read-only") Schichten („Layern"), die einzeln verteilt, referenziert und auf diese Art und Weise wiederverwendet werden können. Beim Start eines Containers werden die Schichten eines Image im Sinne eines Overlay-Filesystems zu einem Ganzen überlagert und durch eine zusätzliche beschreibbare Schicht, in der alle Änderungen, die durch den oder die im Container laufenden Prozesse hervorgerufen werden, gespeichert sind *(siehe Abbildung 3)*. Dieses Prinzip ist vergleichbar mit dem der Life-CDs, bei der die „Read-only"-CD durch ein beschreibbares Dateisystem ergänzt und dem Anwender, der praktisch von oben auf diesen Stapel von Dateisystemen schaut, als Einheit präsentiert wird.

Docker verwenden

Das inzwischen „Docker Engine" genannte Docker-Kernprojekt verwendet die im vorigen Abschnitt beschriebenen Funktionalitäten und stellt darauf aufbauend ein (mehr oder weniger) RESTful-API über den „Docker-Daemon" für folgende Funktionen bereit:

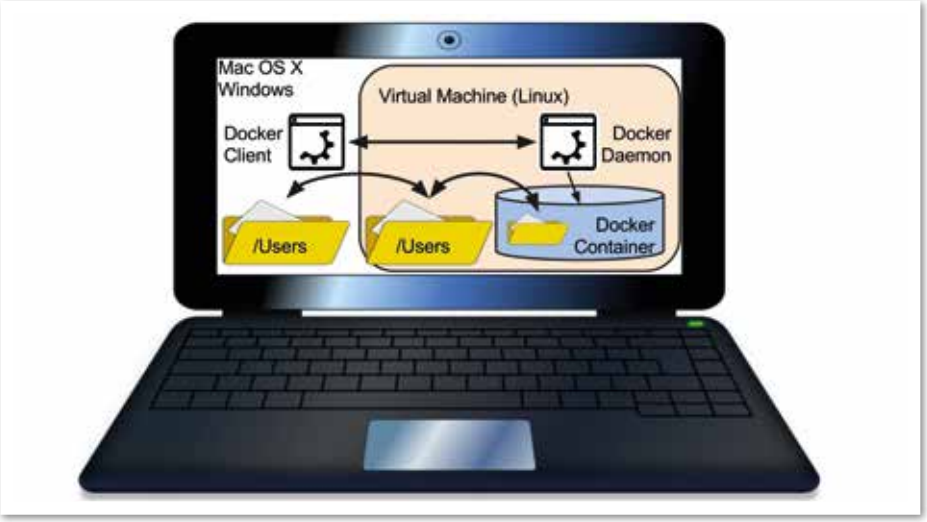


Abbildung 5: Aufbau einer Entwicklungsumgebung mit Docker

- Starten von Prozessen
- Verwalten der Images und der Container
- Organisation der (Netzwerk-)Verbindungen der Container untereinander und zur Außenwelt

Der ebenfalls in diesem Projekt enthaltene Kommandozeilen-Client (CLI) „docker" verbindet sich über einen Socket (lokal unter Linux) oder per Http(s) mit diesem Daemon *(siehe Abbildung 4)*.

Leider gibt es für die beiden am häufigsten auf Entwicklerarbeitsplätzen anzutreffenden Betriebssysteme Windows und Mac OS X (noch) keine native Implementierung des Docker-Daemon, sodass entweder auf eine Hilfskonstruktion unter Einsatz einer auf geringen Ressourcenverbrauch zugeschnittenen Linux-VM wie dem relativ bekannten Projekt „Boot2Docker" und einem trickreichen Mapping von Verzeichnissen *(Abbildung 5* verdeutlicht das Prinzip) oder auf einen Linux-Rechner zurückgegriffen werden muss.

```
HTTP/1.1 200 OK
Content-Length: 12
Content-Type: text/
plain;charset=UTF-8
Date: Tue, 29 Sep 2015 09:09:22
GMT
Server: Apache-Coyote/1.1
X-Application-Context: application

Hello World!
```

Listing 1

Das Projekt „Docker Machine" mit dem gleichnamigen Kommandozeilen-Client erzeugt und verwaltet im Zusammenwirken mit einer Vielzahl von Virtualisierungs- und Cloud-Anbietern, darunter Oracle VirtualBox für den lokalen Arbeitsplatz oder Amazon EC2, Microsoft Azure, Digital Ocean oder Google, in relativ kurzer Zeit fertig konfigurierte Docker-Hosts. Natürlich können sowohl die VMs als auch der Docker-Daemon manuell erstellt und konfiguriert werden, die

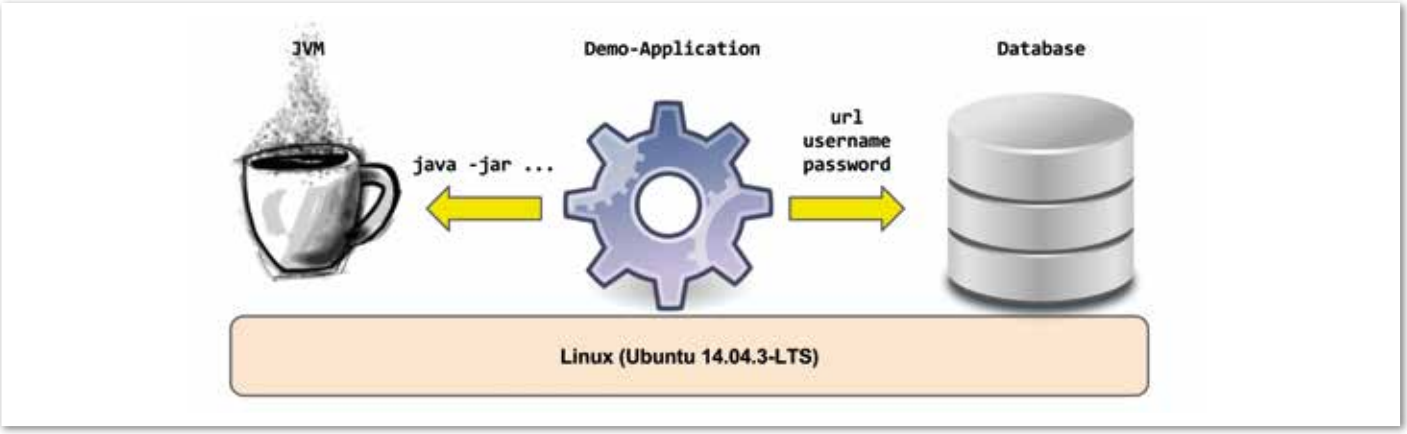


Abbildung 6: Architektur der Beispielanwendung

```
docker ps -a
CONTAINER ID  IMAGE  COMMAND  CRE-
ATED  STATUS  PORTS  NAMES
```

Listing 2

```
# s01-docker-compose.yaml
mysql:
  image: mysql:5.6.26
  expose:
    - "3306"
  ports:
    - "3306:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=9876
    - MYSQL_USER=test
    - MYSQL_PASSWORD=1234
    - MYSQL_DATABASE=test
```

Listing 3

Anleitung hierfür würde allerdings den Umfang dieses Artikels sprengen.

Ausgangspunkt: Die Spring-Boot-Anwendung

Das Beispiel ist im Grunde eine relativ einfache Java-Web-Anwendung, die auf dem Spring-Boot-Framework basiert, eine MySQL-Datenbank zur Datenspeicherung verwendet und zwei Http-Services (die Bezeichnung „REST" wird in diesem Zusammenhang nicht verwendet) anbietet: Bei Aufruf der URL „/" werden der statische Text „Hello World" ausgegeben sowie unter „/todos" die in der Datenbank gespeicherten To-dos gelesen und zurückgeliefert *(siehe Abbildung 6)*.

Für die weiteren Betrachtungen sind zwei Eigenschaften des Projekts wesentlich: Es besteht aus mehreren Komponenten (der JVM, der MySQL-Datenbank und der Anwendung selbst) und per Default werden Spring-Boot-Projekte als startbares „Fat-Jar-Archive" verpackt, also als eine Datei, die die Anwendung selbst einschließlich aller Abhängigkeiten (inklusive Embedded Tomcat-Server, aber ohne JVM) enthält. Nachdem mithilfe von Maven *[12]* per „mvn clean package" das erwähnte Projekt-Artefakt erstellt wurde, kann es im Anschluss mit dem relativ einfachen Kommandozeilenaufruf „java -jar target/demo-helloworld-web-1.0-Local.jar" gestartet werden. Die Funktionalität der Anwendung testet man am einfachsten unter Verwendung von Kommandozeilen-orientierten Tools wie „curl" oder – noch besser – „httpie" *[13]* per *„http http://local-host:8080"*. *Listing 1* zeigt die entsprechende Ausgabe.

Der vollständige Sourcecode der Beispielanwendung steht auf Bitbucket *[14]* zur Verfügung. Da die Entwicklung einer Spring-Boot-Anwendung nicht im Fokus dieses Artikels ist, wird für weitere Details auf die sehr gute Dokumentation des Projekts verwiesen *[18]*.

Stufe 1: MySQL im Container

In den Betrachtungen zur Beispiel-Anwendung wurde bislang völlig außer Acht gelassen, wie und woher die notwendigen externen Bausteine wie die MySQL-Datenbank auf beziehungsweise in die Entwicklungsumgebung gelangen. Die klassische Variante der nativen Installation auf dem Entwicklungsrechner hat einen wesentlichen Nachteil: Mit der Zeit werden es immer mehr Tools und manchmal schließen sich verschiedene Versionen der gleichen Software auch gegenseitig aus.

Besser wäre es natürlich, wenn diese Tools separat und isoliert voneinander installiert und einfach rückstandslos entfernt werden könnten – genau das lässt sich mit Docker und Containern erreichen. Dazu erstellen wir als Erstes mittels „Docker Machine" einen lokalen Docker-Host und richten unsere Kommandozeile entsprechend ein, sodass der Docker-Client weiß, mit welchem Docker-Daemon er sprechen soll: „docker-machine create --driver=virtualbox demo" und „eval $(docker-machine env demo)".

Im Anschluss prüfen wir, ob wir uns mit dem Docker-Daemon der VM „demo" verbinden können, indem wir uns die vom Docker-Daemon gestarteten und verwalteten Prozesse (Container) anzeigen lassen. Diese Liste sollte unmittelbar nach der Installation leer sein *(siehe Listing 2)*.

Auch wenn wir im Moment nur einen Container erstellen, verwenden wir gleich „Docker Compose" *[09]*, das aufbauend auf der „Docker Engine" die Verknüpfung mehrerer Container auf einem Host erlaubt. Die notwendigen Informationen erhält „Docker Compose" aus einer speziellen Konfigurationsdatei im YAML-Format *[6, 19]*, die für unseren Anwendungsfall wie in *Listing 3* aussieht.

Mit „mysql" wird auf der obersten Hierarchieebene ein gleichnamiger Service definiert, der von einem oder mehreren Containern realisiert wird und auf dem Image „mysql:5.6.26" basiert, das automatisch aus der zentralen öffentlichen Docker-Image-Registry, dem „Docker Hub" *[20]* (vergleichbar mit Maven-Central), geladen wird. Mithilfe des Schlüsselworts „ports" wird der interne Port 3306 auf dem Host unter der gleichen Portnummer verfügbar gemacht und am Ende werden noch einige Umgebungsvariablen gesetzt, die die Autoren des Image für die Konfiguration vorgesehen haben *[11]*.

```
docker-compose \
  -f ./src/main/docker/s01-docker-
compose.yaml -p dmo stop
docker-compose \
  -f ./src/main/docker/s01-docker-
compose.yaml -p dmo rm
```

Listing 4

```
# Auszug aus s02-docker-compose.
yaml
app:
  image: java:8
  ports:
    - "8080:8080"
  environment:
    - SPRING_PROFILES_
ACTIVE=initdb
  working_dir: /opt/demo-hellow-
orld-web
  links:
    - mysql:mysql
  entrypoint: [
      "java",
      "-Djava.security.egd=file:/
dev/./urandom",
      "-jar",
"/opt/demo-helloworld-web/demo-
helloworld-web-1.0-Local.jar"
    ]
  volumes:
    - ../../../target:/opt/demo-
helloworld-web:ro
```

Listing 5

```
# Auszug aus application.yml
spring:
  datasource:
    # muss auf einer Zeile ste-
hen
    url: jdbc:mysql://
${MYSQL_PORT_3306_TCP_
ADDR:jsd2015.local}:
${MYSQL_PORT_3306_TCP_PORT:3306}/
${MYSQL_ENV_MYSQL_DATABASE:test}
    username: ${MYSQL_ENV_MYSQL_
USER:test}
    password: ${MYSQL_ENV_MYSQL_
PASSWORD:1234}
```

Listing 6

Ein einfacher Aufruf auf der Kommandozeile aus dem Startverzeichnis des Projekts erzeugt und startet („up") den Container als Daemon („-d"): „docker-compose \ -f ./src/main/docker/s01-docker-compose.yaml -p dmo up –d". Nach kurzer Wartezeit können wir mit den üblichen Tools wie der MySQL-Workbench und natürlich aus unserer Anwendung heraus auf die Datenbank zugreifen. Die dazu notwendige IP der „demo"-VM erfahren wir durch einen Aufruf von „Docker Machine": „docker-machine ip demo", 192.168.99.103. Wenn wir die MySQL-Datenbank nicht mehr benötigen, können wir sie sehr schnell und rückstandslos entfernen *(siehe Listing 4)*.

Das wir aber die MySQL-Datenbank im weiteren Verlauf noch verwenden wollen, müssen wir das nicht unbedingt ausprobieren. Falls doch: Auch nicht so schlimm. Einfach „Docker Compose" wieder mit „up -d" aufrufen und eine neue (leere) Datenbank wird erstellt.

Stufe 2: Java im Container

Nachdem wir die MySQL-Datenbank in einen Container verpackt haben, wollen wir das natürlich auch mit dem Baustein „JVM" vollziehen. Im Gegensatz zur Vorgehensweise aus Stufe 1, bei der wir den durch den Container angebotenen „Service" über einen Netzwerkzugriff benutzen, muss der Java-Prozess direkten Zugriff auf das „.jar"-File besitzen.

Da dieses Artefakt während der Entwicklung relativ häufig neu erstellt wird, sparen wir uns für den Moment das direkte Einpacken und stellen stattdessen das Ausgabeverzeichnis des Projektes virtuell im Container zur Verfügung. Dazu ergänzen wir die Konfigurationsdatei um einen weiteren Service namens „app" *(siehe Listing 5)*.

Ausgangspunkt ist das Docker-Image „java:8" *[21]*, dass die JVM enthält. Danach stellen wir den Port 8080 auch auf dem Host-Rechner zur Verfügung und verbinden den Service „mysql" unter dem gleichnamigen Alias. Dadurch werden zusätzliche Informationen als Umgebungsvariable bereitgestellt, die wir in der Konfiguration unseres Projektes verwenden können *(siehe Listing 6)*.

Das Schlüsselwort „entrypoint" definiert das beim Start des Containers ausgeführte Kommando, während „volumes" die bereits angesprochene Bereitstellung des „target"-Verzeichnisses vom Host in den Container beschreibt. Im Shell-Skript „s02.docker-build-and-run-dev.sh" im Startverzeichnis des Projekts *[14]* ist ein typischer Ablauf vom Bauen des Projekts durch Maven über das Erzeugen und Starten der eben definierten „Docker Composition", die Kontrolle der Log-Datei (Hinweis: Beenden der Log-Ausgabe mittels Ctrl-C) und das Testen der beiden URLs bis zum Stoppen der Container dokumentiert.

Stufe 3: Unpacked FatJar

In Projekten, die interpretierte Sprachen wie Python, JavaScript, HTML oder CSS verwenden, kann man mit diesem Stand schon halbwegs vernünftig arbeiten. Natürlich würde man in diesem Fall nicht das Verzeichnis mit den Kompilaten, sondern den Source-Tree im Container bereitstellen. Mit „vernünftig arbeiten" ist in erster Linie gemeint, dass durch Mappen Änderungen an den Sourcen zumindest theoretisch unmittelbar auch im Container sichtbar und damit testbar werden (in der Praxis gibt es einige etwas trickreichere Aufgabenstellungen, je nachdem, welcher Virtualisierer für die Docker-Host-VM zum Einsatz kommt).

Aufgrund der Art und Weise, wie die JVM Klassen lädt, macht eine analoge Arbeitsweise, die anstelle der Sourcen auf den entsprechenden Kompilaten aufbaut, weniger Freude, da relativ häufig, wenn auch nicht immer, die Anwendung zum Laden der Klassen neu gestartet werden muss. Eine Ausnahme bildet unter Umständen das Produkt „JRebel" der Firma ZeroTurnaround, jedoch wurde das im Rahmen der Arbeiten zu diesem Artikel nicht weiter untersucht.

Eine mögliche Realisierung für unser Beispielprojekt kann anhand des Shellskripts „s03.docker-build-and-run-dev.sh" und der „Docker Compose"-Konfigurationsdatei „s03-docker-compose.yaml" nachvollzogen werden. Dazu wird durch das Maven-Profil „unzipFatJar" der Inhalt des Fat-Jar-Files in ein Unterverzeichnis ausgepackt. Beim Bereitstellen der Kompilate im Container werden anstelle des Basis-Ausgabeverzeichnisses „target" die entsprechenden Unterverzeichnisse des ausgepackten Jar-Files und die des inkrementellen Eclipse-Compilers verlinkt,

```
# Änderungen an Sourcen durchführen und speichern
# Service "app" stoppen
docker-compose \
  -f ./src/main/docker/s03-docker-compose.yaml -p dmo stop app
# Service "app" starten
docker-compose \
  -f ./src/main/docker/s03-docker-compose.yaml \
  -p dmo start app
```

Listing 7

```
# Dockerfile für das Image "mapp/demo-helloworld-web04"
FROM java:8
MAINTAINER Bernd Fischer "bfischer@mindapproach.de"
ENV MODIFIED_AT 2015-09-26_1845

COPY demo-helloworld-web.jar /opt/demo-helloworld-web/
```

Listing 8

```
# bitte aus dem Startverzeichnis
# des Beispielprojektes aufrufen
docker build -t mapp/demo-hello-
world-web04:latest \
             -f $(pwd)/target/
workdir-docker/Dockerfile \
             $(pwd)/target/work-
dir-docker
```

Listing 9

```
app:
 image: mapp/demo-helloworld-
web04:latest
  ports:
    - "8080:8080"
  environment:
    - SPRING_PROFILES_
ACTIVE=initdb
  working_dir: /opt/demo-hello-
world-web/
  links:
    - mysql:mysql
  entrypoint: [
"java",
"-Djava.security.egd=file:/
dev/./urandom",
"-jar",
"/opt/demo-helloworld-web/demo-
helloworld-web.jar"
]
```

Listing 10

sodass Änderungen, auch an Java-Sourcen, nach einem Neustart des Containers zur Verfügung stehen *(siehe Listing 7)*.

Stufe 4: Docker-Image für die Produktion

Für die Installation von Test-/QS- und Produktiv-Umgebungen ist es nicht sinnvoll, das Fat-Jar-File und den Java-Container getrennt voneinander auszuliefern. Besser ist es, so wenig wie möglich Artefakte berücksichtigen zu müssen. Daher packen wir das Jar-File und JVM gemeinsam in ein Docker-Image, dessen Aufbau in einem sogenannten „Dockerfile" beschrieben wird *(siehe Listing 8)*.

Die einzig wichtigen Zeilen sind die erste und die letzte. Mit dem Schlüsselwort „FROM" drücken wir aus, dass das bereits bekannte Image „java:8" verwendet und durch das Kopieren des Fat-Jar-Files erweitert werden soll. Damit das Build des Docker-Image reibungslos funktioniert, speichern wir beide, das Dockerfile und das Jar-File, in einem gemeinsamen und ansonsten leeren Verzeichnis ab. Diese Funktionalität ist durch das Profil „buildDockerWorkDir" bereitgestellt. Im Anschluss verwenden wir „Docker Engine", um ein neues Image zu erstellen *(siehe Listing 9)*.

Um das neue Docker-Image in „Docker Compose" verwenden zu können, passen wir die Konfiguration wie in *Listing 10* an. An die Stelle des Docker-Image „java:8" ist unser eigenes getreten und die Bereitstellung von Artefakten, des Fat-Jar-Files, ist entfallen, da diese bereits im Image enthalten sind. Wer möchte, kann anhand der Skriptdatei „s04.docker-buildPlain-and-run.sh" wieder eine entsprechende Abfolge von Build- und Test-Schritten nachvollziehen.

Stufe 5: Docker-Image via Maven

In der letzten Stufe wollen wir unser Beispielprojekt noch etwas verschönern, indem wir das Erstellen des Docker-Image in den Maven–basierten, automatischen Build-Zyklus einbinden. Dazu brauchen wir als Erstes ein Maven-Plug-in. Roland Huss hat uns in dankenswerter Weise bereits die meiste Arbeit abgenommen, indem er nicht nur selbst ein nach Ansicht des Autors sehr gutes Maven-Docker-Plug-in geschrieben, sondern auch einen ziemlich objektiven Vergleich mit anderen Plug-ins veröffentlicht hat *[3, 22]*.

Die notwendige Erweiterung der Maven-Konfiguration findet sich im Profil „buildDockerImageWithMaven". Dabei wird im Gegensatz zur Stufe 4 kein separates Dockerfile verwendet, sondern alle Informationen direkt in der Plug-in-Konfiguration definiert. Optional lässt sich durch weitere Einstellungen detailliert steuern, welche Dateien in das Image gepackt werden. Ansonsten besteht der Unterschied zum Dockerfile aus Stufe 4 vor allem in der Anzahl der notwendigen Zeilen und den vielen spitzen Klammern.

Die „Docker Compose"-Konfiguration unterscheidet sich von ihrem Vorgänger nur im Namen des erstellten Docker-Image, für den wir zur Unterscheidung einen etwas anderen Namen wählten. Auch gilt ansonsten das Gleiche wie für Stufe 4: Anhand des Shellskripts „s05.docker-buildAllWithMaven-and-run.sh" können die verschiedenen Schritte zum Bauen des Fat-Jar-Files, das Erstellen des Docker-Image mit Maven sowie das Starten, Testen und Stoppen der „Docker-Composition" nachvollzogen werden.

Lessons Learned

Aus der beschriebenen „Dockerisierung" einer Java-Web-Anwendung kann aus Sicht des Autors die Schlussfolgerung gezogen werden, dass sich Docker vor allem für folgende Dinge eignet:

- Die Bereitstellung von Tools und Komponenten, wie in unserem Beispiel der MySQL-Datenbank
- Das Verpacken und das Deployment von Anwendungen
- Die relativ einfache Definition und Erstellung von Anwendungsumgebungen

Andere typische Aufgaben eines Entwicklers, speziell auf der Ebene von Unit- oder Module-Tests, bleiben im Wesentlichen der klassischen lokalen Arbeitsweise vorbehalten. Vor allem aufgrund der einfachen Definition von ganzen Anwendungslandschaften könnte der Einsatz von Docker auch außerhalb von Entwicklerarbeitsplätzen, etwa in CI-/CD-Pipelines, durchaus positive Effekte bringen.

