JavaFX SwingNode: Migration von Java-Swing-Anwendungen ▸ 12

S&S

JAVA Mag

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 8.2013

Java™ magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Apache Solr 4.3
Reif für Cloud und Agile? ▸75

AWS Auto Scaling
Automatisches Skalieren in der Amazon Cloud ▸50

AeroGear
JBoss goes Mobile ▸26

W-JAX 13
Alle Infos hier im Heft!
▸ 51

Vaadin

… mit Clojure ▸ 45
… mit OSGi ▸ 30, 36

Android Studio

Was kann das neue Google-Tool? ▸ 108
JetBrains-Interview zur Partnerschaft mit Google ▸ 113

Apache Hadoop

DIY-Hadoop-Cluster mit Cubieboards ▸ 18
Cascading-Bibliothek für komplexe Hadoop-Jobs ▸ 14

4 194586 709801 08

Der große Wurf?

Kurz vor Redaktionsschluss wurde sie offiziell freigegeben: die Spezifikation für Java EE 7. Nach drei Jahren kann die Java-Community wieder auf einen neuen Enterprise-Standard zurückgreifen. Zwar wurde bekanntlich das große Thema „Cloud" ausgeklammert – zu unreif seien die Technologien für Provisioning, Multi-Tenancy und Elastizität in der Cloud derzeit noch, hatte Oracles Linda DeMichiel im September 2012 erklärt. Nichtsdestotrotz sind zahlreiche, teils seit langem überfällige Aktualisierungen vorgenommen und vier neue Komponenten integriert worden: Java-API for WebSocket 1.0 (JSR-356), Java-API for JSON Processing 1.0 (JSR-353), Concurrency Utilities for Java EE 1.0 (JSR-236) und Batch Applications for the Java Platform 1.0 (JSR-352).

Im Gespräch mit JAXenter nennt Oracles Anil Gaur (Vice President Software Development) drei Hauptthemen für das Java-EE-7-Release: HTML5-Support (beispielsweise im so genannten „HTML5-friendly JSF Markup"), die Steigerung der Entwicklerproduktivität (etwa durch den generalüberholten Java Messaging Service und die Verallgemeinerung des Annotationskonzepts) und die Umsetzung wichtiger Enterprise-Anforderungen (wie der von Spring Batch inspirierte und maßgeblich von IBM beigesteuerte JSR-352 für Batch-Anwendungen).

Fast zeitgleich mit der Freigabe der Java-EE-7-Spezifikationen wurden auch zwei reale Codeartefakte mit Java-EE-7-Support released: Zum einen die Java-EE-7-Referenzimplementierung GlassFish 4.0, die ab sofort auf einer neu aufgemachten Homepage zur Verfügung steht. Zum anderen wurde die NetBeans-Entwicklungsumgebung in ihrer Version 7.3.1 auf Java EE 7 eingestellt.

Apropos IDEs – wo NetBeans steht, ist meist auch Eclipse nicht fern. Und so kommt in Oracles offizieller Pressemitteilung Eclipse-Foundation-Direktor Mike Milinkovich zu Wort, der den vollen Java-EE-7-Support im kommenden Eclipse Kepler verspricht: „Eclipse Kepler will ship with Java EE 7 support, thanks to the hard work of the Eclipse Web Tools Project (WTP) team. Early builds are available today, with a final build expected by the end of June, making this the quickest we have ever supported a new Java EE platform release."

Der große Wurf?

Ist Java EE 7 nun der große Wurf, auf den wir gewartet haben? Nun, sicherlich hätte sich der eine oder andere ein „mutigeres" Release gewünscht. Beispielsweise kommentierte die London Java Community die JMS-2.0-Spezifikation mit den Worten, JMS 2.0 hätte durchaus einen ambitionierteren Scope haben können: „The LJC views the messaging space as one in which further standardisation is possible and desirable, and urges interested JCP members to explore possibilities in this space".

Doch immerhin kommt JMS 2.0 nach über neun Jahren der Stagnation zustande – so manch einer hätte das nicht mehr für möglich gehalten.

Wollten wir also eine abschließende Bewertung von Java EE 7 vornehmen, dann kommt mir das Prädikat „pragmatisch" als Erstes in den Sinn. Statt sich in kontroversen Diskussionen über einen Cloud-Standard zu verlieren, der ohnehin nur für die wenigsten relevant wäre, hat man die realistischen und von einem breiten Konsens getragenen Baustellen abgeschlossen und dafür gesorgt, dass die Java-Community nicht (ganz) den Anschluss an die aktuell so dynamische Webentwicklung verliert.

Und somit steht der schnellen Verbreitung der Java-EE-7-Plattform eigentlich nichts im Wege. Viele der sechzehn Java-EE-6-konformen Application Server werden es in den nächsten Monaten GlassFish gleichtun und die Unterstützung für Java EE 7 ins Programm aufnehmen. Selbst der große Enterprise-Konkurrent Spring hat angekündigt, in Spring 4, das übrigens noch in diesem Jahr kommen soll, Java EE 7 zu unterstützen. Ob die Erweiterungen in Java EE 7 allerdings Grund genug für Sie sind, Ihre bestehende Enterprise-Anwendung zu migrieren, müssen Sie selbst für sich entscheiden.

Auch wenn es manchmal etwas länger dauert: Oracle gelingt es, Releases abzuschließen. Diese Leistung gilt es zu würdigen. Und so schließe ich mit dem offiziellen Statement von Oracles Cameron Purdy (Vice President of Development, Oracle): „Java EE continues to be hugely popular, with continuing strong developer adoption and we're very excited about Java EE 7. This is a great release with strong technology updates that meets the demands of today's enterprises."

Viel Spaß beim Lesen des Java Magazins wünscht,

Hartmut Schlosser, Redakteur JAXenter

@JAXenter

Vaadin 7

Eine Plattform ist eine Anwendung, die nichts kann, aber alles möglich macht. Man muss allerdings nicht eclipse.org heißen, um eine Plattform bauen und betreiben zu können. Dass es auch anders geht, zeigt das Projekt Ripla. Dessen Ziel ist es, eine Plattform mit OSGi und Vaadin zu bauen. Ein anderes Projekt, lunifera.org, stellt eine Vaadin-7-OSGi-Bridge zur Verfügung. Außerdem zeigen wir, dass nichts näher liegt, als Vaadin 7 für die Oberfläche einer Clojure-Applikation zu verwenden.

Big Data für Groß und Klein

Denkt man an Big Data und Hadoop, hat man das Bild von endlosen Serverracks in überdimensionalen Rechenzentren vor Augen. Dass es auch ohne Serverpark geht, zeigt die Selbstbauanleitung für ein Hadoop-Cluster auf Basis des Raspberry-Pi-Klons Cubieboard. Außerdem werfen wir einen Blick auf die Java-Bibliothek Cascading für komplexe Hadoop-Jobs.

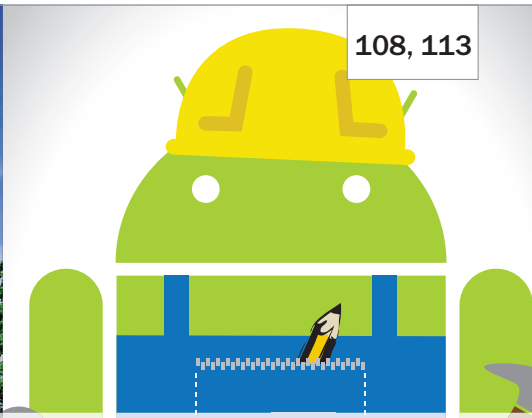
Leserbriefe und Feedback zu den Artikeln des Java Magazins bitte an redaktion@javamagazin.de.

AWS Auto Scaling

Beim Thema Skalierung betrachtet ein Entwickler häufig nur den Anwendungsaspekt. Die Infrastruktur ist aber keineswegs zu vernachlässigen, denn sie ist die Basis für eine Anwendungsskalierung. Erfahren Sie, wie sich mithilfe von Amazon EC2 automatisiert Infrastrukturressourcen anhand von Kriterien wie CPU-Last hinzu- bzw. abschalten lassen.

JavaFX-3D-API

Ab JDK 1.8 wird JavaFX um eine Dimension reicher: Es bekommt ein verbessertes 3-D-API, das es relativ einfach macht, sich virtuelle 3-D-Welten zu erschaffen und die eigene Applikation etwas aufzupeppen. Robert Ladstätter entwirft in seinem Beitrag ein Programm, das baumähnliche Strukturen auf den Bildschirm zeichnet – keine abstrakten, sondern tatsächliche Bäume.



Android Studio

Während der Google I/O Keynote kam überraschend Tor Norbye vom Android-Tools-Team auf die Bühne, um „die neue Android IDE: Android Studio“ vorzustellen. Wir sichten die bisher bekannten Informationen und werfen einen Blick auf „den Neuen“. In einem Interview gab uns Dmitry Jemerov von JetBrains Einblicke in das gemeinsame Projekt mit Google.

Frisch von JAXenter

News, Kommentare und Interviews von www.jaxenter.de

Red Hat startet PaaS-Angebot OpenShift Online

Die PaaS-Lösung von Red Hat „OpenShift Online“ hat die Betaphase verlassen und ist ab sofort als kommerzieller Dienst operabel. OpenShift Online basiert auf dem Open-Source-Projekt OpenShift Origin, das im Mai 2012 veröffentlicht wurde. Betrieben wird OpenShift Online auf den Amazon Web Services (AWS) und kommt mit Service-Support von Red Hat daher. Unterstützt werden mehrere Sprachen und Umgebungen, darunter Java, Ruby, PHP, Python, Node.js und Perl.

Ashesh Badani, Red Hats General Manager für OpenShift, kommentiert den neuen Service wie folgt: „As PaaS matures, enterprise developers want world-class support so they can focus less on system administration and more on coding. Our new OpenShift Online brings enterprise-grade services to public PaaS, and gives Red Hat the industry's most complete PaaS portfolio.“

Auf ZDnet kommt indes auch Konkurrent Sacha Labourey, CEO von CloudBees, zu Wort, der die Schwierigkeit betont, glaubhaft den Sprung vom „traditionellen“ Softwareanbieter zum Online-Service-Provider zu vollziehen: „We are very happy to welcome OpenShift to the fast-growing public PaaS market. It will be very interesting to see how they manage the challenging switch from being a traditional software vendor to becoming an online service provider. It's an entirely different business model.“

Dabei kann der OpenShift-Lösung ein gewisses Momentum in der Szene nicht abgesprochen werden. Angaben Red Hats zufolge wurden bereits über eine Million Anwendungen mit OpenShift entwickelt. OpenShift Online ist für einen Einstiegspreis von 20 US-Dollar pro Monat zu haben.

▶ https://www.openshift.com/products/online

Oracle reagiert auf Communitykritik: TZUpdate wieder kostenlos

Mit dem Update der Java-Standard-Edition 6 Anfang des Jahres verschwand ein kleines aber wichtiges Tool aus den JDKs. Statt die bisher kostenlose Version des Timezone Updaters (TZUpdate) nutzen zu können, sollten die User einen kostenpflichtigen Supportvertrag mit Oracle abschließen, was bei vielen auf großes Unverständnis stieß. Deshalb stellt Oracle das Tool ab sofort wieder gratis zur Verfügung.

Der TZUpdater ist ein wichtiger Bestandteil von Programmen, die auf korrekte und genaue Zeitangaben angewiesen sind. Mit dem Werkzeug lassen sich nämlich die Java-Zeitzonen mit der Referenzdatenbank TZ Database abgleichen.

Dass das TZUpdate nun wieder öffentlich zur Verfügung steht, ist nicht zuletzt der Community zu verdanken. Diese machte in einer regen Diskussion ihrem Unmut Luft, wie auch Noel Trout, der mit seiner Firma Programme für die private Luftfahrt entwickelt und somit auf genaue Zeitangaben an verschiedenen Flughäfen mit verschiedenen Zeitzonen angewiesen ist. Gerade kleine Firmen hätten unter dem Zwang zum kostenpflichtigen Supportprogramm zu leiden gehabt.

Oracles Henrik Stahl zufolge ging das Entfernen nicht konform mit der eigenen Update-Policy und sei somit als eine Art „Versehen“ zu werten. Selbst das Oracle-JDK 7 konnte ohne die TZUpdates-Komponente nicht mehr ohne Lizenzvertrag aktuell gehalten werden. Stahl entschuldigt sich deshalb für die „Verwirrung“ und dankt der Community, allen voran Zeit-Library-Spezialisten Stephen Colebourne, für die „technische Analyse“ aus Sicht der Community.

▶ http://bit.ly/13vCM4x

IntelliJ IDEA 12.1.4 erschienen

IntelliJ-IDEA-User dürfen sich über ein kleines Update ihrer Entwicklungsumgebung freuen. Mit IntelliJ IDEA 12.1.4 werden siebzig Issues behoben. Performanceprobleme in der Android-Komponente, fehlerhafte Shortcut-Zuweisungen für Ant-Targets und Formatierungsfehler in Groovy-Kommentaren gehören zu den ausgemerzten Schwachstellen. Außerdem gab es die Aktualisierung von Log4j und die Verbesserung der JavaFX-CSS-Analyse.

Das Update ist entweder über frühere 12.x-Versionen zugänglich oder steht auf der IntelliJ-IDEA-Webseite in Form eines Installer-Downloads zur Verfügung. Die Entwicklungsumgebung aus dem Hause JetBrains ist weiterhin in einer freien Community-Edition sowie in der kommerziellen Ultimate-Variante zu haben. Die Release-Notes zur Version klären über alle Fixes auf.

▶ http://bit.ly/15SLifZ

Spring Data Release Train Babbage überarbeitet Commons-Modul

Das Spring-Data-Projekt hat einen neuen Meilenstein erreicht. Beim jetzt verfügbar gemachten ersten „Service Milestone Release“ wurden vier Module erneuert: Spring Data Commons 1.6 M1, Spring Data JPA 1.4 M1, Spring Data MongoDB 1.3 M1 und Spring Data Neo4j 2.3 M1.

Der Fokus des Sammelreleases lag auf der Verbesserung des Spring-Data-Commons-Projekts, das die Basis für alle anderen Spring-Data-Module darstellt. Hier wurde auf die Querydsl-3.x-APIs aktualisiert. Die Mapping-Metadaten-Implementierung wurde im Hinblick auf Performanceoptimierungen überarbeitet; die MongoDB- und Neo4j-Module sollen dadurch Mapping-Operationen um 20 Prozent schneller erledigen.

Arbeit wurde auch in die Paginierung und den Websupport gesteckt, was sich vor allem bei der Kombination mit Spring HATEOAS bemerkbar machen soll.

Das vorliegende Spring-Data-Update wird im Rahmen des Release Trains „Babbage“ veröffentlicht. Der gemeinsame Releasezug soll dafür sorgen, dass Module für die verschiedenen NoSQL-Datenbankanbindungen für Spring auf einer gemeinsamen Grundlage entwickelt werden.

Für die nächste Zeit kündigt Oliver Gierke einige neue Featureerweiterungen an, darunter die Unterstützung des Aggregation-Frameworks in MongoDB, CDI-Integration im Neo4j-Modul sowie die Einbeziehung der Spring-Data-REST-Komponente in den Release Train.

▶ http://bit.ly/1bndU1S

Apache Tomcat mit Cross-Origin Resource Sharing

Alle vier bis sechs Wochen erfolgt ein neues Update von Apache Tomcat 7.0, der Open-Source-Variante des Java-Servlets. Dem jüngsten Update 41 liegen neben Bugfixes auch neue Features bei. Dazu zählt auch der Support für Cross-Origin Resource Sharing, einer sicheren Möglichkeit, Ressourcen unterschiedlicher Quellen zu verknüpfen. Außerdem wurde ein Bug im Anti-Resource-Locking-Feature behoben, bei dem Originalversionen von Webanwendungen gelöscht wurden. Und wer sein Deployment über Ant vollzieht, der darf sich über die Unterstützung des Version-Attributs freuen, das die Arbeit mit dem textbasierten Interface der Manageranwendung erlaubt.

Mehr Änderungen wurden im Changelog festgehalten. Zum Download gelangen Sie über die Seite von Apache.org. Wer noch auf eine der älteren Versionen 5.5 oder 6.0 setzt, der kann sich mit dem Migration Guide befassen und sein System auf den aktuellen Stand bringen.

▶ http://www.w3.org/TR/cors/

JSF-Bibliothek OmniFaces vereinfacht HTML-Messages

Die JSF-Komponenten-Library OmniFaces steht in Version 1.5 bereit. Zwei wichtige Erweiterungen hat das Projekt erhalten: Zum einen ist es nun möglich, JSF- und HTML-Code als Output-Format-Parameter anzugeben. Zum anderen wurden Escapable FacesMessages eingeführt, die an mehrere Komponenten gerichtet werden können und ein Rendering ohne Markup erlauben.

Für den letztgenannten Punkt wurde die Standard-JSF-Anweisung <*h:messages*> durch <*o:messages*> und die folgenden Optionen erweitert:

- Possibility to specify multiple client IDs space-separated in the for attribute.
- Control HTML escaping by the new escape attribute.
- Control iteration markup fully by the new var attribute which sets the current FacesMessage in the request scope and disables the default table/list rendering.

Der häufigen Anforderung, HTML in FacesMessages zu nutzen, kann man so recht einfach durch die Escape-Bedingung <*o:messages escape="false"*/> gerecht werden. Eine Liste der Änderungen und Bugfixes in OmniFaces 1.5 finden Sie unter http://bit.ly/13z0uP3.

Autor des Monats

Florian Pirchner ist selbstständiger Softwarearchitekt, lebt und arbeitet in Wien. Aktuell beschäftigt er sich als Project Lead in einem internationalen Team mit dem Open-Source-Projekt „Iunifera.org – OSGi-services for business applications". Auf Basis von Modellabstraktionen und der Verwendung von OSGi-Spezifikationen soll ein hoch erweiterbarer und einfach zu verwendender Kernel für Business-applikationen geschaffen werden.

Wie bist du zur Softwareentwicklung gekommen?

Ich habe schon früh damit angefangen. Bereits am Commodore 64 habe ich meine ersten kleinen Applikationen geschrieben. Damals hatte ich mich noch damit zufrieden gegeben den Cursor per Joystick am Bildschirm zu bewegen. Dann ist eins zum anderen gekommen und nun bin ich Entwickler aus Leib und Seele.

Was ist für dich der schönste Aspekt in der Softwareentwicklung?

Die Abwechslung. Kein Projekt ist wie das andere. Immer kommen neue Technologien auf den Markt. Eclipse, Apache, JBoss ... sind im Open-Source-Bereich ja sehr engagiert. Es ist super spannend, immer wieder aufs Neue lernen zu können.

Was ist für dich ein weniger schöner Aspekt?

Wie in jedem anderen Job auch gibt es Tage, die sich hin ziehen. Allerdings sehe ich das gelassen, denn der nächste Tag kommt bestimmt.

Wie und wann bist du auf Java gestoßen?

Lange habe ich erstmal RPG auf der iSeries im ERP-Bereich programmiert. In meiner ehemaligen Firma wurde dann zusätzlich eine Java-basierte ERP-Lösung vertrieben und das war mein erster beruflicher Kontakt mit Java. In die Open-Source-Welt hat mich Ekkehard Gentz gebracht. Durch ihn lernte ich Open Source kennen und schätzen.

Wenn du für einen Tag König der Java-Welt wärst, was würdest du verändern?

Ich bin ein großer Verfechter der liberalen Softwarelizenzen wie Apache License, Eclipse Public License. Ich persönlich entwickle jede Software nur auf Basis der APL oder EPL. Gerne würde ich mir wünschen, dass die GPL auch ein wenig offener wird, da es mir unmöglich ist, GPL-lizenzierte Projekte in meine Open-Source-Projekte zu integrieren.

Was ist zurzeit dein Lieblingsbuch?

Mit Romanen schaut es bei mir eher schlecht aus; wenn überhaupt, dann lese ich sie nur im Urlaub. Zu 95 Prozent besteht meine Lektüre aus Fachbüchern. Das aus meiner Sicht beste Buch, das ich je gelesen habe, ist „OSGi in Depth" aus dem Manning-Verlag.

Was machst du in deinem anderen Leben?

Wenn ich Zeit habe, dann liebe ich es, zu gärtnern. Angefangen von Orangenbäumen bis hin zu Liebstöckl. Es ist sehr entspannend, Zeit in das Wohlergehen der Pflanzen zu investieren. Dafür sind sie dann auch sehr schön grün, blühen und tragen viele Früchte.

JUG-Kalender* Neues aus den User Groups

WER?	WAS?	WO?
JUG **Stuttgart**	03.07.2013 – Workshop „Java für Entscheider"	http://jugs.org
JUG **Stuttgart**	04.07.2013 – 16. Java-Forum Stuttgart	http://jugs.org
JUG **Saxony**	05.07.2013 – HTML5 aus Oracle-Sicht	http://jugsaxony.org
JUG **Hessen**	10.07.2013 – Web Development – You're doing it wrong	http://jugh.de
JUG **Düsseldorf**	11.07.2013 – JavaFX – Swing war gestern	http://rheinjug.de
JUG **Darmstadt**	18.07.2013 – Continuous Documentation mit Maven	http://jugda.wordpress.com
JUG **Augsburg**	18.07.2013 – Ruby/Rails	http://www.jug-augsburg.de
JUG **Frankfurt**	31.07.2013 – Fehlervermeidung in Java durch generierte Constraint-Klassen	http://jugf.de
JUG **Ostfalen**	15.08.2013 – Spring MVC Integration Testing	http://jug-ostfalen.de
JUG **Ostfalen**	29.08.2013 – Gute Zeilen, schlechte Zeilen – Regeln für wartbare Programme	http://jug-ostfalen.de

*Alle Angaben ohne Gewähr. Da Termine sich kurzfristig ändern können, überprüfen Sie diese bitte auf der jeweiligen JUG-Website.

REST und HTTP

von Stefan Tilkov

Das vorliegende Buch von Stefan Tilkov liegt nun schon seit zwei Jahren in der 2. Auflage vor. Dennoch hat es nicht an Aktualität verloren, denn als Grundlagenwerk ist es zeitlos. REST ist ein Architekturstil, bei dem mittels eindeutiger Bezeichner (URI) und HTTP-Verben, insbesondere *get*, *post*, *put* und *delete*, auf Ressourcen zugegriffen wird. Eine Ressource ist dabei ein Teil einer Anwendung, ein Service. Und die Navigation innerhalb der Anwendung erfolgt mittels Hyperlinks. Das ganze Web funktioniert so: Im Browser sieht der Anwender Inhalt und Information, nämlich Links, wie es weitergeht. Wenn statt eines Menschen eine Applikation dieses Prinzip nutzt, entstehen so beliebige Anwendungen. Was sich recht abstrakt anhört, erläutert der Autor sehr anschaulich und in einer Weise, die einen als Leser die Begeisterung des Autors für diesen Architekturstil spüren lässt und stellenweise regelrecht mitreißt – so man sich mitreißen lässt. Denn wer hier Rezepte für eine bestimmte Programmiersprache, sei es Java, C# o. Ä. sucht, der wird kaum fündig werden. Als Grundlagenwerk kommt das Buch abstrakter daher; es bewegt sich rein auf der HTTP-Ebene – aber dennoch mit starkem Praxisbezug.

Was bietet Stefan Tilkov nun konkret? In der Einleitung klärt er erst einmal die Frage: Warum überhaupt REST? Hier geht es um lose Kopplung, Interoperabilität, Wiederverwendung und Performance. An dieser Stelle gibt es kaum mehr als Schlagworte, doch erfährt der Leser im Laufe des Buchs mehr. Ganz kurz geht er auf die Geschichte von REST ein, um dann die Grundprinzipien zu erläutern. Hier kommt auch HTTP als Basis mit ins Spiel. Mittels einer Fallstudie, die er später noch einmal aufgreift und erweitert, stellt Tilkov den Bezug zur Praxis her. Es folgen ausführliche Erläuterungen zu Themen wie Ressourcen, den oben erwähnten Verben und ihrer Bedeutung für das Web im Allgemeinen und REST-Applikationen im Speziellen. Repräsentationsformate wie XML, CSV, JSON, RSS, Atom und mehr beleuchtet er im Hinblick auf ihre Verwendung für REST und schließt gleich noch eine Fallstudie zu AtomPub an. Weitere Themen sind Sicherheit, Caching, Sessions und, nicht zu vergessen, die Dokumentation. Schließlich geht er noch auf die Architektur REST-basierter Applikationen und REST als SOA ein – nicht ohne dies mit SOAP und WS-* zu vergleichen.

Neben Abbildungen, Tabellen und HTTP-Listings sind in dem Band keine konkreten Codebeispiele zu finden. Es ist also nicht unbedingt ein Buch für diejenigen unter uns, die stets mit der Tastatur in der Hand lesen und sofort nachvollziehen möchten. Allerdings ist es locker und interessant geschrieben, und es werden die Grundlagen für den späteren Praxiseinsatz gelegt. In welcher Sprache dann die konkreten Applikationen erstellt werden, spielt eine untergeordnete Rolle. Schließlich geht es erst einmal darum, die Prinzipien hinter REST zu verstehen. Und das dürfte nach dieser Lektüre der Fall sein.