Quellen und weiterführende Links

[1] J. Gray, A Conversation with Werner Vogels: Learning form the Amazon technology platform, 2006: https://queue.acm.org/detail.cfm?id=1142065

[2] Jez Humble, David Farley, Continuous Delivery: Reliable Software Releases Through Build, Test and Deployment Automation, 2010, Addison-Wesley

[3] Git Repository Docker-Maven-Plugin von Ronald Huss: https://github.com/rhuss/docker-maven-plugin

[4] Christoph Arnold, Michel Rode, Jan Sperling, Andreas Steil: KVM Best Practices, 2012, dpunkt.verlag, ISBN 978-3-86491-117-0

[5] Docker Homepage: https://www.docker.com

[6] Docker Documentation: https://docs.docker.com

[7] Docker Engine: https://github.com/docker/docker

[8] Docker Machine: https://github.com/docker/machine

[9] Docker Compose: https://github.com/docker/compose

[10] Docker Toolbox: https://github.com/docker/toolbox

[11] Docker-Repository MySql-Datenbank: https://hub.docker.com/_/mysql

[12] Homepage des Apache Maven Projektes: http://maven.apache.org

[13] Homepage des Projektes „httpie": http://httpie.org

[14] Git-Repository der Sourcen des Beispielprojekte: https://bitbucket.org/mindapproach/demo-helloworld-web

[15] Issue #1085 „Epic: Windows Support" im Docker-Compose-Projekt auf Github: https://github.com/docker/compose/issues/1085

[16] Diskussion zum Thema „How to install docker-compose on Windows" auf Stackoverflow: http://stackoverflow.com/questions/29289785/how-to-install-docker-compose-on-windows

[17] Brian Puglisi, „How to get Docker Toolbox (and Compose) working on Windows 10": http://brianpuglisi.com/how-to-get-docker-compose-working-on-windows-10/

[18] Spring Projekt, Dokumentationen, Tutorials und Guides: https://spring.io/docs

[19] YAML-Format: https://de.wikipedia.org/wiki/YAML

[20] Docker Hub: https://hub.docker.com

[21] Docker-Repository JVM: https://hub.docker.com/_/java

[22] Ronald Huss, Docker Maven Plugin Shootout: https://github.com/rhuss/shootout-docker-maven

Bernd Fischer

bfischer@mindapproach.de

Bernd Fischer beschäftigt sich seit seinem Studium der Elektrotechnik mit Software-Entwicklung und -Architektur und arbeitet heute als CTO bei der MindApproach GmbH in Dresden. Über Assembler, Fortran, Pascal, C/C++ kam er vor rund fünfzehn Jahren zur Progammiersprache Java und ist ihr seither weitestgehend treu geblieben. Durch seine langjährige Tätigkeit als Entwickler, Architekt und Projektleiter hat er positive wie negative Auswirkungen unterschiedlicher Herangehensweisen in der Software-Entwicklung hautnah kennengelernt und schätzt heute besonders die Vorzügen von DevOps und Continuous Delivery. Neben seiner hauptberuflichen Tätigkeit ist er in der JUG Saxony und seit Neuesten in der Docker-Dresden-Usergroup aktiv.

Infrastruktur-Deployment mit Ansible und Docker

Robert Reiz, VersionEye GmbH

Docker ist eine sehr leichtgewichtige Virtualisierungs-Technologie, mit der Software-Entwickler nicht nur einzelne Anwendungen, sondern einen kompletten Stack einrichten können. Dies vermeidet komplett Unterschiede zwischen verschiedenen Umgebungen. Die neue Technologie bringt aber auch neue Herausforderungen mit sich, etwa die Orchestrierung der einzelnen Docker-Container. Für dieses Problem gibt es bereits verschiedene Lösungen; eine der flexibelsten ist Ansible, ein modernes Tool für Konfigurationsmanagement. Dieser Artikel zeigt, wie VersionEye das Infrastruktur-Deployment mit Docker und Ansible auf der AWS-Cloud durchführt.

Vor 2.000 Jahren wurden Waren über das Mittelmeer transportiert, indem man diese einfach auf dem Schiff gestapelt hat, etwa Vasen, Porzellan und andere zerbrechliche Waren. Wenn ein Sturm aufkam, ging vieles zu Bruch und der Schaden war groß. Im Jahr 1956 erfand Malcom McLean dann den 40-Fuß-Container (ISO 668) und revolutionierte damit den globalen Handel. Heute werden zwei Drittel des globalen Handels über Container abgewickelt.

In der Logistik gibt es das Problem der Transport-Matrix. Man muss wissen, wie man welche Ware auf welchem Transportsystem/Lager platziert, ohne dass dabei die Ware beschädigt wird *(siehe Abbildung 1)*.

Der 40-Fuß-Container hat die Komplexität dieses Problems extrem reduziert. Seitdem muss man nur noch wissen, wie man seine Ware im Container platziert. Das Problem des Tranports und der Lagerung der Container ist bereits gelöst *(siehe Abbildung 2)*.

In der Software-Entwicklung gibt es ein ähnliches Problem. Auf der Y-Achse stehen jede Menge Technologien und auf der X-Achse unterschiedliche Umgebungen, wie Entwicklung (localhost), Test, Staging und Produktion. In der Regel sieht es so aus, dass zwar überall die gleiche Software läuft, allerdings in unterschiedlichen Versionen *(siehe Abbildung 3)*.

Wer kennt das nicht? Der Entwickler entwickelt unter Windows und arbeitet mit JDK 1.8.14-win32. Auf dem Testsystem läuft JDK 1.8.1 auf Linux 64bit, in der Produktion vielleicht noch JDK 1.7 auf Unix. Dann treten auf dem Produktivsystem unvorhergesehene Fehler auf und vom Entwickler hört man nur: „Auf meinem Rechner läuft alles".

Unterschiede in den Versionen führen oft zu Fehlern, die nur sehr schwer zu reproduzieren sind. Eine weitere Fehlerquelle sind die unterschiedlichen Implementierungen. JDK

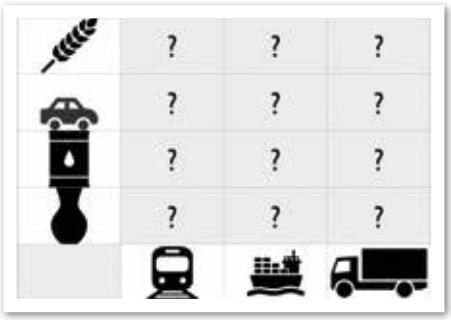


Abbildung 1: Das Problem der Logistik-Matrix ...

Abbildung 2: ... und deren Lösung ...

und MySQL sind zwar theoretisch auf allen Systemen gleich, in der Realität gibt es jedoch große Unterschiede. Das JDK muss für Windows mit einem ganz anderen Compiler und anderen Flags kompiliert werden als auf Linux/Unix. So gibt es auch Bugs im JDK, die nur unter Windows oder Linux auftauchen. Das Ganze macht die Verteilung von Software-Anwendungen nicht gerade einfach. Aber alle diese Probleme lassen sich mit Docker lösen *(siehe Abbildung 4)*.

Ähnlich wie in der Logistik bietet Docker die Möglichkeit, seine komplette Anwendung mit allen Abhängigkeiten in einen Container zu packen. Anstatt eine „.war"- oder „.ear"-Datei in eine Infrastruktur zu installieren, kann man mit Docker Teile der Infrastruktur selbst installieren. Anstatt die „.war"- oder „.ear"-Datei auf einem Application-Server zu installieren, liefert man einen Docker-Container aus, der bereits alles beinhaltet, um einen Service zur Verfügung zu stellen. Dazu gehören auch das JDK und ein leichtgewichtiger Application-Server. Ein Docker-Image/-Container beinhaltet sogar ein kleines Mini-Betriebssystem wie Ubuntu oder RedHat.

Build, Ship, Run

Die Idee hinter Docker ist, dass man einen Service/Prozess in einem Container bereitstellt. Dieser beinhaltet alles, was der Service/Prozess zur Laufzeit braucht. Solch ein Container wird einmal gebaut und in einem zentralen Repository abgelegt. Andere Server/Umgebungen können sich das Image von dort herunterladen und ausführen *(siehe Abbildung 5)*.

Der Vorteil gegenüber dem herkömmlichen Deployment auf einem Java-Applikations-Server besteht darin, dass der Entwickler mehr Kontrolle über die Abhängigkeiten hat und das JDK Teil des Deploy-Artefakts ist. Somit ist auf jeden Fall sichergestellt, dass auf allen Umgebungen die exakt gleiche Version des JDK läuft – nicht nur die exakt gleiche Version, sondern auch die exakt gleiche Implementierung.

Über Docker

Docker ist ein Open-Source-Projekt aus dem Jahre 2013 und basiert auf dem Linux-LXC-Interface. Ein Docker-Container hat seinen eigenen Namespace und läuft in einer „cgroup", teilt sich aber ansonsten die Ressourcen mit dem Host-System. Docker ist ein Prozess-Container mit erweiterten Funktionen und einem eigenen Ecosystem. Zurzeit funktioniert Docker nur auf Linux und nicht auf Windows. Microsoft hat allerdings angekündigt, Docker nativ auf Windows 10 zu unterstützen.

Das Projekt wurde von „dotCloud" ins Leben gerufen. Dahinter steht ein PaaS-Anbieter aus den USA und einer der Konkurrenten von Heroku. Die Macher von „dotCloud" starteten das Docker-Projekt, um die Infrastruktur für ihre PaaS-Lösung besser verwalten zu können. Doch schnell zeigte sich, dass das Interesse an Docker viel größer war als an „dotCloud" selbst. So kam es, dass sie „dotCloud and CloudControl" in Berlin verkauften, um sich voll und ganz auf das Docker-Projekt zu konzentrieren. Seit dem Jahr 2013 hat die Firma hinter Docker 162 Millionen Dollar Risikokapital für Entwicklung und Marketing eingesammelt. Mittlerweile gibt es ein ganzes Ökosystem um Docker herum.

Dockerfile

Das Dockerfile ist eine simple Textdatei, die ein Docker-Image beschreibt. Dies entspricht dem fest definierten Zustand eines Linux-Systems. *Listing 1* zeigt das Dockerfile für ein NGinx-Docker-Image; NGinx ist ein Webserver, ähnlich wie Apache HTTPD.

Abbildung 3: übertragen auf die Programmierung

Abbildung 4: ... und die Lösung mit Docker

```
FROM        ubuntu:14.10
MAINTAINER  Robert Reiz <reiz@versioneye.com>

ENV LANG en_US.UTF-8

RUN apt-get update
RUN apt-get install -y --force-yes -q nginx

ADD nginx.conf /etc/nginx/nginx.conf

CMD nginx

EXPOSE 80
```

Listing 1

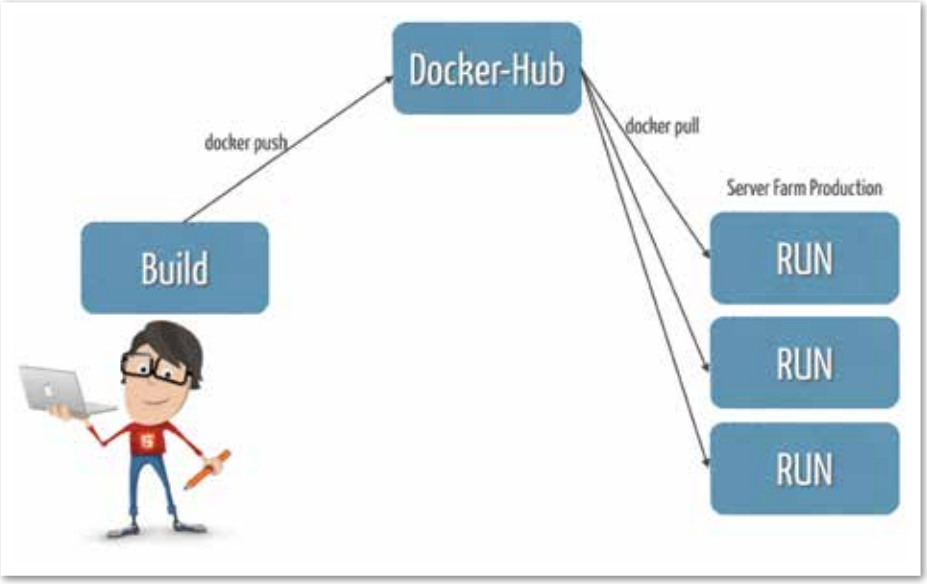


Abbildung 5: Build, Ship, Run

Jedes Dockerfile muss von einem Docker-Base-Image erben. Das Base-Image ist in der ersten Zeile mit „FROM" definiert. In diesem Beispiel ist das Base-Image Ubuntu 14.10. Das Schlüsselwort „MAINTAINER" beschreibt den Autor der Datei. Mit „ENV" können beliebig viele Umgebungsvariable gesetzt sein. „RUN" führt Befehle in der Shell aus. Im Beispiel wird das System aktualisiert und NGinx installiert.

Mit „ADD" können Dateien/Verzeichnisse vom Host-System in das Docker-Image hineinkopiert werden. Im Beispiel wird eine „nginx.conf"-Konfigurationsdatei vom Host in das Docker-Image nach „/etc/nginx/nginx.conf" kopiert. Der „CMD"-Befehl legt fest, welcher Prozess gestartet werden soll, wenn der Docker-Container ausgeführt wird. Im Beispiel ist es der „nginx"-Prozess. „EXPOSE" definiert, welche Ports zum Host geöffnet werden sollen, hier Port 80, der Standard-HTTP-Port.
Das Dockerfile in diesem Beispiel ist sehr simpel, aber das Prinzip wird klar. Das sehr einfache Vokabular kann auch wesentlich komplexere Docker-Images beschreiben.

Build, Push, Run

Das Docker-Image wird nun mit dem Befehl „docker build –t versioneye/nginx:1.0.0" gebaut. Dieser muss in dem Verzeichnis ausgeführt werden, in dem das Dockerfile liegt. Mit „docker build" wird generell ein Docker-Image gebaut. Mit der Option „-t" kann man dem Build einen Namen/Tag geben. In diesem Fall würde das Image „versioneye/nginx" heißen und hätte den Tag 1.0.0. Das Docker-Image liegt nun auf dem Host und hat einen definierten Zustand. Der Befehl „docker push versioneye/nginx:1.0.0" schiebt das gebaute Image in ein zentrales Docker-Repository.

Wenn nicht anders konfiguriert, schiebt Docker das Image nach „http://hub.docker.com", quasi ein GitHub für Docker-Images. Standardmäßig sind alle Repositories auf Docker-Hub public und kostenlos. Ab sieben Dollar im Monat kann man auch private Docker-Repositories dazubuchen. Docker-Hub ist komplett Open Source und kann natürlich auch als On-Premise-Lösung betrieben werden. Ein Server in der Produktionsumgebung könnte das eben gebaute Image mit „docker pull versioneye/nginx:1.0.0" herunterladen.
Das Image wird mit „docker run versioneye/nginx:1.0.0" gestartet. Der „run"-Befehl nimmt ein Image und startet davon einen laufenden Docker-Container. Beim Starten eines Docker-Containers wird immer der Prozess gestartet, der in dem Dockerfile mit dem Schlüsselwort „CMD" definiert ist, im Bespiel der „nginx"-Prozess.

Es ist recht einfach, ein Docker-Image zu bauen, in der Infrastruktur zu verteilen und zu starten. Der Vorteil liegt klar auf der Hand. Anstatt reinen Applikationscode zu bauen und zu verteilen, baut und verteilt man Docker-Images, die ausführbar sind und alle Abhängigkeiten bereits mitbringen. Die Hosts/Server in der Produktionsumgebung müssen bis auf den Docker-Daemon nichts vorinstalliert haben – kein JDK, kein Java Application Server und auch kein Maven oder Gradle.

Es muss also lediglich Docker auf den Servern installiert sein, alle anderen Prozesse laufen dann in Docker-Containern. Bei einem Deployment/Update wird einfach ein neues Docker-Image heruntergeladen und gestartet. Somit kann man auch bei jedem Deployment/Update etwa das JDK updaten. Das ist damit nur eine weitere Abhängigkeit im Docker-Image.

Volumes

Ein laufender Docker-Container speichert keinen Zustand. Wenn er beendet wird, sind alle Änderungen verloren, die während der Laufzeit erfolgt sind. Diese Eigenschaft ist extrem unvorteilhaft für Datenbanken, die im Container laufen sollen. Beim Starten eines Containers kann man aber Verzeichnisse und auch einzelne Dateien in den Container verbinden. So lassen sich Dateien über mehrere Docker-Container hinweg persistent halten. Im folgendem Beispiel werden mit „docker run –v /mnt/mongodb:/data –d versioneye/mongodb:3.6.6" MongoDBs im Container gestartet und mit „/mnt/mongodb" vom Host in den Container auf „/data" verbunden. MongoDB legt standardmäßig seine Daten unter "/data" ab. So stellt man sicher, dass im Container nur ein Prozess läuft, die Daten dauerhaft auf dem Host System gespeichert sind und nicht verloren gehen.

Networking

Bei VersionEye laufen seit Februar 2014 nicht nur die Java-Prozesse, sondern auch die Backend-Systeme wie MongoDB, ElasticSearch, Memcached und RabbitMQ als Docker-Container in der Infrastruktur. Beim Start eines Docker-Containers baut Docker ein internes Netzwerk auf und jeder laufende Container bekommt eine eindeutige IP-Adresse in diesem Netzwerk. Diese ist unabhängig von der IP-Adresse des Host-Systems.

Typischerweise besteht eine komplette Anwendung nicht nur aus einem Java-Prozess, sondern auch aus einer Datenbank. Wenn nun eine Java-Anwendung und eine Datenbank in zwei unterschiedlichen Containern laufen, muss der Java-Prozess die IP-Adresse des Datenbank-Containers kennen. Dafür gibt es verschiedene Möglichkeiten. Wenn alle Container auf dem gleichen Host laufen, kann man Container linken, wodurch automatisch Umgebungsvariablen injiziert werden, die IP-Adresse und Port der jeweilig anderen Container enthalten.
Sobald die Container auf unterschiedlichen Hosts laufen (wie bei VersionEye), dann muss man mit anderen Mitteln arbeiten. Für gewöhnlich sind die IP-Adressen von Backend-Systemen fix und ändern sich auch nicht so schnell. Wenn das der Fall ist, kann man die IP-Adressen beim Starten der Container über Umgebungsvariablen setzen, etwa mit „docker run --env DB_PORT_27017_TCP_ADDR=192.168.1.10 -d versioneye/crawlj:1.0.0". Beim Starten des „versioneye/crawlj"-Containers wird die Umgebungsvariable „DB_PORT_27017_TCP_ADDR" auf den Wert „192.168.1.10" gesetzt. Der Java-Prozess im Container kann beim Booten auf diese Umgebungsvariable zugreifen und weiß somit, wo sich die Datenbank befindet.

Ansible

Ansible ist ein Tool für Konfigurationsmanagement, ähnlich wie Chef, Puppet oder SaltStack. Allerdings ist Ansible „The new kid on the block" mit frischen Ansätzen. Die wichtigsten Unterschiede zu anderen Tools wie Chef und Puppet sind:

- Kein Master-Server erforderlich
- Auf den Servern muss kein Agent installiert sein
- Konfiguration/Scripting in Yaml
- Sehr einfach zu lernen

Mit Ansible kann man sehr einfach Server provisionieren. So konnte man mit Ansible auf zehn Servern parallel Docker installieren. Das Gute dabei ist, dass Ansible via SSH arbeitet. Man muss also keinen Ansible-Client auf den Servern installieren, um dort etwas mit Ansible installieren zu können. Die einzige Voraussetzung ist, sich via SSH auf den Server anzumelden.
In Ansible sind Anweisungen in sogenannten „Plays" und „Playbooks" organisiert. Ein Play hat immer einen Namen/eine Dokumentation sowie ein Modul, das die tatsächliche Arbeit ausführt. *Listing 2* zeigt ein Beispiel.

Die zwei „Plays" nutzen beide das „apt"-Modul, um einmal die Debian-Packet-Da-

```
- name: update debian packages
  apt: update_cache=true

- name: install lxc-docker
  apt: name=lxc-docker
state=present
```

Listing 2

```
- hosts: app_servers
  user: ubuntu
  sudo: true
  roles:
  - git
  - docker
```

Listing 3

tenbank zu aktualisieren und dann Docker zu installieren. Es gibt einige Tausend Ansible-Module für alle möglichen Situationen. Wenn man trotzdem kein passendes Modul findet, kann man einfach das „shell"-Modul nutzen und damit beliebige Shell-Komandos remote auf dem Server ausführen.

Ähnlich wie Maven gibt auch Ansible eine feste Verzeichnisstruktur vor, um Plays, Rollen, Playbooks und Inventroy zu organisieren. In der Regel werden Ansible-Anweisungen in Rollen organisiert und diese in Playbooks bestimmten Servern zugewiesen. So könnte man eine Rolle „docker" haben, um Docker zu installieren, und diese Rolle bestimmten Servern zuweisen. *Listing 3* zeigt ein sehr einfaches Playbook, um Git und Docker auf allen App-Servern zu installieren.

Dieses Playbook könnte „setup_docker.yml" heißen. Die Rollen sind Verzeichnisse, in denen mindestens eine „tasks/main.yml"-Datei liegen muss mit den Anweisungen für die Rolle. In der ersten Zeile wird auf „hosts: app_servers" verwiesen. Ansible hat eine Inventory-Datei, in der die IP-Adressen von Servern in Gruppen organisiert sind. Die dazugehörige Inventory-Datei könnte wie in *Listing 4* aussehen.

Ein Playbook kann mit „ansible-playbook setup_docker.yml" ausgeführt werden. Damit würde Ansible auf allen Servern in der „app_servers"-Gruppe sicherstellen, dass Git und Docker installiert sind. Wenn auf einem Server Docker schon installiert ist, dann wird es nicht erneut installiert. Ansible stellt nur sicher, dass ein bestimmter Zustand auf dem Server hergestellt ist.

```
[app_servers]
192.168.1.11
192.168.1.12
192.168.1.13

[db_servers]
192.168.1.14
192.168.1.15
```

Listing 4

Docker mit Ansible deployen

Ansible hat natürlich auch ein Docker-Modul, mit dem man Docker-Images herunterladen und Docker-Container starten kann. Die folgende Anweisung würde sicherstellen, dass der „versioneye/crawlj"-Container läuft *(siehe Listing 5)*.

```
- name: versioneye/crawlj container
  docker:
    name: crawlj
    image: versioneye/crawlj:1.0.0
    state: started
    env:
      DB_PORT_27017_TCP_ADDR: {{DB_IP}}
      DB_PORT_27017_TCP_PORT: {{DB_PORT}}
```

Listing 5

In den letzten zwei Zeilen werden die Umgebungsvariablen für die Datenbank gesetzt; wie man hier gut sehen kann, stehen dort keine festen Werte. Ansible selber hat auch ein Konzept für globale Variablen. Die IPs und Ports für die Backend-Systeme sind in einer zentralen Konfigurationsdatei hinterlegt und mit der Klammer-Notation kann man in allen Plays/Playbooks darauf zugreifen.

Bei VersionEye existieren unterschiedliche Ansible-Rollen für jedes Docker-Image. Es gibt für jedes Image eine Rolle, um es zu bauen und zu pushen, sowie unterschiedliche Rollen, um das Image in unterschiedlichen Umgebungen zu starten. So bestehen folgende Ansible-Rollen für das „crawlj"-Image

- docker_crawlj_build
- docker_crawlj_run_prod
- docker_crawlj_run_ent

```
- hosts: crawl_j
  user: ubuntu
  sudo: true
  roles:
  - docker_crawlj_build
  - docker_crawlj_run_prod
```

Listing 6

„prod" steht für „production" und „ent" für „Enterprise". Das Playbook, um „crawlj" auf „production" einzurichten, heißt „deploy_crawlj.yml" und sieht wie in *Listing 6* aus. Zuerst wird ein neues Image mit dem neuen Applikations-Code gebaut und gepusht. Die zweite Rolle „pullt" das neue Image auf die Server und startet es mit den Umgebungsvariablen für „production".