Michael Müller

Stefan Tilkov

REST und HTTP

Einsatz der Architektur des Web für Integrationsszenarien

266 Seiten, 36,90 Euro
dpunkt, 2011
ISBN 978-3-89864732-8

Dart in Action

von Chris Buckett

In der Dart-Szene ist Chris Buckett mittlerweile bekannt wie ein bunter Hund. Stunden, nachdem Google die Verfügbarkeit der neuen Sprachen bekanntgegeben hatte, gründete er seinen Blog dartwatch.com, in dem er sich ausschließlich mit Dart beschäftigt. Der Manning-Verlag wurde auf den überaus aktiven Entwickler aufmerksam, und ein Jahr später erschien „Dart in Action".

Überhaupt ein Buch zu schreiben erfordert eine Menge Arbeit und Geduld. Zudem ist Dart eine Art bewegliches Ziel. Die Dart-Entwickler wollen eine moderne Sprache erschaffen und nehmen (noch) keine Rücksicht auf „Abwärtskompatibilität". Im letzten Jahr blieb kein Stein auf dem anderen. Die ständigen Änderungen hätten durchaus dazu führen können, dass dieses Buchprojekt im Sande verläuft. Doch Chris Buckett blieb eisern und schrieb große Teile davon immer wieder um.

Was man in den Händen hält, ist das Buch von jemandem, der die Sprache und auch deren Entwicklung in und auswendig kennt. Das merkt man deutlich, wenn man es liest. Es wirkt, als würde sich der Autor schon seit zehn Jahren mit dem Thema beschäftigen und eine kurze Zusammenfassung seines Dart-Wissens in diesem Buch präsentieren.

„Dart in Action" ist nichts für „Sprachtheoretiker" oder jemanden, der eine Art Referenz sucht. Es ist ein Buch für den, der Dart möglichst schnell einsetzen möchte; eine praktische Einführung, die zeigt, was es alles in dem bereits sehr großen Dart-Universum gibt. Buckett zeigt, was man kennen muss, sei es den Dart-Editor oder auch, wie man mit Abhängigkeiten und „Pub", dem Dart-Package-System, arbeitet.

Das Gute daran ist, dass man kaum merkt, wie schnell man sich an das neue Ökosystem gewöhnt. Die Sprache ist angenehm und leicht zu lesen, aber nie langweilig. Buckett blickt über den Tellerrand, verrennt sich aber nicht in ausufernden Trivialitäten. Er wird nie belehrend, sondern bleibt der nette Kollege von nebenan, der einem sein neuestes Spielzeug zeigt. Er vergisst dabei nicht, an den richtigen Stellen grundsätzliche Best Practices zu erwähnen, auch unabhängig von Dart. Gerade für jüngere Entwickler sind das großartige Hilfestellungen.

Der Inhalt ist komplett und ausgewogen. Es werden die wichtigsten Kernkonzepte von Dart erklärt, auch „Futures" und „Completers". Letzteres ist nicht nur wichtig, sondern auch eine Art Standard, wenn man für Browser entwickeln will. Anders als andere Autoren erklärt Buckett sogar, wie man asynchronen Code automatisiert testen kann.

Es geht weiter mit einer Einführung in „Single Page"-Anwendungen. Das sind Anwendungen, die nur aus einer HTML-Seite bestehen, wie G-Mail. Neben den üblichen „Cookies" wird auch auf Browser-Datenbanken eingegangen oder darauf, wie man seine App in den Chrome Webstore bekommt – Themen, die nicht gerade an jeder Straßenecke besprochen werden.

Auch serverseitiges Dart kommt nicht zu kurz. Dart kann nämlich das Apache CouchDB Interface nutzen, um Daten zu speichern und auch – ähnlich Node.js – HTTP-Anfragen zu beantworten. Wer dann noch das letzte Kapitel über „Isolates" liest, der weiß dann auch, wie man nebenläufig in Dart programmiert.

Insgesamt ist es ein sehr gelungenes Buch. Obwohl manche der Inhalte bereits wieder veraltet sein dürften, ist es lesens- und kaufenswert. Vieles bleibt eben gleich, auch wenn sich die Details ändern. Das Buch macht Spaß und motiviert, sofort selbst ein paar Zeilen Code in Dart zu schreiben. Für andere Autoren wird es schwer werden, „Dart in Action" zu übertreffen.

Christian Grobmeier

Chris Buckett

Dart in Action

424 Seiten, Print 44,99 Dollar,
E-Book 35,99 Dollar
Manning Publications Co., 2013
ISBN 978-1617290862

Swing Controls in JavaFX umschreiben

JavaFX SwingNode

Oracle hat die Stimmen aus der Entwicklergemeinde erhört und stellt ab sofort mit SwingNode eine Node im JavaFX Scene Graph zur Verfügung, die die Migration von Java-Swing-Anwendungen nach JavaFX erheblich erleichtern soll. Ein kurzer Überblick.

von Gerrit Grunwald

Da JavaFX langsam auch für Non Early Adopter interessant wird, stellt sich die Frage der Migration von bestehenden Java-Swing-Anwendungen nach JavaFX. Bislang gab es als Antwort lediglich das JFXPanel. Dabei handelt es sich um die Möglichkeit, JavaFX in Java-Swing-Anwendungen zu integrieren. Das JFXPanel verhält sich dabei wie ein JPanel für Swing, kann jedoch eine JavaFX Scene darstellen. Diese Möglichkeit ist zwar interessant, aber leider nicht das, was sich die Entwickler gewünscht hätten. Üblicherweise bestehen die Swing-Anwendungen im Feld aus einer Vielzahl eigener, sehr komplexer Controls, die man nur sehr ungern auf einmal in JavaFX umschreiben möchte. Für eine „sanfte“ Migration wünscht man sich eher den umgekehrten Weg, sprich: Man möchte gerne eine neue JavaFX-Anwendung aufsetzen und dann die mühsam erstellten Swing-Komponenten in JavaFX integrieren. Mit diesem Ansatz ist es dann möglich, langsam die bestehenden Swing Controls in JavaFX umzuschreiben.

Glücklicherweise hat Oracle hier auf die Nachfrage der Entwicklergemeinde gehört und diesen Migrationspfad mit der so genannten SwingNode zur Verfügung gestellt. Diese stellt eine Node im Scene Graph dar, die die Fähigkeit hat, Swing-Komponenten auf der Basis von JComponent zu rendern.

Bei beiden Migrationspfaden muss man aber immer bedenken, dass man es hier mit zwei unabhängigen Rendering-Threads zu tun hat, die synchronisiert sein wollen. JavaFX kommt hier mit dem FX Application Thread und Swing mit dem Event Dispatch Thread daher. Da beide Threads vollkommen unabhängig voneinander sind, muss der Entwickler dafür sorgen, dass Interaktionen zwischen ihnen über die entsprechenden Methoden in den jeweiligen Thread „eingefüttert“ werden. In JavaFX verwendet man hierzu die Methode

```
Platform.runLater(new Runnable() {
  @Override public void run() {
    // Your code here
  }
});
```

und in Java Swing

```
SwingUtilities.invokeLater(new Runnable() {
  @Override public void run() {
    // Your code here
  }
});
```

Als einfaches Beispiel soll nun in Listing 1 ein Java Swing JButton in eine JavaFX SwingNode eingebettet werden und durch Drücken einen Text in einem JavaFX Label umschalten.

Schon beim Blick auf dieses sehr einfache Beispiel lässt sich erahnen, dass es kein einfaches Unterfangen ist, eine bestehende Swing-Anwendung nach JavaFX zu portieren. Man muss dabei sehr genau darauf achten, dass man keine Threading-Probleme bekommt und sich immer im richtigen Thread befindet. Wer also nun erwartet hat, dass mit dem Einzug der SwingNode ins JDK 8 (verfügbar im so genannten Developer Preview seit b89 und größer) alles viel einfacher wird, wird feststellen, dass sich zwar prinzipiell Swing-Komponenten in JavaFX einbinden lassen, man jedoch immer noch vor dem Problem der Thread-Synchronisierung steht, was man nicht unterschätzen sollte.

Als Fazit kann man festhalten, dass von Oracle nun beide Möglichkeiten der Migration zur Verfügung gestellt werden (JavaFX in Swing mittels JFXPanel und Swing in JavaFX mittels SwingNode) und somit nun jeder selbst wählen kann, was für ihn der beste Weg ist. Man sollte aber beachten, dass SwingNode momentan noch am Anfang steht. Auf die erste stabile Version werden wir aber sicherlich nicht lange warten müssen.

Gerrit Grunwald ist passionierter Java-Entwickler und beschäftigt sich bevorzugt mit JavaFX, Swing und HTML5. Dabei liegt sein Hauptaugenmerk auf der Entwicklung von Controls. Des Weiteren ist er Mitgründer und Leader der Java Usergroup Münster sowie Community-Co-Lead der JavaFX-Community.

Listing 1

```
public class Migration extends Application {
  private Label   javaFxLabel;
  private Button  javaFxButton;
  private JLabel  swingLabel;
  private JButton swingButton;

  @Override public void init() {
    javaFxLabel  = new Label("No Button pressed");
    javaFxButton = new Button("click JavaFX");
    javaFxButton.setOnAction(new
        EventHandler<javafx.event.ActionEvent>() {
      @Override public void handle(javafx.event.
                          ActionEvent actionEvent) {
        javaFxLabel.setText("JavaFX Button
                                        pressed");
        SwingUtilities.invokeLater(new Runnable() {
          @Override public void run() {
            swingLabel.setText("JavaFX Button
                                        pressed");
          }
        });
      }
    });
  }

  @Override public void start(Stage stage) {
    final SwingNode swingNode = new SwingNode();

    // Call on Event Dispatch Thread
    SwingUtilities.invokeLater(new Runnable() {
      @Override public void run() {
        swingLabel = new JLabel("no Button
                                        pressed");

        swingButton = new JButton("click Swing");
        swingButton.addActionListener(new
                                  ActionListener() {
          @Override public void
                    actionPerformed(ActionEvent e) {
            swingLabel.setText("Swing Button
                                        pressed");
            // Call on FX Application Thread
            Platform.runLater(new Runnable() {
              @Override public void run() {
                javaFxLabel.setText("Swing Button
                                        pressed");
              }
            });
          }
        });
        JPanel panel = new JPanel();
        panel.add(swingLabel);
        anel.add(swingButton);
        panel.setVisible(true);
        swingNode.setContent(panel);
      }
    });

    StackPane pane = new StackPane();
    pane.setPrefSize(300, 120);
    // Uncomment if NOT on OS X due to rendering
    // problems with SwingNode
    //swingNode.setTranslateY(-40);

    HBox fxControls = new HBox();
    fxControls.setSpacing(10);
    fxControls.getChildren().addAll(javaFxLabel,
                                        javaFxButton);

    fxControls.setTranslateY(40);
    pane.getChildren().addAll(swingNode,
                                        fxControls);

    Scene scene = new Scene(pane, 300, 120);

    stage.setScene(scene);
    stage.show();
    stage.setTitle("Swing Migration");
  }

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

Big Data mit Cascading

Elefantenzähmen leicht gemacht

Cascading ist eine Java-Bibliothek, die das Schreiben von komplexen Hadoop-Jobs zum Kinderspiel macht. Cascadings Abstraktion von Big Data Streams zu Rohrleitungssystemen schafft Transparenz und erleichtert die Modularisierung und das Testen der einzelnen Programmkomponenten. Also warum nicht zur Abwechslung aus einem Elefanten eine Mücke machen?

von Boris von Loesch und Peter Siemen

Das Thema Big Data hat sich in den letzten Jahren zu einer festen Größe der Industrie entwickelt. Mit den stetig sinkenden Preisen für Speichermedien ist es immer selbstverständlicher für Unternehmen geworden, großzügig alle Signale aufzuzeichnen, die Aufschluss über Kunden, über die Effizienz von Unternehmensprozessen oder die Performance von eingesetzten Softwaremodulen geben können.

Um Datenmengen in Größenordnungen von Terabytes zu verarbeiten, hat sich das 2005 von Doug Cutting und Mike Cafarella entwickelte Softwarepaket Apache Hadoop [1] als De-facto-Standard bewährt. Für einige Stunden am Tag ein Hadoop Cluster in der Cloud von einem der diversen Anbieter zu mieten ist inzwischen einfacher geworden, als ein Outlook-Adressbuch mit einem Android-Telefon zu synchronisieren.

Ein einfaches Hadoop-Programm wie zum Beispiel das gern verwendete „Hello World"-Äquivalent – in der Hadoop-Welt: *WordCount* – zu schreiben und in der Cloud laufen zu lassen, stellt ebenfalls für die wenigsten Programmierer eine ernst zu nehmende Hürde dar. Sobald die Anforderungen an ein Hadoop-Programm komplexer werden, wird es jedoch schnell sehr aufwändig und mühsam, an jeden Berechnungsschritt im MapReduce-Paradigma zu denken. Chris Wensel sah diese Problematik sehr früh und startete 2008 das Open-Source-Projekt Cascading [2].

Cascading: Elefantenzähmen leicht gemacht

Cascading ist eine anwendungsorientierte Abstraktionsschicht auf Hadoop bzw. MapReduce mit dem Ziel, komplexe Hadoop-Jobs transparenter zu machen. Es stellt diverse häufig gebrauchte Komponenten bereit und lässt sich wesentlich leichter testen und warten als die resultierenden einzelnen MapReduce-Jobs. Bei Cascading werden die Datenverarbeitungsschritte als Flows modelliert. Ein Flow setzt sich aus einer oder mehreren Pipes (*cascading.pipe.Pipe*) zusammen. Innerhalb der einzelnen Pipes können Operationen zur Manipulation des Datenstroms konfiguriert bzw. implementiert werden. Input- bzw. Outputkanäle werden als *Source* und *Sink Taps* bezeichnet.

Alle durch das System zu verarbeitenden Daten werden als Tuple zu Tuple Streams zusammengefasst. Ein Tuple wiederum besteht aus einer beliebigen Anzahl von serialisierbaren Objekten, den *Fields* des Tuples.

Über ein gewähltes Key *Field* kann ein Tuple Stream gruppiert werden. Es gibt daher zwei grundlegende Arten von Operationen: Solche, die auf einzelne Tuple eines Tuple Streams angewandt werden (*cascading.operation.Function, cascading.operation.Filter*) und andere, die auf Gruppen von Tuples arbeiten (*cascading.operation.Aggregator, cascading.operation.Buffer*).

Pipes: Gas, Wasser, Elektrik für die Cloud

Wie schon erwähnt, werden Operationen innerhalb definierter Pipe-Abschnitte ausgeführt. Die verschiedenen Abschnitte sind als Spezialisierungen der Pipe-Klasse implementiert. Es gibt die *Each-*, *Every-*, *GroupBy-*, *HashJoin-*, *CoGroup-* und *Merge*-Spezialisierung.

Innerhalb eines *Each*-Pipe-Abschnitts können Funktionen oder Filter auf einzelne Tuples angewandt werden. In einem *Every*-Pipe-Abschnitt können Aggregatoren oder Buffer auf Gruppen von Tuples angewendet werden.

In einem *GroupBy*-Pipe-Abschnitt wird ein Tuple Stream über ein gewähltes *Field* gruppiert. Die Ausgabe eines *GroupBy*-Pipe-Abschnitts wird daher typischerweise von einem *Every*-Pipe-Abschnitt gelesen werden, nicht aber von einem *Each*-Pipe-Abschnitt.

Mit der *HashJoin*-, *CoGroup*- und *Merge*-Pipe können mehrere Pipes zusammengefügt werden. *HashJoin* und *CoGroup* implementieren beide einen Join über eines oder mehrere Fields. *HashJoin* ist die performantere Variante, mit der Einschränkung, dass die rechte Seite des zu joinenden Tuple Streams komplett im Hauptspeicher gehalten wird. Die *CoGroup*-Pipe verzichtet auf diese Performanceoptimierung und kann daher auf beliebig große Tuple Streams angewandt werden. Mit der *Merge*-Pipe können zwei Tuple Streams mit den gleichen Fields zu einem Stream zusammengefasst werden.

Funktionen, Filter, Aggregatoren

Cascading liefert ein Füllhorn einzelner Funktionen, Filter und Aggregatoren, mit deren Hilfe komplexe Datenverarbeitungs-Pipelines zusammengesetzt werden können. Einige davon werden wir im nächsten Abschnitt kennenlernen, wenn es darum geht, die Hörgewohnheiten von Internetradionutzern zu analysieren.

Cascading Pipelines testen

Cascading liefert erstklassige Methoden zum Testen von Pipeline-Komponenten frei Haus. Durch Erben von *cascading.CascadingTestCase* stehen unter anderem die Funktionen *invokeFunction()*, *invokeFilter()*, *invokeAggregator()* und *invokeBuffer()* zur Verfügung. Die ge„sink"ten Tuple-Ergebnisse der Cascading-Operatoren können dann z. B. mit JUnit gegen die Erwartungswerte getestet werden. Wir haben in Listing 1 ein solches Beispiel aufgeführt.

Sag mir, was du hörst, und ich sage dir, wer du bist

2011 veröffentlichte die Firma Echo Nest in Zusammenarbeit mit LabROSA an der Columbia University den „One Million Song Dataset", der umfangreiche Informationen zu einer Million Songs zusammenträgt. Teil dieses Datensatzes ist eine anonymisierte *tsv*-Datei, die die Hörgewohnheiten von mehr als einer Millionen User abbildet [3]. Dieser Datensatz soll als Grundlage dienen, um die Fähigkeiten von Cascading anhand kleiner Beispiele zu illustrieren.

Hits, Hits, Hits

Das Programm in Listing 2 erstellt eine Hitparade, indem für jeden Song die User gezählt werden, die diesen Song mindestens zweimal gehört haben. Um das Pro-

Listing 1

```
@Test
public void testEmittedSongPairs() {
  Fields fields = MSDReco.USER_SONG_COUNT;
  TupleEntry[] argumentsArray = new TupleEntry[] {
    new TupleEntry(fields, new Tuple( "userid-1", "songid-1", "1")),
    new TupleEntry(fields, new Tuple("userid-1", "songid-2", "1")) };

  TupleListCollector collector = invokeBuffer(new MSDReco.PairEmitter(),
                                              argumentsArray, Fields.ALL);

  Iterator<TupleEntry> iter = collector.entryIterator();
  Set<Tuple> pairs = new HashSet<Tuple>();
  while (iter.hasNext()) {
    pairs.add(iter.next().getTuple());
  }

  Assert.assertTrue(pairs.contains(new Tuple("songid-1", "songid-2")));
  Assert.assertTrue(pairs.contains(new Tuple("songid-2", "songid-1")));
  }
```

Listing 2

```
public class Hitparade {
  public static Fields USER_SONG_COUNT = new Fields("userId", "songId",
                                                    "count");
  public static Class[] USER_SONG_COUNT_CLASSES = new Class[] {String.
class, String.class, Long.class};
  public static Fields SONG_COUNT = new Fields("songId", "count");

  public static void main(final String[] args) {
    String trainTriplets = args[0];
    String outputPath = args[1];

    Tap userActionLogsSource = new Hfs(new TextDelimited(USER_SONG_
            COUNT, false, "\t", USER_SONG_COUNT_CLASSES), trainTriplets);

    Tap hitlistSink = new Hfs(new TextDelimited(SONG_COUNT, false, "\t"),
    outputPath, SinkMode.REPLACE);

    Pipe userActionLogsPipe = new Pipe("user-song-input");
    userActionLogsPipe = new Retain(userActionLogsPipe, SONG_COUNT);
    userActionLogsPipe = new Each(userActionLogsPipe, new
                              ExpressionFilter("count >= 2", Integer.class));
    userActionLogsPipe = new GroupBy(userActionLogsPipe, new
                                                    Fields("songId"));

    Pipe hitlist = new Every(userActionLogsPipe, new Count());
    hitlist = new GroupBy(hitlist, new Fields("count"), true);

    FlowDef flowDef = new FlowDef()
    .addSource(userActionLogsPipe, userActionLogsSource)
    .addTailSink(hitlist, hitlistSink);

    Properties properties = new Properties();
    AppProps.setApplicationJarClass(properties, Hitparade.class);

    Flow flow = new HadoopFlowConnector(properties).connect(flowDef);
    flow.complete();
  }
}
```

gramm mit Hadoop zu starten, genügt es, eine JAR-Datei (*MSD.jar*) mit allen Abhängigkeiten zu erstellen und (nachdem die Datei *train_triplets.txt* auf das HDFS kopiert wurde) das Programm via

```
hadoop jar MSD.jar Hitparade <path>/train_triplets.txt <outputPath>
```

auf dem Hadoop Cluster laufen zu lassen. Sollten Sie gerade keinen Hadoop Cluster zur Hand haben, können Sie die *Main*-Methode auch direkt mit Java starten; Cascading verwendet in diesem Fall automatisch Hadoop im *local mode*.

Die Daten sind in einer Textdatei gespeichert, wobei jede Zeile einem Eintrag entspricht. Die Felder *userId*, *songId* und die Anzahl der Abspielungen sind mittels Tabulator getrennt. Beim Definieren des Source Taps kann deshalb auf das *TextDelimited*-Schema zurückgegriffen werden, das pro Zeile einen Datensatz erwartet. Neben dem Trennzeichen müssen zusätzlich nur die in der Datei enthaltenen *Fields* angegeben werden. Auf die gleiche Weise wird das Ausgabeformat des Sinks definiert.

Als Nächstes wird der Verarbeitungsprozess beschrieben: Mithilfe des *Retain*-Operators werden nur die für diese Aufgabe interessanten Felder *songId* und *count* behalten. Um die Songs zu verwerfen, die ein User nur einmal gehört hat, wird ein *Expression*-Filter verwendet. Dieser bietet die Möglichkeit, einfache Filterkriterien direkt als String zu definieren. Für die Auswertung verwendet Cascading den Janino-Compiler, um die Expression beim Start des Programms in Java-Bytecode zu übersetzen.

Denkt man an das MapReduce-Paradigma, würden typischerweise beide Schritte in einem Mapper ausgeführt werden. Das Gruppieren der Einträge nach *songId* und das Zählen der Abspielungen sind hingegen Aufgabe eines Reducers. In Cascading entspricht ein Reducer der Anwendung einer *GroupBy*- und anschließend einer *Every*-Operation mit einem *Buffer* oder einem *Aggregator*. Im Beispiel wurde der *count*-Aggregator verwendet, um die Anzahl der Hörer pro Song zu zählen.

Alle Einzelteile, also Zulauf, das Rohrsystem und der Abfluss sind angelegt. Es muss nur noch definiert werden, wie die Teile zusammengestöpselt werden. Dies erledigt man bei Cascading mit der Definition eines *FlowDef*. Die eigentliche Arbeit, nämlich das Übersetzen der gesamten Pipeline in MapReduce-Schritte und das Ausführen mit Hadoop, erledigt schließlich ein *FlowConnector*.

Nach der Ausführung des Programms erhalten wir unsere Hitparade in Form einer Liste aller Song-IDs und der Anzahl der User, die den Song mindestens zweimal gehört haben.

Durch das Einfügen eines Joins mit der Liste der Titel kann die Ausgabe um den Titel erweitert werden. Cascading stellt hierzu den *CoGroup*- und den *HashJoin*-Operator zur Verfügung.

In unserem Fall ist die Songliste so klein, dass wir den effizienteren *HashJoin* verwenden können.

Für die Songdaten, die auch auf der Webseite des One Million Song Dataset verfügbar sind, müssen wir nun noch ein zusätzliches Source Tap und eine Eingangspipe anlegen. In Cascading ist dies mit wenigen Zeilen erledigt:

```
Tap songDataSource = new Hfs(new TextDelimited(TRACKID_SONGID_
                                ARTIST_NAME, false, "<SEP>"), songDataPath);
Pipe songDataPipe = new Pipe("song-data-input");
hitlist = new HashJoin(hitlist, new Fields("songId"), songDataPipe,
   new Fields("songId"), new Fields("songId", "count", "trackId", "songId2",
                                                    "artist", "songName"));
```

Listing 3: Pair Emitter Buffer

```
public static class PairEmitter extends BaseOperation<Tuple> implements Buffer<Tuple> {
  public PairEmitter() {
    super(new Fields("songId1", "songId2"));
  }

  public void prepare(FlowProcess flowProcess, OperationCall<Tuple> operationCall) {
    operationCall.setContext(Tuple.size(2));
  }

  public void operate(FlowProcess flowProcess, BufferCall<Tuple> bufferCall {
    Iterator<TupleEntry> songsListen = bufferCall.getArgumentsIterator();
    List<TupleEntry> songsListenList = new LinkedList<TupleEntry>();
    while (songsListen.hasNext()) {
      songsListenList.add(new TupleEntry(songsListen.next()));
    }
    Tuple tuple = bufferCall.getContext();
    for (TupleEntry song1 : songsListenList) {
      String song1Id = song1.getString("songId");
      for (TupleEntry song2 : songsListenList) {
        String song2Id = song2.getString("songId");
        if (!song1Id.equals(song2Id)) {
          tuple.set(0, song1Id);
          tuple.set(1, song2Id);
          bufferCall.getOutputCollector().add(tuple);
        }
      }
    }
  }
}
```

Listing 4

```
Pipe userActionLogsPipe = new Pipe("user-song-input");
userActionLogsPipe = new GroupBy(userActionLogsPipe, new Fields("userId"));
Pipe songPairPipe = new Every(userActionLogsPipe, new PairEmitter(), Fields.RESULTS);
Pipe maxCount = new CountBy(songPairPipe, new Fields("songId1", "songId2"),
                                                    new Fields("count"));
maxCount = new GroupBy(maxCount, new Fields("songId1"), new Fields("count"), true);
maxCount = new Every(maxCount, new First(), Fields.RESULTS);
```

Die Ausgabefelder wurden hier explizit angegeben. Dies ist notwendig, da das Feld *songId*, welches in beiden Pipes definiert ist, sonst doppelt vorkäme, was bei Cascading nicht erlaubt ist. In den Songdaten gibt es teilweise mehrere Einträge zu einer *songId*, z. B. wenn der Titel auf unterschiedlichen Alben vorkam. Nach dem Join führt dies teilweise auch zu doppelten Einträgen in der Hitliste. Glücklicherweise stellt Cascading mit dem *Unique*-Operator eine Funktion zur Verfügung, um dieses Problem zu lösen. Wir ergänzen also noch:

```
hitlist = new Unique(hitlist, new Fields("songId"));
```

Am Ende müssen wir dem *FlowConnector* nur noch angeben, dass die Pipe „song-date-input“ mit dem Tap *songDataSource* verbunden werden soll, indem wir eine weitere Source hinzufügen:

```
FlowDef flowDef = new FlowDef()
  .addSource(userActionLogsPipe,userActionLogsSource)
  .addSource(songDataPipe, songDataSource)
  .addTailSink(hitlist, hitlistSink);
```

Wenn in dem Ausgabeformat der *Sink* die Felder *artist* und *songName* hinzugefügt wurden, finden sich nach Ausführung der so abgewandelten Hitparade in der Ausgabedatei auch die zusätzlichen Songinformationen.

Gleich und Gleich gesellt sich gern

Das vorherige Beispiel ist so simpel, dass der Einsatz eines so mächtigen Frameworks wie Hadoop/Cascading möglicherweise übertrieben erscheint. Ähnliche Anwendungsfälle, wie etwa das Filtern und Verarbeiten sehr großer Mengen von Log-Daten, können jedoch häufig durch Einsatz dieser Werkzeuge gut skalierbar gemacht werden.

Für unser nächstes Beispiel wollen wir Cascading durch eine eigene Operation erweitern. Das in Listing 3 und 4 abgebildete Programm schreibt zu jedem Lied den Song heraus, der am häufigsten mit dem Ursprungssong zusammen in der Playlist eines Users auftritt. Hauptzutat ist die Klasse *PairEmitter* (Listing 3), ein *Buffer*, der – im Anschluss an die Gruppierung nach Usern – für jeden User alle bei ihm auftauchenden Songpaare ausgibt. Jede Operation in Cascading hat ein *Context*-Objekt, das am Anfang in der *prepare*-Methode initialisiert werden kann. In unserem Fall erzeugen wir ein Tuple der Länge 2, das für die Rückgabe der Paare verwendet wird. Im Gegensatz zu Aggregatoren, wie z. B. *sum*, *count* etc., wird die *operate*-Methode eines Buffers nur einmal für jeden unterschiedlichen Eintrag des Gruppierungsfelds – in diesem Beispiel für jeden User – aufgerufen. Auf die Eingabe-Tuples, also die Songs, die der User gehört hat, greift man dann mithilfe eines *Iterators* zu, und die Ausgabe-Tuples, alle auftretenden Songpaare, werden in den *OuputCollector* geschrieben.

Die *Main*-Methode gruppiert die Tuples zuerst nach User und wendet dann den *PairEmitter* an. Die Ausgabe der Pipe besteht dann aus allen Songpaaren. Diese müssen nur noch gezählt und anschließend die am häufigsten auftretende Kombination pro Song als Ergebnis ausgeschrieben werden. Dies machen wir wieder mit Cascadings Bordmitteln. In diesem Fall verwenden wir zuerst den *CountBy*-Partial-Aggregator – Cascading verwendet keine Combiner, sondern mit den Partial-Aggregatoren eine flexiblere Technik, auf die wir im Rahmen dieses Artikels aber leider nicht näher eingehen können. Anschließend sortieren wir die Ergebnisse und geben zu jedem Song das erste Paar aus. Hierzu verwenden wir den *First*-Operator. Last, but not least definieren wir den Flow und führen den Prozess aus. Da in unserem *PairEmitter* sehr viele Songkombinationen erzeugt werden, läuft der Job auf einem Rechner mehrere Stunden, skaliert aber hervorragend dank des zugrunde liegenden Hadoop-Frameworks.

Fazit

Cascading liefert eine klar designte Abstraktionsschicht für Hadoop. Dem Entwickler erspart der Einsatz von Cascading, seine gesamte möglicherweise sehr komplexe Big-Data-Pipeline in MapReduce-Schritte zu übersetzen, wodurch eine enorme Produktivitätssteigerung bewirkt werden kann. Andererseits liefert Cascading genügend feingranulare Stellschrauben, mit denen die resultierenden MapReduce-Programme „getunt“ werden können – Cascading verdigitalisiert somit das Beste aus beiden Welten. In den nächsten Jahren wird von dieser Java-Bibliothek sicherlich noch einiges zu hören sein.

Dr. Boris von Loesch ist promovierter Mathematiker und arbeitet zurzeit als Datenanalyst und Java-Entwickler bei der Zalando GmbH. Schon seit vielen Jahren entwickelt und betreut er außerdem ehrenamtlich die Open-Source-Eclipse-Plug-ins Texlipse und Pdf4Eclipse.

Peter Siemen ist freiberuflicher Machine Learning/Software Consultant. Als Full-Stack Developer sammelt er seit über fünf Jahren Erfahrung bei der Implementierung von maßgeschneiderten Machine-Learning-Lösungen in High-Traffic-Umgebungen.

(Zusammen implementieren Boris und Peter zurzeit die nächste Generation der Recommendation Engine für die Zalando GmbH.)

Links & Literatur

[1] http://hadoop.apache.org

[2] http://www.cascading.org

[3] http://labrosa.ee.columbia.edu/millionsong/tasteprofile

Wie man ein Personal Hadoop-Cluster mit Cubieboards aufbaut

Elefantenbaby

Denkt man an Big Data und Hadoop, hat man in der Regel das Bild von endlosen Server-Racks in Gängen von überdimensionalen Rechenzentren vor Augen. Dass es auch ohne Serverpark geht, zeigt die Selbstbauanleitung für ein Hadoop-Cluster auf Basis des Raspberry-Pi-Klons Cubieboard.

von Ramon Wartala

Heimcomputer waren in den achtziger Jahren das, was Smartphones und Tablets zu Beginn des 21. Jahrhunderts sind: persönliche Computer für die tägliche Freizeit und Arbeit. Daten- und rechenintensive Anwendungen sind auf derartigen Geräten nur selten ohne die Hilfe von Cloud-Diensten zu realisieren. Angetrieben werden Dienste dieser Art von einer Vielzahl an Servern, die oftmals als Cluster den ausfallsicheren Zugriff der mobilen Endgeräte auf ihre Daten und Anwendungen ermöglichen. Diese Server stehen in der Regel in großen Rechenzentren, zu denen man als Privatperson und Nutzer dieser Dienste nur äußerst selten bis überhaupt nicht Zutritt erlagen kann.

Konnte man in den 70er und 80er Jahren mit etwas elektrotechnischem Grundverständnis Computer wie den *Altair 8800* [1] oder den *TRS-80* [2] noch selbst zusammenbauen, so ist der Bau eines kompletten Computer-Clusters für Privatpersonen nur selten sinnvoll. Auch wenn Services wie die *Amazon Elastic Compute Cloud* [3] oder *ProfitBricks* [4] ihre Hardware zu On-Demand-Preisen anbieten, bei denen rein nach genutzter Zeit abgerechnet wird, entstehen für den Privatanwender dabei erhebliche Kosten.

2012 kehrte mit dem als Schulungscomputer konzipierten Einplatinenrechner Raspberry Pi [5] etwas vom Flair der Homecomputer-Gründerjahre zurück. Die kreditkartengroße Platine beherbergt alle nötigen Anschlüsse, wie einen HDMI-Ausgang, SD-Kartenleser, Ethernet- und USB-Ports.

Angetrieben wird der kostengünstige Kleincomputer von einer ARM-CPU und 256 bzw. 512 MB Hauptspeicher. Der Umgang mit einem solchen Einplatinencomputer erinnert nicht nur an die Elektronikkästen aus dem eignen Kinder- oder Jugendzimmer, sondern fühlt sich auf der Softwareseite auch wie ein waschechter Server an. Zugang bekommt man über angepasste Linux-Versionen mit oder ohne Desktop remote oder direkt über eine angeschlossene Tastatur plus Maus und Monitor. Neben der Nutzung als Streaming-Client lassen sich auch einfache Serverdienste für das eigene LAN realisieren. Durch den großen Erfolg des Raspberry Pi wurden auch andere auf den wachsenden Markt in diesem Computersegment aufmerksam. Schnell kam der Wunsch

nach mehr Hauptspeicher, einem SATA-Anschluss für Massenspeicher und anderen Erweiterungen auf. So startete Tom Cubie von Allwinner [6] 2012 mit anderen das nach ihm benannte *Cubieboard*. Neben einer doppelten Hauptspeicherausstattung von 1 GB besitzt das Cubieboard einen SATA-Anschluss für den Betrieb von einfachen 2,5-Zoll-Notebook-Festplatten oder SSDs (**Abb. 1**).

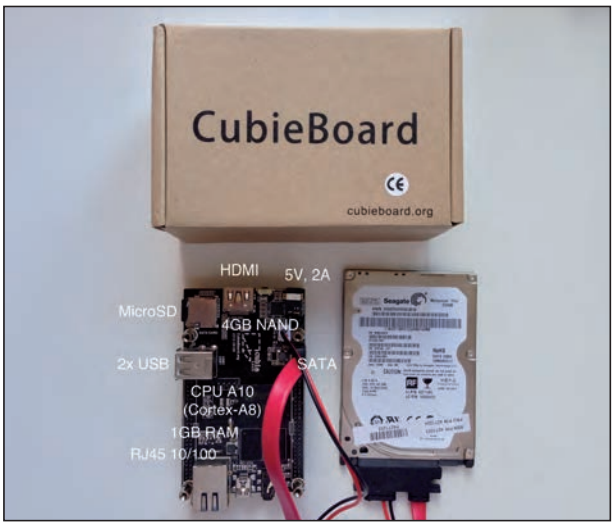


Abb. 1: Anschlussfreudiges Cubieboard, hier mit angeschlossener 2,5-Zoll-Notebook-SATA-Festplatte

Ideenklau

Den ersten Anstoß, aus Einplatinencomputern ein persönliches Cluster aufzubauen, lieferte eine Verlautbarung der Universität von Southampton [7], einen Supercomputer aus 64 Raspberry Pis gebaut zu haben. Bei meiner Recherche fand ich den Blogpost „Hadoop Cluster running on Raspberry Pis“ von Bryan Wann [8], der bereits das MapReduce-Framework Hadoop auf Raspberry Pis umgesetzt hatte. Doch in den ersten Tests enttäuschte nicht nur der fehlende I/O-Durchsatz der SDHC-Karten, sondern auch die MapReduce-Geschwindigkeit, die dem fehlenden RAM zugeschrieben wurde. Es sprach also einiges dafür, einen anderen Einplatinencomputer für das eigene Cluster zu nutzen. Die Wahl fiel schließlich auf das bereits erwähnte Cubieboard, das neben mehr RAM auch über einen SATA-I-Anschluss mit theoretisch 150 MB/s verfügt. Zum Vergleich: SDHC-Karten schaffen in der Regel zwischen 2 MB/s und 20 MB/s. Der Formfaktor lässt sich mit der Größe einer Zigarettenschachtel vergleichen. Das Cubieboard ist somit nur wenig größer als der kreditkartengroße Raspberry Pi. Kostenseitig ist das Cubieboard zwar ganze 28 Euro teurer als das Model B des Raspberry Pi, kommt in dieser Ausführung aber bereits mit den nötigen SATA-Kabeln, 1 GB RAM, 4 GB NAND und einem einfachen Gehäuse daher.

Grundinstallation

Das Cubieboard hat werksseitig Android installiert. Ebenfalls im 4 GB großen NAND-Speicher befinden sich einige Anwendungen; darunter auch ein Webbrowser. Mit seiner Hilfe ist es möglich, sich die App *BerryBoot* [9] zu installieren, die sich als Bootloader für Raspberry Pis, Cubiboards und andere A10- und Android-basierte Systeme nutzen lässt. Nach der Installation und dem Start der Android-App bietet BerryBoot eine menügestützte Installation der gängigen Linux-Distributionen. Dabei werden die Dateien aus dem Netz geladen und auf eine im System befindliche Speicherkarte oder Festplatte geschrieben. Von hier aus kann das gewählte System gestartet werden. Für unseren Zweck wollen wir die *Debian Wheezy Distribution* nutzen. Die Installation auf die SD-Karte erledigen wir über die BerryBoot-Auswahl. Nach dem Reboot können wir uns mithilfe von SSH mit den Cubieboards verbinden:

```
$ ssh pi@<IP-Adresse-des-Cubieboards>
```

Das Default-Passwort ist dabei *raspberry*. Die IP-Adresse sollte sich leicht über den (W-)LAN-Router herausfinden lassen:

```
$ uname -a
Linux raspberrypi 3.4.24-a10-aufs+ #33 PREEMPT Sun Feb 24
21:17:26 CET 2013 armv7l GNU/Linux
```

Versuchsaufbau

Um aus Cubieboards ein einfaches Hadoop-Cluster zu bauen, sind mindestens drei Cubieboards, drei 4-GB-microSD-Karten und drei 2,5-Zoll-SATA-Platten mit möglichst niedriger Leistungsaufnahme nötig. Dazu kommt für die Vernetzung der Rechner noch ein handelsüblicher 10/100-Ethernet-Switch, der im lokalen LAN installiert ist (Kasten: „Einkaufsliste“). Auf jedem Cubieboard wird, wie bereits beschrieben, Debian Wheezy für Raspberry Pi auf die 4-GB-microSD-Karten installiert und gebootet. Wenn alle Cubieboards im LAN erreichbar sind, sollte auch die Anmeldung über SSH möglich sein. Als Nächstes sollen die Notebookfestplatten eingerichtet werden. Nach dem Boot-Vorgang sollten sich folgende Zeilen in der Datei */var/log/dmesg* finden lassen, wenn die Festplatte erkannt und das entsprechende Kernel-Modul geladen wurde:

```
...
5.630000] ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
[    5.630000] ata1.00: ATA-8: ST320LT007-9ZV142, 0004LVM1,
                                                        max UDMA/133
[    5.640000] ata1.00: 625142448 sectors, multi 16: LBA48 NCQ
                                                        (depth 31/32)
[    5.650000] ata1.00: configured for UDMA/133
[    5.680000] scsi 0:0:0:0: Direct-Access     ATA      ST320LT007-9ZV14 0004
                                                        PQ: 0 ANSI: 5
[    5.690000] sd 0:0:0:0: [sda] 625142448 512-byte logical blocks:
                                                        (320 GB/298 GiB)
[    5.690000] sd 0:0:0:0: [sda] 4096-byte physical blocks
[    5.700000] sd 0:0:0:0: [sda] Write Protect is off
[    5.700000] sd 0:0:0:0: [sda] Mode Sense: 00 3a 00 00
[    5.700000] sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled,
                                                        doesn't support DPO or FUA
[    5.750000]  sda: sda1
[    5.760000] sd 0:0:0:0: [sda] Attached SCSI disk
...
```

Installiert wird jeweils eine Festplatte pro Cubieboard mit folgenden Kommandos:

```
$ sudo mkdir /mnt/disk1
$ sudo mount /dev/sda1 /mnt/disk1

$ sudo fdisk /dev/sda
> n (add new partition)
> enter
> enter
> enter
> enter
> w

$ sudo mkfs.ext4 /dev/sda1
```

Damit die Festplatte beim nächsten Reboot automatisch erkannt wird, ist folgender Eintrag in die Datei */etc/fstab* nötig:

```
/dev/sda1  /mnt/disk1   ext4   defaults,noatime  0    3
```

Netzwerkinstallation

Bei der Nutzung mehrerer Cubieboards im Netzwerk kann es nützlich sein, sowohl einen einfachen Alias für jeden Rechner zu vergeben, damit man nicht immer die komplette IP-Adresse eingeben muss. Dazu tragen wir in die */etc/hosts*-Datei die gültige IP-Adresse der Cubieboards (über *ifconfig* zu erfahren) und einen beliebigen Namen ein. Hier zum Beispiel die Namen bekannter Kommissare:

```
192.168.178.65  schimanski
192.168.178.62  brunetti
192.168.178.67  lestrade
```

Die Hostnamen ändert man mit dem Befehl

```
sudo hostname NAME
```

auf den jeweiligen Cubieboards. Um sich bequem von einem Rechnerknoten mit einem anderen zu verbinden, sollte man Public-Key-Authentifizierung bei der SSH-Verbindung konfigurieren. Dazu müssen auf jedem Rechner mit

```
$ ssh-keygen -t rsa -P ""
$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

die nötigen Privat- und Public-Schlüssel erzeugt und dann auf alle anderen Cubieboards verteilt werden. Das funktioniert am einfachsten mit

```
pi@brunetti$ ssh-copy-id -i .ssh/id_rsa.pub pi@schimanski
pi@brunetti$ ssh-copy-id -i .ssh/id_rsa.pub pi@lestrade
...
```

Ein *pi@brunetti$ ssh schimanski* sollte nun ohne Passwortabfrage klappen. Ist dies für alle Cubieboards erfolgt, kann die weitere Softwareinstallation durchgeführt werden.

Java für ARM

Die wichtigste Softwarekomponente für den Betrieb eines Hadoop-Systems ist neben dem eigentlichen Betriebssystem eine funktionierende Java-Infrastruktur. Im Dezember letzten Jahres präsentierte Oracle mit Java for ARM eine erste Version des JDK 8 speziell für die ARM-Prozessoren [10]. Diese soll als Grundlage für die weitere Installation des Hadoop-Clusters zum Einsatz kommen. Ein

```
$ wget http://www.java.net/download/JavaFXarm/jdk-8-ea-b36e-linux-arm-
                                    hflt-29_nov_2012.tar.gz --no-check-certificate
```

lädt die benötigte Version herunter, die sich mit

```
$ tar xvzf jdk-8-ea-b36e-linux-arm-hflt-29_nov_2012.tar.gz
$ sudo mv jdk1.8.0 /usr/local/java
```

installieren lässt. Fehlt nur noch, den Installationspfad über die Umgebungsvariablen dem Log-in bekannt zu machen. Dies geschieht am einfachsten mit

```
$ sudo vi /etc/profile
```

und dem Hinzufügen der Zeilen

```
export JAVA_HOME=/usr/local/java
export PATH=$PATH:$JAVA_HOME/bin
```

Ob alles richtig installiert ist, lässt sich mit einem einfachen

```
$ java -server -version
```

testen.

Hadoop-Installation

Hadoop gibt es mittlerweile in einer Reihe von Geschmacksrichtungen. Ähnlich wie bei Linux unterscheidet man auch bei Hadoop so genannte Distributionen. Diese sind sowohl von Firmen wie Cloudera [11] oder Hortonworks [12] als auch direkt über die Seiten der Apache Foundation [13] zu beziehen. Letztes wollen wir nutzen, um möglichst herstellerunabhängig zu bleiben. Jedes Cubieboard bekommt dafür die kompletten Hadoop-Binaries installiert. Von einem ausgewählten Apache-Mirror geschieht dies am einfachsten mit

```
$ wget http://ftp.halifax.rwth-aachen.de/apache/hadoop/core/stable/
                                                    hadoop-1.0.4-bin.tar.gz
```

und das Auspacken mit

```
$ tar xvzf hadoop-1.0.4-bin.tar.gz
$ sudo mv hadoop-1.0.4 /usr/local/.
```

```
$ sudo ln -s /usr/local/hadoop-1.0.4 /usr/local/hadoop
$ sudo chown -R pi.pi /usr/local/hadoop
```

Die eigentliche Konfiguration der Hadoop-Software ist auch auf ausgewachsenen Cluster-Installationen kein Spaziergang. Es muss eine Reihe von Diensten auf jedem einzelnen Cluster-Knoten konfiguriert und in der richtigen Reihenfolge gestartet werden. Jedes Hadoop-System besteht aus dem Dateisystem (HDFS) und Diensten für die verteilte Ausführung von Anwendungen (MapReduce-Framework). Diese beiden Teile werden durch unterschiedliche Daemons verwaltet. Das verteilte Dateisystem HDFS wird über die Dienste *NameNode*, *SecondaryNameNode* und *DataNode* betrieben. Die eigentlichen MapReducer-Anwendungen laufen auf den Diensten *TaskTracker* und *JobTracker*.

Prinzipiell können alle Daemons auf allen Rechnerknoten laufen. Um eine optimale Verteilung zu erreichen, wollen wir folgende Zuordnung von Diensten und Rechnerknoten realisieren:

```
schimanski: NameNode, SecondaryNameNode, JobTracker
brunetti  : DataNode, TaskTracker
lestrade  : DataNode, TaskTracker
```

Hadoop-Konfiguration

Hadoop verfügt über eine Reihe von Konfigurationsdateien und Einstellungsmöglichkeiten. Zum Glück werden insgesamt nur wenige Änderungen an den Standardeinstellungen benötigt, um das Cluster in Betrieb zu nehmen. Ausgangspunkt beim Start aller Hadoop-Dienste stellt die Datei *hadoop-env.sh* dar, die die Java-Umgebung konfiguriert. Als Minimaleinstellung muss darin der Pfad zur Java-Installation bekannt gegeben werden:

```
sudo vi /usr/local/hadoop/conf/hadoop-env.sh

export JAVA_HOME=/usr/local/java
```

Wenn man schon mal dabei ist, sollte man auch gleich die richtige Heapsize und die nötige HotSpot-Runtime einstellen mit:

```
export HADOOP_HEAPSIZE=512
export HADOOP_OPTS=-server
```

Als Nächstes ist die Konfiguration der Hadoop-Dienste an der Reihe. Dazu werden Änderungen in den Dateien *core-site.xml* (Listing 1) und *hdfs-site.xml* (Listing 2) für das Dateisystem HDFS sowie *mapred-site.xml* (Listing 3) für das MapReduce-Framework vorgenommen.

Um Hadoops Dateisystem nutzen zu können, müssen sowohl ein temporäres Verzeichnis definiert als auch der Hostname der NameNode und der Replikationsfaktor für die Blockreplikation eingestellt werden. Letzterer

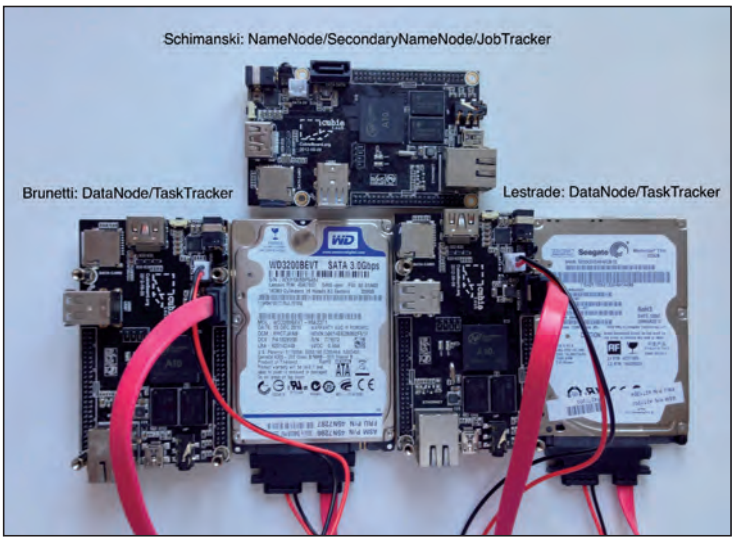


Abb. 2: Drei Cubieboards ermöglichen den Betrieb eines einfachen Hadoop-Clusters

gibt an, wie viele Rechnerknoten während der Ausführung einer MapReduce-Anwendung ausfallen dürfen. Bei zwei Cubieboards, die das HDFS über zwei DataNodes aufspannen, tragen wir eine 2 als Replikationsfaktor ein. Das Cubieboard mit Namen *schimanski* wird als NameNode die Dateiverwaltung und Speicherung der Dateisystem-Metadaten (Datei- und Verzeichnisnamen etc.) übernehmen.