Fazit

Die Kombination Docker + Ansible hat sich bei VersionEye bewährt und läuft sehr stabil. Zurzeit läuft VersionEye auf fünfzehn Docker–Containern, verteilt auf zwölf EC2-Instanzen in der Amazon Cloud. Die exakt gleichen Docker-Images kommen übrigens auch auf der VersionEye Enterprise VM bei Kunden zum Einsatz. Zum Provisionieren der Enterprise VM wird natürlich auch Ansible benutzt, mit leicht unterschiedlichen Rollen. Docker und Ansible haben das Deployment sehr stark vereinfacht und vor allem Unterschiede zwischen Entwicklung, Test, Enterprise und Produktionsumgebung ausgemerzt.

Robert Reiz
reiz@versioneye.com

Robert Reiz schloss im Jahr 2005 als Dipl.-Informatiker an der HS Mannheim erfolgreich sein Studium ab. Er gründete im Jahr 2008 die PLOIN GmbH, ein Java-Beratungshaus, und verkaufte es im Jahr 2010 erfolgreich. Im Jahr 2011 wirkte er bei einem Startup in San Francisco mit und erweiterte seinen Horizont. Seit 2012 arbeitet er an VersionEye.

Good bye Swing – hello JavaFX

Christoph Rein und Stefan Kühnlein, OPITZ CONSULTING GmbH

Im Zeitalter von iOS und Android fordern Benutzer verstärkt optisch moderne und ansehnliche Anwendungen. Die Erwartungen gehen dabei von komplexen Farbverläufen, Transformationen (Drehungen und Zoom), animierten Übergängen bis hin zur flüssigen Bedienbarkeit von Applikationen per Touchscreen sowie Responsive Design. Solche Anforderungen können ältere Technologien nicht mehr erfüllen. Insbesondere bei der Entwicklung von Desktop-Anwendungen waren die Möglichkeiten mit Java zuletzt eher beschränkt. Das Abstract Window Toolkit (AWT) gilt bereits seit Längerem als veraltet und wird daher kaum noch verwendet. Alternativ wurden noch Swing und das Standard Widget Toolkit (SWT) eingesetzt, die mit ihrem Oberflächen-Design jedoch etwas verstaubt wirken.

Mit Einführung des Release 8 zur Entwicklung von Benutzeroberflächen soll JavaFX zum neuen Technologie-Standard erhoben werden. Da die Weiterentwicklung von Swing zur gleichen Zeit eingestellt wurde, löst die neue Framework-Version den Standard damit formal ab. Im Gegensatz zu Swing beherrscht JavaFX den Umgang mit modernen Features sehr gut. Damit ist diese Technologie aktuell die erste Wahl für die Entwicklung von neuen, modernen Desktop-Anwendungen. In vielen Fällen ist jedoch auch die Erweiterung bestehender Swing-Applikationen oder die Wiederverwendung von Swing-Komponenten in JavaFX eine bevorzugte Alternative. Mit einer Gegenüberstellung von Swing und JavaFX zeigt dieser Artikel die wichtigsten Vorteile des neuen Standards und Möglichkeiten zur Migration von Swing-Komponenten in JavaFX oder in die andere Richtung auf.

Grundgerüst und Aufbau

In Swing stellt der Frame das Standard-Fenster für eine GUI dar. Es wird durch die Klasse JFrame repräsentiert und erzeugt *(siehe Abbildung 1)*. Alle anderen Container und Komponenten sind darin aufgenommen. Den Einstiegspunkt eines Frames und der entsprechenden Container-Hierarchie bildet ein „RootPane", der in Form eines „JPanel" definiert wird und das Layout bestimmt. Wenn die Anwendung zusätzliche Layouts benötigt, müssen dafür weitere JPanels erstellt und dem Top-Level-Panel angefügt werden. Der RootPane enthält dann den sogenannten „LayeredPane", über den auf den „ContentPane" und die eigentlichen Elemente einer Swing-Anwendung zu-

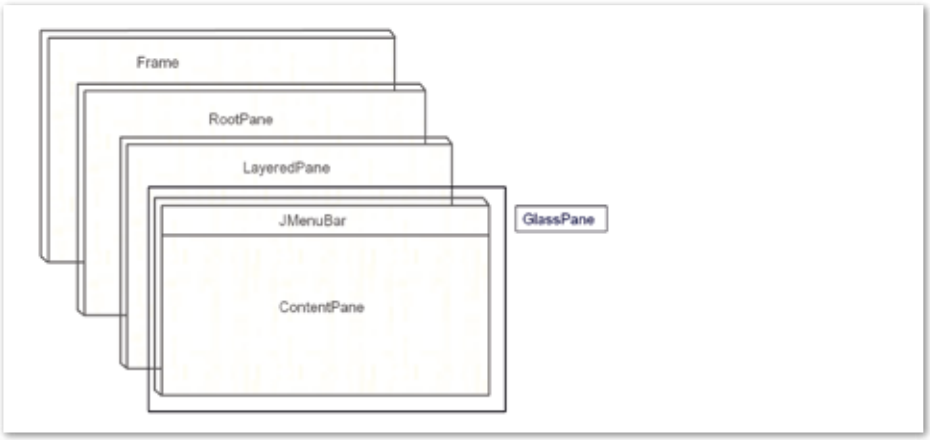


Abbildung 1: Swing Frame

gegriffen werden kann. Die oberste Schicht bildet der „GlassPane", mit dem eine unsichtbare Struktur über das GUI gelegt und so bestimmte Events abgefangen werden können. Der Zugriff erfolgt normalerweise nur auf den ContentPane und dessen Inhalt.

Im Gegensatz zu Swing ist die Grundstruktur einer JavaFX-Applikation an die eines Theaterstücks angelehnt. Als Rahmen steht eine „Stage" zur Verfügung, auf der die Anwendung gezeigt wird. Zu Beginn gibt es immer eine Stage in Form von Fenstern, aus der sich bei Bedarf weitere Stages öffnen lassen. Daneben existieren verschiedene „Scenes", die auf die Stage geladen werden können. Jede Scene enthält einen „Scene Graph" *(siehe Abbildung 2)*. Dieser bildet das Grundgerüst einer JavaFX-Anwendung und besteht aus genau einem Hauptknoten („Root Node") sowie mehreren Unterknoten („Leaf Node" oder „Branch Node"). Der Root Node kann einen oder mehrere Unterknoten enthalten, die entweder selbst Knoten aufnehmen („Branch Node") oder einen Endknoten darstellen können („Leaf Node").

Der „Scene Graph" ist also eine Sammlung aller Elemente, die die Benutzeroberfläche bilden. In JavaFX sind Layouts immer Unterklassen der Node-Klasse, sie enthalten also eine Reihe von Nodes. Die Art der enthaltenen Nodes ist dabei ebenfalls beliebig. Das Ergänzen von neuen Layouts gestaltet sich so um einiges flexibler als mit dem Hinzufügen von JPanels in Swing. *Listing 1* zeigt ein einfaches Beispiel einer Swing-Klasse, *Listing 2* dieselbe Klasse in JavaFX; die *Abbildung 3* und 4 stellen die entsprechenden Ergebnisse dar.

```
public class HelloWorldSwing
{
    public static void
main(String[] args)
    {
        JFrame frame = new
JFrame();
        frame.setTitle("Hello
Swing");
        JPanel panel = new JPan-
el();
        JButton button = new
JButton("Hello Swing");
        panel.add(button);
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Listing 1

Abbildung 3: „Hello World" mit JavaFX

Abbildung 4: „Hello World" mit Swing

Entwicklung von Oberflächen

Die GUI-Programmierung in Swing gestaltet sich gerade bei größeren Projekten sehr aufwändig und mühsam: Jede Komponente muss einzeln erzeugt und explizit angeordnet werden. Auf diese Weise entstehen sehr schnell komplexe, unübersichtliche und extrem schwer zu wartende Applikationen. Alternativen zu dieser Art der Oberflächen-Programmierung bieten deklarative Oberflächen. Verschiedene Open-Source-Lösungen wie SwiXml *[3]* bieten für diese Fälle entsprechende Ansätze an. Oracle hingegen hat keinen Standard entwickelt, der die deklarative Entwicklung von Benutzeroberflächen unterstützt.

Mit FXML bietet Java erstmals die Möglichkeit, grafische Komponenten schnell deklarativ und unabhängig von der Programmlogik zu definieren. Dabei handelt es sich um eine XML-basierte Definition zur Implementierung von deklarativen JavaFX-Oberflächen. Die Grundidee ist, dass eine FXML-Datei im Sinne des MVC-Patterns oder auch des MVVM-Patterns nur die UI beschreibt. Die Logik wird dann im Controller implementiert, der die entsprechenden UI-Elemente über die Annotation „@FXML" injiziert. Damit bietet Java eine Alternative zum klassischen API, in dem Knoten in Bäume angehängt werden.

Mit der XML-basierten Definition der Benutzeroberfläche ermöglicht FXML eine hierarchische Gliederung. Damit kann die Aufteilung einer GUI in Container und Kom-

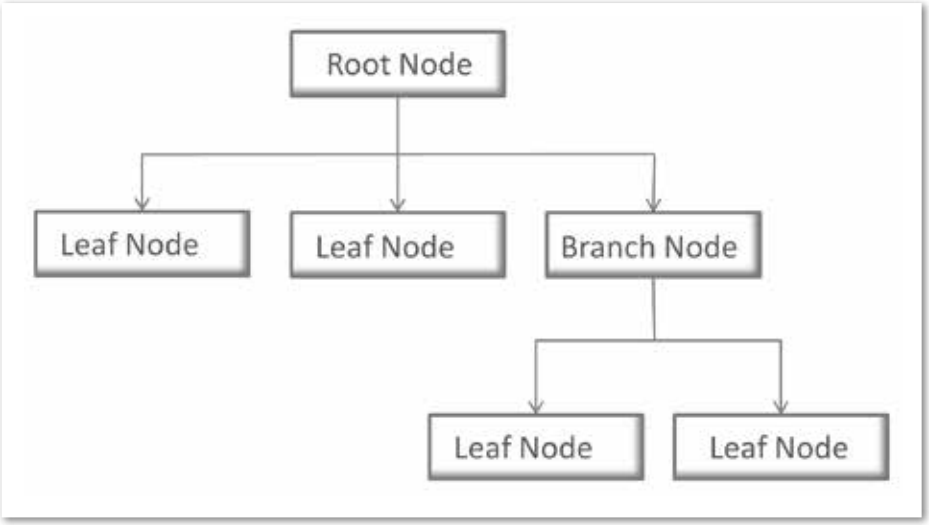


Abbildung 2: JavaFX-Scene-Graph

```
public class HelloWorldFx extends
Application {

   public static void
main(String[] args) {
        launch(args);
    }

   @Override
    public void start(Stage pri-
maryStage) throws IOException {
        primaryStage.
setTitle("JavaFX");

        Button button = new But-
ton();
        button.prefHeight(200);
        button.setText("Hello
Fx");
        StackPane root = new
StackPane();
        root.getChildren().
addAll(button);
        Scene scene = new
Scene(root, 300, 150);
        primaryStage.
setScene(scene);
        primaryStage.show();
    }
}
```

Listing 2

ponenten gut dargestellt werden. Für den Entwurf von Oberflächen existieren bereits Tools, mit denen die GUI-Container grafisch erstellt und im XML-Format abgespeichert sind. Das bekannteste Tool ist der „Scene Builder" von Gloun *[1]*. Für die Erstellung von GUI-Containern mit dem Scene Builder sind keine Java-Kenntnisse notwendig, sodass dieses Tool bereits während der Analyse-Phase eines Projekts zur Spezifikation von Desktop-Anwendungen verwendet werden kann. *Listing 3* zeigt eine einfache FXML-Datei zur Validierung von Benutzernamen und Passwort, die mithilfe des Scene Builder definiert wurde.

Ein Layout in JavaFX gestalten

Die individuelle Gestaltung von Oberflächen in Swing ist unter Umständen mit sehr viel Aufwand verbunden. Will man beispielsweise eine Hintergrund-Grafik einbinden, muss das JPanel dafür erst erweitert werden. Für die Definition einer Schriftart ist es notwendig, erst ein Font-Objekt anzulegen, das erst nach Beenden der Anwendung oder des JFrame aus dem Speicher gelöscht wird. Selbst wenn diese Hürden überwunden sind, ist man mit den Möglichkeiten von Swing recht eingeschränkt. Komplexe Farbverläufe oder einfache Effekte wie Schatten oder Reflektionen können entweder nur mit viel Aufwand oder gar nicht dargestellt werden. Anderen modernen Oberflächen-Frameworks hinkt Swing daher auch mit den neuesten Versionen hinterher.

Mit der Möglichkeit zur Einbindung von Cascading Style Sheets (CSS) bietet die JavaFX-Bibliothek ein weiteres sehr nützliches Feature. Das Aussehen der JavaFX-Anwendung kann damit flexibel und dynamisch verändert werden. Das Design kann entweder direkt im Code oder vollkommen unabhängig von der Logik in einer eigenen CSS-Datei gekapselt werden. Praktisch jedes Element einer Oberfläche ist mit CSS individuell definierbar; Styles für Elemente wie zum Beispiel eine Schriftart können jedoch auch einfach global definiert werden und sind so allgemein gültig. Ein unternehmensspezifisches Corporate Design kann damit beispielsweise einmal definiert und einfach in die jeweilige Scene integriert werden. Der Nutzer kann selbst entscheiden, welche CSS-Datei er der Scene hinzufügt, es ist aber ebenfalls möglich, eine Datei global für alle Scenes einzubinden.

```
<BorderPane xmlns="http://javafx.com/javafx/8.0.40" xmlns:fx="http://
javafx.com/fxml/1" fx:controller="application.ListDemoController">
   <center>
      <GridPane BorderPane.alignment="CENTER">
         <children>
            <Label text="Username:"></Label>
            <Label text="Password:" GridPane.rowIndex="1"></Label>
            <TextField GridPane.columnIndex="1"></TextField>
            <TextField GridPane.columnIndex="1" GridPane.rowIndex="1"></
TextField>
         </children>
      </GridPane>
   </center>
</BorderPane>
```

Listing 3

```
public class HelloSwingLayout extends JFrame {

  public HelloSwingLayout() {
     this.setTitle("Swing Layout");
     JPanel bgPanel = new JPanel() {

        public void paintComponent(Graphics g) {
           super.paintComponent(g);
           Image bgImage = null;

           try {
bgImage = ImageIO.read(new File("images/background.jpg"));
           } catch (IOException e) {
              e.printStackTrace();
           }
              g.drawImage(bgImage, 50, 50, this);
        }
     };
     bgPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
     Font font = new Font("Serif", Font.PLAIN, 24);
     JButton btn = new JButton("Hello Swing");
     btn.setFont(font);
     bgPanel.add(btn);
     this.add(bgPanel);
     this.setBounds(0, 0, 300, 150);
     this.setVisible(true);
  }
}
```

Listing 4

```
.root {
    -fx-font-size: 14pt;
    -fx-font-family: "Tahoma";
    -fx-background-image: url("background.jpg");
}

.button {
    -fx-text-fill: #0000ff;
    -fx-border-radius: 20;
    -fx-padding: 5;

}
```

Listing 5

Listing 4 zeigt eine einfache CSS-Datei, die ein Hintergrundbild einbindet sowie das Style einer Komponente anpasst. Mit „scene.getStylesheets().add ()" kann die Datei der jeweiligen Scene zugefügt werden. Um in Swing dasselbe Ergebnis zu bekommen, ist dagegen etwas mehr Aufwand nötig. *Listing 5* zeigt dazu, wie eine Grafik mithilfe von „paintComponent ()" und eines Image-Objekts eingebunden werden kann. Im Vergleich zu JavaFX ist dafür wesentlich mehr Code nötig. Für die Änderung der Schriftart

des Buttons muss, wie bereits erwähnt, erst ein Objekt erzeugt werden.

Properties und Bindings

Die Methoden und Konzepte in Swing sind für eine statische Kontrolle und Modifikation der Oberfläche ausgelegt. Moderne Benutzeranforderungen gehen jedoch dahin, Daten unmittelbar bei der Eingabe zu validieren und Änderungen sofort zu registrieren. In JavaFX wurden dafür die Konzepte der Properties und Bindings entwickelt und mit einigen nützlichen Features ausgestattet. JavaFX-Properties basieren dabei auf dem bekannten JavaBean Model, durch das bisher Properties eines Objekts repräsentiert wurden. Mit diesem Model wurden Namenskonventionen eingeführt, die für die Konsistenz über verschiedene Projekte hinweg sorgen.

Das JavaFX-API bietet für bestimmte Klassen bereits implementierte Properties. Es bietet eine Reihe von Property-Basisklassen für primitive Datentypen.

Bestimmte Klassen wie „javafx.scene.shape.Rectangle" verfügen bereits über Properties für „arcHeight", „arcWidth", „height" oder „width". Jede Property besitzt dabei standardmäßig eine „get()"/„set()"-Methode. Um über die Änderung einer Property informiert zu werden, kann zusätzlich ein „ChangeListener()" implementiert sein, der dazu sowohl den alten als auch den neuen Wert der Property ausgeben kann. Der Listener funktioniert dabei ähnlich wie der „PropertyChange Listener()" aus dem JavaBean Model.

Mit Bindings bringt JavaFX ein weiteres enorm mächtiges und einfaches Konzept mit, mit dem verschiedene Properties aneinander gebunden werden können. Eine „Property a" kann damit an eine „Property b" gebunden werden, sodass bei der Änderung von „a" die Property „b" mit dem neuen Wert von „a" aktualisiert wird. Werte können dabei mit „bind()" und „bindBidirectional()" sowohl in eine als auch in beide Richtungen gebunden werden. „unbind()"entfernt ein Binding zudem ganz einfach aus dem Code. *Listing 6* zeigt ein einfaches bidirektionales Binding zweier Slider. Dabei werden die Eigenschaften „valueProperty()" der beiden Slider aneinander gebunden, sodass bei Veränderung eines Werts der jeweils andere Wert ebenfalls verändert wird.

Neben solchen einfacheren Beispielen kann es auch vorkommen, dass das Standard-API nicht ausreichend ist. Für diesen Fall steht dem Entwickler das sogenannte „Low-Level-Binding" zur Verfügung. Damit wird eine Klasse erweitert und die Methode „computeValue()" überschrieben, um so den aktuellen Wert des Binding zurückzugeben. *Listing 7* zeigt dazu ein Beispiel mit einer angepassten Subklasse von „DoubleBinding()". Durch „super.bind()" werden die Abhängigkeiten an „DoubleBinding" übergeben. *[2]*

Migration von Swing- und JavaFX-Komponenten

Mit Einführung und wachsender Beliebtheit von JavaFX stellt sich die Frage, ob man bereits bestehende Swing-Applikationen mit JavaFX kombiniert. Dabei kann sowohl die Integration alter Swing-Anwendungen in ein neues JavaFX-Projekt gewünscht sein als auch eine Erweiterung von Swing durch neue Komponenten.

Die Gründe für eine Kombination aus beiden Technologien können unterschiedlich sein. Manchmal decken bestehende Swing-Anwendungen neuere Features nicht ab, in anderen Fällen kann es sinnvoll sein, komplexe Swing-Komponenten in eine neue JavaFX-Anwendung zu übertragen, statt diese neu zu implementieren. Auf diese Weise sinkt der Entwicklungsaufwand enorm.

Um Swing-Komponenten auch weiterhin nutzen zu können oder um alte Anwendungen noch zu erweitern, wurden zwei Erweiterungen eingeführt, die einerseits ermöglichen, JavaFX in Swing zu migrieren, und andererseits erlauben, Swing-Komponenten in JavaFX einzubetten.

JFXPanel

Für die Einbettung von JavaFX-Komponenten in Swing-Anwendungen wurde das JFXPanel entwickelt. Es verhält sich dabei grundsätzlich wie ein JPanel, kann aber eine JavaFX-Scene darstellen. Als Teil der Swing-Hierarchie kann das JFXPanel jedem JFrame hinzugefügt werden. Statt der Stage mit „Stage.setScene()" ein Scene-Objekt anzuhängen, lässt sich dem Panel mit „setScene()" eine JavaFX-Komponente hinzufügen; die Methode akzeptiert eine Instanz von JavaFX-Scene. Mit „getScene()" erhält man die Scene, die in dem JFXPanel enthalten ist. So können alle JavaFX-Komponenten in Swing dargestellt werden, inklusive ihrer Features.

Listing 8 zeigt ein einfaches Beispiel zur Einbindung von JavaFX-Code. Das entsprechende Ergebnis ist in der *Abbildung 5* dargestellt. Im oberen Teil wird dabei das JFXPanel

```
Slider slider1 = SliderBuilder.create().layoutX(50).layoutY(50).build();
Slider slider2 = SliderBuilder.create().layoutX(50).layoutY(100).build();

slider1.valueProperty().bindBidirectional(slider2.valueProperty());
```

Listing 6

```
public class Main {

    public static void main(String[] args) {

        final DoubleProperty a = new SimpleDoubleProperty(1);
        final DoubleProperty b = new SimpleDoubleProperty(2);
        final DoubleProperty c = new SimpleDoubleProperty(3);
        final DoubleProperty d = new SimpleDoubleProperty(4);

        DoubleBinding db = new DoubleBinding() {
             {
                super.bind(a, b, c, d);
            }
             @Override
            protected double computeValue() {
                return (a.get() * b.get()) + (c.get() * d.get());
            }
        };
        System.out.println(db.get());
        b.set(3);
        System.out.println(db.get());
    }
}
```

Listing 7

Abbildung 5: Swing-Fenster mit einem JavaFX-Control

erzeugt und dem JFrame hinzugefügt. In der Methode „createScene()" können sämtliche JavaFX-Komponenten wie gewohnt eingebunden werden. So ist auch die Nutzung von Bindings innerhalb eines JFrame ohne Weiteres möglich. Auch können die Komponenten mithilfe von CSS verändert sowie Bilder oder Media-Content eingebunden werden. Dafür wird in „run()" mit „getScene()" auf die aktuelle Scene zugegriffen und so eine CSS-Datei genutzt. Natürlich lässt sich auf diese Weise auch eine FXML-Datei einbinden, um so deklarative Oberflächen und damit verbunden auch Tools wie Scene Builder zu nutzen.

Ein „InputEvent" wird automatisch von Swing an JavaFX weitergegeben. Bei einer Änderung von JavaFX-Content übernimmt Swing diese ebenfalls. Beachten muss man dabei, dass beide Technologien auf unabhängigen Threads arbeiten. JavaFX arbeitet mit dem „FXApplicationThread", Swing mit dem „Event Dispatch Thread" (EDT).

Zwar beinhaltet die JavaFX-Bibliothek verschiedene Threads, der Application-Thread sticht jedoch dadurch hervor, dass auf ihm alle Events verarbeitet, Animationen ausgeführt sowie alle Knoten erstellt und verändert werden. Um bei der Integration eine Runtime-Exception zu vermeiden, müssen die beiden Threads synchronisiert werden.

Abbildung 6: JavaFX-Fenster mit einem Swing-Control

```
public class JFXPanelExample {
  private static void initAndShowGUI() {

      JFrame frame = new JFrame("JavaFX in Swing");
      final JFXPanel fxPanel = new JFXPanel();
      frame.add(fxPanel);
      frame.setSize(300, 200);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      Platform.runLater(new Runnable() {
         @Override
         public void run() {
            initFX(fxPanel);
             fxPanel.getScene().getStylesheets().add(getClass()
.getResource("application.css").toExternalForm());
         }
      });
  }

  private static void initFX(JFXPanel fxPanel) {
      // This method is invoked on the JavaFX thread
      Scene scene = createScene();

      fxPanel.setScene(scene);
  }
  private static Scene createScene() {
      Group root = new Group();
      Scene scene = new Scene(root, Color.ALICEBLUE);

      Button btn = new Button();
      btn.setText("JavaFX in Swing");
      btn.setLayoutX(25);
      btn.setLayoutY(10);

Slider slider1 = SliderBuilder.create().layoutX(25).layoutY(75).build();
Slider slider2 = SliderBuilder.create().layoutX(25).layoutY(125).build();

          slider1.valueProperty().bindBidirectional(slider2.valueProperty());

          root.getChildren().addAll(btn, slider1,e slider2);

      return (scene);
  }

  public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            initAndShowGUI();
         }
      });
  }
}
```

Listing 8

Durch die Erstellung einer JFXPanel-Instanz wird implizit die JavaFX-Runtime gestartet. Die Methode „initFX()" erstellt eine JavaFX-Scene auf dem JavaFX-Thread. Das Konstrukt „Platform.runLater (){ }" – eingebettet in die „initFX"-Methode – sorgt dafür, dass die Methode auf dem FX-Thread aufgerufen wird und nicht auf dem Standard-EDT-Thread.

SwingNode

Der vorherige Abschnitt hat gezeigt, wie JavaFX in eine bestehende Swing-Anwendung integriert werden kann. Die umgekehrte Integration ist ebenfalls möglich. Um Swing-Komponenten in JavaFX zu integrieren, gibt es die Klasse „SwingNode" mit den Methoden „setContent()" und „getContent()". Diese fügen dem SwingNode JComponents hinzu. Beide Methoden können dabei sowohl vom EDT als auch vom FX-Thread aufgerufen werden. Im Gegensatz dazu muss beim Zugriff auf eine Swing-Komponente auf die Synchronisierung der Threads geachtet werden. In diesem Fall muss das Setzen auf dem Swing EDT erfolgen, standardmäßig erfolgt der Aufruf in einer JavaFX-Anwendung auf dem FXApplication-Thread. *Listing 9*

```
public class SwingNodeExample extends Application {

         @Override
         public void start(Stage stage) {

          final SwingNode swingNode = new SwingNode();
              createAndSetSwingContent(swingNode);
              StackPane pane = new StackPane();
              pane.getChildren().add(swingNode);
              stage.setScene(new Scene(pane, 100, 50));
              stage.show();
         }

         private void createAndSetSwingContent(final SwingNode swingNode) {
              SwingUtilities.invokeLater(new Runnable() {
                  @Override
                  public void run() {

                    JButton button = new JButton("Hello JavaFX");
                    swingNode.setContent(button);

                  }
              });
         }

         public static void main(String[] args) {
              launch(args);
         }
     }
```

Listing 9

zeigt beispielhaft, wie ein „swing.Jbutton" in eine JavaFX-Anwendung integriert werden kann. Das zugehörige Ergebnis ist in der *Abbildung 6* dargestellt. Dabei wird eine SwingNode-Instanz erstellt, der dann ein Swing JButton zugefügt wird. Der Aufruf „SwingUtilities.invokeLater()" startet den EDT und erstellt einen neuen JButton. Um auf JavaFX-Elemente zugreifen zu können, muss wiederum „Platform.runLater(Runnable)" gestartet werden.

Eine Komponente wie der JButton oder ein komplexer JTable kann so in JavaFX weiterverwendet werden. Die ursprünglichen Eigenschaften eines Swing-Elements bleiben dabei erhalten. Auch das standardmäßige Swing-Design wird mit übernommen, wodurch optisch eine deutliche Abgrenzung zur moderneren JavaFX-Oberfläche entsteht

Fazit

Mit JavaFX steht Java-Entwicklern eine moderne Technik zur Verfügung, um grafisch ansprechende Applikationen zu erstellen. Eine Vielzahl weiterer Komponenten und Konzepte wie JavaFX-MediaSupport oder die Möglichkeit, HTML-Content in Java-Applikationen zu rendern, lassen es zu, moderne und stabile Anwendungen zu entwickeln. Durch deklarative Oberflächen können UIs einfach erstellt, verändert oder angepasst werden. Mit der Einbindung von CSS wird das Aussehen einer grafischen Darstellung zudem noch weiter individualisiert und optimiert. Mithilfe von Bindings lassen sich außerdem Werte auf einfache Weise voneinander abhängig machen.

Die Verbindung zu Swing schafft weiteren Spielraum im Hinblick auf bereits bestehende Anwendungen. Mit dem Einsatz von JavaFX in bestehende Swing-Applikationen ergeben sich Vorteile, wie die einfache und schnelle Umsetzung neuer Features und die zeitnahe Nutzbarkeit der Komponenten durch den Anwender. Jede Komponente kann dabei Schritt für Schritt ausgetauscht werden. Die Weiterverwendung bestehender komplexer Swing-Komponenten und der dadurch vermiedene Nachbau in JavaFX können den Entwicklungsaufwand senken.

Referenzen

[1] http://gluonhq.com/open-source/scene-builder
[2] https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm
[3] http://www.swixml.org

Christoph Rein
christoph.rein@opitz-consulting.com

Christoph Rein hat an der Universität Regensburg Wirtschaftsinformatik studiert und ist seit dem Jahr 2015 als Consultant für die OPITZ CONSULTING GmbH tätig. Dort beschäftigt er sich im Java-Umfeld speziell mit der Modernisierung von Benutzeroberflächen in JavaFX, unter anderem für einen großen Kunden aus Deutschland.

Stefan Kühnlein
stefan.kühnlein@opitz-consulting.com

Stefan Kühnlein ist seit Oktober 2013 als Solution Architect für die OPITZ CONSULTING GmbH tätig. Sein Schwerpunkt liegt im Entwurf von SOA-basierten Architekturen sowie in der Auswahl moderner Software-Architekturen und entsprechender technischer Frameworks. Aktuell unterstützt er gerade ein größeres Modernisierungsprojekt, dessen Frontend mit JavaFX entwickelt wird.

2000 Zeilen Java oder 50 Zeilen SQL?

Lukas Eder, Data Geekery GmbH

Wer sich in Java User Groups und bei Java-Konferenzen zum Thema SQL informiert, der kommt unweigerlich zu der Schlussfolgerung „Java-Entwickler können kein SQL" … und das kann man auch niemandem verübeln. Wer heutzutage mit Java programmiert, wird kaum mehr mit der SQL-Sprache konfrontiert.

Man muss sich mal vergegenwärtigen, dass wir Entwickler (oder unsere Kunden) jedes Jahr Beträge in Millionenhöhe an Firmen wie Oracle, Microsoft, IBM oder SAP für deren Datenbank-Systeme ausgeben – oft nur, um 90 Prozent der darin enthaltenen Features zu ignorieren. Dies nur für ein bisschen CRUD und ein bisschen ACID, das wir über ORMs wie Hibernate bekommen. Wir haben vergessen, warum diese Datenbanken so teuer sind. Wir haben gar nicht aufgepasst, was in den neueren SQL-Standards wie SQL:1999, SQL:2003, SQL:2008 oder SQL:2011 alles passiert ist. Standards, die allesamt von JPA bisher nicht berücksichtigt *[1]* wurden. Wenn Java-Entwickler nur wüssten, wie einfach und befriedigend es sein kann, Tausende Zeilen fehlerhaften Legacy Java Codes durch wenige Zeilen SQL zu ersetzen.

Der Erfolg der SQL-Sprache

Die Data Geekery GmbH spricht auf Java-Konferenzen über SQL im Allgemeinen, denn SQL und Java – insbesondere Java 8 – passen sehr gut zusammen, egal welches der folgenden APIs man verwendet:

- JDBC
- jOOQ
- Spring JDBC
- MyBatis
- Apache DbUtils
- JDBI
- Groovy SQL

Das Befriedigende an der Aufklärungsarbeit ist zu sehen, wie erstaunt die meisten Java-Entwickler sind, wenn sie erkennen, was mit SQL alles möglich ist *[2]*. Die Stärke von SQL lässt sich sehr gut am Beispiel eines sogenannten „Running Total" aufzeigen. Dazu ein Beispiel mit Kontobuchungsdaten in einer Bankapplikations-Datenbank *(siehe Listing 1)*. Man wird sofort feststellen, dass der Saldo bei jeder einzelnen Buchung fehlt. Er soll berechnet werden *(siehe Listing 2)*.

Angenommen, der aktuelle Saldo auf jedem Bankkonto ist bekannt. Dann kann man den „AMOUNT" von jeder Buchung verwenden und diesen vom aktuellen Saldo subtrahieren. Falls der aktuelle Saldo nicht bekannt ist, könnte man auch vom ursprünglichen Saldo („null") ausgehen und alle „AMOUNT"-Werte bis heute aufaddieren *(siehe Listing 3)*. Der Saldo auf jeder Buchung lässt sich also durch eine der beiden Formeln berechnen *(siehe Listing 4)*. So einfach entsteht ein „Running Total", fast wie in Microsoft Excel.

Die meisten würden wahrscheinlich eine kleine Java-Methode aus ihrem Ärmel schütteln, die alle Beträge im Memory behält. Wir würden Unit Tests schreiben, etliche rechnerische Bugs fixen, uns über den Typ BigDecimal ärgern etc. Dann würden wir plötzlich feststellen, dass die Saldoberechnungs-Funktion auch in einem uralten Perl-Script gebraucht wird, in dem wir in einem Batch-Job alle Kontoauszüge als PDF erstellen. Danach brauchen wir noch spezielle Berechnungsvarianten, wenn ein Bankmitarbeiter in seiner Administrationsansicht eigene Filter- und Sortierkriterien anwendet, die die Berechnung verkomplizieren.

```
| ID   | VALUE_DATE | AMOUNT |
|------|------------|--------|
| 9997 | 2014-03-18 |  99.17 |
| 9981 | 2014-03-16 |  71.44 |
| 9979 | 2014-03-16 | -94.60 |
| 9977 | 2014-03-16 |  -6.96 |
| 9971 | 2014-03-15 | -65.95 |
```

Listing 1

```
BALANCE(ROWn) = BALANCE(ROWn+1) +
AMOUNT(ROWn)
BALANCE(ROWn+1) = BALANCE(ROWn) -
AMOUNT(ROWn)
```

Listing 4

```
| ID   | VALUE_DATE | AMOUNT |  BALANCE |
|------|------------|--------|----------|
| 9997 | 2014-03-18 |  99.17 | 19985.81 |
| 9981 | 2014-03-16 |  71.44 | 19886.64 |
| 9979 | 2014-03-16 | -94.60 | 19815.20 |
| 9977 | 2014-03-16 |  -6.96 | 19909.80 |
| 9971 | 2014-03-15 | -65.95 | 19916.76 |
```

Listing 2

```
| ID   | VALUE_DATE | AMOUNT |  BALANCE |
|------|------------|--------|----------|
| 9997 | 2014-03-18 |  99.17 | 19985.81 |
| 9981 | 2014-03-16 | +71.44 |=19886.64 | n
| 9979 | 2014-03-16 | -94.60 |+19815.20 | n + 1
| 9977 | 2014-03-16 |  -6.96 | 19909.80 |
| 9971 | 2014-03-15 | -65.95 | 19916.76 |
```

Listing 3

Wenige von uns würden dieselbe Arbeit in PL/SQL, T-SQL oder einer anderen prozeduralen Sprache erledigen. Möglicherweise würden wir den Saldo direkt in der Datenbank festschreiben und bei jedem „INSERT" oder „UPDATE" von Buchungen für alle Buchungen neu berechnen.

Im Sinne dieses Artikels gibt es aber eine viel einfachere Lösung in SQL. Im Folgenden werden verschiedene, relativ einfache Lösungsansätze in SQL besprochen. Alle diese Beispiele sind als Skript *[3]* verfügbar und können in einer freien Oracle-XE-Instanz sofort ausgeführt werden.

Berechnung mit verschachteltem „SELECT"

Wer sich auf dem Wissensstand von SQL-92 *[4]* aus Uni-Zeiten bewegt, würde das „Running Total" eventuell mit einem verschachtelten „SELECT" berechnen – konkret mit einer „correlated subquery".

Nehmen wir mal an, wir haben eine View namens „V_TRANSACTIONS", die bereits die „ACCOUNTS"-Tabelle der „ACCOUNT_TRANSACTIONS"-Tabelle hinzufügt, um den „ACCOUNTS.CURRENT_BALANCE" zur Verfügung zu haben. *Listing 5* zeigt eine entsprechende Abfrage.

In diesem Beispiel verwenden wir sogenannte „Row Value Expression"-Prädikate *[5]*, um auf einfache Weise mehrere Werte gleichzeitig miteinander zu vergleichen – also „VALUE_DATE" und „ID" auf einen Schlag. Falls die Datenbank diese Form von Prädikaten nicht unterstützt und falls man dabei nicht jOOQ verwendet, um diese Prädikate zu emulieren *[6]*, dann kann man auch ein äquivalentes Prädikat schreiben *(siehe Listing 6)*.

Im Grunde genommen wird dabei einfach für jede einzelne Buchung ein verschachteltes „SELECT" ausgeführt, das die Summe aller „AMOUNT"-Werte von Buchungen berechnet, die neuer sind als die vorher genannte Buchung (siehe *Listing 7* und *Abbildung 1*).

Die Performance einer solchen Abfrage sieht leider gar nicht gut aus. In diesem Beispiel werden für das Prädikat „ACCOUNT_ID = 1" (in Zeile 9) 1.101 Werte mit einem effizienten „INDEX RANGE SCAN" herausgefiltert, also gar nicht viel. Trotzdem entstehen auf Zeile 3 mehr als eine Million Rows im Memory. Es scheint so, als ob der Algorithmus zum Berechnen dieser Summe von „O(n²)"-Komplexität, also ein sehr naiver Algorithmus ist. Noch viel schlimmer wirkt sich das auf den „A-Time"-Wert aus. Diese Abfrage dauert fast eine Sekunde, was nicht annähernd schnell ist *[7]*. Wahrscheinlich könnte man diese Abfrage tunen, aber generell riskiert man mit „correlated subqueries" immer wieder diese quadratische Komplexität. Versuchen wir also etwas anderes.

Berechnung mit rekursivem SQL

Niemand schreibt gerne rekursives SQL. Doch es funktioniert. Der Einfachheit halber nehmen wir an, dass wir zusätzlich zu unseren Buchungsdaten auch noch eine praktische „TRANSACTION_NR" haben, um die Transaktionen in Sortierreihenfolge bequemerweise aufzulisten. Dies wird uns helfen, die Rekursionskriterien zu vereinfachen. *Listing 8* zeigt das Resultat einer neuen View „V_TRANSACTIONS_BY_TIME". Bereit? Wie

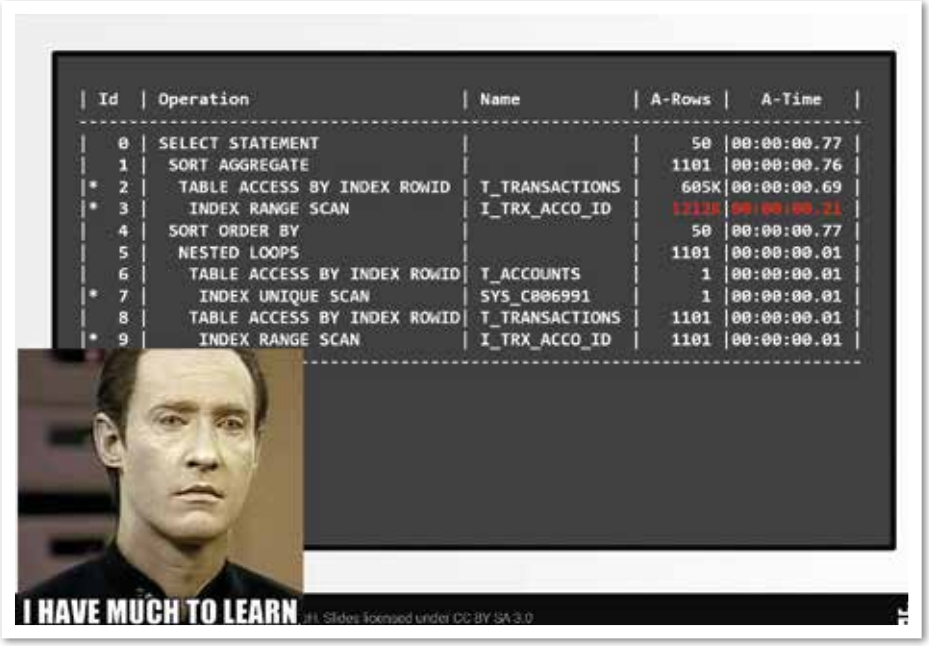


Abbildung 1: Ausführungsplan für verschachteltes „SELECT"

```
SELECT
  t1.*,
  t1.current_balance - (
    SELECT NVL(SUM(amount), 0)
    FROM v_transactions t2
    WHERE t2.account_id = t1.account_id
    AND  (t2.value_date, t2.id) >
         (t1.value_date, t1.id)
  ) AS balance
FROM     v_transactions t1
WHERE    t1.account_id = 1
ORDER BY t1.value_date DESC, t1.id DESC
```

Listing 5

```
SELECT
  t1.*,
  t1.current_balance - (
    SELECT NVL(SUM(amount), 0)
    FROM v_transactions t2
    WHERE t2.account_id = t1.account_id
    AND ((t2.value_date > t1.value_date) OR
         (t2.value_date = t1.value_date AND
          t2.id         > t1.id))
  ) AS balance
FROM     v_transactions t1
WHERE    t1.account_id = 1
ORDER BY t1.value_date DESC, t1.id DESC
```

Listing 6

ID	VALUE_DATE	AMOUNT	TRANSACTION_NR
9997	2014-03-18	99.17	1
9981	2014-03-16	71.44	2
9979	2014-03-16	-94.60	3
9977	2014-03-16	-6.96	4
9971	2014-03-15	-65.95	5

Listing 8

ID	VALUE_DATE	AMOUNT	BALANCE
9997	2014-03-18	-(99.17)	+19985.81
9981	2014-03-16	-(71.44)	19886.64
9979	2014-03-16	-(-94.60)	19815.20
9977	2014-03-16	-6.96	=19909.80
9971	2014-03-15	-65.95	19916.76

Listing 7

liest man nun den SQL-Code in *Listing 9*? Es ist eigentlich nicht schwierig. Eigentlich genommen verwenden wir hier einfach eine sogenannte „Common Table Expression ORDERED_WITH_BALANCE" (so etwas wie eine Tabellen-Variable), die rekursiv definiert ist *(siehe Listing 10)*.

Wie man sieht, wird im ersten Subselect des „UNION ALL"-Ausdrucks die „CURRENT_BALANCE" verwendet, und zwar nur für die allererste „TRANSACTION_NUMBER". Damit wird die Rekursion sozusagen initiiert. Im zweiten Subselect des „UNION ALL"-Ausdrucks wird die Differenz zwischen dem Saldo der vorherigen Buchung und dem Betrag der vorherigen Buchung berechnet *(siehe Listing 11)*.

Da es sich um eine Rekursion in den „ORDERED_WITH_BALANCE Common Table Expression" handelt, wird diese Berechnung so lange fortgeführt, bis die letzte Buchung erreicht ist. Auch diese Abfrage ist leider nicht performt. Sogar noch schlimmer, wir haben nun zu einem gewissen Zeitpunkt des Algorithmus elf Millionen A-Rows im Memory *(siehe Abbildung 2)*. Auch diese Abfrage lässt sich noch optimieren. Ein Teil des Problems entstand dadurch, dass die „TRANSACTION_NUMBER"-Hilfsspalte mit der Abfrage direkt berechnet wurde, statt sie im Vorfeld zu berechnen. Rekursives SELECT ist zwar möglich, aber oft schwierig.

Berechnen durch Fenster-Funktionen

Dimitri Fontaine aus der PostgreSQL Community schreibt in einem sehr lesenswerten Artikel über Fenster-Funktionen: „Es gibt SQL vor Fenster-Funktionen und SQL nach Fenster-Funktionen" *[8]*. *Listing 12* zeigt die wahrscheinlich beste Lösung für dieses Problem.

Im Grunde genommen geschieht hier genau dasselbe wie bei der Lösung mit dem verschachtelten SELECT. Man subtrahiert die Summe aller „AMOUNT" über ein Fenster von Rows, das folgende Kriterien erfüllt:

```
WITH ordered_with_balance (
  account_id, value_date, amount,
  balance, transaction_number
)
AS (
  SELECT t1.account_id, t1.value_date,
t1.amount,
         t1.current_balance,
t1.transaction_number
  FROM   v_transactions_by_time t1
  WHERE  t1.transaction_number = 1

  UNION ALL

  SELECT t1.account_id, t1.value_date,
t1.amount,
         t2.balance - t2.amount,
t1.transaction_number
  FROM   ordered_with_balance t2
  JOIN   v_transactions_by_time t1
  ON     t1.transaction_number =
         t2.transaction_number + 1
  AND    t1.account_id = t2.account_id
)
SELECT   *
FROM     ordered_with_balance
WHERE    account_id= 1
ORDER BY transaction_number ASC
```

Listing 9

```
SELECT
  t.*,
  t.current_balance - NVL(
    SUM(t.amount) OVER (
      PARTITION BY t.account_id
      ORDER BY     t.value_date DESC,
                   t.id         DESC
      ROWS BETWEEN UNBOUNDED PRECEDING
           AND     1         PRECEDING
    ),
  0) AS balance
FROM     v_transactions t
WHERE    t.account_id = 1
ORDER BY t.value_date DESC,
         t.id         DESC
```

Listing 12

```
WITH ordered_with_balance (
  account_id, value_date, amount,
  balance, transaction_number
)
AS (
  SELECT t1.account_id, t1.value_date, t1.amount,
         t1.current_balance, t1.transaction_number
  FROM   v_transactions_by_time t1
  WHERE  t1.transaction_number = 1

  UNION ALL

  SELECT t1.account_id, t1.value_date, t1.amount,
         t2.balance - t2.amount, t1.transaction_number
  FROM   ordered_with_balance t2
  JOIN   v_transactions_by_time t1
  ON     t1.transaction_number =
         t2.transaction_number + 1
  AND    t1.account_id = t2.account_id
)
SELECT   *
FROM     ordered_with_balance
WHERE    account_id= 1
ORDER BY transaction_number ASC
```

Listing 10

```
WITH ordered_with_balance (
  account_id, value_date, amount,
  balance, transaction_number
)
AS (
  SELECT t1.account_id, t1.value_date, t1.amount,
         t1.current_balance, t1.transaction_number
  FROM   v_transactions_by_time t1
  WHERE  t1.transaction_number = 1