Listing 1: /usr/local/hadoop/conf/core-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/mnt/disk1/tmp</value>
  <description></description>
</property>
<property>
  <name>fs.default.name</name>
  <value>hdfs://schimanski:54310</value>
  <description></description>
</property>
</configuration>
```

Listing 2: /usr/local/hadoop/conf/hdfs-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>mapred.job.tracker</name>
  <value>localhost:54311</value>
  <description></description>
</property>
</configuration>
```

Rechnerknoten, die die Dienste NameNode und JobTracker beherbergen, werden im Hadoop-Sprachgebrauch auch als *Master-Knoten* bezeichnet. Welche Rechnerknoten im Cluster diese Aufgaben wahrnehmen, wird über die Datei

```
$ sudo vi /usr/local/hadoop/conf/masters
```

definiert. In unserem Beispiel muss *schimanski* als Master eingetragen werden. Entsprechend werden die DataNodes und TaskTracker in der Datei

```
$ sudo vi /usr/local/hadoop/conf/slaves
```

benannt. In unserem Beispiel *brunetti* und *lestrade*.

Dateisystem einrichten

Sind alle Konfigurationsdateien eingerichtet, kann Hadoops Dateisystem formatiert werden. Dazu teilt man dies der NameNode über die Kommandozeile mit:

```
$ /usr/local/hadoop/bin/hadoop namenode -format
```

Ist die Formatierung abgeschlossen, kann mithilfe von

```
/usr/local/hadoop/bin/start-dfs.sh
```

das HDFS gestartet werden. Ob alle Dienste richtig gestartet sind, lässt sich über ihre Log-Dateien in Erfahrung bringen – bei der NameNode über

```
tail -f /usr/local/hadoop/logs/hadoop-pi-namenode-schimanski.log
```

und bei den DataNodes über

```
tail -f /usr/local/hadoop/logs/hadoop-pi-datanode-lestrade.log
tail -f /usr/local/hadoop/logs/hadoop-pi-datanode-brunetti.log
```

Darüber hinaus lässt sich mit einem

```
$ jps
```

auf dem jeweiligen Cubieboard herausfinden, welche Java-Prozesse gestartet sind.

Listing 3: /usr/local/hadoop/conf/mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
  <name>dfs.replication</name>
  <value>2</value>
</property>
</configuration>
```

Dateisystem testen

Um das verteilte Dateisystem zu testen, können wir mit folgenden Kommandos eine einfache Textdatei hineinkopieren. Dazu holen wir uns aus dem Projekt Gutenberg eine beliebige Textdatei (hier Grimms Märchen) mit

```
$ wget http://www.gutenberg.org/cache/epub/2591/pg2591.txt -o
                                                grimms-fairy-tales.txt
```

und kopieren diese mit

```
$ /usr/local/hadoop/bin/hadoop fs -copyFromLocal grimms-fairy-tales.txt
                                                grimms-fairy-tales.txt
```

Hat bis hierher alles funktioniert, sollten wir ein erstes MapReduce-Programm auf unsere Daten loslassen. Dazu müssen noch die entsprechenden Dienste gestartet werden mit

```
$ /usr/local/hadoop/bin/start-mapred.sh
```

Auch hier können wir den Start der TaskTracker-Dienste

```
tail -f /usr/local/hadoop/logs/hadoop-pi-tasktracker-lestrade.log
tail -f /usr/local/hadoop/logs/hadoop-pi-tasktracker-brunetti.log
```

und JobTracker-Dienste

```
tail -f /usr/local/hadoop/logs/hadoop-pi-jobtracker-schimanski.log
```

über die jeweiligen Log-Dateien verfolgen.

MapReduce testen

Um erste Tests mit dem neuen Hadoop-Cluster zu machen, bieten sich die in Hadoop enthaltenen Beispielanwendungen an. Eine Liste aller Beispiele lässt sich mit

```
$ /usr/local/hadoop/bin/hadoop jar /usr/local/hadoop/hadoop-examples-
                                                1.0.4.jar
```

anzeigen. Das „hello world“ auf Hadoop-Seite ist ein MapReduce-Programm zum parallelen Zählen von Wörtern und ihrem Vorkommen in einem Text. Um die Wörter in den Grimmschen Märchen zu zählen, reicht ein

```
$ /usr/local/hadoop/bin/hadoop jar /usr/local/hadoop/hadoop-examples-
        1.0.4.jar wordcount /user/pi/grimms-fairy-tales.txt /user/pi/out
```

Die Verteilung der Wörter kann über

```
$ /usr/local/hadoop/bin/hadoop fs -getmerge /user/pi/out
                                                grimms-fairy-tales_out.txt
```

als Datei in das lokale Dateisystem geholt werden.

Fazit

Der Aufbau eines privaten Hadoop-Clusters für Forschung und Lehre ist mithilfe des Einplatinensystems Cubieboard mit vertretbarem zeitlichem und finanziellem Aufwand möglich. Natürlich sind die I/O-Zugriffszeiten nicht zu vergleichen mit ausgewachsenen Serversystemen, aber dafür benötigt ein Cubieboard-Cluster auch deutlich weniger Strom und finanzielle Mittel. Die Schnittstellen zum System unterscheiden sich in keiner Weise von denen eines ausgewachsenen Hadoop-Clusters. Und Firmen wie HP haben die Vorteile der stromsparenden ARM-CPUs im Serverbereich auch bereits erkannt: Mit dem Projekt Moonshot [14] soll demnächst ein solcher Server auf den Markt kommen.

Ramon Wartala ist Diplom-Informatiker und arbeitet als Director Technology für die Online-Marketing-Agentur Performance Media Deutschland GmbH. Anfang 2012 erschien sein Buch „Hadoop – Zuverlässige, verteilte und skalierbare Big-Data-Anwendungen“.

Links & Literatur

[1] http://de.wikipedia.org/wiki/Altair_8800

[2] http://de.wikipedia.org/wiki/Tandy_TRS_80_Model_1

[3] http://aws.amazon.com/de/elasticmapreduce/

[4] https://www.profitbricks.com/de/en/the-iaas-company/press/20130319-profitbricks-and-mapr-benchmark-for-hadoop-within-a-virtualised-data-centre/

[5] http://www.raspberrypi.org

[6] http://www.allwinnertech.com

[7] http://www.southampton.ac.uk/~sjc/raspberrypi

[8] http://binaryfury.wann.net/2013/02/hadoop-cluster-running-on-raspberry-pis/

[9] http://www.berryterminal.com/doku.php/berryboot

[10] http://jdk8.java.net/fxarmpreview/index.html

[11] http://www.cloudera.com/content/cloudera/en/products/cdh.html

[12] http://hortonworks.com/products/hortonworksdataplatform/

[13] http://hadoop.apache.org

[14] http://www.zdnet.de/88150615/projekt-moonshot-warum-hp-auf-neue-servertechnik-setzt/

Einkaufsliste

Ein Hadoop-Cluster aus Cubieboards aufzubauen erfordert neben den eigentlichen Einplatinencomputern nur noch Teile, die sich in der Regel aus alten Beständen nutzen lassen. Für den Minimalaufbau sind folgende Komponenten notwendig:

- 3 x Cubieboards (3 x 64,80 Euro = 194,40 Euro)
- 3 x 5V-, 2A-Netzteile (z. B. von Thomann à 7,60 Euro = 22,80 Euro)
- 3 x 4-GB-microSDHC-Karten für Linux (3 x 5,90 Euro = 17,70 Euro)
- 3 x 2,5-Zoll-SATA-Platten (z. B. Seagate Momentum Thins 320 GB, ca. 50 Euro pro Stück)
- 1 x Fast-Ethernet-Switch (zwischen 15 und 25 Euro)
- 3 x Netzwerkkabel (3 bis 5 Euro)

Anzeige

Anzeige

Mobile Anwendungen mit AeroGear

JBoss goes Mobile

AeroGear ist ein JBoss-Communityprojekt, das den Fokus auf die mobile Entwicklung legt. Das Projekt bietet eine Bibliothek und verschiedene Utilities für Android, iOS und JavaScript. Wir werfen einen ersten Blick auf AeroGear.

von Matthias Weßendorf

Die Entwicklung von mobilen Anwendungen muss nicht unbedingt komplexer sein als nötig. Eine häufige Hürde ist die Bereitstellung einer Anwendung für verschiedene Plattformen. Für den Java-Entwickler scheint Android noch die gängigste Wahl zu sein, allerdings lauern auch hier Tücken, die nicht mit den Aufgabenstellungen von Swing- oder gar Enterprise-Java-Anwendungen zu vergleichen sind. Neue, oder zusätzliche Sprachen, wie JavaScript oder gar Objective-C können ebenfalls Probleme bereiten.

Unterschiedliche Frameworks

Ein durchaus größeres Problem als die verschiedenen Sprachen sind allerdings die völlig unterschiedlichen Frameworks, die es für die entsprechenden Plattformen gibt. Eine mobile Anwendung soll in erster Linie eine fachliche Anforderung umsetzen, doch diese gilt es in Quellcode zu formulieren. Für den Zugriff auf eine HTTP-Schnittstelle wird in Android häufig das "*HttpURLConnection*"-API genutzt, bei iOS-Anwendungen hingegen das AFNetworking Framework [1] oder die systemeigene NSURLConnection-Klasse [2]. Innerhalb von mobilen Webanwendungen wiederum kommt jQuery oder das *XmlHttpRequest*-Objekt zum Einsatz. Diese Frameworks haben völlig unterschiedliche APIs, was nicht nur daran liegt, dass sie in verschiedenen Programmiersprachen erstellt wurden. Diese Unterschiede machen es nicht leichter, eine fachliche Anforderung für die unterschiedlichen mobilen Plattformen zu programmieren.

Listing 1

```
// create "business object"
Car someCar = new Car(.....);

// save, via HTTP POST:
carsPipe.save(someCar, new Callback<Car>() {

  @Override
  public void onSuccess(Car serverData) {
    // work with the data....
  }
  @Override
  public void onFailure(Exception e) {
    // show error...
  }
});
```

AeroGear: einheitliches API

Das AeroGear-Projekt [3] bietet ein einheitliches API für Android, iOS und JavaScript. Die Konzepte der Bibliothek sind plattformübergreifend und somit leicht auf andere Plattformen zu übertragen. In der Praxis bedeutet das, dass das API dieselbe Funktionalität bereitstellt, dabei allerdings die Gegebenheiten der jeweiligen Programmiersprache honoriert. In Java wird ein Callback beispielsweise durch ein Interface ausgedrückt, bei Objective-C stattdessen ein so genannter Block (vergleichbar mit einem Closure) genutzt. Die derzeitigen Schwerpunkte von AeroGear liegen auf HTTP-Kommunikation, Offlinespeicherung und verschiedenen Sicherheitsthemen wie OTP [4] oder Authentifizierung.

HTTP-REST-Zugriff

Für den Zugriff auf RESTful Web Services bietet AeroGear die Pipe-Schnittstelle. Sie abstrahiert den eigentlichen HTTP-Zugriff auf eine HTTP-Ressource und stellt im Wesentlichen CRUD-Funktionalität bereit. Sämtliche Pipe-Objekte werden mittels der Pipeline-Klasse erstellt, die ebenfalls die Verwaltung der einzelnen Pipes übernimmt. Nehmen wir an, es gibt einen REST-Service *http://server.com/cars*, der über entsprechende CRUD-Funktionalität, wie in [5] beschrieben, funktioniert. Hier der Code für die AeroGear-Android-Plattform [6]:

```
// set up the baseURL and the Pipeline "Factory":
URL baseURL = new URL("http://server.com");
Pipeline pipeline = new Pipeline(baseURL);

// create Connection to "http://server.com/cars"
Pipe carsPipe = pipeline.pipe(Car.class);
```

Innerhalb von drei Zeilen wird das Pipe-Objekt für den oben genannten URL erstellt. Anschließend können Daten vom REST-Service gelesen oder zu ihm übertragen werden (Listing 1).

Ein einfaches POJO-Objekt wird mittels der *save*-Methode zum Server geschickt. Die Tatsache, dass intern HTTP-POST verwendet wird, ist völlig transparent. Neben dem eigentlichen Objekt wird eine anonyme Implementierung des Callback-Interface übergeben. Im Falle eines erfolgreichen Speicherns (z. B. HTTP 201 (created)), wird die *onSuccess()*-Methode aufgerufen. Verschiedene Fehler werden innerhalb von *onFailure()* signalisiert.

Wird dieser Code aus einer Android Activity oder einem Fragment aufgerufen, muss man die Eigenschaften des Android Activity Lifecycle [7] beachten. Bei einem eingehenden Anruf wird die Anwendung in den Hintergrund befördert, und (via HTTP) gelesene Daten können verloren gehen oder gar einen Fehler verursachen, wenn der Callback das UI der sich im Hintergrund befindenden Android Activity verändern will. Ähnliches passiert bei einer „Konfigurationsänderung", wie beispielsweise dem Drehen des Telefons. Hier wird die aktuelle Android Activity zerstört und das Betriebssystem erzeugt eine neue. Die Daten der anonymen Callback-Implementierung gehen hier ebenfalls verloren. Auch hier bietet AeroGear Hilfe in Form von speziellen abstrakten Hilfsimplementierungen des *Callback*-Interface an [8].

Neben den CRUD-Features wird auch das Thema Hypermedia angegangen: AeroGears Pipe unterstützt verschiedene Arten für das Scrollen durch große Datenmengen („Pagination"). Dabei wird unter anderem der Web Linking RFC umgesetzt [9] (Listing 2).

Für die Pagination wird der *read()*-Methode neben dem üblichen Callback ein *ReadFilter*-Objekt [10] mitgegeben. Neben diesen Parametern können weitere Feineinstellungen bei der Erzeugung des Pipe-Objekts mit der *PageConfig*-Klasse [11] angegeben werden, wie beispielsweise die Info, welche Linking-Variante genutzt werden soll (z. B. RFC 5988 oder innerhalb des Response).

Listing 2

```
...
ReadFilter readFilter = new ReadFilter();
readFilter.setLimit(2); // max. two cars per response

// HTTP GET for two cars
carsPipe.read(readFilter, new Callback<List<Car>>() {

  @Override
  public void onSuccess(List<Car> response) {
    PagedList<Car> pagedCars = (PagedList<Car>) response;

    // performs HTTP-GET for next items
    data.next(....callback.....);

  }
  @Override
  public void onFailure(Exception e) {
    // show error...
  }
});
```

Hat der Server die (HTTP-)Anfrage der Android-Anwendung beantwortet, wird wie erwartet die *onSuccess()*-Methode aufgerufen. Hier wird das List-Ergebnis auf ein type-cast auf ein *PagedList*-Objekt vorgenommen. Der Aufruf von *next()* bezweckt das Versenden einer weiteren Anfrage an den Server. In diesem Beispiel werden die nächsten zwei Autos vom Server geladen.

Apple iOS

Das oben beschriebene Pipe-/Pipeline-API liegt auch für die iOS-Plattform vor [12]. Auch hier bietet das Pipe-Objekt die bekannten *read*- oder *save*-Funktionen, allerdings angepasst an die „Besonderheiten" von Objective-C (Listing 3).

Innerhalb weniger Zeilen wird das Pipe-Objekt für den gewünschten URL erstellt. Die Pipeline „Factory" verfügt, wie in der Android-Variante, über eine *pipe*-Methode. Hier handelt es sich um einen so genannten Block (vergleichbar mit Java Closures (JDK 8)). Dieser Block bekommt genau ein Objekt übergeben, um die Konfiguration (z. B. Endpointname oder Authentifizierungsmodule) zu bestimmen. Anschließend können Daten vom REST-Service gelesen oder zu ihm übertragen werden:

```
// create "business object"
NSDictionary *car = @{@"maker": @"VW", @"type": @"Golf"};

// save, via HTTP POST:
[carsPipe save:car success:^(id responseObject) {
    // work with the data....
} failure:^(NSError *error) {
    // show error...
}];
```

Ähnlich wie in der Android-Variante wird ein „Auto"-Objekt zum Server geschickt. Statt einer eigenen Klasse wird eine Map genutzt, die das Auto repräsentiert. Die *success*- und *failure*-Callbacks sind ebenfalls mithilfe von Blocks realisiert. Beide Blocks haben jeweils genau ein Argument.

Xcode Template

Das Starten von neuen Projekten ist oft müßig. In AeroGear gibt es deshalb ein Xcode Template [13], gezeigt in **Abbildung 1**. Dieses Template erzeugt ein ein-

Listing 3

```
// set up the baseURL and the Pipeline "Factory":
NSURL *baseURL = [NSURL URLWithString:@"http://server.com"];
AGPipeline *pipeline = [AGPipeline pipelineWithBaseURL:baseURL];

// create Connection to "http://server.com/cars"
id<AGPipe> carsPipe = [pipeline pipe:^(id<AGPipeConfig> config) {
  [config setName:@"cars"];
}];
```

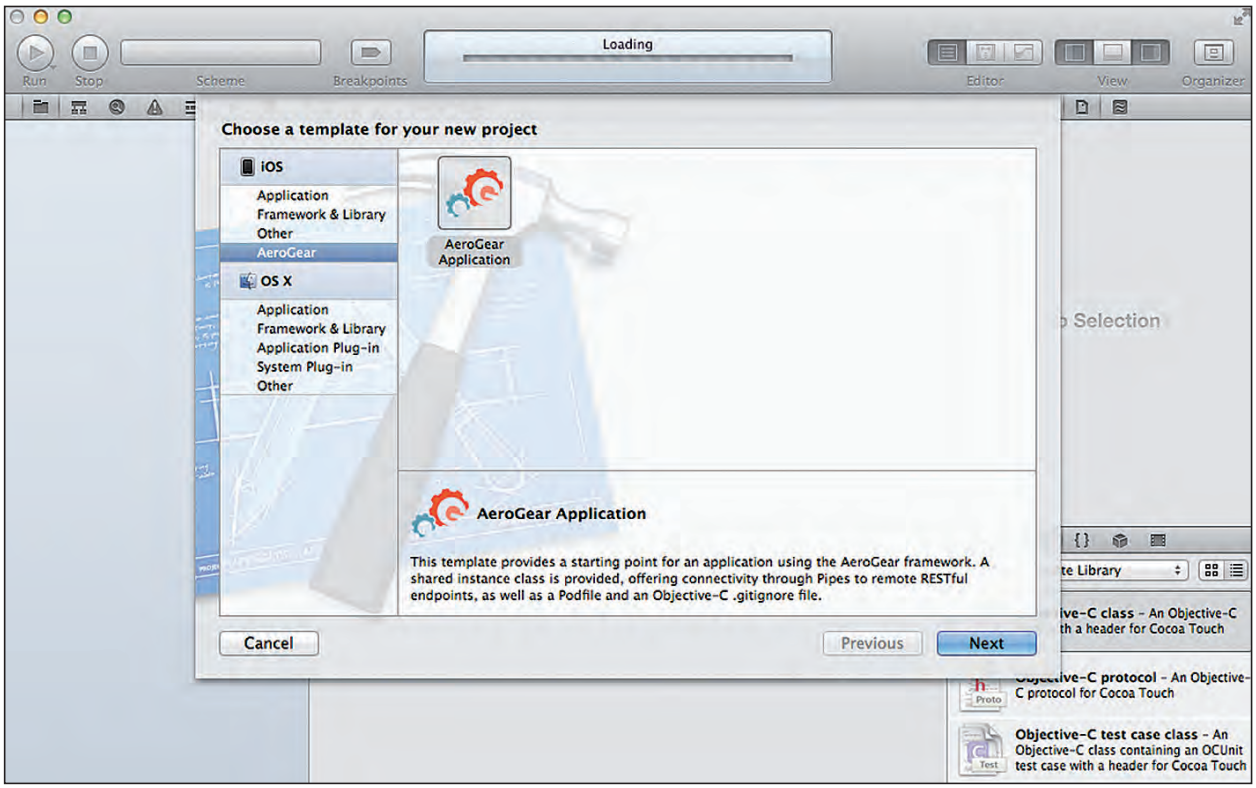


Abb. 1: Xcode Template

faches Beispiel auf Basis der AeroGear-iOS-Bibliothek. Nachdem das Projekt erzeugt wurde, fehlt noch ein kleiner Schritt: Die Abhängigkeiten müssen mittels CocoaPods [14] installiert werden. CocoaPods ist im weiten Sinne vergleichbar mit Apache Maven:

```
pod install
```

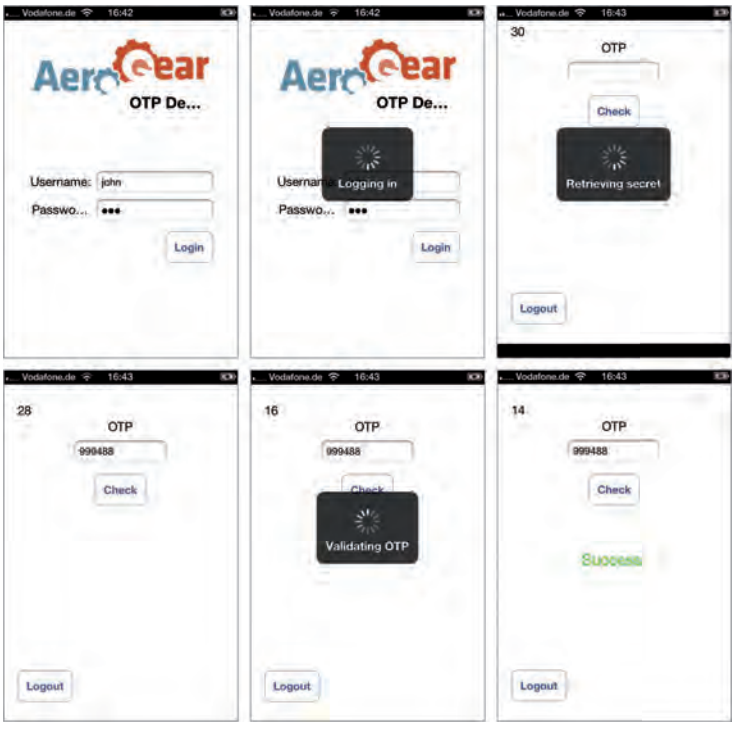


Abb. 2: iOS-Client-Demo für OTP

Mit diesem Aufruf werden die Abhängigkeiten geladen und ein so genannter Workspace wird erzeugt. Nach dem Öffnen dieses Workspace kann die Anwendung direkt auf dem eigenen Telefon gestartet werden. Der erste wichtige Schritt für eine eigene iOS-Anwendung.

Neben Android und iOS wird ebenfalls JavaScript für mobile Webanwendungen angeboten. Auch hier entspricht das API den bereits vorgestellten Konzepten. Für die Callbacks werden hier allerdings JavaScript-Funktionen verwendet:

```
jsCarsPipe.read({

  success: function( data ) {...}

});
```

Offline-Storage

Die verschiedenen Clientplattformen bieten unterschiedliche Möglichkeiten, Daten zu speichern:

- Android: Memory, SQLite
- iOS: Memory, PList
- JavaScript: Memory, SessionStorage, LocalStorage

Auch hier bietet das Store-API plattformübergreifende Konzepte, wiederum angepasst an die Besonderheiten der verwendeten Programmiersprache, wie hier im Fall von JavaScript:

```
var userStore = AeroGear.DataManager({
    name: "users",
```

```
    type: "SessionLocal"
}).stores.users;

userStore.save(someObject);
```

2-Phasen-Authentifizierung und OTP

Neben einem gewöhnlichen Login-Module bietet AeroGear Support für *One Time Password* (OTP) [15]. Mit der 2-Phasen-Authentifizierung kann der eigentliche Login um einen zusätzlichen Schutzmechanismus erweitert werden. Neben dem vom Benutzer gewählten Passwort ist ein generierter OTP-Token notwendig. In AeroGear wird die TOTP-(Time-Based-One-Time-Password-)Variante umgesetzt [16]. Die Java-Bibliothek kann sowohl clientseitig (Android) als auch serverseitig eingesetzt werden. Der Server erzeugt dabei ein *Secret* und schickt dieses zum Client:

```
@javax.inject.Inject
@org.jboss.aerogear.security.auth.Secret
private Instance<String> secret;
....
Totp totp = new Totp(secret.get());
```

Der Client muss nun anhand des empfangenen Geheimnisses, beispielsweise via SMS, den Token für den zweiten Schritt der Anmeldung erzeugen (Android):

```
String secret = // received secret
// initialize OTP
Totp generator = new Totp(secret);
// generate token
String token = generator.now();
```

Der Client schickt diesen Token nun zum Server, damit dieser – nach erfolgreicher Validierung – den Login-Versuch frei gibt:

```
@javax.inject.Inject
@org.jboss.aerogear.security.auth.Secret
private Instance<String> secret;
...
Totp totp = new Totp(secret.get());
// check if the TOKEN was valid
boolean result = totp.verify(tokenFromMobileDevice);
```

Abbildung 2 zeigt den Prozess auf Basis des iOS-Client-Demos [17]. Eine JavaScript-Variante ist derzeit in Arbeit.

AeroGear.NEXT

Die aktuelle Version 1.0 legt die Basis für kommende AeroGear-Features. Derzeit arbeiten die Entwickler an neuen Funktionalitäten wie beispielsweise „Data-Sync“, „Offline“ (z. B. CoreData für iOS oder ContentProvider Stores für Android) sowie auch plattformübergreifende Push Notificationen für APNs [18], GCM [19] oder SimplePush [20]. Der AeroGear Unified Push Server bietet eine einheitliche HTTP-Schnittstelle (und Java-API), um eine Push Notification auf unterschiedliche Endgeräte zu senden (**Abb. 3**) [21].

Eine Besonderheit hier ist die Entwicklung eines JavaScript-API für SimplePush, das außerhalb des FirefoxOS nutzbar ist. Neben der JavaScript-Bibliothek stellt das Projekt auch eine Implementierung des SimplePush-Protokolls bereit.

In der zweiten Jahreshälfte 2013 darf mit dem ersten 1.x.y-Release von AeroGear gerechnet werden. Die Entwickler freuen sich über Feedback und Diskussion auf der Mailingliste des Projekts [22].

Abb. 3: Push Notification

Matthias Weßendorf arbeitet bei Red Hat, wo er sich mit WebSocket, HTML5 und weiteren Themen rund um das Next Generation Web beschäftigt.

http://matthiaswessendorf.wordpress.com

@mwessendorf

Links & Literatur

[1] https://github.com/AFNetworking/AFNetworking

[2] http://bit.ly/pWRHib

[3] http://aerogear.org

[4] http://en.wikipedia.org/wiki/One-time_password

[5] http://www.infoq.com/articles/rest-introduction

[6] http://aerogear.org/docs/guides/aerogear-android/

[7] http://developer.android.com/training/basics/activity-lifecycle/index.html

[8] http://aerogear.org/docs/specs/aerogear-android/org/jboss/aerogear/android/Callback.html

[9] http://tools.ietf.org/html/rfc5988

[10] http://aerogear.org/docs/specs/aerogear-android/org/jboss/aerogear/android/ReadFilter.html

[11] http://aerogear.org/docs/specs/aerogear-android/org/jboss/aerogear/android/pipeline/paging/PageConfig.html

[12] https://github.com/aerogear/aerogear-ios

[13] https://github.com/aerogear/aerogear-ios-xcode-template

[14] http://cocoapods.org

[15] http://aerogear.org/docs/guides/AeroGear-OTP/

[16] http://tools.ietf.org/html/rfc6238

[17] https://github.com/aerogear/aerogear-otp-ios-demo

[18] http://bit.ly/11dht58

[19] http://developer.android.com/google/gcm/index.html

[20] https://wiki.mozilla.org/WebAPI/SimplePush

[21] https://github.com/matzew/pushee

[22] http://aerogear.org/community

Projekt Ripla: eine Plattform mit OSGi und Vaadin bauen

Modular auf ganzer Linie

Eine Plattform ist eine Anwendung, die nichts kann, aber alles möglich macht. Das Potenzial eines solchen Ansatzes führt Eclipse meisterhaft vor. Man muss allerdings nicht eclipse.org heißen, um eine Plattform bauen und betreiben zu können.

von Dr. Benno Luthiger

Die Stärke des Plattformansatzes wurde von Eclipse beispielhaft durchgespielt. Die Eclipse Rich Client Platform (RCP) wird von einem verhältnismäßig knappen Satz an OSGi Bundles gebildet. Die eigentliche Anwendung entsteht dadurch, dass die Plattform durch eine beliebige Anzahl von Komponenten aus dem Eclipse-Ökosystem angereichert wird. So existieren unzählige Anwendungen, die im Kern alle aus Eclipse bestehen, aber ganz unterschiedliche Verwendungszwecke haben und ebenso unterschiedliche Anwendungsprobleme lösen.

Während das Plattformmodell im Fat-Client-Bereich, d. h. im Bereich der Applikationen, die auf einem PC installiert werden, weite Verbreitung gefunden hat, werden im Bereich der Webapplikationen die meisten Anwendungen in traditioneller Weise ausgeliefert und installiert. Das mag damit zusammenhängen, dass der Deployment-Aufwand von Webapplikationen im Vergleich zu Fat-Clients von untergeordneter Bedeutung ist. Im Fat-Client-Modell muss die Anwendung auf jedem Kundenrechner installiert werden. Dies gilt auch für jede neue Version der Anwendung. Das allein macht den Vorteil eines Plattformansatzes augenfällig. Besteht die Applikation aus einem Kern, z. B. Eclipse RCP, und einer Sammlung von Komponenten, die in einem lockeren Verbund an diesen Kern angehängt werden, so besteht die Möglichkeit, eine solche Applikation aus sich heraus zu aktualisieren. Es reicht aus, dass der Applikationskern weiß, wie die umgebenden Komponenten auszuwechseln sind. Auf diese Weise kann der Installationsaufwand auf die einmalige Installation der Applikation reduziert werden. Alle Aktualisierungen erfolgen aus der installierten Plattform heraus.

Webapplikationen dagegen sind üblicherweise Thin-Clients. Die eigentliche Applikation wird auf einem

zentralen Server installiert. Was die Benutzer sehen, wird in einem Browser dargestellt. Bei Webapplikationen beschränkt sich das Deployment also auf die Installation auf dem zentralen Server. Damit ist das Deployment, verglichen mit den übrigen Aufwendungen zur Entwicklung der Anwendung, von untergeordneter Bedeutung. Dies hat dazu geführt, dass man bei Webapplikationen zumeist noch in Unikaten denkt und effizientere Applikationsentwicklungsmodelle nicht in Erwägung gezogen werden.

Der Vorteil einer Plattform zeigt sich, wenn man den Blick von der Deployment-Problematik abwendet und untersucht, welche Vorteile sich ergeben, wenn Applikationen nicht als Unikate, sondern als Produkte – oder besser noch: im Rahmen von Produktlinien – erzeugt werden.

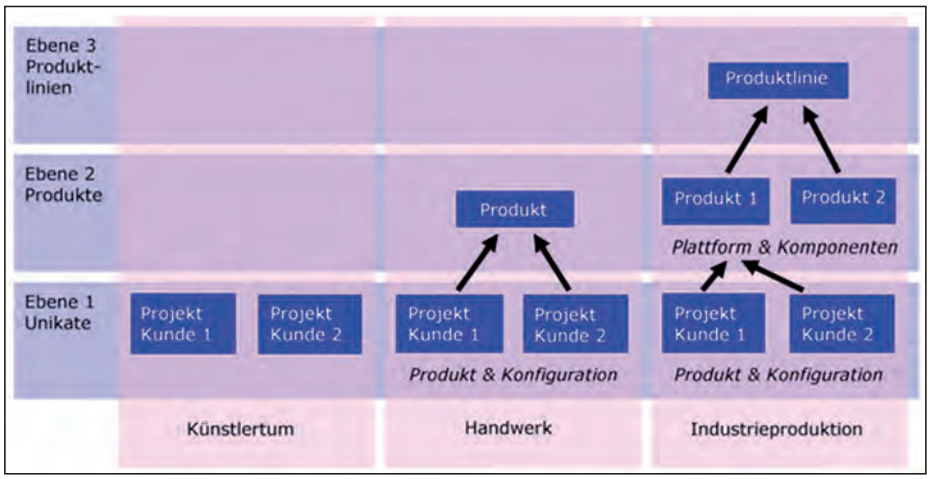


Abb. 1: Abstraktionsebenen der Entwicklung (R. Zacharias, [1])

Produktlinien

In einem 2007 im Java Magazin erschienenen Artikel erläutert Roger Zacharias den Produktlinienansatz [1]. Produktlinien sind eine zweite Abstraktionsebene in der Softwareentwicklung. Bei einer rasch und aus dem Handgelenk heraus erzeugten Softwarelösung, die ausschließlich die unmittelbar gestellten Anforderungen löst, handelt es sich um ein Unikat. Wenn ein Anbieter verschiedene Kunden hat, wird er schnell feststellen, dass ein Arbeiten mit Unikaten zu aufwändig wird. Um seinen Aufwand zu reduzieren, wird der Anbieter versuchen, Gemeinsamkeiten in den verschiedenen Projekten zu finden und diese Invarianten auf eine erste Abstraktionsebene zu verschieben. Aus Unikaten werden Produkte: Die Gemeinsamkeiten sind abstrahiert und die Variabilität wird durch projektspezifische Anpassungen und Konfigurationen abgedeckt.

Wenn ein Unternehmen mehrere Produkte aus einem ähnlichen fachlichen Bereich anbietet, kann es diesen Abstraktionsprozess fortsetzen. Die Invarianten der Produkte werden auf die nächste Ebene ausgelagert: auf die Ebene der Produktlinie (**Abb. 1**). Hat der Anbieter Produktlinien eingerichtet, muss er innerhalb der Produkte nur noch die Unterschiede und Einzigartigkeiten managen. Die Softwareprodukte werden wie in einer Fertigungsstraße fabriziert: Mit dem gleichen Herstellungsprozess und unter gemeinsamer Verwendung unterschiedlicher Komponenten werden unterschiedliche Softwareanwendungen erzeugt. Dies führt dazu, dass die Fertigungstiefe verringert wird, weil konsequent darauf geachtet wird, die Produkte aus existierenden Komponenten zusammenzubauen. Damit können die knappen Ressourcen für die Kernkompetenzen des Unternehmens eingesetzt werden.

Für einen Anbieter ist es aus verschiedenen Gründen vorteilhaft, Produktlinien zu realisieren: Da ein Softwareprodukt, das einer Produktlinie entstammt, zum größten Teil aus bereits gut getesteten Komponenten

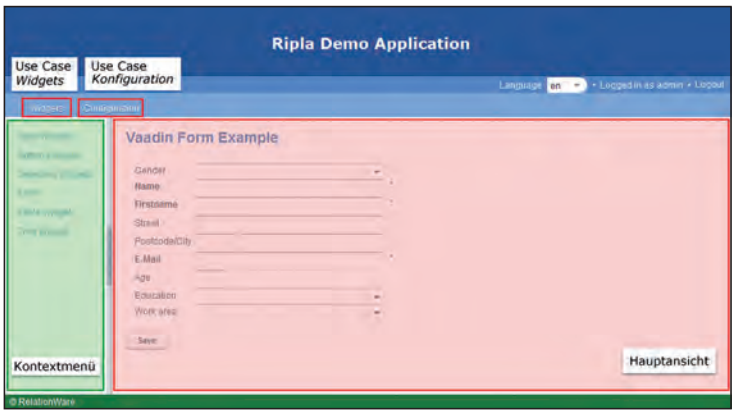


Abb. 2: Ripla UI

besteht, wird die Qualität eines solchen Produkts automatisch höher sein als diejenige eines Unikats. Ein Unternehmen, das seine Produkte über Produktlinien herstellt, wird eine größere Anzahl an Produkten managen können als ein Unternehmen, das über keinen solchen Produktionsprozess verfügt. Der wichtigste Vorteil von Produktlinien ist, dass damit die Produktentwicklung bis zur Einführung (*time to market*) drastisch reduziert wird, wodurch sich der Anbieter einen erheblichen Wettbewerbsvorteil erarbeitet.

Der Ausflug in die Softwareproduktionstechnologie ist im Zusammenhang mit unseren Überlegungen zu einer Plattform aus folgenden Gründen relevant: Eine Plattform bildet die zentrale Infrastruktureinheit von Softwareproduktlinien. Erst mit einer Plattform kann ein Softwareanbieter seinen Produktionsprozess als Produktlinie organisieren, denn es ist der Zweck einer Plattform, unterschiedliche Produkte aus einer bestimmten Menge an Komponenten zu assemblieren.

OSGi als Plattformbasis

Eclipse verwendet bekanntlich OSGi als Modularisierungstechnologie. Es ist deshalb naheliegend, OSGi auch für eine Webplattform zu verwenden.

OSGi definiert eine Sammlung von Diensten, die unterschiedliche Aspekte der Softwaremodularisierung unterstützen. Für den Plattformbetrieb ist in erster Linie *Declarative Services* (DS) von Bedeutung. Im typischen Vorgehensfall besteht OSGi DS aus einem Dienst (*Service*), der durch ein Interface beschrieben wird, aus einer beliebigen Anzahl von Dienstanbietern, die diese Schnittstelle implementieren und einem Servicekunden (*Client*), der die implementierten Dienste für eine spätere Verwendung referenziert. Diese Funktionsweise ist genau das, was wir für eine dynamisch erweiterbare Plattform brauchen. Die über ein Interface beschriebenen Dienste definieren die Erweiterungspunkte der Plattform. Die OSGi Bundles implementieren diese Dienste, und die Plattform selbst wirkt als Servicekunde, der die angebotenen Dienste auf eine sinnvolle Art und Weise verwendet. Ganz nebenbei realisiert OSGi DS auch *Dependency Injection*. Der Serviceclient, in unserem Fall die Plattform, bekommt die Erweiterungen injiziert. Er kennt von den Erweiterungen nur die Schnittstellen; die Instanziierung der konkreten Implementierungen wird durch den OSGi Container besorgt. Somit ist auch die für einen Plattformbetrieb notwendige lose Koppelung sichergestellt.

Listing 1