  UNION ALL

  SELECT t1.account_id, t1.value_date, t1.amount,
         t2.balance - t2.amount, t1.transaction_number
  FROM   ordered_with_balance t2
  JOIN   v_transactions_by_time t1
  ON     t1.transaction_number =
         t2.transaction_number + 1
  AND    t1.account_id = t2.account_id
)
SELECT   *
FROM     ordered_with_balance
WHERE    account_id= 1
ORDER BY transaction_number ASC
```

Listing 11

Abbildung 2: Ausführungsplan für rekursives SQL

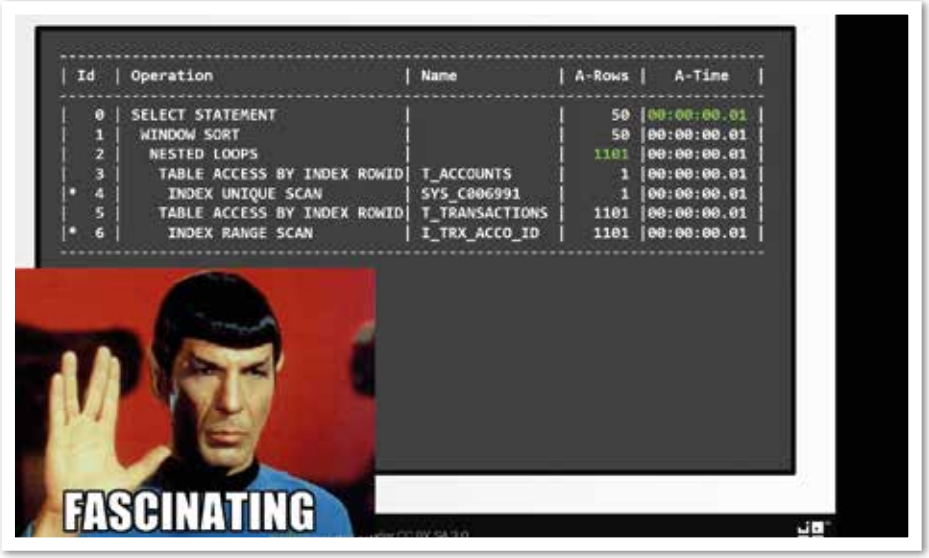


Abbildung 3: Ausführungsplan für Fenster-Funktionen

```
SELECT account_id, value_date, amount, balance
FROM (
  SELECT id, account_id, value_date, amount,
         current_balance AS balance
  FROM   v_transactions
) t
WHERE account_id = 1
MODEL
  PARTITION BY (account_id)
  DIMENSION BY (
    ROW_NUMBER() OVER (
      ORDER BY value_date DESC, id DESC
    ) AS rn
  )
  MEASURES (value_date, amount, balance)
  RULES (
    balance[rn > 1] = balance[cv(rn) - 1]
                    - amount [cv(rn) - 1]
  )
ORDER BY rn ASC
```

Listing 14

- Das Fenster beinhaltet alle „ROWS" in derselben „PARTITION" wie die aktuelle „ROW" (dieselbe „ACCOUNT_ID")
- Das Fenster ist nach demselben Sortierkriterium sortiert wie die Buchungen
- Das Fenster beschränkt sich auf alle „ROWS", die strikt vor der aktuellen „ROW" liegen

Listing 13 zeigt das Ergebnis. Diese Lösung ist nun performt *(siehe Abbildung 3)*. Schneller geht es in dieser Darstellung von Oracle-Execution-Plänen nicht. Über Fenster-Funktionen könnte man sich stundenlang unterhalten. Sie sind wahrscheinlich das am meisten unterschätzte und unterverwendete SQL-Feature *[9]*.

Berechnen mittels Oracle-MODEL-Klausel

Die Oracle-MODEL-Klausel ist als Sahnehäubchen für SQL-Nerds gedacht *(siehe Listing 14)*. Diese Abfrage wird folgendermaßen gelesen:

- *PARITION*
  Partitioniert nach der „ACCOUNT_ID" (wie bei den Fenster-Funktionen)
- *DIMENSION*
  Dimensioniert nach ihrer Sortier-Reihenfolge, also der Buchungs-Reihennummer
- *MEASURE*
  Liefert die berechneten Werte für „VALUE_DATE", „AMOUNT" und „BALANCE", wobei „VALUE_DATE" und „AMOUNT" direkt von den Ausgangsdaten genommen werden und unverändert bleiben
- *RULES*
  Definiert einen neuen Wert für „BALANCE" bei jeder Buchung, deren Buchungs-Reihennummer größer als „1" ist. Der Wert berechnet sich durch den „BALANCE"-Wert der vorherigen Buchungs-Reihennummer (CV = „Current Value")

Wem das trotzdem zu abstrakt ist, der betrachte doch *Abbildung 4*. Bei den RULES

```
| ID   | VALUE_DATE |  AMOUNT |  BALANCE |
|------|------------|---------|----------|
| 9997 | 2014-03-18 | -(99.17)|+19985.81 |
| 9981 | 2014-03-16 | -(71.44)| 19886.64 |
| 9979 | 2014-03-16 |-(-94.60)| 19815.20 |
| 9977 | 2014-03-16 |   -6.96 |=19909.80 |
| 9971 | 2014-03-15 |  -65.95 | 19916.76 |
```

Listing 13

Abbildung 5: Ausführungsplan für die MODEL-Klausel

handelt es sich um eine Berechnungsregel, ähnlich wie man diese auch bei Microsoft Excel oder bei anderen Spreadsheet-Lösungen angeben könnte. Dabei kann man mit Formeln auf beliebige andere Zell-Inhalte verweisen *(siehe Listing 14)*.

Jedes Mal, wenn man ein Problem hat, das man mit Excel lösen könnte, sollte man daran denken, dass sich dieses Problem auch mit der Oracle-MODEL-Klausel lösen lässt. Diese Lösung ist ziemlich performt *(siehe Abbildung 5)*. Wer noch nicht genug gesehen hat, kann sich noch einen anderen, sehr interessanten Use-Case für die MODEL-Klausel ansehen *[10]*. Auch das sehr informative, offizielle Oracle-Whitepaper zur MODEL-Klausel weitet den Horizont *[11]*.

Fazit

Die gezeigten Beispiele sind nur die Spitze des Eisbergs. Mit SQL lässt sich (fast) alles direkt in der Datenbank berechnen. Dabei gilt es natürlich immer zu beachten, ob dies im Rahmen der Architektur-Landschaft Sinn ergibt. Das ist viel öfter der Fall, als einem erfahrenen Java-EE-Architekten bisweilen lieb ist. Denn wenn die Datenmenge wächst, ist es viel einfacher und effizienter, einen Algorithmus in die Nähe der Daten zu schieben (zum Beispiel in Form von SQL oder Stored Procedures) als die Daten in die Nähe des Algorithmus.

Abbildung 4: Das erinnert an Microsoft Excel

Diese „Running Total"-Beispiele wurden im Zusammenhang mit einer mittelgroßen Schweizer E-Banking-Lösung evaluiert. Die entsprechende Bank erfreut sich einer Buchungs-Tabelle von mehreren Milliarden Rows. Die Ad-hoc-Saldoberechnung über Fenster-Funktionen stört die produktive Datenbank nicht weiter – ist aber sehr viel einfacher zu warten und anzuwenden als jede in Java implementierte Lösung.

Im Unternehmen des Autors heißt es: „SQL is a device whose mystery is only exceeded by its power". Manchmal wird auch gerne Winston Churchill zitiert, der gesagt haben soll: „SQL is the worst form of database querying, except for all the other forms". Tatsächlich – wie die MODEL-Klausel gezeigt hat, kann SQL schon ziemlich extrem sein.

Aber sobald man einige einfachere Tricks und Ausdrücke erlernt hat (hauptsächlich Fenster-Funktionen), wird man mit SQL sehr viel produktiver arbeiten als mit vielen anderen Technologien, insbesondere, wenn die Aufgabe rechnerischer Natur ist – egal ob über einfachere oder komplexere Datenmengen. Unbedingt dran denken: Der Execution Planner ist meist besser als man selbst im Eruieren eines effizienten Algorithmus. Also die Datenbank in Aktion setzen.

Weitere Informationen

[1] http://blog.jooq.org/2013/11/13/popular-orms-dont-do-sql
[2] http://www.jooq.org/java-8-and-sql, http://www.jooq.org/training
[3] https://github.com/jOOQ/Talks/blob/master/NoSQL-No-SQL/running-totals.sql
[4] http://www.andrew.cmu.edu/user/shadow/sql/sql1992.txt
[5] http://blog.jooq.org/2012/12/26/row-value-expressions-and-the-between-predicate
[6] http://www.jooq.org/doc/latest/manual/sql-building/conditional-expressions/comparison-predicate-degree-n
[7] http://blog.jooq.org/2013/08/12/10-more-common-mistakes-java-developers-make-when-writing-sql
[8] http://tapoueh.org/blog/2013/08/20-Window-Functions
[9] http://blog.jooq.org/2013/11/03/probably-the-coolest-sql-feature-window-functions
[10] http://stackoverflow.com/questions/12243488/oracle-sql-analytic-query-recursive-spread-sheet-like-running-total
[11] http://www.oracle.com/technetwork/middleware/bi-foundation/10gr1-twp-bi-dw-sqlmodel-131067.pdf

Lukas Eder
lukas.eder@datageekery.com

Lukas Eder ist Gründer und Geschäftsführer der Data Geekery GmbH, der Firma hinter jOOQ. Er ist aktiv an der Entwicklung von jOOQ beteiligt und berät Kunden zu verschiedenen Themen an der Java-und SQL-Schnittstelle sowie auch zur PL/SQL-Entwicklung.

Puppet für Entwickler

Sebastian Hempel, IT-Consulting Hempel

Kein Administrator wird heute einen Server manuell konfigurieren. Stattdessen wird die gewünschte Konfiguration mit Systemen wie Puppet, Chef oder Ansible „programmiert". Ein Programmaufruf bringt einen neuen Server in den gewünschten Zustand. Diese Art der Konfiguration ist nicht nur für den Betrieb, sondern auch für den Entwickler interessant, denn auch er möchte die Vorteile von schnell gebauten Systemen und reproduzierbaren Konfigurationen nutzen. Der Artikel stellt das Konfigurationsmanagement-System Puppet anhand von Beispielen aus der Welt der Software-Entwicklung vor und zeigt anschließend die Konfiguration von Diensten mit Puppet.

Puppet ist eine Software zur automatisierten Konfiguration von Servern und darauf laufenden Diensten. Luke Kanies hat sie im Jahr 2005 in Ruby programmiert. Er war mit dem damals bereits verfügbaren System „CFEngine" nicht zufrieden und programmierte daraufhin seine Version eines Konfigurations-Management-Tools. Puppet ist von Anfang an Open Source. Die verwendete Lizenz wurde im Jahr 2011 von der GPL auf die Apache-Lizenz geändert. Seit dieser Zeit gibt es eine kommerzielle Enterprise-Version von Puppet. Die Open-Source- und die Enterprise-Version unterscheiden sich nicht im Funktionsumfang.

Puppet wird primär zur Konfiguration von „unixoiden"-Systemen eingesetzt. Seit einiger Zeit können auch Systeme unter Windows mit Puppet verwaltet werden. Durch den grundsätzlich anderen Aufbau von Windows im Vergleich zu Linux, MacOS X & Co. lassen sich die Konfigurationen aber nicht untereinander austauschen. Nachfolgend ist die Konfiguration von Systemen unter Linux beschrieben. Von Puppet Labs werden Pakete für die folgenden Distributionen zur Verfügung gestellt:

- Red Hat, CentOS, Fedora
- Debian, Ubuntu

SUSE kümmert sich selbst um die Bereitstellung von Paketen. Für Arch Linux stellt die Community Pakete bereit.

Deklaration der Konfiguration

Puppet arbeitet deklarativ. Es werden keine Schritte und Operationen beschrieben, die zum Erreichen der Konfiguration durchgeführt werden müssen. Stattdessen wird mithilfe einer Domain Specific Language (DSL) der gewünschte Ziel-Zustand der Konfiguration beschrieben. Puppet ermittelt den aktuellen Ist-Zustand des Systems und vergleicht ihn mit dem definierten Soll-Zustand. Aus der Differenz werden die auszuführenden Operationen ermittelt und ausgeführt.

Puppet arbeitet dabei „idempotent". Auch bei mehrfacher Ausführung von Puppet werden keine Änderungen am System durchgeführt, wenn der definierte Soll-Zustand erreicht ist. Puppet führt nur die notwendigen Änderungen durch, um den Soll-Zustand zu erreichen. Dadurch werden bei jedem Lauf von Puppet auch alle in der Zwischenzeit manuell vorgenommenen Änderungen rückgängig gemacht.

Dateien und Konfigurationen, die nicht in Puppet beschrieben sind, werden von Puppet nicht beeinflusst. So lassen sich neben den mit Puppet verwalteten Diensten weiterhin herkömmlich konfigurierte Dienste betreiben.

Die Puppet-Architektur

Puppet besteht aus der zentralen Server-Komponente, dem Master und dem auf den zu konfigurierenden Systemen installierten Agent. Der Master wird auf einem eigenen Server im Netzwerk ausgeführt. Er kümmert sich um die Erstellung des Katalogs aus den Modulen und Manifesten. Die Manifeste beschreiben die gewünschte Konfiguration.

Der Agent wird auf allen zu konfigurierenden Systemen ausgeführt. Er konfiguriert das System mit den Informationen aus dem Katalog, die ihm der Master übermittelt hat. Der Katalog definiert den gewünschten Soll-Zustand, der auf dem System des Agent hergestellt werden soll *(siehe Abbildung 1)*.

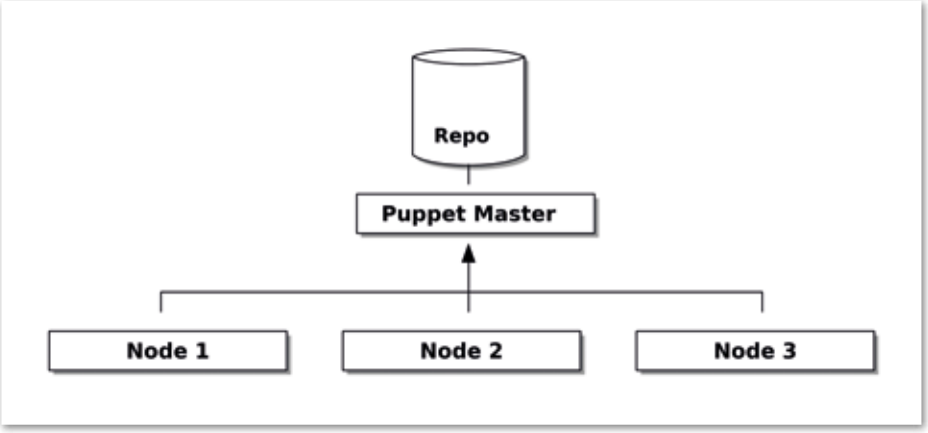


Abbildung 1: Die Puppet-Architektur

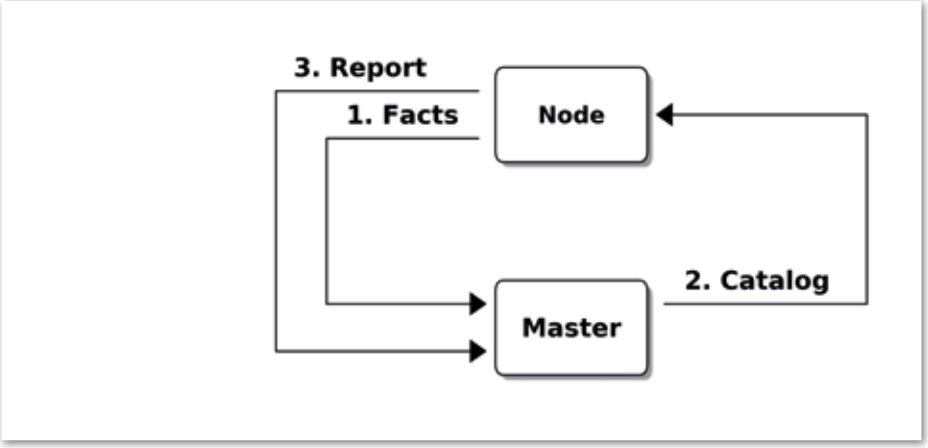


Abbildung 2: Schritte in einem Puppet-Lauf

Die Kommunikation zwischen Agent und Master wird immer vom Agent initiiert. Sie erfolgt über HTTPs. Zur Absicherung der Kommunikation sind sowohl der Agent als auch der Master mit Zertifikaten versehen. Die notwendige CA wird von Puppet selbst verwaltet. Ohne gültiges Zertifikat der Puppet CA erhält kein Agent eine Konfiguration.

Die in der DSL von Puppet definierte Konfiguration ist im Puppet-Repository gespeichert. Dabei handelt es sich um eine vorgegebene Verzeichnis-Struktur, in der die einzelnen Module und Manifeste gespeichert werden.

Anmerkung: Es besteht die Möglichkeit, Puppet auch ohne einen Master zu betreiben. Dann übernimmt der Agent auch die Erstellung des Katalogs. In diesem Fall muss der Agent Zugriff auf das Puppet-Repository haben.

Der Puppet-Lauf

Der Agent kann zeitgesteuert etwa alle 30 Minuten von selbst starten. Neben dem manuellen Start des Agent wird zur Steuerung der Ausführung oft auf Cron zurückgegriffen. Ein Puppet-Lauf unterteilt sich in die folgenden Schritte *(siehe Abbildung 2)*:

- *Facts ermitteln*
  Der Agent ermittelt den Wert/Zustand aller sogenannten „Facts". Dabei handelt es sich um Werte, die den aktuellen Zustand des Systems beschreiben. Es gibt Facts über die konfigurierten IP-Adressen, die Anzahl und Art der verfügbaren CPUs und Informationen über das Betriebssystem. Neben diesen vorgegebenen Facts können eigene Facts erstellt werden, um etwa den Stage des Systems (Entwicklung, Integration, Abnahme etc.) oder die Location des Servers zu ermitteln.
- *Facts an Master senden*
  Die gesammelten Facts werden durch einen „POST"-Request an den Puppet-Master übertragen. Dieser übermittelt die Facts an eventuell vorhandene weitere Systeme (wie Puppet DB, Foreman) und nutzt die Werte selbst zur Erstellung des Katalogs.
- *Katalog erstellen*
  Der Master erstellt aus allen dem Node zugeordneten Modulen den Katalog. Dieser beinhaltet alle definierten Ressourcen und deren Attribute. In den Modulen kann in Abhängigkeit von den ermittelten Facts die Konfiguration/der Katalog angepasst werden.
- *Katalog an Agent senden*
  Der vom Master erstellte Katalog wird als Antwort auf den „POST"-Request an den Agent/Client zurückgeliefert. Ist es bei der Erstellung des Katalogs zu einem Fehler gekommen, wird stattdessen ein HTTP-Status-Code „400" an den Agent übermittelt.
- *Katalog anwenden*
  Der Agent wendet den vom Master empfangenen Katalog auf das System an. Er vergleicht die im Katalog definierten Ressourcen und deren Attribute mit den zugehörigen Ressourcen auf dem System und führt eventuell notwendige Änderungen durch.
- *Report an Master senden*
  Bei der Anwendung des Katalogs wird ein Report erstellt und dieser vom Agent als letzte Aktion wieder durch einen „POST"-Request an den Master übermittelt.

Die Puppet-Language

Mit der Puppet-Language wird die Konfiguration beschrieben. Das zentrale Element ist die „Ressource", die kleinste zu definierende Einheit einer Konfiguration. Als Ressource wird beispielsweise eine Datei, ein Benutzer oder ein Dienst definiert. Jede Ressource ist mit einem eindeutigen Namen und ein oder mehreren Attributen belegt. Über die Attribute wird die Ressource genauer spezifiziert.

Ein Beispiel für eine File-Ressource: Die Datei „/etc/default/jenkins" soll auf dem System vorhanden sein („ensure present"). Sie hat als Besitzer den Benutzer „root" und ist der Gruppe „root" zugeordnet. Die Zugriffsrechte sind auf „0644" gesetzt. Die Datei soll den Inhalt der für das Modul Jenkins hinterlegten Datei „etc/default/jenkins" haben *(siehe Listing 1)*.

In Abhängigkeit von Facts oder selbst ermittelten Werten können einzelne Ressourcen zum Katalog hinzugefügt oder von diesem entfernt werden. So werden in *Listing 2* die Ressourcen nur dem Katalog hinzugefügt, wenn das Betriebssystem des Systems „Debian" ist.

```
file { '/etc/default/jenkins':
  ensure => present,
  source => 'puppet:///modules/
jenkins/etc/default/jenkins',
  owner  => 'root',
  group  => 'root',
  mode   => '0644',
}
```

Listing 1

```
if $::os['name'] == 'Debian' {
  ...
}
```

Listing 2

```
node buildserver.example.org {
  include 'java'
  include 'jenkins'
}

node /server[1-3].example.org/ {
  include 'jboss'
}
```

Listing 3

Neben „Conditionals" zur bedingten Definition von Ressourcen legt die Puppet-Language verschiedene Elemente zur Kombination von Ressourcen zu umfangreicheren Elementen fest. Sind mehrere Ressourcen zu einer „Class" zusammengefasst, können diese an einem Stück einem System zugewiesen werden. Einem System kann dabei immer nur eine Instanz einer Klasse zugeordnet sein.

Eine weitere Möglichkeit zur Zusammenfassung von Ressourcen ist ein „Define". Im Gegensatz zu einer Klasse können einem System mehrere Instanzen davon zugeordnet sein. Klassen konfigurieren somit nur einmal auf dem System vorhandene Systeme wie beispielsweise den Apache-HTTPD-Daemon. So sind mit „Define" mehrfach vorhandene Elemente wie virtuelle Hosts eines Apache-HTTPD beschrieben.

Module können über die Puppet-Forge geteilt werden. Damit steht eine Sammlung von vorgefertigten Konfigurationen zur Verfügung und das Rad muss nicht immer neu erfunden werden. Man kann die Forge mit einer Sammlung von Bibliotheken vergleichen.

Neben der Beschreibung der Konfiguration werden mithilfe der Puppet-Language auch die Zuordnungen von Klassen oder Modulen zu den zu konfigurierenden Systemen definiert. Die Systeme werden durch einen Node identifiziert. Neben dem Namen des Systems können hier reguläre Ausdrücke verwendet werden, um die gleiche Konfiguration mehreren Systemen zuzuordnen. Dem Node werden dann durch die Funktion „include" die Module zugeordnet *(siehe Listing 3)*.

Mit der ersten Node-Definition werden dem System „buildserver.example.org" die beiden Module „Java" und „Jenkins" zugeordnet. Die zweite Definition umfasst die Server „server1.example.org", „server2.example.org" und „server3.example.org", denen das Modul „Jboss" zugeordnet ist.

Trennung von Code und Daten

Die Elemente der Puppet-Language können so gut wie alle Konfigurationen beschreiben. Ohne weitere Hilfe wird jedoch mit der Sprache das „Wie" (der Code) mit dem „Was" (den zugehörigen Daten) vermischt:

- *Code*
  Aus welchen Elementen besteht eine Konfiguration? Welche Pakete, Dateien und Dienste sind zu verwalten?
- *Daten*
  Mit welchen Konfigurationswerten muss in einer Konfigurationsdatei gearbeitet werden? Welche Werte sind einzutragen?

Die hierarchische Datenbank „Hiera" lagert in Puppet die Konfigurationswerte beispielsweise in „YAML"-Dateien aus. Durch verschiedene Hierarchien können die Werte für einen Rechner, eine Rechner-Gruppe oder ein ganzen Netzwerk gültig sein. Änderungen am Wert einer Konfiguration bedingen nicht mehr eine Änderung des Codes und umgekehrt.

Puppet praktisch einsetzen

Nach dieser Einführung in Puppet werden einige Beispiele für Konfigurationen gegeben. Sie zeigen, wie Puppet ein Team bei der Entwicklung von Software unterstützen kann. Soweit möglich und sinnvoll, wird für die gezeigten Konfigurationen auf fertige Module aus der Puppet-Forge zurückgegriffen. In der Forge sind inzwischen viele Module in hervorragender Qualität verfügbar. Der Aufwand zur Einrichtung der gewünschten Software ist damit sehr gering.

Kern der Infrastruktur eines jeden Software-Entwicklungs-Projekts ist der Build-Server. Dieser kompiliert den von den Entwicklern erstellten Quellcode, testet ihn und prüft ihn mit verschiedenen Verfahren. Häufig wird hierzu der Continuous-Integration-Server Jenkins verwendet. Dieser benötigt neben einer installierten Java-Runtime noch weitere Programme. Puppet kann eine entsprechende Konfiguration erstellen und verwalten.

JDK

Eine der Grundlagen für den Build-Server ist ein installiertes JDK. Das OpenJDK kann auf allen Linux-Systemen direkt aus den Repository der Distribution installiert werden. Das Oracle JDK darf seit einer Änderung der „Operating System Distributor License for Java" (siehe *„http://www.golem.de/1108/86052.html")* nicht mehr von den Distributionen umpaketiert werden. Dieses JDK muss entweder direkt von der Oracle-Homepage heruntergeladen oder in einem lokalen Repository zur Verfügung gestellt werden.

Puppet bietet mit der Ressource „file" die Möglichkeit, Dateien auf den Systemen zu kopieren und diese hierzu auf dem Puppet-Master zur Verfügung zu stellen. Der in den Puppet-Master integrierte Datei-Server ist aber nicht für umfangreiche Dateien ausgelegt. Somit wird nicht empfohlen, umfangreiche Dateien wie das JDK von Java über diesen Weg zu verteilen.

Die Konfiguration des JDK erfolgt am einfachsten mit dem Java-Modul von Puppet Labs „puppetlabs-java" aus der Forge per „include 'java'". Über Hiera wird das Modul konfiguriert. In unserem Fall installieren wir das JDK der Distribution (OpenJDK) per „java::distribution: 'jdk'".

Build-Tool

Das Build-Tool (in diesem Falle Maven) kann ebenfalls mit Puppet verwaltet werden. In vielen Distributionen ist eine (eventuell alte) Version von Maven zur Installation verfügbar. Mithilfe des Puppet-Moduls „maestrodev-maven" kann neben der Installation auch die Konfiguration der „settings.xml" erfolgen *(siehe Listing 4)*.

Die Konfiguration der Mirrors und Servers erfolgt gemäß der Trennung von Code und Daten in Hiera. Die folgenden Einträge in Hiera konfigurieren in der „settings.xml" einen Proxy-Server für das Central-Repository. Zudem werden die beiden (lokalen) Repositories „Snapshots" und „Releases" definiert *(siehe Listing 5)*.

Version Control System

Die Installation und Konfiguration von Git als Version Control System erfolgt direkt ohne die Verwendung eines Moduls aus der Forge. Die Ressourcen sorgen dafür, dass Git aus dem Repository der Distribution instal-

```
include 'maven'
maven::settings { 'global-settings':
  mirrors => hiera('maven::mirrors'),
  servers => hiera('maven::servers'),
}
```

Listing 4

```
---
maven::maven::version: '3.2.5'
maven::mirrors:
  - 'central':
    id: 'central'
    url: 'https://local.nexus.example.com/nexus/content/groups/public/'
    mirrorOf: 'central'
maven::servers:
  - 'snapshots':
    id: 'snapshots'
    url: 'https://loca.nexus.example.com/nexus/content/repositories/snapshots'
    active: 'true'
    username: 'deployment'
    password: 'geheim'
  - 'release':
    id: 'releases'
    url: 'https://loca.nexus.example.com/nexus/content/repositories/releases'
    active: 'true'
    username: 'deployment'
    password: 'nochmehrgeheim'
```

Listing 5

```
package { 'git':
  ensure => installed,
}

file { '/var/lib/jenkins/.gitconfig':
  ensure  => file,
  content => template(git/gitconfig.erb),
  owner   => 'jenkins',
  groupe  => 'jenkins',
  mode    => '0644',
}
```

Listing 6

```
[user]
    name = Build Server
    email = jenkins@<%= @fqdn %>
[push]
    default = current
```

Listing 7

```
i'nclude 'jenkins'

---
jenkins::install_java: false
jenkins::lts: true
jenkins::config_hash:
  JENKINS_HOME:
    value: '/srv/jenkins'
jenkins::plugin_hash:
  git:
    version: '2.3.4'
  credentials:
    version: '1.20'
  ssh-credentials:
    version: '1.10'
  git-client:
    version: '1.15.0'
  scm-api:
    version: '0.2'
  matrix-project:
    version: '1.4'
  mailer:
    version: '1.11
```

Listing 8

liert wird („package"-Ressource). Im Home-Directory von Jenkins sind die wichtigsten Einstellungen für Git in der Datei „.gitconfig" („file"-Ressource) abgelegt *(siehe Listing 6)*.

Der Inhalt der Konfigurationsdatei wird bei der Erstellung des Katalogs aus dem Template „gitconfig.erb" ermittelt. Mit Templates vom Typ „erb" („embedded Ruby") kann der Inhalt einer Datei aus festem Text und Platzhaltern für Werte erstellt werden. Die Werte können aus Variablen der Puppet-Language, aus „Facts" oder aus Hiera stammen. Im Beispiel in *Listing 7* wird als Domain-Part der E-Mail-Adresse der Fact „fqdn" („fully qualified domain name") des Node eingesetzt.

Jenkins

Für die abschließende Konfiguration des CI-Servers Jenkins dient das Modul „rtyler-jenkins" in der Forge. Der Status eines „Puppet Approved"-Moduls stellt sicher, dass sich der Autor des Moduls an alle Konventionen von Puppet-Labs gehalten hat.

Die gesamte Konfiguration der von Jenkins benötigten Plug-ins erfolgt bei diesem Modul in Hiera. Das Modul kümmert sich aber auch um die Installation von Jenkins selbst. In unserem Fall deaktiviert man die Installation von Java, da man sie ja bereits selbst vorgenommen hat. Darüber hinaus fällt die Entscheidung für die Installation der LTS-Version von Jenkins.

Neben den gewünschten Plug-ins sind in der derzeitigen Version des Moduls auch die Abhängigkeiten eines Moduls aufzuführen. Das Update der Plug-ins muss ebenfalls über Puppet erfolgen. Zwar lassen sich über die in Jenkins integrierte Update-Funktion die Module aktualisieren. Beim nächsten Puppet-Lauf werden aber wieder die in Hiera eingetragenen Versionen der Plug-ins installiert *(siehe Listing 8)*.

Weitere Server

Neben den Kernbestandteilen eines Build-Servers können weitere Komponenten wie SonarQube oder Gerrit mit Puppet konfiguriert werden. Es bietet sich auch an, nicht nur den Build-Server, sondern auch alle anderen Server des Teams mit Puppet zu konfigurieren.

Die vom Team erstellten Artefakte werden auf dem Testsystem eingerichtet und dort automatisiert und/oder manuell getestet. Puppet ist ideal dafür geeignet, dieses System zu konfigurieren. Wird ein Server neu gebaut, wird mit einem Puppet-Lauf das System in den definierten Zustand versetzt. Damit ist es nur ein kleiner Schritt, die für die Testsysteme genutzte Konfiguration auch auf alle weiteren Stages bis hin zur Produktion anzuwenden. Wenn der Betrieb („Ops") neben dem Artefakt auch die Konfiguration von den Entwicklern („Dev") übernimmt und eventuell angepasst verwendet, ist ein großer Schritt in Richtung „DevOps" getan.

Sebastian Hempel
shempel@it-hempel.de

Sebastian Hempel ist selbstständiger IT-Consultant und Trainer aus dem Fichtelgebirge. Seit dem Jahr 2003 unterstützt er Kunden bei der Entwicklung und dem Betrieb von Enterprise-Anwendungen. Seine Schwerpunkt liegt bei Java EE auf Linux-Systemen. In Projekten übernimmt er gerne die Konzeption und den Aufbau der Entwicklungs- und Build-Umgebung. In den letzten Jahren beschäftigt er sich mit der DevOps-Bewegung. Neben seiner Tätigkeit als Software-Entwickler hält Sebastian Hempel Trainings zu Java, Java EE und Puppet.

RESTful-Microservices mit Dropwizard

Felix Braun, Acrolinx

Für Microservice-Architekturen sind klassische Application-Server zu schwergewichtig und unflexibel. Container-less Deployment wird daher ein immer größerer Trend; der Servlet-Container ist in die Applikation eingebettet. Mit dem Dropwizard-Framework kann ein eigenständiger RESTful-Webservice in wenigen Minuten entwickelt und einrichtet werden. Da Dropwizard dabei auf Java-Standard-Bibliotheken wie Jetty und Jersey setzt, fällt der Einstieg besonders leicht.

Ryan Kennedy überlegt keine Sekunde, auf die Frage, wie er Dropwizard in einem Satz beschreiben kann: „Ein Framework, das dir hilft, in kürzester Zeit einen produktionsreifen RESTful-Webservice zu entwickeln – Focus on your Features." Dropwizard hilft beim Logging, Monitoring, Packaging und Deployment. Die Idee dahinter ist, auf ausgereifte Java-Standard-Bibliotheken zu setzen und das Rad nicht neu zu erfinden. Dropwizard verwendet Jetty als Servlet-Container, die JAX-RS-Referenz-Implementierung Jersey für RESTful-Webservices und Jackson als JSON-Bibliothek.

Kennedy, früher Infrastructure Engineer bei Yammer, war lange Zeit ein führender Committer bei Dropwizard. Kurz bevor er selbst Yammer verließ, hat er dabei geholfen, das Dropwizard-Projekt unabhängiger von seinem Arbeitgeber zu machen. Inzwischen ist Dropwizard auf GitHub gehostet und es gibt mehr als 160 Contributors, von denen ein gutes Dutzend ständig aktiv ist.

Dieser Artikel zeigt Schritt für Schritt, wie ein RESTful-Webservice mit Dropwizard erstellt wird. Die Wette ist: In der Zeit, in der man einen Kaffee trinkt, kann man mit Dropwizard *[1]* einen kompletten RESTful-Webservice entwickeln. Also, ab in die Küche und Kaffee aufgesetzt. Während der durchläuft, noch kurz ein Blick auf die Geschichte von Dropwizard.

Das erste Release von Dropwizard erschien im Dezember 2011. Yammer, ein führendes Social Network für Unternehmen, stellte seine IT-Infrastruktur von einem Rails-Monolithen auf Java-Microservices um. Als sich beim zweiten und dritten Service das „Copy & Paste" häufte, fingen die Entwickler an, die wiederkehrenden Patterns in ein Framework auszulagern. Die Definition von Abhängigkeiten, das Log-Format, das Starten des Servlet-Containers, all diese Punkte sollten die Entwickler der einzelnen Services nicht beschäftigen. Die Entwicklerteams konnten sich ausschließlich auf ihre Features konzentrieren. Das Projekt wurde ein Erfolg.

Inzwischen laufen bei Yammer mehr als 170 Instanzen von 30 verschiedenen Services *[2]*, alle erstellt mit dem Dropwizard-Framework. Ist der Kaffee inzwischen fertig? Dann geht es los mit dem ersten Dropwizard-Projekt. Wir erstellen ein Maven-Projekt und fügen dann die Dependency „dropwizard-core" hinzu. Wir werden später noch einige andere Dropwizard-Module verwenden und benutzen daher von Beginn an am besten eine Property zum Pflegen der Versionsnummer *(siehe Listing 1)*.

```
<properties>
    <dropwizard.version>0.8.0</dropwizard.version>
</properties>

<dependencies>
    <dependency>
        <groupId>io.dropwizard</groupId>
        <artifactId>dropwizard-core</artifactId>
        <version>${dropwizard.version}</version>
    </dependency>
</dependencies>
```

Listing 1: Maven-POM

In diesem Artikel entwickeln wir die Applikation WizBook: Ein einfaches Telefonbuch, in das Zauberer via REST hinzugefügt und abgerufen werden können. Das komplette Projekt kann auf GitHub eingesehen werden *[3]*. Ein einfacher Dropwizard-Service besitzt die vier Klassen „Entity", „Application", „Configuration" und „Resource".

Dreh- und Angelpunkt des Service ist die Application-Klasse, die von „io.dropwizard.Application" erbt. In deren „Main"-Methode wird der Service gestartet, in der „run()"-Methode sind die einzelnen Teile der Anwendung registriert und verbunden. Zusätzliche Erweiterungen (Bundles) können in der optionalen Methode „initialize()"registriert werden.

Listing 2 zeigt die erste Version der „WizBookApplication". In der „Main"-Methode wird der Service gestartet. In der „run()"-Methode registrieren wir die „WizardResource". Dabei kommt das Environment zum Einsatz, das von Dropwizard per Parameter zur Verfügung steht. Über das Environment werden die Bestandteile und das Verhalten des Service konfiguriert. Der Zugriff auf das „JerseyEnvironment" erfolgt per „jersey()"-Methode. Damit werden Jersey-Features konfiguriert und, wie in diesem Beispiel, Ressourcen registriert. Zudem zeigt das *Listing 2*, wie mit „conf.getGreeting()" auf einen Wert aus der Konfiguration des Service zugegriffen wird, um diesen zur Begrüßung auf der Konsole auszugeben.

Listing 3 zeigt die verwendete Konfigurationsklasse. Felder in dieser Klasse können mit „JSR 303 Bean Validation"-Annotationen versehen werden. Beim Start stellt der Dropwizard-Service sicher, dass die angegebene „YAML"-Konfigurationsdatei die entsprechenden Felder mit validen Werten enthält. *Listing 4* zeigt die Beispiel-Konfiguration mit einem Wert für „greeting" und der minimalen Server-Konfiguration. Der Jetty-Webserver wird den Service als „http" auf dem Port 56789 anbieten. Per „applicationContextPath: /" wird konfiguriert, dass die WizardResource und alle anderen Ressourcen unter dem Rootpath angeboten sind.

Die „WizardResource" *(siehe Listing 5)* wird über JAX-RS-Annotationen spezifiziert. Sie ist unter dem Pfad „wizards" erreichbar, konsumiert und produziert JSON. Die Ressource bekommt im Konstruktor einen WizardStore übergeben, der die Datenhaltung simuliert. Die „addWizard"-Methode fügt eine Entity in diesen Store ein *(siehe Listing 6)*, sie ist über ein POST auf „/wizards" aufrufbar. Die „@Valid"-Annotation an dem Parameter stellt sicher, dass die Methode nur aufgerufen wird, wenn das übergebene JSON alle JSR-303-Constraints erfüllt.

Im Beispiel wird ein POST mit leerem Namen den Status 422 „Unprocessable Entity" zurückgeben. Die „addWizard"-Methode gibt eine Response zurück mit dem Status 201 „created" und der URL, unter der die Entity abgerufen werden kann. Jersey wandelt die relative URL in eine absolute URL um, sodass an dieser Stelle weder Port oder Hostname notwendig sind.

Mit der „getWizard"-Methode lassen sich eingefügte Entitäten abrufen. Bemerkenswert an dieser Methode ist, dass ein „Guava Optional" zurückgegeben wird. Ist dieses nicht vorhanden, dann gibt Dropwizard einen „404"-Status zurück.

Diese wenigen Zeilen in vier verschiedenen Klassen reichen für die erste Version des WizBook-Service aus. Die Entwickler von Dropwizard empfehlen, die Anwendung

```
public class WizBookApplication extends Application<WizBookConfig> {

  private final Logger logger = LoggerFactory
        .getLogger(WizBookApplication.class);

  public static void main(String[] args) throws Exception{
     new WizBookApplication().run(args);
  }

  @Override
  public void run(WizBookConfig conf, Environment environment) throws Exception {
     logger.info(conf.getGreeting() + ", starting WizBook...");

     environment.jersey().register(new WizardResource());
  }
}
```

Listing 2: Application

```
@Path("wizards")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class WizardResource {
private final WizardStore store;

  public WizardResource(WizardStore store) {
     this.store = store;
  }

  @POST
  public Response addWizard(@Valid final Wizard wizard)
        throws URISyntaxException {
     store.store(wizard);
     return Response.created(new URI("/" + escape(wizard.getName()))).build();
  }

  @GET @Path("{name}")
  public Optional<Wizard> getWizard(@PathParam("name") String name) {
     return store.get(name);
  }
}
```

Listing 5: Resource

```
public class WizBookConfig extends
Configuration {

  @NotEmpty
  private String greeting;

  public String getGreeting() {
     return greeting;
  }
}
```

Listing 3: Configuration

```
greeting: Hello

server:
  type: simple
  applicationContextPath: /
  connector:
    type: http
    port: 56789
```

Listing 4: yml-File

als Fat-Jar einzurichten, also als ein „.jar"-File, das alle Klassen enthält, die für die Ausführung des Service benötigt werden. Das vereinfacht das Handling eines Release: Ein Artefakt kann die verschiedenen Level von QA über Staging bis hin zum Live-Betrieb durchlaufen, ohne dass eventuelle Unterschiede bei den installierten Bibliotheken den Betrieb beeinflussen.

Nachdem das Maven-Shade-Plug-in zur POM hinzugefügt wurde (siehe *[4]*), kann der WizBook-Service gebaut und gestartet werden. Der Aufruf „mvn package" erstellt ein Fat-Jar von ca. 17 MB Größe. Der Service wird mit der Kommandozeile „java -jar wizbook-0.1-SNAPSHOT.jar server conf.yml" gestartet. Der erste Parameter „{server,check}" gibt an, ob der Service gestartet oder nur die Konfiguration validiert werden soll. Der zweite Parameter bezeichnet den Pfad zur Konfigurationsdatei.

Nach wenigen Sekunden ist der Service gestartet und der Aufruf der URL *„http://localhost:56789/wizards/ping"* gibt als Ergebnis einen „200 OK"-Status sowie die JSON-Repräsentation der erzeugten Wizard-Klasse zurück *(siehe Listing 7)*.

Der Kaffee neigt sich dem Ende zu, Zeit für ein kurzes Zwischenfazit: Mit wenigen Codezeilen wurde ein RESTful-Webservice entwickelt, der als Fat-Jar auf jeder JVM gestartet werden kann. Ein eingebetteter Jetty hostet die mit JAX-RS definierte Ressource und Jackson sorgt dafür, dass die zurückgegebene Entität als JSON serialisiert wird. Soweit so gut, aber auch nicht überraschend. „Production-ready" nennt sich Dropwizard, da schon in der Standard-Konfiguration Monitoring- und Metrik-Schnittstellen aktiv sind.

Über den standardmäßig aktiven Admin-Port können per HTTP Healthchecks, Metriken und ein Thread-Dump abgerufen werden. Healthchecks sind kleine Tests, um die korrekte Funktionalität des Service sicherzustellen. So könnten die Tests die Erreichbarkeit einer externen Datenquelle oder den ausreichenden Festplatten-Platz prüfen. Die Entwickler von Dropwizard halten Healthchecks für so wichtig, dass der Service beim Hochfahren diese in Ton und Gestaltung subtile Warnung ausgibt: „!!!!!!! THIS APPLICATION HAS NO HEALTHCHECKS. THIS MEANS YOU WILL NEVER KNOW IF IT DIES IN PRODUCTION, WHICH MEANS YOU WILL NEVER KNOW IF YOU'RE LETTING YOUR USERS DOWN. !!!!!!!"

Ein eigener Healthcheck muss die abstrakte „check()"-Methode der Oberklasse implementieren *(siehe Listing 8)*. Als Rückgabe sind „Result.unhealthy()" oder „Result.healthy()" möglich. Parameter sind optional eine Meldung und ein Stacktrace.

Zauberer sind sehr abergläubisch, daher wird in unserem Beispiel überprüft, dass der Service nicht an einem Freitag, dem Dreizehnten, läuft. Dafür wird Joda-Time genutzt, das zu den transienten Abhängigkeiten gehört, die Dropwizard dem Projekt hinzugefügt hat. Registriert wird der Healthcheck unter Angabe eines frei wählbaren Namens in der „run()"-Methode der Application-Klasse „environment.healthChecks().register("WBHealth", new WizBookHealth());".

Die URL *„http://localhost:56789/admin/healthcheck"* liefert nun den Status „200" und die detaillierten Ergebnisse zu allen Healthchecks als JSON *(siehe Listing 9)*. Sobald einer der registrierten Healthchecks „Result.unhealthy()" zurückgibt, wird der Aufruf der Healthcheck-URL den Status „500" inklusive aller Details und Fehlermeldungen zurückgeben.

Healthchecks sind für eine binäre Rückmeldung des Systemzustands gedacht. Metriken können einen genaueren Einblick in die Performance und Nutzung des Service geben. Jeder Dropwizard-Service enthält auch das Metrics-Framework *[5]*. In der Standard-Konfiguration sind bereits ausführliche Informationen über Jetty und die JVM-Performance abrufbar. *Listing 10* zeigt, wie einfach es ist, eine Resource-Methode mit dem Metrics-Framework zu profilen. Eine „@Timed"-Annotation sorgt dafür, dass die Methode so dekoriert wird, dass die Ausführungszeiten gemessen werden.

Die unter *„http://localhost:56789/admin/metrics"* abrufbaren Metriken zeigen, wie oft und mit welcher Rate die „/ping"-Methode aufgerufen wurde *(siehe Listing 11)*. Auch die Ausführungszeiten werden aufgezeichnet und ausgegeben. Hilfreich ist, dass Perzentile berechnet werden, die deutlich aussagekräftiger sind als die mittlere Ausführungszeit. Das Metrics-Framework enthält noch viele weitere Komponenten wie Zähler und Histogramme, die auch ohne Annotationen direkt aus dem Code zum Einsatz kommen können.

```
public class Wizard {

  private final UUID id;
  @NotEmpty
  private final String name;
  private final WIZARD_HAT hat;
     private final String phoneNumber;
...
}
```

Listing 6: Entity

```
 {
"id":"32f7d779-6dec-4553-
b1c7-0827acd52a4a", "name":
"PongWizard","hat":"POINTY_
HAT","phoneNumber":"0800 3245"}.
```

Listing 7: Wizard-Klasse

```
public class WizBookHealth extends HealthCheck {

  @Override
  protected Result check() throws Exception {
     DateTime now = DateTime.now();
     if (now.getDayOfMonth() == 13 && now.getDayOfWeek() == FRIDAY) {
        return Result.unhealthy("Feeling unhealthy on a Friday the 13th.");
     } else {
        return Result.healthy("Alrighty!");
     }
  }
}
```

Listing 8: Healthcheck

```
{
    "WBHealth": {
        "healthy": true,
        "message": "Alrighty!"
    },
    "deadlocks": {
        "healthy": true
    }
}
```

Listing 9: Healthcheck Result

```
@Path("ping")
@Timed
@GET
public Wizard getPongWizard() { ... }
```

Listing 10: Timed Annotation

Ein weiteres interessantes Konzept von Dropwizard sind Tasks, die in der Applikations-Klasse mit „environment.admin().addTask(new HelloTask("Hello"))“ registriert werden können. Der Task ist danach mit einem POST auf die URL *„http://localhost:56789/admin/tasks/Hello“* ausführbar. Tasks eignen sich, um administrative Abläufe anzustoßen, sie sollten nicht für Applikationslogik verwendet werden.

Die bis hierhin vorgestellten Konzepte sind alle Teil des Moduls „Dropwizard-Core“. Weitere Funktionalität erhält Dropwizard über Bundles. Es gibt Dropwizard-Bundles für Service-Discovery, Datenbank-Zugriff per JDBI oder Hibernate, Authentication, Views und einige mehr. Neben diesen offiziellen Erweiterungen sind auf *„http://modules.dropwizard.io“* derzeit 43 Thirdparty-Bundles verlinkt, darunter Bundles zu Guice, Spring Data und MongoDB.

Im Folgenden schauen wir uns die Funktionsweise eines ausgewählten Bundles genauer an. Mit dem Dropwizard-Views-Bundle kann einem Dropwizard-Service ein einfaches Frontend hinzugefügt werden. Dropwizard unterstützt die beiden Template-Engines „Mustache“ und „Freemaker“. Unser WizBook-Service verwendet das Mustache-Framework. Es ist „logic-less“, erlaubt also keinen Kontroll-Fluss, ist dadurch aber sehr einfach einzusetzen.

Das View-Bundle wird mit „bootstrap.addBundle(new ViewBundle<WizBookConfig>());“ inder „initialize()“-Methode der WizBookApplication-Klasse registriert. Vorher wurden noch die Dependencies „dropwizard-views“ und „dropwizard-views-mustache“ in der Datei „pom.xml“ hinzugefügt.

Die WizardView-Klasse ist eine Erweiterung der Klasse „io.dropwizard.views.View“ *(siehe Listing 12)*. Im Konstruktor wird der Name des Templates angegeben, das zum Rendern der HTML-Seite verwendet wird. Die Entität, die gerendert werden soll, wird im Konstruktor übergeben und per Getter zugreifbar gemacht. Im Mustache-Template *(siehe Listing 13)* wird mit dem Platzhalter „{{#wizard}}“ eine Sektion geöffnet. Alle folgenden Platzhalter beziehen sich nun auf Felder dieser Entität der View-Klasse. Einzelne Platzhalter wie „{{id}}“ werden dann durch den Wert der übergebenen Instanz ersetzt. Komplexe Webanwendungen lassen sich so nicht erstellen, doch für kleinere Anwendungsfälle eignen sich diese Views sehr gut.

Listing 14 zeigt, wie eine Instanz der „WizardView“-Klasse von einer „Ressource“-Methode unter dem Pfad „wizards/{name}/view“ zurückgegeben wird. Ein Highlight an dieser Methode ist die „@Produces“-Annotation mit den beiden Media-Types „TEXT_HTML“ und „APPLICATION_JSON“.

Durch die Angabe des Accept-Headers kann der Aufrufer entscheiden, ob das Ergebnis als JSON oder als HTML repräsentiert wird. Ein Browser, der ein „Accept: text/html“-Header-Feld sendet, erhält die HTML-Seite *(siehe Abbildung 1)*. Ein automatisierter Service, der ein „Accept: application/json“-Header-Feld sendet, erhält die reine JSON-Repräsentation der WizardView-Klasse.

```
timers": {
    "de.fb.demo.resource.WizardResource.
getPongWizard":
{
    "count": 59,
    "max": 0.00486,
    "mean": 0.00006363,
    "min": 0.0000357,
    "p50": 0.0000558,
    "p75": 0.0000681,
    "p95": 0.0000775,
    "p98": 0.0000985,
    "p99": 0.000383,
    "p999": 0.000383,
    "stddev": 0.0000466,
    "m15_rate": 0.0617,
    "m1_rate": 0.716,
    "m5_rate": 0.172,
    "mean_rate": 0.107,
    "duration_units": "seconds",
    "rate_units": "calls/second"
}
```

Listing 11: Metrics-Ausgabe

```
public class WizardView extends View {

  private final Wizard wizard;

  public WizardView(Wizard wizard) {
     super("wizard.mustache");
     this.wizard = wizard;
  }

  public Wizard getWizard() {
     return wizard;
  }
}
```

Listing 12: WizardView

```
public class WizardView extends View {

  private final Wizard wizard;

  public WizardView(Wizard wizard) {
     super("wizard.mustache");
     this.wizard = wizard;
  }

  public Wizard getWizard() {
     return wizard;
  }
}

Listing 12: WizardView

<html>
  <body>
     <h2>Dropwizard Demo</h2>
     {{#wizard}}
        This is {{name}}.
        <p>
        His id is {{id}} and he wears a {{hat}}.
     {{/wizard}}
  </body>
</html>
```

Listing 13: Mustache-Template

```
@GET
@Path("{name}/view")
@Produces({ MediaType.TEXT_HTML, MediaType.APPLICATION_JSON })
public WizardView getWizardView(@PathParam("name") String name) {
  return new WizardView(getWizard(name));
}
```

Listing 14: View Resource Method

Abbildung 1: Die HTML-Seite des Aufrufs

Fazit

Dieser Artikel zeigt, wie mit Dropwizard in kurzer Zeit ein Stand-alone-RESTful-Microservice entwickelt wird. Der große Vorteil von Dropwizard ist die kluge Auswahl von Standard-Bibliotheken, die bereits sinnvoll verknüpft und konfiguriert sind. Dadurch können sich die Entwicklerteams voll auf ihre Features konzentrieren; die technischen Probleme rund um Deployment und Setup der Anwendung sind bereits gelöst.

Die Performance von Dropwizard ist ausgezeichnet. Die Kombination von Jetty- und Jersey-Applikation ist lange erprobt. Auch das zur JSON-Serialisierung eingesetzte Jackson ist hoch performant und wird durch das von Dropwizard aktivierte Jackson-Modul „Afterburner" noch einmal um dreißig bis vierzig Prozent beschleunigt.

Performance-Tests sind ein weites Feld, daher im Folgenden nur ein paar Tendenzen, die jeder in seiner eigenen Umgebung nachvollziehen kann. Es wurde jeweils eine einfache „Hello World"-Applikation in Dropwizard, Spring Boot und GlassFish geschrieben. Die Server liefen jeweils in der Standard-Konfiguration. Eine Client-Anwendung erzeugte mit einem Thread maximale Last auf den Services.

Wurden Resource-Methoden aufgerufen, die Objekte im JSON-Format zurückgeben, ist Dropwizard deutlich schneller als Spring Boot. Geben die Methoden einen primitiven Typ zurück, so ist Spring Boot etwas schneller als Dropwizard. GlassFish fällt in beiden Kategorien deutlich zurück *[6]*.

Obwohl Dropwizard noch die „0" als Major-Version führt, ist das Framework so stabil, dass es problemlos auf Produktiv-Systemen eingesetzt werden kann. Der beste Beweis dafür ist Yammer; dort wird gerade das Upgrade auf Dropwizard 0.8.1 für alle dreißig Services vorbereitet. Auch der Autor hat im letzten Jahr eine Microservice-Landschaft aus insgesamt dreizehn Dropwizard-Services mitgeplant und aufgebaut. In dieser Zeit gab es so gut wie keine Dropwizard-spezifischen Probleme.

Ein großer Vorteil von Dropwizard, verglichen mit Frameworks wie Spring Boot, ist die Verwendung von Java-Standard-Bibliotheken. Mit JAX-RS vertraute Entwickler brauchen keine neuen Konzepte zu lernen. Ein Risiko für Dropwizard ist, dass die Release-Zyklen etwas nachgelassen haben. Die Community hat den Ausstieg von Yammer aus der Entwicklung bereits aufgefangen. In der Release-Planung ist dies noch nicht ganz gelungen. Doch auf den Mailinglisten wurde angekündigt, dass das aktuelle Release 0.8 der Auftakt zu wieder kürzeren Release-Zyklen sein soll.

Der Trend zu Container-less Deployment von Microservices wird sich weiter verstärken. Mit Dropwizard steht ein schlankes Framework bereit, das die neuen Anforderungen bestens erfüllt.

Links

[1] http://www.dropwizard.io
[2] http://de.slideshare.net/JamieFurness1/dropwizard-at-yammer
[3] https://github.com/fexbraun/wizbook-demo
[4] http://www.dropwizard.io/getting-started.html#building-fat-jars
[5] https://dropwizard.github.io/metrics
[6] http://www.heise.de/developer/artikel/Dropwizard-als-REST-App-Server-2431565.html?artikelseite=6

Felix Braun
an@felixbraun.de

Felix Braun ist Team-Lead Server Development bei der Acrolinx GmbH. Er hat mehr als zwölf Jahre Erfahrung als Java-Entwickler. Derzeit interessiert er sich besonders für Komponenten-basierte Systeme und Cloud-Architekturen. Dabei ist er immer auf der Suche nach schlanken Frameworks und Tools, die es ermöglichen, mit weniger Code mehr zu erreichen.

RESTliche Featuritis – modulare Anwendungen mit JAX-RS

Markus Karg, Head Crashing Informatics

Unter der schlichten Nummer „JSR 339" hat eine von Oracle angeführte Expertengruppe über die Version 2.0 des JAX-RS-Standards beraten. Was dabei herauskam, war weit mehr als ein Java-API für REST-Anwendungen: JAX-RS-2.x-konforme Frameworks erlauben vielmehr ein modulares, Feature-zentrisches Anwendungsdesign mit klarer Trennung von Anwendungsdomäne und technischer Implementierung.

Das Akronym „JAX-RS" steht im Grunde für „Java API for XML RESTful WebServices". Es ist genau genommen schon immer falsch gewesen, denn mit XML hat JAX-RS eigentlich nur am Rande zu tun. Der Standard mag zwar vom Grundsatz her dazu gedacht gewesen sein, hauptsächlich XML-Dokumente zu transportieren, weil dies die vorherrschende Spezies in grauer Vorzeit war, doch da die REST-Architektur (REpresentational State Transfer) ebenso wenig wie das zumeist genutzte HTTP-Protokoll diesen Dokumententyp weder erzwingt noch zumindest empfiehlt, wurde sehr früh das „XML" im Namen gestrichen und dessen bevorzugte Behandlung mit Version 2.1 eingedampft *[1]*.

JAX-RS ist grundsätzlich offen für jeden standardisierten oder selbst erfundenen Dokumententyp, was Spielraum für Anwendungszwecke jenseits von CRUD (Create, Read, Update, Delete) und REST gibt –damit ist praktisch auch das „RS" des Akronyms obsolet. Tatsächlich normiert JAX-RS generell Anwendungs-Frameworks und seit Version 2.0 unter anderem deren Fähigkeit, Anwendungen in Features zu zerlegen.

Once upon a time …

Die REST-Architektur wurde im Jahr 2000 durch Roy Thomas Fieldings Dissertation *[2]* bekannt. Er beschrieb darin den Aufbau des WWW als ein System, bei dem weder Client noch Server einen Kommunikationsstatus vorhalten, sondern sich diesen in Form eines jeweils aktualisierten Dokuments gegenseitig wie Ping-Ping-Spieler zuspielen. Jeder Spieler schaut sich den ihm zugespielten Ball („Request") an, zieht die notwendigen Schlüsse aufgrund seiner Geschäftsregeln, beschriftet den Ball neu und spielt ihn zurück („Response"). Der Vorteil: Es können beliebig viele Teilnehmer mit ebenso beliebig vielen Bällen teilnehmen – das System ist praktisch unbegrenzt skalierbar. Das WWW hat durch sein jahrzehntelang ungebremstes Wachstums die Validität dieser These nachhaltig untermauert.

Auch wenn typischerweise die Mehrzahl der Menschen beim Stichwort „World Wide Web" eher an Browser und Mausklicks denken, ist das REST-Architektur-Muster keineswegs auf menschliche Teilnehmer beschränkt. Im Gegenteil, JAX-RS konzentriert sich gerade eben nicht auf Menschen als Zielgruppe, sondern dient ausschließlich Maschinen dazu, Teil des WWWs zu sein – und zwar nicht nur als Anbieter, sondern auch als Nutzer.

Im Zuge des Industrie-4.0-Hypes sei notabene angemerkt, dass mit dem Wort „Maschinen" explizit nicht nur Computer gemeint sind, sondern tatsächlich jegliche Art von Gerätschaft. Hierzu wurden in den frühen Versionen des JAX-RS-Standards die entsprechenden serverseitigen Verträge definiert, während seit Version 2.0 auch die clientseitigen Schnittstellen definiert sind. Somit ist es möglich, Java-Anwendungen zu schreiben, die gleichzeitig sowohl RESTful-Dienste darstellen als auch selbst wiederum solche nutzen.

Bei der Ausgestaltung des JAX-RS-API bemerkten die Autoren der Spezifikation, dass die alleinige Konzentration auf die objektorientierte Ummantelung des HTTP-Protokolls mittels Java-Schnittstellen nicht genügt, um das REST-Architektur-Muster umfassend abzudecken. Für ein solcherlei beschränktes Ansinnen hätten einige wenige Erweiterungen der Servlet-Spezifikation durchaus genügt. Da REST aber weit mehr als CRUD darstellt, muss JAX-RS in tiefer liegende Anwendungs-

schichten eingreifen. JAX-RS kann sich somit nicht auf die reine Client-Server-Schnittstelle beschränken und muss wiederverwendbare Anwendungs-Bausteine sowie ein Rahmenwerk zur Verbindung selbiger anbieten. Wohlgemerkt, JAX-RS spezifiziert eine Norm für Frameworks, ist selbst jedoch kein Produkt, auch wenn mit Oracle „Jersey" ein solches als Referenz-Implementierung (RI) herangezogen wurde. Die Gratwanderung zwischen Normieren fremder Produkte und Umsetzen eigener Produkte ist jedoch sehr gut gelungen, sodass heute mehrere Anwendungsframeworks unterschiedlicher Prägung in der Lage sind, die Norm in der aktuellen Version umzusetzen, ohne dass jene allzu viel Code vordefiniert.

Neben dem bereits erwähnten „Jersey" *[3]* sind dies beispielsweise „RESTEasy"*[4]* von Red Hat oder Apache „CXF"*[5]*. Alle genannten Anbieter haben, ebenso wie der Autor dieses Artikels, in besagter Expertengruppe „JSR 339" aktiv am Standard mitgearbeitet, wodurch die Kompatibilität, Stichwort „Write Once Run Anywhere" (WORA), gewährleistet ist: Es gibt praktisch keine herstellerspezifische Abhängigkeit in einer JAX-RS-konform programmierten Anwendung.

Lose Kopplung und Separation of Concerns

Wie in ähnlicher Weise bereits von der EJB-Technologie bekannt, wird schon seit Version 1.0 der Spezifikation eine JAX-RS-Anwendung durch das Framework zur Laufzeit aus einzelnen Komponenten montiert, die zur Compile-Zeit technisch keine zwingende Verbindung haben, was gemeinhin als lose Kopplung bekannt ist. Hierzu werden Plain Old Java Objects (POJO) durch simple Auszeichnung mit Java-Annotationen bestimmte Stereotypen zugeschrieben, die das Framework wiederum nutzt, um über die weitere Behandlung der Komponente zu entscheiden.

Allgemein bekannt dürften dabei die Annotationen „@Path" und „@GET" sein, durch die der Stereotyp „HTTP-Ressource" zugeschrieben wird, wohingegen beispielsweise die Zuschreibung des Stereotyps „Entity Provider" mittels „@Provider" und „@Produces" weit weniger bekannt, technisch aber umso mächtiger ist. „Entity Provider" bilden nämlich gemeinsam mit den Parameter-„Converter"-Providern die technische Grundlage für die Separation der fachlichen Anwendungslogik vom technischen Durchführungsweg. Dies bedeutet, eine „saubere" JAX-RS-Anwendung enthält grundsätzlich keinerlei API-Aufrufe, die sich mit rein technischen Problemstellungen wie der Behandlung von HTTP-Request und -Response, Parsen von XML oder JSON, Rendering von PDF oder SVG, Transformation von Query-Parametern in Java-Enums etc. befassen.

Solche Aspekte sind in Provider ausgelagert, die aufgrund obengenannter loser Kopplung wiederum keine Kenntnis von der Anwendungslogik haben und dadurch mehrfach verwendbar sind. Zur Laufzeit bemerkt das Framework, wenn ein Provider zur Lösung einer technischen Problemstellung (etwa der Übersetzung von Java-Klassen in HTTP-Header oder JSON-Dokumente) benötigt wird, und nutzt diesen automatisch, ohne dass die Anwendung dies explizit mitteilt.

Möglich wird dies, da alle Provider während des Ladevorgangs automatisch erkannt und somit vom Framework für den entsprechenden Zweck registriert werden. Komplexe, in der Spezifikation beschriebene Regeln sorgen dafür, dass praktisch „wie von Geisterhand" immer der am besten passende Provider zur Anwendung kommt, sollten mehrere für den gleichen Zweck zu finden sein – beispielsweise wenn ein Client sich per HTTP-Accept-Header bereit erklärt, sowohl XML als auch JSON zu akzeptieren, und dem Framework sowohl ein XML- als auch ein JSON-Renderer bekannt sind. Entsprechend korrekte Parametrisierung der Annotationen durch Attribute wie MIME-Typen etc. ist dazu zwingende Voraussetzung

Aufbauend auf dieser Mechanik kamen mit JAX-RS 2.0 weitere Fähigkeiten des Frameworks hinzu, die eine noch weitergehende Abstraktion der Anwendung von der technischen Umsetzung erlauben (Filter, Interzeptoren, Features und Konfiguration), wodurch der Baukasten nun praktisch komplett ist. Eine Anwendung muss sich somit seither auch nicht mehr um Aspekte wie Zugangsschutz, Verschlüsselung, Kompression, Caching, Konfiguration etc. kümmern, da auch für diese Techniken entsprechende Provider ermöglicht wurden.

Hinzu kommt mit der Einführung der Features die Möglichkeit, Provider zu bündeln und gemeinsam ein- beziehungsweise auszuschalten. So wird es mit der Version 2.0 der Open-Source-Software „WebDAV Support for JAX-RS" beispielsweise möglich sein, das Protokoll „WebDAV" als ein einziges Feature ein- beziehungsweise auszuschalten, obwohl dessen Implementierung durchaus mehrere Komponenten umfasst.

Gerade die Übersichtlichkeit und Verständlichkeit einer Anwendung wird erheblich gesteigert, wenn diese lediglich ein komplettes Themengebiet als notwendig deklariert (etwa „benötigt WebDAV"), sich jedoch nicht mehr selbst darum kümmern muss, welche technischen Komponenten dazu erforderlich sind. Ähnlich könnte eine Anwendung anmelden, das Feature „Kompression" zu benötigen, ohne sich Gedanken über die Vielzahl an möglichen Algorithmen und Datenformaten zu machen, die dabei eine Rolle spielen.

Letztendlich ermöglichen Features es somit, thematisch (nicht technisch) geschnürte Pakete auf einem Markt zu behandeln, da diese nun anwendungsunabhängig sind: Ein späterer Austausch des Features „Einfache Kompressions-Algorithmen" durch ein Feature „Starke Kompressions-Algorithmen" ist binnen Minuten erledigt und zeigt die Mächtigkeit der Spezifikation: Der Kernpunkt der Wiederverwendung ist nicht mehr die Klasse (JavaBean) oder Komponente (EJB), sondern die Fähigkeit (Feature).

Ob sich ein solcher Markt tatsächlich bildet, muss die Zukunft zeigen. Die technischen und organisatorischen Grundlagen sind jedenfalls vorhanden und können auch ohne Handel mit anderen großen Nutzen bringen: Sie eröffnen eine hervorragende Möglichkeit, sauberen, also in der eigentlichen Funktionsdomäne der Anwendung formulierten Code zu schreiben, wodurch spätere technische Anpassungen ohne Änderung der Anwendung und somit ohne das Risiko neuerlicherFehlerquellen durch Querbeziehungen möglich sind.

Features nachrüsten ohne Code-Änderung

Angenommen, es soll eine Anwendung entwickelt werden, die Lotterie-Spielscheine akzeptiert, wobei der Spielschein in der Filiale gescannt und per „HTTPS PUT" an die Anwendung übergeben wird. Ein Servlet müsste nun in einer Nicht-JAX-RS-Anwendung die binäre Bild-Information per Optical Character Recognition (OCR) in Text umwandeln, um den Spieler und seinen Tipp in der Teilnehmer-Datenbank zu registrieren.

Eine JAX-RS-Anwendung hingegen kümmert sich um diesen technischen Sachverhalt nicht selbst. Sie würde auf viel höherer Ebene ansetzen, sich weder um HTTPS noch um das OCR kümmern, sondern lediglich deklarieren, dass sie gewillt ist, ein per „HTTP PUT" zugestelltes Java-Objekt der Klasse „Spielschein" zu akzeptieren. Die Verschlüsselung würde sie üblicherweise tiefer liegenden Schichten überlassen und die Schrifterkennung als zu nutzendes Feature deklarieren, ohne explizit Klassen, Komponenten oder Hersteller zu benennen.

Der Deployer könnte entsprechende Features jederzeit ohne Änderung der Anwendung austauschen oder nachrüsten, sollte dies nötig oder gewollt sein – beispielsweise zur Verbesserung der Erkennungsrate der OCR, um zusätzliche Bildformate wie „PNG" statt „JPG" oder Transfer-Kompression bei nicht komprimierten Bildformaten wie „TIFF" zu unterstützen.

Anwendung und Features können von absolut separierten und sich gegenseitig nicht bekannten Expertenteams entwickelt werden, ohne sich in die Quere zu kommen oder regelmäßige Kompatibilitätstests anstrengen zu müssen – beide stützen sich lediglich auf die JAX-RS-Spezifikation. Diese Entkopplung zieht neue Teams an, die bereits eine JAX-RS-Anwendung betreiben und diese nun mit OCR nachrüsten können – ohne Rücksprache mit dem Lotto-Anbieter oder dem OCR-Team.

Weiterhin angenommen, die Scanner in den Filialen werden nach und nach abgebaut und die Spielscheine zunehmend von einem WWW-Endkunden-Frontend per XML-Dokument an die Anwendung geliefert, so wäre eine Servlet-basierte Anwendung mehrfach umzuschreiben: Die Autoren müssten ihr beibringen, den MIME-Type zu inspizieren, um zwischen dem Aufruf der OCR und dem Anliefern der XML zu unterscheiden. Die JAX-RS-Anwendung hingegen benötigt keine Änderung: Es wird dem Framework (nicht der Anwendung) lediglich ein neuer Provider zur Verfügung gestellt. Seine Deklaration meldet per „@Consumes" die Fähigkeit, JSON zu verarbeiten, und bestätigt dem Framework gegenüber zur Laufzeit, dass dieser Instanzen der Klasse „Spielschein" liefern kann.

Ab nun wertet das Framework (nicht die Anwendung) zur Laufzeit den MIME-Type aus, entscheidet somit pro HTTP-Request, ob es einen Spielschein an „OCR" senden oder aus „JSON" interpretieren muss, und ruft die bestehende, exakt gleiche, unveränderte Verarbeitungsmethode in der Lotto-Anwendung auf. Die Anwendung, obwohl unverändert, hat aber eine neue Fähigkeit („Feature") gelernt.

Moularisieren der Anwendung

Das zugegeben etwas anachronistisch wirkende Beispiel verdeutlicht den großen Vorteil der JAX-RS-Architektur: Anwendungen werden damit nicht nur von der Technik getrennt und befassen sich nur noch mit fachlichen Objekten statt mit technischen; sie werden vielmehr auch in erheblichem Maße modularisiert und sind somit ohne Code-Änderung mit neuen Features nachrüstbar. Diese Fähigkeit, die man bislang nur von proprietären Umgebungen wie beispielsweise „Eclipse" kannte, hat damit nun Einzug in einen offiziellen Java-Standard gehalten und steht somit automatisch auf allen JAX-RS-kompatiblen Frameworks zur Verfügung – und zwar nicht nur Herstellern von Technik-Komponenten wie im Beispiel der OCR oder der Bild-Kompression, sondern eben auch den Anwendungs-Programmierern selbst.

Beim Betrachten einer bestehenden Anwendung, beispielsweise einer Servlet-basierten, finden sich sehr viele Aspekte, die zwar zur Erledigung einer gewünschten Aufgabe konzertiert agieren, jedoch eigentlich nicht zwingend voneinander Kenntnis haben müssen. Das heißt, eine Portierung der Anwendung auf JAX-RS wird dazu führen, dass viele Aspekte nun in eigenständige Provider ausgelagert werden können, folglich aus dem Anwendungscode verschwinden.

Zwar sind diese Provider oftmals weiterhin spezifisch auf eine bestimmte Anwendung zugeschnitten und nicht oder nur bedingt allgemein nutzbar, trotzdem bietet es sich damit an, sie in Features zu gruppieren, beispielsweise um sie gemeinsam ein-/ausschaltbar zu machen – was zuvor vielleicht über viele „if"s gelöst wurde. Letztendlich stellen Provider und Features ja nichts anderes dar als die technische Umsetzung des Strategie-Entwurfsmusters *[6]* (Strategy Pattern) für bestimmte Szenarien. Liegt in der eigenen Anwendung ein solches Szenario vor, ist der Weg zur Auslagerung in einen Provider oder in ein Feature frei.

Die beiden hauptsächlichen Schwierigkeiten bei der Modularisierung einer Anwendung sind hierbei sicherlich sowohl die umfassende Kenntnis der angebotenen Möglichkeiten, also die Frage „Welcher Stereotyp in welchem Szenario", bei der *Tabelle 1* sicherlich eine Hilfestellung sein kann, als auch die als „Separation of Concerns" bezeichnete Denksportaufgabe, eine Aufgabe zunächst in einzelne Bausteine, also Provider, zu zerlegen, um diese im nächsten Schritt in thematisch zusammengehörende Blöcke, also Features, zu kombinieren. Zur Veranschaulichung wird nochmals obengenanntes Beispiel zusätzlicher Datentypen herangezogen.

Butter bei die Fische

Angenommen, eine Anwendung soll mit dem Datentyp „PDF" nachgerüstet werden, beispielsweise um gescannte Formulare in Daten zu wandeln oder um Daten als Druckvorschau downloaden zu können. Anstatt diese Fähigkeiten irgendwie in die Anwendung einzuflicken, werden diese zunächst in Stereotype zerlegt. Ein Message Body Reader für den Datentyp „PDF" dient als Verpackung für den Code, der PDFs analysieren und in Domänen-Objekte, beispielsweise POJOs der

```
@Provider public class LottoPdfSupport implements Feature {
   public boolean configure(FeatureContext context) {
       context.register(LottoscheinPdfParser.class);
       context.register(LottoscheinRenderer.class);
       context.register(PersonalausweisParser.class);
       context.register(PersonalausweisRenderer.class);
       return true; // Feature wurde korrekt registriert.
   }
}
```

Listing 1: Zusammenfassen von Providern zu Features

Unterstütztes Szenario	Zu implementierender Stereotyp
Parsen von Eingangsdokumenten	Message Body Reader
Rendern von Ausgangsdokumenten	Message Body Writer
Parsen von Parametern und Headern	Parameter Converter Provider
Rendern von Parametern und Headern	Parameter Converter Provider
Filtern von Anfragen	Container Request Filter und Reader Interceptor
Filtern von Antworten	Container Response Filter und Writer Inter
Gemeinsames Ein-/Ausschalten	Feature
Feature für bestimmte URLs aktivieren	Dynamic Feature

Tabelle 1: Einige unterstützte Szenarien in JAX-RS-Frameworks

Klasse „Lottoschein", umwandeln kann. Ein Message Body Writer für den Datentyp „PDF" wiederum dient als Verpackung für den Code, der aus „Lottoschein"-Instanzen ein PDF als Druckvorschau rendert. Weitere Entity Provider (also Message Body Reader und Message Body Writer) könnten für die Wandlung von PDF in andere Java-Typen, wie zum Beispiel die Teilnehmeranmeldung mit Kopie des Personalausweises, dienen. Das Gesamtpaket „PDF-Unterstützung" soll nun am Stück nachgerüstet werden und nur gemeinsam ein- beziehungsweise ausgeschaltet werden können. Hierzu wird ein Feature „LottoPdfSupport" programmiert, das die genannten Einzelbausteine gemeinsam registriert *(siehe Listing 1)*.

Sobald sich das Feature „LottoPdfSupport" im Klassenpfad der Anwendung befindet, wird es geladen und aktiviert. Der in *Listing 1* sichtbare Rückgabewert teilt dem Framework mit, ob das Feature aktiviert oder deaktiviert sein möchte. Beispielsweise könnte ein Feature mittels „context.getConfiguration().isEnabled(NotwendigesFeature.class" Abhängigkeiten prüfen und auf diesem Wege den Dienst verweigern, ohne die Gesamtanwendung vom grundsätzlichen Betrieb abzuhalten.

Fazit

Features sind eine sehr nützliche Neuheit in JAX 2.0, um die Zusammenfassung von Providern zu umfassenderen Aspekten zu ermöglichen. Dadurch wird die Anwendung strukturierter und der Code besser verständlich. Fehler werden vermieden, unerwünschte Abhängigkeiten beseitigt, Wiederverwendung gefördert. Sie zeigen exemplarisch, wo der Weg von JAX-RS hinführt: Weg von reinem REST und CRUD, hin zu einem umfassenden Anwendungs-Framework.

Weitere Informationen

[1] JSR 370: „https://www.jcp.org/en/jsr/detail?id=370"

[2] Fielding, Roy Thomas, Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000: „http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm"

[3] Oracle Jersey: „https://jersey.java.net"

[4] Red Hat RESTEasy: „http://resteasy.jboss.org"

[5] Apache CXF „http://cxf.apache.org"

[6] Strategie-Entwurfsmuster: „http://de.wikipedia.org/wiki/Strategie"

Markus Karg
markus@headcrashing.eu

Markus Karg fasziniert das Programmieren, seit er in grauer Vorzeit einen Sinclair ZX Spectrum in die Hand bekam. Heute verantwortet er die Entwicklung eines unabhängigen Software-Herstellers. JAX-RS begleitet der gebürtige Pforzheimer seit Version 0.8 durch Feature Proposals, Mitarbeit an der Referenz-Implementierung „Jersey" und zuletzt in der Expert Group JSR 370. Einige Features von JAX-RS, aber auch von anderen Java-APIs, stammen aus der Feder des JCP-Mitglieds. Seine Freizeit widmet er der Kunst seiner Frau sowie einer nachhaltigen Gesellschaft.

DukeCon ist mehr als eine Javaland-App!

Gerd Aschemann, JUG Darmstadt

Auf der Javaland 2015 wünschten sich viele Teilnehmer vor Ort eine zeitgemäße App, um auf das Programm zuzugreifen, ihre Vortrags-Favoriten zu verwalten und bei Bedarf den Speakern Feedback zu geben. Die JUGs Darmstadt und Kaiserslautern haben sich an die Arbeit gemacht und *„http://DukeCon.org"* gegründet (Twitter „@DukeConference"). Mithilfe von DOAG und iJUG sind ein REST-Service für die Konferenzdaten entstanden, auf dem mehrere Clients aufsetzen. Die Clients sind voll benutzbar und man kann sich jetzt schon damit sein Programm für die Javaland 2016 zusammenstellen. Andere JUGs und Organisationen unterstützen mittlerweile und haben Interesse bekundet, beispielsweise DukeCon für weitere Konferenzen zu nutzen. Die Macher sind offen für neue Mitstreiter und Ideen.Die Plattform nutzt zahlreiche aktuelle Technologien und erkundet den einen oder anderen Trend. Beim Buzzword-Bingo wird in vorderster Linie mitgespielt: Spring Boot, Docker, HTML5, Flex, Jasmine, Meteor, KeyCloak (SingleSignOn/SocialLogin), Offline-Fähigkeit/Browser-Caching, localStorage, Cordova, Android, iOS, Groovy, REST, Lombok, Continuous Integration/Delivery, ElasticSearch/Logstash/Kibana, JPA, Maven, Jenkins, Nexus, Puppet, Vagrant etc.

Auf der Javaland 2016 gibt es ein Community-Event am Montag-Abend vor Konferenz-Beginn. Jeder Konferenz-Teilnehmer ist eingeladen, in die Plattform reinzuschnuppern (bitte Community-Programm und Anmeldung beachten). Die Beteiligten zeigen, wie sie die Software entwickeln, durch die Delivery-Pipeline schieben und auf ihrer automatisiert aufgebauten und gepflegten Infrastruktur betreiben. Die Teilnehmer können einfach mal was ausprobieren, sei es von dem, was gemacht wurde oder in diesem Umfeld etwas Neues. Vielleicht mag jemand eine DukeCon-UI mit JavaFX bauen oder alles auf Java EE umstellen oder endlich mal von Postgres nach MongoDB migrieren, oder, oder, oder? Die Macher freuen sich auf rege Diskussionen und Erfahrungsaustausch.

Alle Quellen und weiteren Verweise finden sich auf Github unter *„https://github.com/dukecon"* und natürlich auf der Homepage des Projekts. Eine Unterstützung des Projekts ist auch unabhängig von der Konferenzteilnahme möglich: Alles ist Open Source. In der nächsten Ausgabe der Java aktuell wird die Plattform-Architektur ausführlich vorgestellt.

Gerd Aschemann
gerd@aschemann.net

„Vor diesem Hintergrund ist die starke Community das Rückgrat von Java …“

Usergroups bieten vielfältige Möglichkeiten zum Erfahrungsaustausch und zur Wissensvermittlung unter den Java-Entwicklern. Sie sind aber auch ein wichtiges Sprachrohr in der Community und gegenüber Oracle. Wolfgang Taschner, Chefredakteur Java aktuell, sprach darüber mit Björn Martin von der Java User Group Karlsruhe.

Wie ist die Java User Group Karlsruhe organisiert?
Björn Martin: Unsere JUG wird zur Zeit von einer Dreiergruppe geleitet, der Florian Hopf, Michael Prieß und ich angehören. Wir teilen uns die Organisation untereinander auf. Im wesentlichen beschränkt sich das bei uns auf das Kontaktieren von Speakern für mögliche Vorträge und die Koordination, also das Kümmern um eine Lokalität, das Sponsoring danach – wir gehen gewöhnlich noch auf dessen Kosten ein Bierchen trinken – und ein wenig Marketing. Viel von der Kommunikation findet digital statt, aber wir treffen uns auch regelmäßig, um aktuelle Themen durchzusprechen.

Was sind die Ziele Java User Group Karlsruhe?
Björn Martin: Seit es die JUG Karlsruhe gibt war das Ziel immer, eine kostenlose Möglichkeit zu bieten, sich im Bereich „Java“ und drumherum auf dem Laufenden zu halten. Das klappt jetzt schon ein knappes Jahrzehnt ganz gut.

Wie viele Veranstaltungen gibt es pro Jahr?
Björn Martin: Wir haben jeden Monat einen Vortrag. Das hat sich ganz gut eingependelt. Und da wir ja nicht die einzige User Group in Karlsruhe sind, braucht es auch nicht mehr.

Was bedeutet Java für euch?
Björn Martin: Für mich ist Java eine Sprache, bei der viele Rahmenbedingungen passen. So konnte sich durch den freien Zugang und die in der Sprache verankerte Modularität schon früh ein sehr großes Open-Source-Ökosystem an Bibliotheken bilden, welches auch heute noch seinesgleichen sucht. Das macht es relativ einfach, für so ziemlich jeden Anwendungsfall eine Bibliothek zu finden, welche nicht nur frei zugänglich, sondern auch sehr erprobt ist.

Welchen Stellenwert besitzt die Java-Community für euch?
Björn Martin: Ohne die Java-Community gäbe es das eben erwähnte Open-Source-Ökosystem nicht in dieser Form. Vor diesem Hintergrund ist die starke Community das Rückgrat von Java.

Wie sollte sich Java weiterentwickeln?
Björn Martin: Dazu wird jeder Entwickler seine Sicht haben. Ich persönlich finde die aktuell betriebene, sehr konservative und überlegte Weiterentwicklung, die Oracle aktuell verfolgt, genau richtig. Es gibt diverse Konstrukte wie die Öffnung der JVM für andere Sprachen, welche es der Community ermöglicht, in der eigenen Geschwindigkeit neues schneller auszuprobieren. Scala ist hier ein gutes Beispiel. Bei den Enterprise-Bibliotheken zeigt Spring, wie Java EE in schnell geht. All das basiert aber auf einem stabilen, durchdachten Kern. Das kann gerne so bleiben.

Wie sollte Oracle eurer Meinung nach mit Java umgehen?
Björn Martin: Oracle macht es richtig, wenn man die technische Herangehensweise betrachtet, also die Weiterentwicklung der Sprache. Politisch muss Oracle noch lernen, was es bedeutet, mit einer freien Community zu interagieren. Negative Beispiele wie der Umgang mit den Test-Lizenzen für das JDK der Apache-Foundation oder die politisch bedingten Forks von OpenOffice (LibreOffice) und Hudson (Jenkins) zeigen, dass hier noch das nötige Feingefühl fehlt.

Wie sollte sich die Community gegenüber Oracle verhalten?
Björn Martin: Die Community sollte nie vergessen, dass Oracle – wie alle anderen Unternehmen auch – Geld verdienen möchte. Nur verdientes Geld kann auch wieder ausgegeben werden. Vor dem Hintergrund sollte an mancher Stelle Nachsicht geübt werden.

Björn Martin
jug@bjoern-martin.de

Zur Person: Björn Martin
Björn Martin ist leidenschaftlicher Entwickler und seit seinem Studium an der FH Karlsruhe der Sprache Java verschrieben. Seit gut zehn Jahren ist er als Entwickler, Architekt und Manager in Karlsruhe tätig. Aktuell leitet er ein Entwicklungs-Team bei der BrandMaker GmbH. Zusätzlich ist er in der Java User Group Karlsruhe als Co-Organisator aktiv.

Entwurfsmuster – Das umfassende Handbuch

gelesen vom Daniel Grycman

Dieses Buch widmet sich auf 643 Seiten dem sehr theorielastigen Thema der Informationstechnologie. Diese Rezension beschreibt, ob dem Autor der Wechsel zwischen Theorie und Praxisbezug gelingt. Vorab eine persönliche Anmerkung: In der gesamten Rezension verwendet der Autor grundsätzlich die englische Bezeichnung des jeweiligen Musters.

Der Verfasser des Buches, Martin Geirhos, ist dem Autor das erste Mal beim Lesen des Buchs „IT-Projekt-Management" aufgefallen, in dem er sich auf eine sehr lockere Art und Weise mit der Theorie des IT-Projektmanagements beschäftigt. Entsprechend hatte er auch gewisse Erwartungen an das Werk.

Das Buch gliedert sich in acht Kapitel. Die Muster in den Kapiteln zwei bis sieben werden durch eine kurze Erklärung erläutert, danach folgt ein kleiner Steckbrief mit dem deutschen und englischen Namen. Im Anschluss daran liefert der Autor ein UML-Diagramm über das jeweilige Muster. Nun folgt eine ausführliche Erklärung des Musters und entsprechender Anwendungsfälle, die durch Java das Muster veranschaulichen. Mögliche Alternativen und weitere Überlegungen werden danach erläutert.

Das erste Kapitel beinhaltet eine Einführung in das Thema „Entwurfsmuster" und es wird auch die „Gang of Four" vorgestellt. Martin Geirhos legt auch direkt dar, dass nicht alle vorgestellten Muster von der Gang of Four stammen. Im zweiten Kapitel präsentiert er die Erzeugungsmuster („Factory", „Singleton", „Multiton", „Abstract Factory", „Builder" und „Prototype"). Das dritte Kapitel beschäftigt sich mit Strukturmustern („Adapter", „Bridge", „Composite", „Decorator", „Facade", „Flyweight" und „Proxy"). Den Verhaltensmustern ist das vierte Kapitel geschuldet. In den Unterkapiteln geht es um „Chain of Responsibility", „Command", „Interceptor", „Interpreter", „Iterator", „Mediator", „Memento", „Observer", „State", „Strategy", „Template Method" und „Visitor". Dies ist auch das längste Kapitel des Buchs mit insgesamt 163 Seiten.

Im nächsten Kapitel beschäftigt sich der Autor mit Mustern im Zusammenhang von verteilten Architekturen. Neben den acht Irrtümern von „Distributed Computing" werden auch „SOA", „Event Sourcing" und „Command Query Responsibility Segregation" (CQRS) erläutert. Das sechste Kapitel setzt sich mit Mustern beim Umgang mit Daten innerhalb einer Anwendung auseinander. Martin Geirhos greift hier die folgenden Muster auf: „Unit of Work", „Transactions", „Data Transfer Objects", „Table Data Gateway", „Row Data Gateway", „Identity Map/Function", „Optimistic Locking", „Pessimistic Locking" und „Inheritance".

Im vorletzten Kapitel werden noch die drei bekanntesten Muster für GUIs („Model View Controller", „Model View Presenter" und „Model View ViewModel") erläutert. Den Abschluss bildet das Kapitel zu Design- und Entwicklungsprinzipien. Zu Anfang beschäftigt sich der Autor mit Merkmalen schlechten Designs, dem SOLID-Prinzip und dem agilen Manifest. Ebenso werden konkrete Designprinzipien, wie KISS und DRY, und Design Smells erläutert.

Fazit

Insgesamt handelt es sich um ein sehr gelungenes Buch. Dem Autor Martin Geirhos gelingt es mit Leichtigkeit, Theorie und Praxis-Bezug zu verknüpfen. Jeder Software-Entwickler sollte, wenn nicht schon das GoF-Buch vorhanden ist, ein Exemplar auf seinem Schreibtisch haben. Es eignet sich nicht nur für Programmier-Anfänger zur Vertiefung und Festigung von Wissen, sondern auch für Praktiker als Nachschlagwerk.

Titel:	Entwurfsmuster – Das umfassende Handbuch
Verlag:	Rheinwerk Verlag
Umfang:	643 Seiten
Preis:	39,90 Euro, eBook (downloadbar) nicht im Preis enthalten
ISBN:	978-3-8362-2762-9

Daniel Grycman
daniel.grycman@bilsteingroup.com

Die iJUG-Mitglieder auf einen Blick

Java User Group Deutschland e.V.
www.java.de

DOAG Deutsche ORACLE-Anwendergruppe e.V.
www.doag.org

Java User Group Stuttgart e.V. (JUGS)
www.jugs.de

Java User Group Köln
www.jugcologne.eu

Java User Group Darmstadt
http://jugda.wordpress.com

Java User Group München (JUGM)
www.jugm.de

Java User Group Metropolregion Nürnberg
www.source-knights.com

Java User Group Ostfalen
www.jug-ostfalen.de

Java User Group Saxony
www.jugsaxony.org

Sun User Group Deutschland e.V.
www.sugd.de

Swiss Oracle User Group (SOUG)
www.soug.ch

Berlin Expert Days e.V.
www.bed-con.org

Java Student User Group Wien
www.jsug.at

Java User Group Karlsruhe
http://jug-karlsruhe.mixxt.de

Java User Group Hannover
www.jug-h.de

Java User Group Augsburg
www.jug-augsburg.de

Java User Group Bremen
www.jugbremen.de

Java User Group Münster
www.jug-muenster.de

Java User Group Hessen
www.jugh.de

Java User Group Dortmund
www.jugdo.de

Java User Group Hamburg
www.jughh.de

Java User Group Berlin-Brandenburg
www.jug-berlin-brandenburg.de

Java User Group Kaiserslautern
www.jug-kl.de

Java User Group Switzerland
www.jug.ch

Der iJUG möchte alle Java-Usergroups unter einem Dach vereinen. So können sich alle Java-Usergroups in Deutschland, Österreich und der Schweiz, die sich für den Verbund interessieren und ihm beitreten möchten, gerne unter office@ijug.eu melden.

iJUG Verbund

www.ijug.eu

Impressum

Herausgeber:
Interessenverbund der Java User Groups e.V. (iJUG)
Tempelhofer Weg 64, 12347 Berlin
Tel.: 030 6090 218-15
www.ijug.eu

Verlag:
DOAG Dienstleistungen GmbH
Fried Saacke, Geschäftsführer
info@doag-dienstleistungen.de

Chefredakteur (VisdP):
Wolfgang Taschner, redaktion@ijug.eu

Redaktionsbeirat:
Ronny Kröhne, IBM-Architekt;
Daniel van Ross, FIZ Karlsruhe;
André Sept, InterFace AG;
Jan Diller, Triestram und Partner

Titel, Gestaltung und Satz:
Elena Rankova
DOAG Dienstleistungen GmbH

Anzeigen:
Simone Fischer
anzeigen@doag.org

Mediadaten und Preise:
http://www.doag.org/go/mediadaten

Druck:
adame Advertising and Media GmbH
www.adame.de

Titelfoto: © lotus_studio/ de.fotolia.com
Foto S.8: © AllebaziB / de.fotolia.com
Foto S. 15: © serrnovik / 123rf.com
Foto S. 26: © alphaspirit / 123rf.com
Foto S. 36: © Binkski / de.fotolia.com
Foto S. 40: © peshkova / de.fotolia.com

Inserentenverzeichnis

www.ijug.eu

JETZT ABO BESTELLEN

Sichern Sie sich 4 Ausgaben für 18 EUR

Für Oracle-Anwender und Interessierte gibt es das Java aktuell Abonnement auch mit zusätzlich sechs Ausgaben im Jahr der Fachzeitschrift DOAG News und vier Ausgaben im Jahr Business News zusammen für 70 EUR. Weitere Informationen unter www.doag.org/shop/

FAXEN SIE DAS AUSGEFÜLLTE FORMULA
0700 11 36 24 39

ODER BESTELLEN SIE ONLINE
go.ijug.eu/go/abo

Interessenverbund der Java User Groups e.V.
Tempelhofer Weg 64
12347 Berlin

Java aktuell

+++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN +++ AUSSCHNEIDEN +++

☐ **Ja**, ich bestelle das Abo Java aktuell – das IJUG-Magazin: 4 Ausgaben zu 18 EUR/Jahr

☐ **Ja**, ich bestelle den kostenfreien Newsletter: Java aktuell – der iJUG-Newsletter

ANSCHRIFT

Name, Vorname

Firma

Abteilung

Straße, Hausnummer

PLZ, Ort

GGF. ABWEICHENDE RECHNUNGSANSCHRIFT

Straße, Hausnummer

PLZ, Ort

E-Mail

Telefonnummer

X

Die allgemeinen Geschäftsbedingungen* erkenne ich an, Datum,

*Allgemeine Geschäftsbedingungen:
Zum Preis von 18 Euro (inkl. MwSt.) pro Kalenderjahr erhalten Sie vier Ausgaben der Zeitschrift "Java aktuell - das iJUG-Magazin" direkt nach Erscheinen per Post zugeschickt. Die Abonnementgebühr wird jeweils im Januar für ein Jahr fällig. Sie erhalten eine entsprechende Rechung. Abonnementverträge, die während eines Jahres beginnen, werden

Early Bird:

Tickets ab sofort verfügbar!

8. bis 10. März 2016 | im Phantasialand | Brühl bei Köln

Die Konferenz der Java-Community!

Präsentiert von: DOAG Deutsche ORACLE-Anwendergruppe e.V. | Heise Medien

Community Partner: iJUG Verbund

www.JavaLand.eu