```
public interface IUseCase {
  Package getControllerClasses();
  IControllerSet getControllerSet();
  IMenuItem getMenu();
  IMenuSet[] getContextMenus();
}
```

Wie könnte nun eine Plattform für Webanwendungen aussehen? Das wichtigste Merkmal einer Plattform ist, dass deren Funktionsweise im Betrieb nicht zur Entwicklungszeit festgelegt ist. Stattdessen wird die Funktionalität einer Plattform kurz vor Inbetriebnahme zusammenkomponiert und kann im laufenden Betrieb aus- und umgebaut werden. Im Gegensatz zu klassischen (Web-)Anwendungen ist die Deployment-Einheit eines Plattformbetriebs, wie oben ausgeführt, nicht ein WAR-, EAR- oder EXE-File, sondern eine Komponente in Form eines OSGi Bundles.

Wird eine neue Komponente zur Plattform hinzugefügt, so soll das für den Benutzer in irgendeiner Form sichtbar werden, z. B. in Form eines neuen Menüpunkts in der Menüleiste. Klickt der Nutzer auf diesen Menüeintrag, so wird beispielsweise ein Dialog geöffnet, in den er verschiedene Informationen eingeben kann. Diese werden durch die in der Komponente mitgelieferte Geschäftslogik verarbeitet und in einer Resultatansicht angezeigt.

Eine grundlegende Bedingung für den Plattformbetrieb – das machen diese Ausführungen deutlich – besteht also darin, dass eine Komponente nicht nur Geschäftslogik, sondern auch Anzeigelogik enthält. Die Plattformkomponenten müssen die Fähigkeit besitzen, *view contributions* wie Menüeinträge, Dialoge, Ansichten oder Editoren beisteuern zu können. Diese grundlegende Bedingung hat einschneidende Konsequenzen auf die verwendbaren Webframeworks. Frameworks wie z. B. JSF, die zwingend in einem WAR- oder EAR-File ausgeliefert werden müssen, sind nicht plattformtauglich.

Von den 34 Webframeworks, die Wikipedia in der Java-Sektion auflistet, arbeiten lediglich drei mit OSGi zusammen: Apache Sling, Eclipse RAP und Vaadin [2], [3]. Apache Sling ist ein REST-Framework, das stark an *Java Content Repository (JCR)* als Persistenztechnologie gebunden ist. Darüber, wie eine Weboberfläche gestaltet ist, sagt Apache Sling nichts aus. Somit verbleiben Eclipse RAP und Vaadin.

Die Ripla-Plattform

Im Projekt *Ripla* (Rich Platform project for Java web applications [4]) habe ich den Plattformgedanken für

Webanwendungen prototypisch durchgespielt. In diesem Projekt habe ich mich entschieden, Vaadin zu verwenden.

Bei der Entwicklung von Ripla habe ich mich vom Gedanken der Vorgehensfälle (*Use Cases*) leiten lassen. Ein Use Case beschreibt alle Benutzerinteraktionen, die zur Lösung eines spezifischen Problems notwendig sind, orientiert sich somit also stark an der Benutzersicht. In Ripla soll sich ein Use Case als Erstes als Pull-down-Menü manifestieren. Die Einträge in diesem Menü entsprechen unterschiedlichen Aspekten eines Vorgehensfalls. Ein Klick auf einen Menüeintrag soll in der Hauptansicht beispielsweise ein Eingabeformular anzeigen, damit der Benutzer diesen Aspekt des Vorgehensfalls bearbeiten kann. Gleichzeitig soll im entsprechenden Bereich ein zu dieser Ansicht passendes Kontextmenü angezeigt werden. Eine solche Anwendungsoberfläche ist weit verbreitet und hat sich in vielen Fällen als angemessen erwiesen. **Abbildung 2** zeigt eine Ripla-Anwendung mit zwei Vorgehensfällen: *Widgets* und *Konfiguration*.

Ripla definiert nun ein Use-Case-Interface, das von Komponenten in OSGi Bundles implementiert werden kann. Mithilfe von OSGi *Declarative Services* wird es damit möglich, einen vollständigen Use Case in ein Bundle zu packen und als Service bei der Ripla-Applikation zu registrieren. Wird die Anwendung um einen Ripla Use Case erweitert, d. h. wird der Applikation ein neues OSGi Bundle hinzugefügt, so erscheint ein neuer Menüpunkt in der von Ripla generierten Webseite.

Die Definition des Interface (Listing 1) lässt erahnen, wie die Ripla-Plattform neue Use Case Bundles einbindet. Ripla verwendet das bewährte *Model-View-Controller*-Muster (MVC). Die Controller-Klasse enthält die Anzeigelogik und referenziert allenfalls die Geschäftslogik. Mit dieser Logik kann der Controller das Modell auswerten und zu einem *View Model* verarbeiten. Zusätzlich weiß der Controller, welche Ansicht (*View*) dieses Modell unter welchen Bedingungen zur Ansicht bringen kann.

In Ripla werden die Controller-Klassen über die Methoden *getControllerClasses()* bzw. *getControllerSet()* übergeben. Die erste Version erwartet eine Referenz auf ein Java-Package, das die Controller-Klassen enthält. In diesem Fall müssen Letztere mit *@UseCaseController* annotiert sein, um von der Plattform erkannt und registriert zu werden. Die zweite Version gibt eine *ControllerSet*-Instanz zurück. Dieses Set referenziert alle Controller-Klassen des Use Case Bundles direkt. Jede Use-Case-Controller-Klasse besitzt eine *run()*-Methode, die den Controller laufen lässt und als Resultat eine Vaadin-Komponente zurückgibt, die im Browser die gewünschte Ansicht anzeigt.

Bei der nächsten Methode des Use-Case-Interface geht es um das Menü. Die Methode *getMenu()* gibt ei-

nen Menüeintrag in Form einer *IMenuItem*-Instanz zurück. Bei dieser Menü-Item-Instanz handelt es sich um den Wurzelknoten eines beliebig strukturierten Menübaums. Dieser Wurzelknoten definiert das Pull-down-Menü in der Menüleiste der Anwendung.

Ein Menü besteht aus verschiedenen Menüeinträgen, die in verschachtelten Untermenüs angeordnet sind. In Vaadin sind solche Menüeinträge mit einem Menükommando verbunden. Ein Vaadin-Menükommando besitzt eine einzige Methode *menuSelected()*. Diese Methode wird aufgerufen, wenn der Menüeintrag geklickt wird. Beim Aufruf der Kommandomethode wird der geklickte Menüeintrag als Parameter übergeben. Relevant ist dies, weil jeder Menüeintrag eine eindeutige Identifikation besitzt. Wenn wir bei der Registrierung des Use Case Bundles und beim Aufbau des Menüs eine eindeutige Zuordnung der Menüeinträge mit den Controller-Klassen gebildet haben, können wir nun im Menükommando mithilfe der Identifikation des Menüeintrags die zugeordnete Controller-Klasse finden, aus dieser eine Objektinstanz bilden, diese laufen lassen und das Resultat in der Hauptansicht der Applikation anzeigen.

Die letzte Methode der Use-Case-Schnittstellendefinition dient dazu, die Kontextmenüs eines Use Case Bundles zu registrieren. Ein Kontextmenü zeigt die Aktionen an, die im Zusammenhang mit der aktuell in der Hauptansicht gezeigten Aktion sinnvoll sind. Die Anzeige des Kontextmenüs soll durch den Controller gesteuert werden. Analog zu einem Klick auf einen Menüeintrag soll auch ein Klick auf einen Eintrag im Kontextmenü einen neuen Controller anstoßen, der die Anzeige in der Hauptansicht auswechselt. Um dieses Verhalten zu erreichen, wird in Ripla ausgiebig der OSGi *Event Admin Service* verwendet. Dieser Service ist hier angebracht, weil er speziell auf die Kommunikation zwischen Bundles ausgerichtet wurde.

Mit *getContextMenus()* definiert jedes Use Case Bundle ein Set von Kontextmenüs. Diese werden in der Ripla-Plattform unter einer eindeutigen Adresse registriert. Will ein Controller zu seiner Hauptansicht ein bestimmtes Kontextmenü anzeigen, so schickt er in seiner *run*-Methode eine entsprechende Botschaft an den Event Handler und gibt dieser Botschaft die Identifikation des Kontextmenüs mit. Voraussetzung für dieses Verhalten ist, dass der Controller nach seiner Instanziierung von der Plattform eine Instanz des OSGi Event Admins bekommen hat. Der Event Handler der Plattform empfängt diese Botschaft und kann mit der darin enthaltenen Information das gewünschte Kontextmenü finden und im dafür vorgesehenen Bereich des Browserfensters anzeigen. Ein Klick auf einen Eintrag im Kontextmenü löst wiederum eine Botschaft an den Event Handler aus. Diesmal wird der Plattform mitgeteilt, dass ein neuer Controller aktiviert werden soll. Dieser Botschaft wird der voll qualifizierte Name der gewünschten Controller-Klasse mitgegeben. Die Plattform instanziiert und aktiviert die derart bezeichnete Klasse. Damit kann der neue Controller das Geschehen übernehmen, die notwendige Logik ausführen und mit den entsprechenden Daten die Hauptansicht aktualisieren.

Auf diese Weise führt die Applikation eine Benutzerinteraktion aus, ohne dass der Entwickler explizit eine Verbindung zwischen Menüeinträgen und View-Komponenten programmiert hätte. Den Menüeinträgen ist keine Referenz auf die Controller-Klassen einprogrammiert; trotzdem führt ein Klick auf einen solchen Eintrag dazu, dass der entsprechende Controller aktiviert wird. Das gewünschte Verhalten entsteht alleine dadurch, dass der Entwickler die Schnittstelle *IUseCase* implementiert und die Plattform diese Schnittstelle vernünftig auswertet. Weil *IUseCase* gleichzeitig einen OSGi-Service definiert, können die Komponenten, die diesen Service anbieten, zur Laufzeit zur Plattform zugeschaltet werden und diese mit neuer Funktionalität bestücken.

Das Projekt Ripla zeigt, dass mit wenig Code und dank OSGi eine voll funktionsfähige Plattform für Webanwendungen erzeugt werden kann. Neben OSGi Declarative Services und dem Even Admin Service verwendet Ripla den User Admin Service, um die Anzeige der Menüeinträge abhängig von den Zugriffsrechten der Benutzer zu steuern, sowie den Metatype und Preferences Service. Als weiteres Merkmal verfügt Ripla über Skinning-Funktionalität. Wie Use-Case-Funktionalität können auch Skins über eigenständige Bundles bei der Plattform registriert werden. Auf diese Weise kann nicht nur die Funktionalität, sondern auch das Aussehen der Anwendung zur Laufzeit und vollständig unabhängig vom übrigen Code gesteuert werden. Auch in dieser Hinsicht setzt Ripla den Produktlinienansatz um: Auch, wenn unterm Strich eine große Übereinstimmung herrscht, soll sich ein Produkt für den Kunden A maximal vom Produkt für den Kunden B unterscheiden.

Ripla ist in der aktuellen Form bereit für einen produktiven Einsatz. Das Projekt kann aber auch als Anregung dienen, eine eigene Plattform für Webanwendungen zu entwickeln. Der Autor ist offen für Anregungen, wie dieses Projekt weiterentwickelt werden kann.

Dr. Benno Luthiger ist Softwareentwickler und Fachperson für Open Source bei den Informatikdiensten der ETH Zürich. Er hat über die Motivation von Open-Source-Entwicklern promoviert.

benno.luthiger@id.ethz.ch

Links & Literatur

[1] Zacharias, Roger (2007): „Industrialisierung, wir kommen!“, in Java Magazin 3.2007, S. 69–79

[2] Wikipedia: „Comparison of web application frameworks“: http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks

[3] Stamer, Jan (2013): „The Big Picture“, in Eclipse Magazin 1.2013, S. 70–74

[4] Ripla-Projekt, Projektseite: github.com/aktion-hip/ripla, Webseite: aktion-hip.github.com/ripla/

Schnelle Entwicklung von Vaadin-Applikationen auf OSGi-Basis

OSGi fürs Web-UI

Vaadin 7 setzt in Bezug auf Web-UI-Technologien neue Maßstäbe. Allerdings fehlte bis vor Kurzem eine OSGi-Integration. Das Open-Source-Projekt „lunifera.org – OSGi-services for business applications“ hat es sich zur Aufgabe gemacht, der Community eine sehr flexible Implementierung zur Verfügung zu stellen. Vaadin-Applikationen lassen sich per OSGi Configuration Admin beschreiben und auf HttpServices deployen.

von Florian Pirchner

Der OSGi-Spezifikation folgend, stehen dem Entwickler zwei Möglichkeiten zur Verfügung, um Webressourcen wie Servlets, Ressourcen etc. einem Webserver zugänglich machen zu können: zum einen die Web-Applications-Spezifikation und zum anderen die HTTP-Service-Spezifikation. Beide werden im OSGi Service Compendium beschrieben [1].

Diese beiden Spezifikation unterscheiden sich in ihren Modellen sehr stark. Die HTTP-Service-Spezifikation definiert einen OSGi Service, der dazu dient, Servlets, Filter und Ressourcen registrieren zu können. Allerdings stellt sie keine Möglichkeit zur Verfügung, um Konfigurationen analog einer *web.xml* vorzunehmen. Auf den *ServletContext* und evtl. Listener kann kein Einfluss genommen werden. Der *HttpService* wird von einem Provider zur Verfügung gestellt, wie bspw. *org.eclipse.equinox.http.servlet*. Einstellungen bzgl. Context Path müssen bereits beim Start des Jetty-Servers durch *org.eclipse.equinox.http.jetty* per *context.path* Property konfiguriert werden.

Um den in der JEE-Welt stark verwendeten Webapplikationen – die auf Basis von *web.xml* beschrieben werden – einen einfachen Zutritt in die OSGi-Welt zu bieten, wurde die „Web-Applications-Spezifikation“ definiert. Mithilfe dieser Spezifikation können *web.xml*-Dateien verwendet werden, um alle notwendigen Konfigurationen vorzunehmen. Die Basis bietet in diesem Fall ein Web-Bundle (WAB). Dieses entspricht einem normalen Bundle, angereichert mit zusätzlichen Headern im Bundle-Manifest. Durch die in einem WAB enthaltenen Informationen können Webapplikationen analog zum JEE-Ansatz gestartet werden.

In diesem Artikel wird ausschließlich auf die Verwendung der HTTP-Service-Spezifikation eingegangen. Im Zuge des Open-Source-Projekts *lunifera.org – OSGi-services for business applications* gab es reifliche Überlegungen, ob die Web-Applications-Spezifikation verwendet werden soll. Wegen der höheren Flexibilität des *HttpServices* haben wir uns dagegen entschieden und eine Implementierung einer Vaadin-7-OSGi-Bridge auf Basis des *HttpService* umgesetzt.

Gerne möchte ich Ihnen nun einen Einblick geben, wie die Bridge implementiert wurde und wie einfach Sie diese in Ihren eigenen Projekten verwenden können. Ziel der Umsetzung war es, dem Entwickler eine möglichst einfache und dennoch sehr flexible Implementierung auf Basis von OSGi-Spezifikationen zu bieten, um Vaadin 7 optimal in OSGi-Umgebungen nutzen zu können.

Dieser Artikel setzt Basiswissen in OSGi und den Serviceerweiterungen OSGi-DS (Declarative Services) [2] sowie OSGi-CM (Configuration Admin) [3] voraus. Unter „Links & Literatur“ am Ende des Artikels finden Sie hierzu Informationsquellen.

Die grundlegende Architektur (**Abb. 1**) ist denkbar einfach. Es existiert ein OSGi Service mit dem Namen *VaadinApplicationService*. Dieser Service enthält die notwendigen Informationen und kapselt sämtlich Schritte, die notwendig sind, um Vaadin 7 erfolgreich an einem *HttpService* zu registrieren. Dazu ist die Registrierung von Servlets, Ressourcen und Filtern notwendig.

Der *VaadinApplicationService* kann mittels OSGi-CM erzeugt und konfiguriert werden. Informationen wie der URL-Alias bzw. der Name des Service werden als Properties übergeben. Um mehrere Instanzen erstellen zu können, wird auf das Konzept der *ManagedServiceFactory* von OSGi-CM zurückgegriffen. Die *VaadinServiceFactory* ist dafür verantwortlich, auf Basis von OSGi-CM-„Commands“ neue Services zu erstellen und bestehende zu verändern bzw. zu entfernen.

Welche Implementierung der *HttpService* zur Verfügung stellt, ist erst einmal unerlässlich. Allerdings benötigt *VaadinApplicationService* eine Erweiterung des

HttpService, um auch Filter registrieren zu können (Kasten: „ExtendedHttpService“).

Nachdem die Webressourcen am *ExtendedHttpService* registriert wurden, stehen diese auch im *ServletContext* eines Webservers zur Verfügung und können per Browser angesprochen werden.

Als notwendiges Servlet wurde *org.lunifera.web.vaadin.osgi.servlet.VaadinOSGiServlet* implementiert, welches am *HttpService* registriert wird und *VaadinServlet* erweitert.

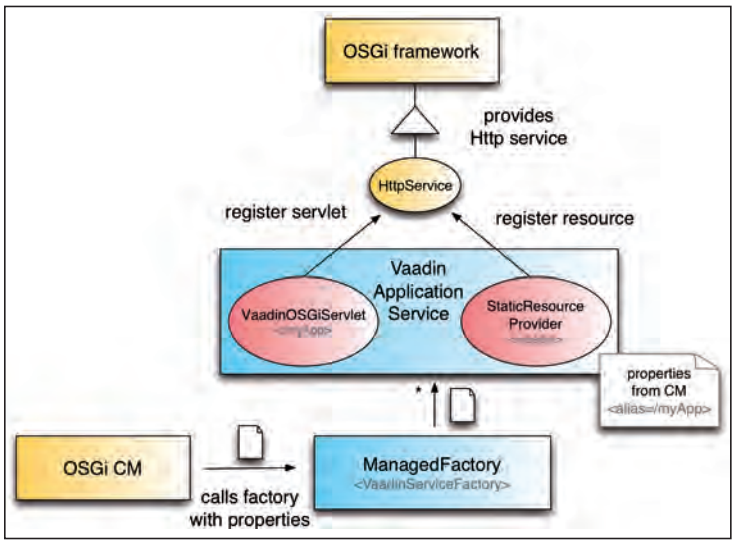


Abb. 1: Architektur

Vaadin-UI-Instanzen

Sobald das Vaadin-Servlet einen Request vom Browser erhält, muss eine *VaadinSession* erstellt werden, die als Abstraktion der Verbindung zum Browser für einen User gesehen werden kann. Für jeden einzelnen geöffneten Browser-Tab erstellt die *VaadinSession* eine Instanz von *com.vaadin.UI*. Dies ist der Punkt, an dem Vaadin mittels *UI#init()* den Custom-Code ausführt, der schlussendlich die darzustellende Applikation beschreibt.

Im Zuge der Vaadin-OSGi-Bridge wurde darauf geachtet, dass die Erzeugung der UI-Instanz durch eine OSGi-DS *ComponentFactory* durchgeführt wird. *ComponentFactories* erlauben es, auf Basis einer Konfiguration in XML-Form beliebig viele Instanzen eines Service zu erstellen. Diese werden automatisch in der Service Registry registriert. Einer der großen Vorteile ist, dass in den erzeugten Instanzen das Whiteboard-Pattern verwendet werden kann, um weitere OSGi Services automatisch in die gerade eben erstellte Serviceinstanz per Method Injection setzen lassen zu können. Method Injection bedeutet, dass eine *bind*- und eine *unbind*-Methode definiert und dann vom Framework aufgerufen werden. Die referenzierten Services werden ebenfalls in der XML-Konfiguration definiert und durch das Framework automatisch injiziert. Die Verwendung des Whiteboard-Patterns zeigt auch den großen Vorteil dieser Umsetzung: Beliebige Services stehen – sofern diese vorhanden sind – nach Instanziierung einer neuen UI-Instanz zur Verfügung. Zum Beispiel kann ein *PreferenceService* direkt in die UI-Instanz injiziert werden. Sollten referenzierte Services gestoppt werden, werden diese automatisch aus der UI-Instanz entfernt, und bei Start eines Service wird dieser analog der Konfiguration wieder in den Service injiziert. Der Aufbau wird im Folgenden nochmals im Detail erklärt (**Abb. 2**).

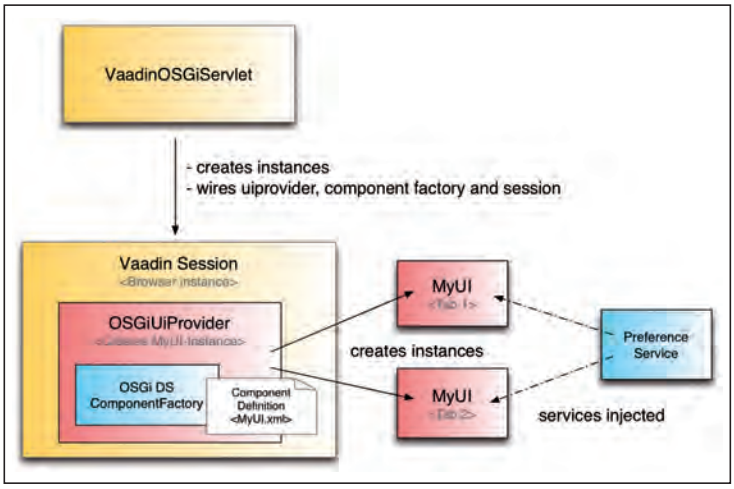


Abb. 2: Erstellung von Vaadin-UI-Instanzen

Definition UI-Klasse

Mittels OSGi-DS wird der *MyUI*-Service definiert. Hierzu wird einerseits eine Implementierung von *MyUI.class* benötigt, die Vaadin den Entry Point in Form einer *UI#init()*-Implementierung bereitstellt. Zum anderen muss eine OSGi-DS-Component-Definition in Form einer XML-Konfiguration erstellt werden, die durch OSGi-DS verwendet wird, um mehrere Instanzen von

ExtendedHttpService

Um Filter am *HttpService* registrieren zu können, wurde eine Equinox-proprietäre Implementierung verwendet. *ExtendedHttpService* stellt eine Erweiterung des *HttpService* dar und erlaubt auch, Filter zu registrieren. Aus diesem Grund hat *VaadinApplicationService* derzeit eine Abhängigkeit zu *equinox.http.servlet*, was die Verwendung der Equinox-*HttpService*-Implementierung notwendig macht. Sobald ein einheitlicher Service seitens OSGi zur Verfügung steht, wird *VaadinApplicationService* auf den neuen Service umgestellt. Aktuell arbeiten wir daran, sowohl Equinox- als auch Apache-*ExtendedHttpService*s zu unterstützen, was allerdings den Einsatz von *lunifera.runtime.web.http* notwendig macht, um geeignete *HttpServices* zur Verfügung stellen zu können.

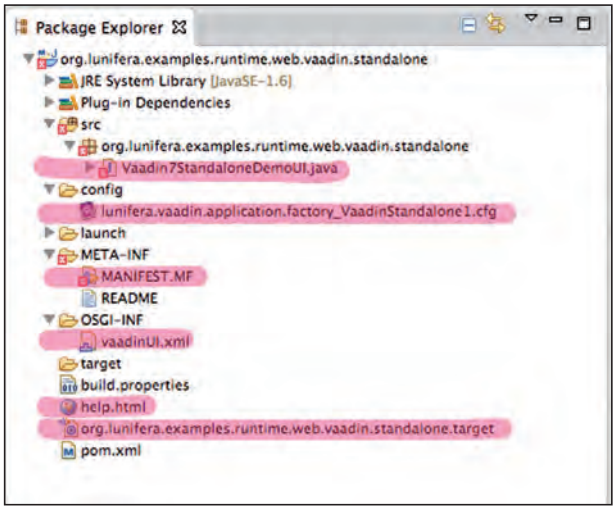


Abb. 3: Workspace nach dem Import des Beispiels

MyUI.class erstellen zu können und die benötigten Services beschreibt.

UI-Provider

Eine *VaadinSession* instanziiert UI-Klassen über so genannte *UI-Provider*. Instanzen dieser werden an der Vaadin-Session registriert. Bei Bedarf verwendet die Session diese, um neue Instanzen von *MyUI.class* anzufordern. Der große Trick bei der Implementierung ist, dass *OSGiUiProvider* an der Session registriert werden. Diese verwenden intern wiederum die definierte OSGi-DS-*ComponentFactory*, um Instanzen von *MyUI.class* als OSGi Services zu erstellen.

Wiring

Die Verbindung zwischen *VaadinSession*, *OSGiUiProvider* und UI-Implementierung wird automatisch vorgenommen. Der Entwickler muss nur bedingt darauf achten, dass die Konfigurationen in der Component-Definition korrekt sind. Alles andere wird automatisch vom *VaadinOSGiServlet* erledigt. Es stellt sicher, dass *OSGiUiProvider* mit der *ComponentFactory* versorgt und dass die *OSGiUiProvider* an der richtigen *VaadinSession* registriert werden.

Das Standalone Example im Überblick

Um eine eigene Vaadin-Applikation erstellen zu können, halten Sie sich am besten an das „Standalone Example", das unter [4] zum Download bereit steht. Es enthält die notwendige UI-Implementierung, die Component-Definition, eine Target-Platform-Definition und Konfigurationen für OSGi-CM. Nach der Installation des Beispiels in den Workspace sollte Ihr Projekt wie in **Abbildung 3** aussehen. Wichtig für die Betrachtung sind die rot markierten Artefakte.

Vaadin7StandaloneDemoUI.java: Dieses stellt die Implementierung des *com.vaadin.UI* dar und somit den Entry Point für Vaadin. Vaadin führt die Methode *init()* aus, um die Custom-Applikation zu instanziieren:

```
@Theme(Reindeer.THEME_NAME)
public class Vaadin7StandaloneDemoUI extends OSGiUI {
  @Override
  public void init(VaadinRequest request) {
    // build your UI
    ...
```

Es muss allerdings darauf geachtet werden, dass die Klasse *OSGiUI* erweitert.

lunifera.vaadin.application.factory_VaadinStandalone1.cfg: Hier sind die Properties enthalten, die von OSGi-CM verwendet werden, um die Vaadin-OSGi-Bridge zu konfigurieren (Kasten: „Config Admin Properties"). Natürlich können Sie in Ihrer eigenen Applikation auch direkt OSGi-CM verwenden, was die Verwendung dieses Property-Files unnötig machen würde. Inhalt des Beispiels:

```
# configuration to start a vaadin service
lunifera.web.vaadin.name=VaadinStandaloneDemo
```

Config Admin Properties

Die Verwendung von OSGi-CM sieht vor, dass auf Basis einer Persistence-ID (PID) eine Konfiguration geladen wird. Mithilfe dieses *Config*-Objekts können neue Properties an den Ziel-Service gesendet werden. Die Methode *update()* des Service hat diese zu verarbeiten und darauf zu reagieren.

Um die Verwendung des Config Admins zu vereinfachen, wurde das Bundle *org.lunifera.runtime.component.configuration.manager* implementiert. Die Klasse *SystemConfiguration-Component* dient als Config-Manager und liest Verzeichnisse in startenden Bundles aus. Dazu wird das Extender-Pattern verwendet, das einen zusätzlichen Eintrag in der *MANIFEST.MF*-Datei benötigt. Durch die Angabe von

```
Lunifera-Config: {Pfad zum Config-Folder}
```

erkennt der Config-Manager, dass in diesem Verzeichnis OSGi-CM-Konfigurationen liegen. Alle Files mit der Endung *.cfg* werden ausgewertet und automatisch in aufbereiteter Form an OSGi-CM weitergeleitet. Dies ermöglicht eine sehr schnelle Konfiguration von Services. Jede *.cfg*-Datei steht für eine eigenständige Konfiguration. Dabei muss der Name der Datei folgender Syntax entsprechen:

```
{Factory-PID}_{Unique_ID}.cfg
```

Im Falle des *VaadinApplicationServices*:

```
lunifera.vaadin.application.factory_{ID}.cfg
```

Die ID wird verwendet, um beim nächsten Start des OSGi-Servers prüfen zu können, ob bereits eine Konfiguration im Configuration Store von OSGi-CM bereitsteht. Wenn ja, wird kein neuer Service mehr erzeugt, zumal der existierende bereits gestartet wird.

```
lunifera.web.vaadin.uialias=standalone
lunifera.web.vaadin.widgetset=com.
vaadin.DefaultWidgetSet
```

```
vaadinUI.xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    factory=
     "org.lunifera.web.vaadin.UI/org.lunifera.examples.runtime.web.vaadin.standalone.Vaadin7StandaloneDemoUI"
     immediate="false" name="org.lunifera.examples.runtime.web.vaadin.standalone.ui">
    <implementation class=
     "org.lunifera.examples.runtime.web.vaadin.standalone.Vaadin7StandaloneDemoUI"/>
</scr:component>
```

Abb. 4: „ComponentFactory"-Definition für UI-Klasse

Mithilfe des Property-Files werden der Name des *VaadinApplicationServices*, der URL-Alias und das zu verwendende Widget Set definiert. Hiermit hat dieser Service alle Informationen, um das Servlet und die Ressourcen korrekt am *HttpService* registrieren zu können.

MANIFEST.MF: Das bekannte Bundle-Manifest. Diesem wurden zusätzliche Require-Bundle-Definitionen hinzugefügt, um das Erzeugen einer Launch-Konfiguration zu vereinfachen.

Ebenfalls enthält es den „Lunifera-Config"-Header, um das *ExtenderPattern* möglich zu machen.

vaadinUi.xml: Die OSGi-DS-*ComponentFactory*-Definition sorgt dafür, dass OSGi-DS mehrere Instanzen der *VaadinUI*-Services erzeugen kann. Es enthält lediglich die Definition der *VaadinUI.class* aus dem Bundle (**Abb. 4**).

Auf zwei Elemente in der Definition muss eingegangen werden:

- „implementation class" definiert die UI-Klasse, somit die Instanz, die von der Factory erzeugt werden soll.
- „factory" enthält zwei verschiedene Informationen:
  - Zum einen eine Konstante *org.lunifera.web.vaadin.UI*. Diese wird von der OSGi-Bridge verwendet, um die *ComponentFactory* laden zu können.
  - Und nach dem „/" nochmals die UI-Klasse. Da die von der OSGi-Bridge verwendete DS-*ComponentFactory* keinen Zugriff auf „implemention class" gibt, allerdings der Vaadin-*UIProvider* bereits vor der Instanziierung der Klasse Zugriff auf das *Class*-Objekt benötigt, muss der Klassenname zusätzlich im *factory*-Element angegeben werden. Mithilfe des *BundleContext* der *ComponentFactory* kann die Klasse mit dem korrekten Bundle Classloader geladen werden.

help.html: Sollten Sie sich bei der Installation des Examples fragen, wie Sie am besten vorgehen, werfen Sie einen Blick in die *help.html*. Alle notwendigen Schritte stehen dort beschrieben.

org.lunifera.examples.runtime.web.vaadin.standalone.target: Dieses File ist die P2-Target-Definition. Es definiert alle notwendigen Bundles und den Ort, um diese automatisch downloaden zu können.

Ausführung des Beispiels

Download des Beispiels:

1. Laden Sie die Datei *VaadinStandalone_OSGi_example.zip* unter [5] herunter.
2. Öffnen Sie eine neue Eclipse-Instanz.
3. Im Package-Explorer: Rechts-Klick, Import | Existing Projects into Workspace | Select Archive file | Browse, dann das heruntergeladene Beispiel auswählen, finish klicken.

Target Platform setzen

Um sicherstellen zu können, dass alle notwendigen Bundles geladen sind, öffnen Sie bitte die Target-Definition *org.lunifera.examples.runtime.web.vaadin.standalone.target*. Der Target-Definition-Editor beginnt sofort mit dem Download aller notwendigen

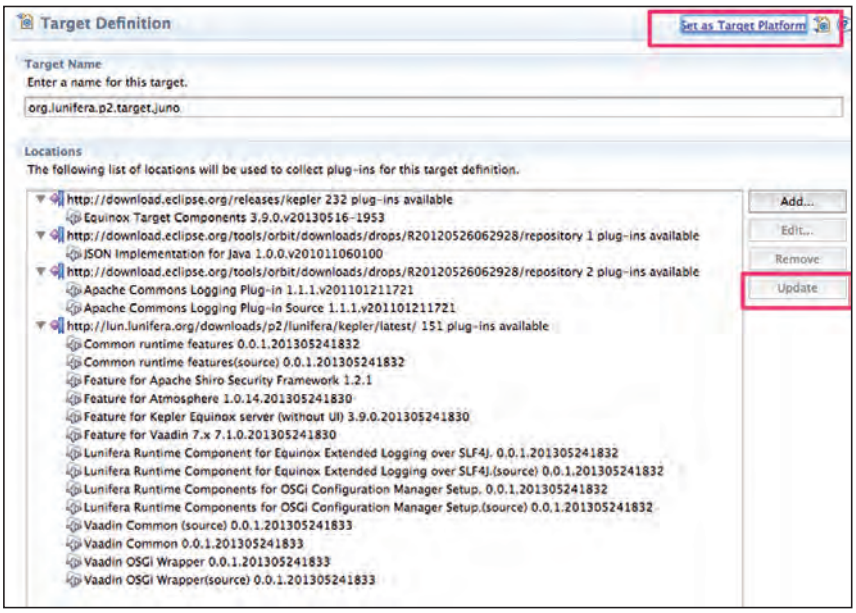


Abb. 5: Aufgelöste Target Platform

Bundles. Sollte es zu dem (wahrscheinlichen) Fall kommen, dass Bundles nicht gelöst werden können, selektieren Sie bitte das fehlerhafte Repository und klicken rechts auf Update. **Abbildung 5** zeigt die aufgelöste Target Platform.

Sobald die Target Platform aufgelöst ist, aktivieren Sie diese bitte mit Set as Target Platform rechts oben. Nach diesem Schritt sollten keine Fehler mehr im Projekt angezeigt werden.

Beispiel starten

Das Beispiel wird mit einer vorkonfigurierten „Launch-Konfiguration" geliefert. Dazu öffnen Sie in Eclipse die Run Configurations und selektieren unter OSGi Framework die Konfiguration vaadinstandalone. Mit Validate Bundles prüfen Sie nochmals, ob Fehler angezeigt werden. Falls ja, dann bitte Add Required Bundles klicken, um alle notwendigen Bundles hinzuzufügen. Der Button Launch startet Vaadin 7. Über den URL http://localhost:8084/standalone sollte Ihnen ein sehr einfaches Vaadin-UI angezeigt werden (**Abb. 6**).

Abb. 6: Das Ergebnis im Browser

Start-Properties Jetty

Um den Port und den Context Path für Jetty festlegen zu können, müssen diese als Properties beim Start des OSGi Frameworks als *vm*-Argumente mitgegeben werden. In der Launch-Konfiguration auf dem Tab Arguments wurde der Port mittels

```
-Dorg.osgi.service.http.port=8084
```

konfiguriert. Der Kontextpfad kann mit folgender Property definiert werden:

```
-Dorg.eclipse.equinox.http.jetty.context.path=/myPath
```

Zusammenfassung

Die Vaadin-7-OSGi-Bridge erlaubt eine sehr schnelle Entwicklung von Vaadin-Applikationen auf OSGi-Basis. Wie bereits oben erklärt, benötigt diese aufgrund der Filter die Equinox-*HttpService*-Implementierung. Allerdings ist das lunifera.org-Team bereits an der Umsetzung einer Lösung, die auch den Apache *ExtHttpService* unterstützen wird.

Ebenfalls existieren im lunifera.org-Projekt weitere Features, die das Deployment auf unterschiedlichen *HttpServices* erlauben und ebenfalls die Definition weiterer Jetty-Serverinstanzen sowie das Mapping von Jetty-Server –> *ServletContext* –> *VaadinApplication* unterstützen. Die einzigartigen Möglichkeiten in Bezug auf die Verwendung von OSGi-CM zur Konfiguration von Vaadin-Applikationen zeigen sich sehr schön unter der Verwendung dieser lunifera.org-Erweiterungen (*lunifera.jetty* und *lunifera.http*). Im laufenden Betrieb können Vaadin-Applikationen auf einen anderen *ServletContext* umgehängt werden. Die Vaadin-Applikation und der Jetty-Server fahren kontrolliert herunter, ein Redeployment wird durchgeführt und die Services werden wieder gestartet.

Um Vaadin-Applikationen manuell starten und stoppen zu können, wurden Equinox Console Commands implementiert – ein hilfreiches Werkzeug, um feststellen zu können, ob Vaadin-Applikationen bereitstehen. Über die zusätzlichen Features und deren Console Commands stehen detaillierte Informationen unter [6] bereit.

Florian Pirchner ist selbstständiger Softwarearchitekt, lebt und arbeitet in Wien. Aktuell beschäftigt er sich als Project Lead in einem internationalen Team mit dem Open-Source-Projekt „lunifera.org – OSGi-services for business applications". Auf Basis von Modellabstraktionen und der Verwendung von OSGi-Spezifikationen soll ein hoch erweiterbarer und einfach zu verwendender Kernel für Businessapplikationen geschaffen werden.

Links & Literatur

[1] http://www.osgi.org/Download/Release4V43

[2] http://wiki.osgi.org/wiki/Declarative_Services

[3] http://wiki.osgi.org/wiki/Configuration_Admin

[4] http://lun.lunifera.org/downloads/examples/vaadin/

[5] http://lun.lunifera.org/downloads/examples/vaadin/VaadinStandalone_OSGi_example.zip

[6] http://lun.lunifera.org/docu/contents/00-Main.html

Anzeige

Anzeige

Anzeige

Vaadin 7 für die Oberfläche einer Clojure-Anwendung

Vaadin meets Clojure

Im Java Magazin 6.2013 wurde die neue Major-Version 7 des RIA-Frameworks Vaadin vorgestellt. Mit Vaadin 7 können ansprechende Webapplikationen schnell und einfach entwickelt werden. Die funktionale Sprache Clojure gibt und hält ähnliche Versprechen in Sachen Einfachheit und Schnelligkeit bei der Entwicklung. Sie bietet außerdem hervorragende Interoperabilität mit Java. Was liegt also näher, als Vaadin 7 für die Oberfläche einer Clojure-Applikation zu verwenden?

von Tobias Bayer

In diesem Artikel wird anhand eines einfachen RSS-Readers beschrieben, wie die Oberfläche einer Clojure-Webapplikation mit Vaadin 7 erstellt werden kann. Der RSS-Reader bietet ein Eingabefeld, in das der Benutzer den URL eines RSS-Feeds einträgt. Ein Klick auf den Fetch-Button lädt den Feed und parst ihn. Die Applikation stellt die Titel der einzelnen Feedelemente in einer Tabelle dar. Klickt der Benutzer auf einen Tabelleneintrag, werden der Feedinhalt und der Link zum vollständigen Artikel unterhalb der Tabelle angezeigt (**Abb. 1**).

Die Webapplikation konfigurieren

Vaadin 7 weist gegenüber den früheren Versionen Änderungen im Bootstrap-Prozess auf. Das Framework liefert nun ein eigenes Servlet mit, das mit einer Klasse initialisiert werden muss, die von *com.vaadin.ui.UI* ableitet. Diese Klasse ist der Einstiegspunkt für den Start der Webapplikation. Sie enthält eine *init*-Methode, in der der Entwickler das UI aufbaut. Das Vaadin Servlet wird mit dieser Klasse in der *web.xml* konfiguriert (Listing 1).

Hier ergibt sich die erste Besonderheit im Zusammenspiel mit Clojure: Clojure kompiliert den Code erst zum Zeitpunkt der Ausführung. Für eine Vaadin-Webapplikation muss die UI-Klasse aber bereits beim Deployment der Applikation vorhanden sein. Der Entwickler steht also vor der Aufgabe, diese Klasse aus seinem Clojure-Code erzeugen zu müssen. Clojure bietet hierfür Ahead-of-Time-Kompilierung (AOT) an. Mit dieser Funktionalität wird ein Clojure Namespace bereits zur Kompilierungszeit in JVM Bytecode übersetzt.

Bei unserem RSS-Reader soll der Namespace *rss-cljvaadin.rssapplicationui* alle UI-relevanten Anteile enthalten. Der Namespace enthält also auch die *init*-Methode und soll deshalb beim Paketieren der Webapplikation per AOT übersetzt werden, damit er von Java aus verwendet werden kann. Dazu fügt man im Namespace-Makro die Option *:gen-class* mit dem Namen der Klasse ein, die aus diesem Namespace generiert werden soll. Die Angabe eines Namens für die zu erzeugende Klasse ist optional. Gibt man ihn nicht an, heißt die Klasse genau wie der Namespace. Da sich aber die Namenskonventionen von Java für Klassen und Clojure für Namespaces unterscheiden, ist es ratsam, die *:name*-Option zu verwenden. Außerdem lässt sich mit *:extends* angeben, von welcher Klasse die generierte Klasse ableiten soll. In unserem Beispiel ist das *com.vaadin.ui.UI* (Listing 2).

Damit der Namespace vorab kompiliert wird, gibt man in der Projektdefinition die Option *:aot* mit dessen Namen an (Listing 3).

Die Applikation bauen und packen

Die Webapplikation wird mithilfe zweier Build-Tools gebaut und gepackt. Das Kompilieren übernimmt *Leiningen*. Dieses versteht sich als Standard-Build-Tool für Clojure-Projekte und kann ein POM-File für Maven automatisch generieren. Maven packt die Applikation

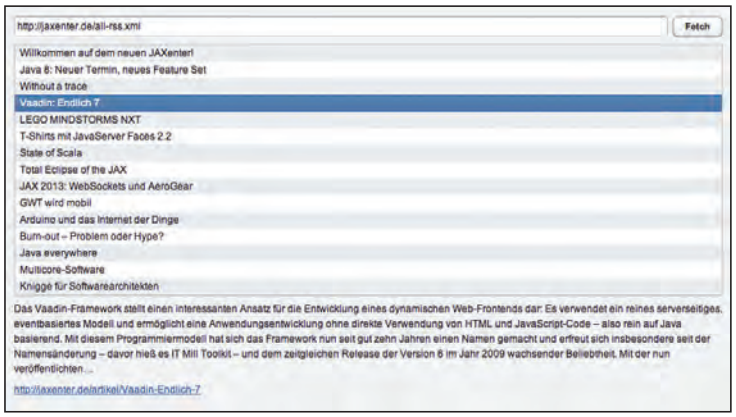


Abb. 1: Oberfläche des RSS-Readers

Listing 1: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">
   <display-name>Clojure Vaadin 7 RSS Reader</display-name>
   <context-param>
      <description>Vaadin production mode</description>
      <param-name>productionMode</param-name>
      <param-value>false</param-value>
   </context-param>
   <servlet>
      <servlet-name>vaadinServlet</servlet-name>
      <servlet-class>com.vaadin.server.VaadinServlet</servlet-class>
      <init-param>
         <description>Vaadin UI</description>
         <param-name>UI</param-name>
         <param-value>rsscljvaadin.RSSApplicationUI</param-value>
      </init-param>
   </servlet>
   <servlet-mapping>
      <servlet-name>vaadinServlet</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Listing 2: Namespace-Deklaration

```
(ns rsscljvaadin.rssapplicationui
  ...
  (:gen-class
   :name rsscljvaadin.RSSApplicationUI
   :extends com.vaadin.ui.UI))
```

Listing 3: Projektdefinition

```
(defproject rsscljvaadin "1.0.0-SNAPSHOT"
  :dependencies  [[org.clojure/clojure "1.5.0"]
                  [com.vaadin/vaadin-server "7.0.3"]
                  [com.vaadin/vaadin-client-compiled "7.0.3"]
                  [com.vaadin/vaadin-themes "7.0.3"]
                  [javax.servlet/servlet-api "2.5"]]
  :aot [rsscljvaadin.rssapplicationui])
```

Listing 4: Bauen und Packen der Applikation

```
lein compile
lein pom
mvn war:war
```

in ein WAR. Um den Packprozess möglichst einfach zu halten, wird in diesem Beispiel das Standardlayout für Webprojekte verwendet. Die *web.xml* befindet sich also in *src/main/webapp*. Damit kann Maven die zuvor durch Leiningen erzeugten Klassendateien, die verwendeten Bibliotheken und die *web.xml* zu einem deploybaren WAR packen (Listing 4). Diese WAR-Datei kann in einem beliebigen Servlet-2.5-kompatiblen Container (z. B. Tomcat) deployt werden.

Das UI mit Vaadin erzeugen

Die Beispielapplikation verwendet den Namespace *rss*, der einfache Funktionen zum Parsen eines RSS-Feeds enthält. Auf die Implementierung dieses Namespaces wird im Kasten „RSS-Parsen mit Clojure“ eingegangen.

Vorerst ist nur wichtig zu wissen, dass die Funktion *rss/fetch-feed* eine Liste von Feedelementen in Form von Maps mit den Schlüsseln *:description*, *:title* und *:link* liefert.

Diese Einträge sollen nun anhand eines Vaadin-UI dargestellt werden. Listing 5 zeigt die Funktion *create-main-layout*, die unser Layout erzeugt, und die von Vaadin aufgerufene Funktion *-init*, die unser Layout in den Root Container der Webapplikation einfügt. Alle

Listing 5: Layouterzeugung

```
(defn- create-main-layout
  []
  (let [content-label (create-feed-content-label)
        url-field (create-url-field)
        link-label (create-link-label)
        feed-table (create-feed-table content-label link-label)
        fetch-button (create-button
                      "Fetch"
                      #(display-feed
                        (.getValue url-field)
                        feed-table))]
    (doto (VerticalLayout.)
      (.setMargin true)
      (.setSpacing true)
      (.addComponent (doto (HorizontalLayout.)
                       (.setWidth "100%")
                       (.setSpacing true)
                       (.addComponent
                       (doto url-field
                         (.setWidth "100%")))
                       (.setExpandRatio url-field 1)
                       (.addComponent fetch-button)))
      (.addComponent feed-table)
      (.addComponent content-label)
      (.addComponent link-label))))

(defn -init
  [main-ui _]
  (doto main-ui (.setContent (create-main-layout))))
```

UI-Komponenten werden in einem *VerticalLayout* angeordnet. Um die einzelnen Elemente dem Layout hinzuzufügen, verwenden wir das Clojure-Makro *doto*. Dieses wertet sein erstes Argument aus und führt alle weiteren als Argumente angegebenen Funktionen mit dem Ergebnis der ersten Auswertung als erstes Argument aus. Am Ende wird das Ergebnis der Auswertung des ersten Arguments zurückgegeben. Im Beispiel wird erst der Konstruktor von *VerticalLayout* aufgerufen. Konstruktoraufrufe für Java-Klassen erfolgen in Clojure mit dem Funktionsaufruf *(<Klassenname>.)*. Der Rückgabewert der Funktion ist das neu erzeugte Objekt, in unserem Beispiel ein Objekt vom Typ *VerticalLayout*. Auf diesem Objekt werden nun durch *doto* nacheinander *set-* und *add*-Methoden aufgerufen, um das Layout zu konfigurieren und die UI-Komponenten hinzuzufügen. Am Ende gibt *doto* (und damit die Funktion *create-main-layout*) das fertige Layout mit allen Kind-Komponenten zurück.

In der *let*-Form werden die einzelnen Kind-Komponenten erzeugt, die dann durch *doto* in das Layout eingefügt werden. Beispielhaft wollen wir in Listing 6 die Erzeugung des FETCH-Buttons betrachten. Die Funktion *create-button* erhält zwei Argumente: Das erste Argument ist die Buttonbeschriftung, das zweite ist die Funktion, die ausgeführt werden soll, wenn der Button

RSS-Parsen mit Clojure

In Clojure lässt sich ein RSS-Feed mit wenigen Zeilen Code parsen. Am Ende steht für die Beispielapplikation eine Liste von Maps mit den Schlüsseln *:description*, *:title* und *:link*:

```
(defn fetch-feed
  [url]
  (map #(hash-map
        :title (title %)
        :link (link %)
        :description (description %))
         (items (xml-seq (xml/parse url)))))
```

Die Funktion *fetch-feed* erzeugt zunächst per *xml/parse* eine Baum-Repräsentation des RSS-Feeds. Die Elemente des Baums sind Maps, die jeweils die Schlüssel *:tag* (XML-Tag), *:attrs* (XML-Attribute) und *:content* (Inhalt des Tags) enthalten. *:content* kann dabei wieder eine Liste solcher Maps enthalten, um die Baumstruktur darzustellen. Für unseren RSS-Reader sind nur die Elemente mit dem Tag *items* interessant, da wir nur diese in der Tabelle darstellen wollen. Um den von *xml/parse* zurückgegebenen Baum nicht selbst rekursiv durchlaufen zu müssen, verwenden wir die Funktion *xml-seq*. Diese gibt eine Sequenz aller Knoten zurück, indem sie den Baum rekursiv durchläuft. In der Rückgabe ist also zunächst der oberste Knoten der XML-Struktur mit allen Kindern enthalten. Danach folgen alle Kinder des Knotens und auf diese jeweils wieder deren Kinder (**Abb. 2**).

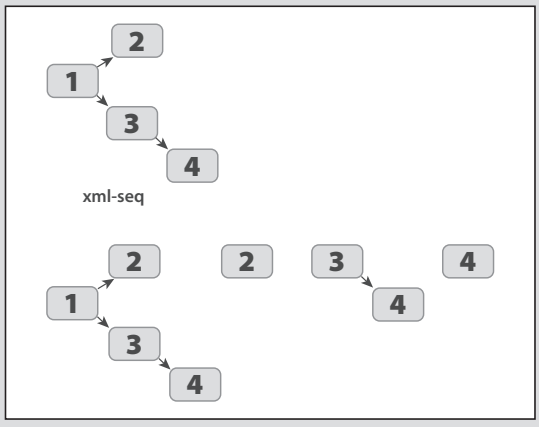


Abb. 2: Baumsequenzierung

Diese Sequenz ist – wie häufig in Clojure – „lazy". Das heißt, dass das rekursive Aufspalten des XML-Baums keinen zusätzlichen Speicherplatz benötigt, weil die Elemente der Sequenz erst dann berechnet werden, wenn darauf zugegriffen wird (in unserem Fall beim Filtern der Elemente):

```
(defn- items
  [feed]
  (filter-tag feed :item))

(defn- filter-tag
  [content tag]
  (filter #(= (:tag %) tag) content))
```

Die Funktion *items* filtert nun diese Sequenz auf alle Einträge, deren *:tag* „item" lautet. Nur diese Elemente des RSS-Feeds sind für uns interessant. Die gefilterten Elemente werden dann in eine Liste von Maps überführt, deren Schlüssel die drei oben genannten (*:description*, *:title* und *:link*) sind:

```
(defn- node-value
  [node]
  (first (:content node)))

(defn- subnode
  [node tag]
  (first (filter-tag (:content node) tag)))

(defn- title
  [item]
  (node-value (subnode item :title)))
```

Die Funktion *node-value* gibt den Wert eines Knotens (also den Inhalt der Map für den Schlüssel *:content)* zurück. *subnode* gibt den ersten Knoten mit dem angegebenen Tag unterhalb eines gegebenen Knotens zurück. Die Funktion *title* kombiniert schließlich diese beiden Funktionen, um den Titel eines Feedeintrags zu extrahieren (die fertige Applikation enthält je eine analog gestaltete Funktion *description* und *link)*. Mit der so erzeugten Liste von Maps (Feedeinträgen) kann nun im UI-Namespace der Applikation weitergearbeitet werden.

geklickt wird. Hier wird die Funktion *display-feed* mit dem URL, den der Benutzer in das entsprechende Feld eingetragen hat und der *feed-table* als weiterem Argument (damit die Feedelemente in die Tabelle eingetragen werden können) angegeben. Durch das #-Zeichen wird eine anonyme Funktion erzeugt, die an die Methode *create-button* übergeben wird. *create-button* erzeugt daraufhin einen neuen Button mit der Beschriftung als Konstruktorargument und ruft mit dem zuvor beschriebenen *doto*-Makro die Funktion *add-action* auf. Diese bindet die von uns implementierte Funktion *display-feed* als Listener an den Button, sodass sie ausgeführt wird, sobald der Benutzer den Button klickt. *create-button-click-listener* verpackt die *action*-Funktion in einen Vaadin Listener. Dazu implementiert sie mit *reify* das Vaadin-Interface *Button$ClickListener* so, dass beim Aufruf der Methode *buttonClick* die übergebene *action*-Funktion ausgeführt wird.

Listing 6: Fetch-Button

```
(defn- create-button-click-listener
  [action]
  (reify Button$ClickListener
    (buttonClick
          [_ evt]
          (action))))

(defn- add-action
  [button action]
  (.addListener button (create-button-click-listener action)))

(defn- create-button
  [caption action]
  (doto (Button. caption) (add-action action)))
```

Listing 7: Tabellenbefüllung

```
(defn- create-container
  [items]
  (let [c (IndexedContainer.)]
    (.addContainerProperty c "Title" String nil)
    (.addContainerProperty c "Link" String nil)
    (.addContainerProperty c "Description" String nil)
    (doseq [item items]
      (let [i (.addItem c (Object.))]
        (.setValue (.getItemProperty i "Title") (:title item))
        (.setValue (.getItemProperty i "Link") (:link item))
        (.setValue (.getItemProperty i "Description") (:description item))))
    c))

(defn- display-feed
  [url table]
  (.setContainerDataSource table
(create-container (rss/fetch-feed url))
(java.util.ArrayList. ["Title"])))
```

Die Funktion *display-feed* ruft nun die eingangs erwähnte Funktion *fetch-feed* aus dem *rss* Namespace auf und füllt die Tabelle mit den zurückgegebenen Feedelementen (Listing 7). Dazu erzeugt sie mit *create-container* einen Vaadin *IndexedContainer*, der als Datenquelle für die Tabelle dient. Dem Container werden alle Properties hinzugefügt, die in den übergebenen *items* enthalten sind. Dann werden die einzelnen Feedelemente per *doseq* in einer Schleife durchlaufen und in den Container gesetzt. Die Vaadin-Methode *addItem* erwartet eine eindeutige ID für das neue Element im Container. Da in unserer Beispielapplikation diese ID nicht weiter interessant ist, erzeugen wir jeweils ein neues Java-*Object*, das als Element-ID dient. Obwohl der Container nun alle Attribute der Feedelemente enthält, um den Detailbereich bei einem Klick auf eine Tabellenzeile mit dem Inhalt des Containers zu befüllen, soll in der Tabelle nur der Titel angezeigt werden. Deshalb wird beim Setzen der Datenquelle noch zusätzlich ein Array mit den darzustellenden Spalten übergeben.

Die Reaktion auf einen Klick auf eine Feedzeile in der Tabelle erfolgt nach einem ähnlichen Muster wie der Klick auf den FETCH-Button und kann im vollständigen Quellcode auf GitHub [1] nachgelesen werden. Beim Klick werden in der Detailansicht unterhalb der Tabelle der Feedinhalt und der Link zum Originalartikel dargestellt. Die Informationen werden dabei direkt aus dem Container der Tabelle gelesen.

Fazit

Durch die mächtige Java-Interoperabilität von Clojure ist es einfach, das eigentlich für Java gedachte Vaadin auch für Clojure-Anwendungen einzusetzen. Mit dieser Kombination lässt sich schon mit wenigen Zeilen Code eine kleine und optisch ansprechende Applikation implementieren.

Die Java-Interoperabilität von Clojure hat aber auch ihren Preis: Hat man sich erst einmal an die Ausdrucksstärke und Form einer funktionalen Sprache gewöhnt, fühlt man sich häufig durch den Zugriff auf Java-Klassen und -Objekte zu einen imperativen Stil gezwungen. Für die Integration von Clojure und Swing gibt es das Projekt Seesaw [2], das einen funktionalen Wrapper um die Swing-Bibliotheken anbietet. Eine solche Bibliothek wäre auch für Vaadin wünschenswert.

Tobias Bayer ist Softwarearchitekt und Senior Developer bei der inovex GmbH. Seine Schwerpunkte liegen auf der Entwicklung von Webapplikationen mit Java und mobilen Apps für iOS. Außerdem beschäftigt er sich mit funktionaler Programmierung in Clojure.

tobias.bayer@inovex.de

Links & Literatur

[1] https://github.com/codebrickie/rsscljvaadin

[2] https://github.com/daveray/seesaw

Automatisches Skalieren in der Amazon Cloud

Selbst ist die Cloud!

Wer sich mit dem Thema Skalierbarkeit befasst, sieht sich meist mit zahlreichen Fragen konfrontiert: Wie muss ich meine Java-Anwendung entwickeln, damit diese skaliert? Welche Teile der Anwendung sollte ich auf einzelne Knoten verteilen, welche sollten dupliziert werden, welche aus Konsistenzgründen nur einmal instanziiert? Zunächst muss man aber wissen, wie man Knoten bzw. Ressourcen überhaupt dynamisch je nach Anforderungen bereitstellen und wieder herunterfahren kann. Häufig betrachtet ein Entwickler lediglich den Anwendungsaspekt. Die Infrastruktur ist aber keineswegs zu vernachlässigen, denn sie bildet die Basis für eine Anwendungsskalierung. Erfahren Sie, wie sich mithilfe von Amazon EC2 automatisiert Infrastrukturressourcen, d. h. Knoten anhand von Kriterien wie CPU-Last hinzu- bzw. abschalten lassen.

von Steffen Heinzl, Benjamin Schmeling und Niko Eder

Einer der interessantesten Aspekte von Cloud Computing ist die Elastizität (rapid elasticity). Sie beschreibt die Fähigkeit, abhängig von der Last oder anderen Faktoren dynamisch Instanzen virtueller Maschinen hinzuzufügen bzw. wieder herunterzufahren. Amazon bietet in diesem Bereich verschiedene Dienste an, die eine Anwendung der elastischen Infrastruktur zulassen. Der grundlegende Dienst in diesem Bereich ist die Amazon Elastic Compute Cloud (EC2). EC2 ist ein Dienst, der es erlaubt, über eine Web-Service-Schnittstelle Instanzen von virtuellen Maschinen, so genannte Amazon Machine Images (AMI), zu starten und zu stoppen. Die AMIs können dabei entweder vorkonfiguriert, z. B. mit einem gängigen Betriebssystem (**Abb. 1**) und Webserver verwendet oder auch komplett selbst erstellt werden [1]. Weitere Vorteile von EC2 sind, dass man nur für die Ressourcen

und Dienstleistungen bezahlen muss, die man tatsächlich gebraucht hat (Netzwerkverkehr, Instanzstunden etc.) und Anwendungen und Daten in verschiedene Rechenzentren auf der ganzen Welt verteilen und somit hochverfügbar anbieten kann. Geringere Latenzzeiten sind ein weiterer Vorteil, da das nahegelegenste Rechenzentrum die Anfragen eines Clients entgegennehmen kann und die Last auf die Rechenzentren aufgeteilt wird.

Um eine elastische Infrastruktur richtig nutzen zu können, muss die zu deployende Anwendung horizontal skalierbar sein, d. h. wenn man mehrere Instanzen der Anwendung verwendet, muss sich daraus eine schnellere Abarbeitung des Gesamtarbeitsvolumens ergeben. In der Regel ist das durch die Verteilung von Anfragen mittels eines Load Balancers möglich. Auch hierfür bietet Amazon EC2 bereits einen Dienst an: Elastic Load Balancing. Dadurch wird der eingehende Netzwerkverkehr auf die EC2-Instanzen verteilt.

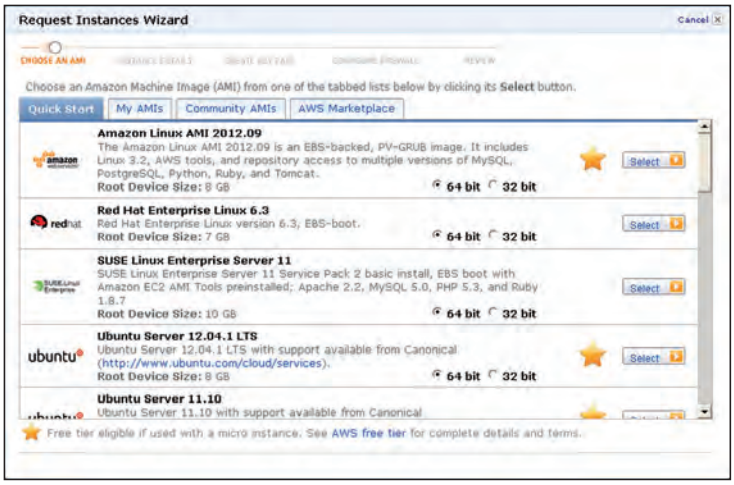


Abb. 1: Auswahl der verschiedenen AMI-Typen im Konfigurationsprozess einer Instanz

Dieser Artikel dreht sich um einen weiteren Baustein, der für die Konfiguration einer skalierbaren Anwendung benötigt wird: automatisches Hoch- und Herunterskalieren. Diese Art der Skalierung wird über einen regelbasierten Ansatz konfiguriert und heißt bei Amazon Auto Scaling.

Ob nun mit oder ohne Load Balancer – mittels hinterlegter Regeln können zusätzliche Instanzen zur Bewältigung von Lastspitzen hinzugeschaltet werden. Für den Endnutzer ist das Zuschalten von Instanzen dabei völlig verborgen. Wird die zusätzliche Rechenleistung nicht mehr benötigt, so lassen sich die hinzugefügten Ressourcen mittels vorher definierter Regeln wieder abschalten. Der Clou an Amazon EC2 ist in so einem Fall die nutzungsgerechte Abrechnung. Demnach muss der Kunde die zusätzliche Rechenleistung nur für die beanspruchte Zeitspanne entrichten. Die teure Anschaffung zusätzlicher Hardware entfällt.

Installation

Amazon stellt für die verschiedenen Dienste diverse Möglichkeiten für eine Konfiguration zur Verfügung: eine Weboberfläche, Kommandozeilentools und/oder Web-Service-Schnittstellen. Da über den Auto-Scaling-Dienst normalerweise ein Regelwerk hinterlegt wird, wann eine Instanz hinzu (oder ab-)geschaltet werden soll, muss man zur Laufzeit nicht selbst auf Änderungen reagieren. Der gängige Weg ist daher, ein Kommandozeilentool zu verwenden, um die Regeln aufzustellen, statt dies über den Programmcode zu erledigen. Eine Einrichtung der Regeln über Eclipse ist aber auch möglich. Alle Developer Tools werden von Amazon auf der Developer-Tools-Seite angeboten [2].

Um das Auto-Scaling-Tool verwenden zu können, muss es zunächst heruntergeladen und entpackt werden. Danach muss die Umgebungsvariable *AWS_AUTO_SCALING_HOME* auf das Verzeichnis, in das das Tool entpackt wurde, gesetzt werden. Eine weitere Umgebungsvariable *AWS_CREDENTIAL_FILE* muss auf eine Datei, die den Secret Key und den Access Key enthält, gesetzt werden. Der Inhalt der Datei könnte wie folgt aussehen:

```
secretKey=xxx
accessKey=xxx
```

Die Credential-Informationen können über das Web von http://aws.amazon.com/security-credentials abgerufen werden. Als Letztes muss noch die Umgebungsvariable *AWS_AUTO_SCALING_URL* gesetzt werden, die angibt, für welche Region die Auto-Scaling-Regeln gesetzt werden sollen, z. B. https://autoscaling.eu-west-1.amazonaws.com für die Region *eu-west-1*.

Nach diesen Einstellungen ist es möglich, Regeln für eine automatische Skalierung festzulegen. Dazu müssen mehrere Schritte durchgeführt werden:

- Anlegen einer *Launch Config*
- Anlegen einer *Scaling Group*
- Anlegen einer oder mehrerer *Scaling Policies*
- Anlegen eines *CloudWatch*-Metric-Alarms [3]

Anlegen einer Launch Config

Innerhalb einer Launch Config wird dem Dienst mitgeteilt, welche Art von Instanzen bei definierten Ereignissen hinzugeschaltet werden sollen. Durch die verschiedenen Parameter wird Folgendes festgelegt:

- der Instanztyp (z. B. t1.Mikro-Instanz) durch *--instance-type*
- die softwareseitige Ausstattung (Betriebssystem, Webserver etc.) durch die Angabe der zu verwendenden AMI durch *--image-id*
- das Monitoring (z. B. CloudWatch oder auch keins) durch *--monitoring*
- die Zugreifbarkeit zukünftiger Instanzen durch *--key*

Abb. 2: Liste der Policies der Scaling Group „MyGroup“

- die Security Group durch *--group*, Letztere regelt zum Beispiel die verfügbaren Ports der VM

Über den folgenden Befehl wird die Launch Config mit der Bezeichnung *MyLC* angelegt:

```
as-create-launch-config MyLC --image-id ami-d0929fa4
--instance-type t1.micro --monitoring-disabled --key EC2_Test
--group quicklaunch-1
```

Diese soll lediglich Instanzen vom Typ *t1.micro* erstellen, auf Basis der verwendeten AMI mit der ID *ami-d31515a7*. Das detaillierte und kostenpflichte Monitoring soll dabei ausgeschaltet bleiben. Gleichzeitig soll auf den Sicherheitsschlüssel *EC2-Test* zurückgegriffen werden sowie auf die Security Group *quicklaunch-1*.

Anlegen einer Scaling Group

Nachdem eine Launch Config für das Auto Scaling eingerichtet wurde, wird im nächsten Schritt eine Auto Scaling Group erstellt. Diese bildet das Herzstück des Auto Scalings und dient der Referenzierung in allen weiteren Vorgängen. Eine Auto Scaling Group beschreibt dabei eine Sammlung von EC2-Instanzen, deren minimale sowie maximale Größe eingangs festgelegt wird. Ihr können wiederum Richtlinien, so genannte Scaling Policies, zugeordnet werden, die dafür sorgen, dass zu der Gruppe weitere Instanzen hinzugefügt oder von der Gruppe Instanzen entfernt werden können. Dies geht selbstverständlich nur bis zu der in der Scaling Group hinterlegten minimalen und maximalen Größe. Gleichzeitig ist ein Verweis auf die zu verwendende Launch Config notwendig.

Für das vorliegende Anwendungsszenario wird eine Scaling Group mit den folgenden Parametern erzeugt:

```
as-create-auto-scaling-group MyGroup --launch-configuration MyLC
--availability-zones eu-west-1c --min-size 1 --max-size 2
--desired-capacity 1
```

Über den Befehl *as-create-auto-scaling-group* wird eine Gruppe mit der Bezeichnung *MyGroup* erzeugt. Deren zugrunde liegende Launch Config ist die im vorherigen Schritt erzeugte *MyLC*. Dies hat zur Folge, dass alle der Launch Group neu hinzugefügten Instanzen mit den Hard- und Softwarespezifikationen gestartet werden, die innerhalb der Launch Config *MyLC* spezifiziert sind. Die *availability zone* regelt wiederum, in welchem Rechenzentrum die Instanzen erzeugt werden sollen. Innerhalb der Launch Config wurde lediglich die Region festgelegt. Für unser Szenario wählen wir die Zone *eu-west-1c*. Die Parameter *min-size* und *max-size* regeln jeweils die minimale sowie maximale Größe der Launch Group. Der minimale Wert muss dabei mindestens eins sein. Über den Parameter *desired-capacity* wird die Anzahl der Instanzen festgelegt, die nach Bestätigung des Befehls *initial* erzeugt werden sollen. In unserem Beispiel wäre dies eine Instanz.

Nach dem erfolgreichen Erstellen einer Scaling Group soll diese im weiteren Vorgehen mit zwei verschiedenen Regelwerken verknüpft werden, die die Skalierung organisieren.

Anlegen einer Scaling Policy

Dazu werden der Scaling Group *MyGroup* zwei so genannte Scaling Policies hinzugefügt. Eine Scaling Policy legt hierbei die Regeln fest, in welchem Rahmen neue Instanzen hinzugefügt werden dürfen. Als Pflichtangaben einer Policy gelten neben der zu referenzierenden Scaling Group die Menge von Instanzen, die nach Auslösung hinzugeschaltet werden sollen, sowie die Art der Änderung, also ob die Anzahl der neu gestarteten oder heruntergefahrenen Menge einem prozentualen Wert oder einer festen Anzahl entsprechen soll. Die Scaling Policy steht im direkten Zusammenhang mit der Scaling Group sowie einem Metric-Alarm – ein Kommando des CloudWatch Command Line Tools, das die Bedingungen für eine Scaling-Aktion festlegt und im folgenden Abschnitt genauer erläutert wird. Eine Scaling Group kann bis zu 25 Scaling Policies besitzen. Für den Anwendungsfall wird der Scaling Group *MyGroup* eine Policy für das Hinzufügen von weiteren Instanzen sowie eine Policy für das Herunterfahren von nicht mehr benötigten Ressourcen über die folgenden beiden Befehle hinzugefügt:

```
as-put-scaling-policy MyScaleUp --auto-scaling-group MyGroup
--adjustment=1 --type ChangeInCapacity --cooldown 60

as-put-scaling-policy MyScaleDown --type ChangeInCapacity
--auto-scaling-group MyGroup "--adjustment=-1" --cooldown 60
```

Über den ersten Befehl wird die Scaling Policy *MyScaleUp* erstellt. Über die Angabe der *--auto-scaling-group* wird dabei auf die Scaling Group *MyGroup* referenziert. Die Maschine weiß somit, welcher Gruppe die benötigten Ressourcen hinzugefügt werden müssen. Im vorliegenden Beispiel soll der Typ der Änderung eine Veränderung der Kapazität bedeuten, die bei Auslösung um eins erhöht wird. Der Parameter *cooldown* ist optional und legt fest, wie vielen Sekunden nach Auslösung einer Policy die nächste Policy-Aktion ausgeführt werden darf. Dies ist besonders bei automatisierten Prozessen von Nutzen, da zum Beispiel nach dem Hinzufügen von neuen Rechenkapazitäten Leistungsspitzen entstehen, die je nach festgelegter Policy ungewollt weitere Ereignisse auslösen können.

Die Scaling Policy steht im direkten Zusammenhang mit der Scaling Group sowie mit einem Metric-Alarm, der die Bedingungen für eine Scaling-Aktion festlegt.

Über den zweiten Befehl wird die Scaling Group *MyScaleDown* erstellt. Das Anlegen der Scaling Policy für das Entfernen von nicht mehr benötigten Instanzen verläuft analog. Neben der Änderung der frei wählbaren Bezeichnung wird lediglich der Parameter *--adjustment=* mit dem Wert *-1* initiiert. Bei Ausführung der Policy soll somit die Anzahl der Instanzen um eine Maschine reduziert werden. Es ist anzumerken, dass der Parameter unter Windows in Hochzeichen gesetzt werden muss.

Nach erfolgreicher Durchführung wurden der Scaling Group *MyGroup* die Scaling Policies *MyScaleUp* und *MyScaleDown* hinzugefügt. Innerhalb des Kommandotools wird daraufhin je eine *arn* ausgegeben. Dies ist eine Zeichenkette, die im weiteren Verlauf gebraucht wird und eine Referenzierung eines CloudWatch-Alarms auf eine konkrete Scaling Policy ermöglicht. Über den Befehl

```
as-describe-policies --auto-scaling-group MyGroup
```

können die Scaling Policies von *MyGroup* aufgelistet werden. Das Ergebnis ist in **Abbildung 2** zu sehen.

Anlegen eines CloudWatch-Metric-Alarms

Im letzten Schritt werden die erstellten Scaling Policies mit einem CloudWatch-Alarm verbunden. Dieser legt die exakten Bedingungen für das Auslösen einer Scaling Policy fest. Hierbei wird eine konkrete CloudWatch-Metrik zur Überwachung ausgewählt. Überschreitet diese einen Regelwert, wird mittels Alarm die Ausführung der jeweiligen Policy angestoßen. Da ein Alarm ein CloudWatch-internes Werkzeug ist, kann dieser nicht innerhalb des Auto Scaling Command Line Tools erstellt werden, sondern bedarf der Installation und Einrichtung des CloudWatch-spezifischen Command Line Tools.

Dessen Installation erfolgt dabei ähnlich der des Auto Scaling Tools; lediglich die Namen der Umgebungsvariablen variieren leicht. Die Umgebungsvariable *AWS_CLOUDWATCH_HOME* gibt an, in welchen Ordner das CloudWatch Command Line Tool (heruntergeladen und) entpackt wurde. Die Umgebungsvariable *AWS_CREDENTIAL_FILE* muss genauso gesetzt werden wie bei der Einrichtung des Auto Scaling Tools. Die Umgebungsvariable *AWS_CLOUDWATCH_URL* gibt an, für welche Region ein Monitoring eingestellt wird, in unserem Fall *eu-west-1* durch den URL https://monitoring.eu-west-1.amazonaws.com.

Beide Policies werden nun mit einem Metric-Alarm verknüpft. Der generelle Aufbau einer Alarmanweisung setzt sich wie folgt zusammen: Über den Befehl *mon-put-metric-alarm* kann ein Alarm erstellt werden. Den wichtigsten Parameter stellt dabei der *--comparison-operator* dar. Er legt fest, ob der Alarm bei Überschreitung oder Unterschreitung einer festgelegten Maßzahl ausgelöst werden soll. Diese wird über den *--threshold*-Parameter auf die gewünschte Größe gesetzt. Eng an diesen Wert gekoppelt ist die *--period*. Sie spezifiziert die

Zeitspanne anhand von Sekunden, in der die Threshold-Zahl innerhalb der Metrik überschritten oder unterschritten werden muss, ehe der Alarm und damit auch die Scaling Policy ausgelöst wird. Um eine eindeutige Referenzierung der gewünschten Scaling Policy zu ermöglichen, ist für den Parameter *--alarm-actions* die *arn* mit anzugeben, die jeweils nach erfolgreicher Erstellung einer Scaling Policy auf der Kommandozeile ausgegeben wurde. Über folgenden Befehl wird ein Metric-Alarm namens *HighCPUAlarm* angelegt, der die Scaling Policy *MyScaleUp* referenziert und damit bei Auftreten des Alarms ein Hochskalieren auslöst:

```
mon-put-metric-alarm HighCPUAlarm --comparison-operator
                GreaterThanThreshold --evaluation-periods 1 --metric-name
                                CPUUtilization --namespace "AWS/EC2"
--period 60 --statistic Average --threshold 50 --alarm-actions
        arn:aws:autoscaling:eu-west-1:316041782606:scalingPolicy:89636ce2-
                facb-4ba3-8296-f6121d01abdf:autoScalingGroupName/
                                MyGroup:policyName/MyScaleUp
--dimensions "AutoScalingGroupName=MyGroup"
```

Die Regelauslösung ist dabei von der CPU-Auslastung der Auto Scaling Group *MyGroup* abhängig, die ab einem überschrittenen Wert von 50 Prozent in einem Zeitfenster von 60 konstanten Sekunden den Alarm auslöst und damit auch die Policy *MyScaleUp*. Dies hätte zur Folge, dass der Scaling Group eine weitere Instanz hinzugefügt wird.

Die Einrichtung eines Metric-Alarms für das Downscaling verläuft analog zum Upscaling durch folgenden Befehl:

```
mon-put-metric-alarm LowCPUAlarm --comparison-operator LessThanThreshold
  --evaluation-periods 1 --metric-name CPUUtilization --namespace "AWS/EC2"
                --period 60 --statistic Average --threshold 49 --alarm-actions
  arn:aws:autoscaling:eu-west-1:316041782606:scalingPolicy:d906797b-518b-
      4cd4-a4d4-09ad599c730d:autoScalingGroupName/MyGroup:policyName/
                                                        MyScaleDown
  --dimensions "AutoScalingGroupName=MyGroup"
```

Ein Unterschied ergibt sich im gewählten Operator *LessThanThreshold*, der dafür sorgt, dass der Alarm bei einer Unterschreitung der Kenngröße ausgelöst wird. Für das Fallbeispiel wurde der Wert auf eine CPU-Auslastung von 49 Prozent gesetzt. Ein weiterer Unterschied ist die Verknüpfung mit der Policy *MyScaleDown*. Diese Verknüpfung sorgt dafür, dass bei Eintreten des Alarms innerhalb der Scaling Group *MyGroup* eine Instanz heruntergefahren wird. Nach erfolgreicher Eingabe wurde den beiden Scaling Policies somit jeweils ein Metric-Alarm zugeordnet. Über den Befehl

```
as-describe-scaling-activities --auto-scaling-group MyGroup
                                --headers --show-long view
```

kann über das Log nachvollzogen werden (**Abb. 3**), ob eine Skalierung erfolgt ist. Über den Befehl

```
as-describe-auto-scaling-instances --headers
```

kann die Anzahl der aktiven Instanzen erfragt werden (**Abb. 4**).

```
C:\Dokumente und Einstellungen\eder>as-describe-scaling-activities --auto-scalin
g-group MyGroup --headers --show-long view
ACTIVITY,ACTIVITY-ID,END-TIME,GROUP-NAME,CODE,MESSAGE,CAUSE,PROGRESS,DESCRIPTION
,UPDATE-TIME,START-TIME
ACTIVITY,822ibe8d-e2dc-4bfd-82d6-7c5351922b55,2013-01-11T12:56:19Z,MyGroup,Succe
ssful,(nil),"At 2013-01-11T12:55:36Z a monitor alarm HighCPUAlarm in state ALARM
 triggered policy MyScaleUp changing the desired capacity from 1 to 2.  At 2013-
01-11T12:55:42Z an instance was started in response to a difference between desi
red and actual capacity, increasing the capacity from 1 to 2.",100,Launching a n
ew EC2 instance: i-858892ce,(nil),2013-01-11T12:55:42.899Z
```

Abb. 3: Logeintrag, der eine Upscaling-Aktivität repräsentiert

```
C:\Dokumente und Einstellungen\eder>as-describe-auto-scaling-instances --headers
INSTANCE  INSTANCE-ID  GROUP-NAME  AVAILABILITY-ZONE  STATE      STATUS   LAUNCH
-CONFIG
INSTANCE  i-858892ce   MyGroup     eu-west-1c         InService  HEALTHY  MyLC
INSTANCE  i-f1c9d3ba   MyGroup     eu-west-1c         InService  HEALTHY  MyLC
```

Abb. 4: Anzahl der gerade laufenden Instanzen

Abb. 5: CloudWatch-CPU-Auslastung

Wie wir sehen können, wurde ein Upscaling einmal ausgeführt, da derzeit zwei Instanzen laufen. **Abbildung 5** zeigt in diesem Zusammenhang die CloudWatch-Metrik der CPU-Auslastung innerhalb der AWS Management Console. Diese verdeutlicht den Rückgang der Rechenlast nach dem Start der zweiten VM.

Da auf ein kostenpflichtiges Monitoring verzichtet wurde, treten ca. fünf Minuten Zeitverzug bei den Messungen auf. Dieser Zeitverzug muss innerhalb des Skalierungsprozesses berücksichtigt werden, um die Skalierung sowie eine realistische Wertausgabe im Zuge des Monitorings zu gewährleisten.

Fazit

In diesem Artikel haben wir die automatische Skalierbarkeit von Amazon EC2 gezeigt. Hierbei wird durch das Zusammenspiel mehrerer Features, wie z. B. Auto Scaling und CloudWatch, eine automatische Skalierbarkeit von Ressourcen erreicht.

Hierzu haben wir zunächst eine Launch Config angelegt, um zu definieren, welche Arten von Instanzen hochgefahren werden sollen. Weiterhin haben wir eine Scaling Group erstellt, die die Launch Config referenziert und u. a. die maximale, minimale und angestrebte Zahl der Instanzen festlegt. Diese Gruppe wird dann wiederum von mehreren Scaling Policies referenziert, die u. a. definieren, wie viele Instanzen beim Skalieren zu- bzw. abgeschaltet werden sollen. Schließlich wurden CloudWatch-Metric-Alarme definiert, die bei bestimmten Bedingungen, wie z. B. bei hoher CPU-Last, eine der definierten Policies auslösen und somit für das automatische Skalieren sorgen. All das ließ sich bequem von der Kommandozeile konfigurieren und erfordert zur Laufzeit keinerlei manuelle Eingriffe. Es ist jedoch auch möglich, die Ereignisse in Logs und Auslastungsvisualisierungen manuell zu beobachten und ggf. einzugreifen, falls dies erforderlich ist. Alles in allem ist Amazon EC2 in Bezug auf automatische Skalierbarkeit eine runde Sache. Wer noch einen Schritt weitergehen möchte und sogar die beschriebenen Konfigurationsschritte automatisch zur Laufzeit durchführen möchte, kann dies mit den angebotenen Java-Programmierschnittstellen unter Eclipse tun [4].

Steffen Heinzl ist Professor an der Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt in den Bereichen Web-Engineering und Cloud Computing. Nach seiner Promotion an der Philipps-Universität Marburg arbeitete er als Forscher bei SAP Research und als IT-Architekt bei der T-Systems International GmbH. Seine Forschungs- und Entwicklungsinteressen liegen in den Bereichen Cloud Computing – insbesondere Platform as a Service – und Web Services. Ferner ist er Autor des Buchs „Middleware in Java“.

Benjamin Schmeling ist Teamleiter der Softwareentwicklung bei UBL Informationssysteme und ist dort zuständig für die Entwicklung von kundenindividuellen Softwarelösungen, Integrationslösungen und Strategy Services. Er beschäftigt sich in diesem Rahmen u. a. mit Themen wie der modellgetriebenen Softwareentwicklung, serviceorientierten Architekturen und Cloud Computing.

Niko Eder ist Studierender an der Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt. Nach erfolgreichem Abschluss des Bachelorstudiums der Wirtschaftsinformatik mit dem Schwerpunkt E-Commerce studiert er im Zuge des konsekutiven Masterprogramms Informationssysteme.

Links & Literatur

[1] Amazon Elastic Compute Cloud: http://aws.amazon.com/ec2

[2] Developer Tools: http://aws.amazon.com/developertools

[3] Amazon CloudWatch: http://aws.amazon.com/cloudwatch

[4] Heinzl, Steffen; Schmeling, Benjamin; Franke, Jonas: „Eclipse, E-Commerce und EC2“, in Eclipse Magazin 4.2013, S. 67–72

Open-Source-BPM für den Werkzeugkasten des Java-Entwicklers

camunda BPM 7.0

Mit camunda BPM gibt es ein BPMS unter Apache-Open-Source-Lizenz. Es ist keine proprietäre Blackbox-BPM-Suite, sondern eine leichtgewichtige „embeddable" Java Process Engine inklusive notwendiger Tools für den Enterprise-Einsatz. Es kann sowohl in Plain-Java-, Spring- oder Tomcat-Umgebungen als auch in Java-EE-Containern, wie JBoss, GlassFish, WebSphere oder WebLogic, eingesetzt werden. Zeit, sich das Ganze einmal genauer anzuschauen und es auch von vergleichbaren Projekten (wie z. B. Activiti) abzugrenzen. In der nächsten Ausgabe folgt dann ein Erfahrungsbericht von Zalando.

von Bernd Rücker

In den letzten Jahren hat sich viel getan im BPM-Bereich. Der Modellierungsstandard BPMN (Business Process Model and Notation) ist weltweit akzeptiert und wird nicht mehr in Frage gestellt. Seit Version 2.0 definiert er auch die so genannte Ausführungssemantik und kann von Process Engines direkt verarbeitet werden – ohne Übersetzung in eine andere Sprache wie BPEL oder jPDL. Auch sind die Engines an sich sehr gereift und dank Open Source für jedermann verfügbar. Ein Trend hat sich dabei in unserer Projektpraxis über die Maßen erfolgreich bewährt: „Embeddable Java Process Engines".

Nachdem wir jahrelang an JBoss jBPM und Activiti mit entwickelt haben, haben wir Anfang des Jahres dann unser eigenes Open-Source-Projekt ins Leben gerufen: camunda BPM. Das Ziel ist es, unsere Vision rund um „BPM + Java" in der Breite zur Verfügung zu stellen – denn die Projekterfahrung hat uns recht gegeben, dass hier offensichtlich Bedarf besteht.

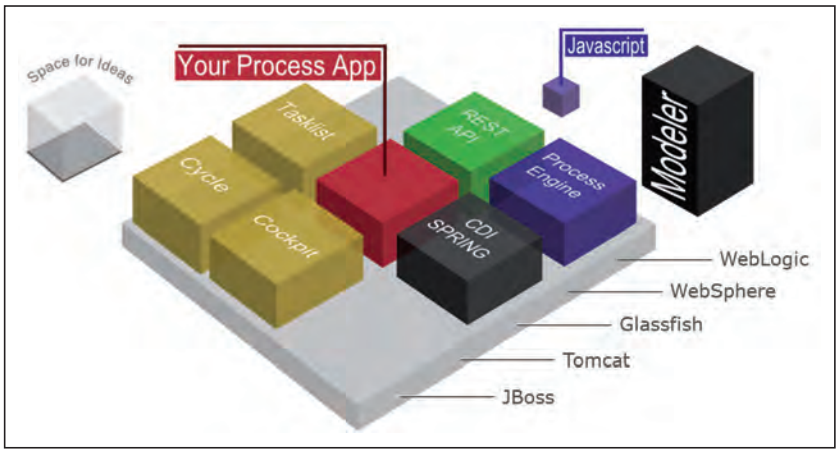


Abb. 1: Komponenten der camunda-BPM-Plattform

Abbildung 1 stellt die Komponenten der camunda-BPM-Plattform dar. Das Projekt liegt zeitgemäß auf GitHub [1]. Aber für alle, die BPM, BPMN oder Workflow-Engines nicht so genau kennen, starten wir mit einem kleinen Beispiel.

Rechnungseingangsprozess

Es mag ein abgegriffenes Beispiel sein – aber dafür ist es übersichtlich. Und ich hatte es auch schon öfter beim Kunden unter der Nase. **Abbildung 2** zeigt den umzusetzenden Prozess; Prozess und Workflow sehe ich von den Begriffen her synonym in diesem Zusammenhang.

Der Prozess wird durch eine eingehende Rechnung gestartet (StartEvent). Daraufhin bekommt die Team-Assistenz eine Aufgabe zugewiesen, um zu bestimmen, wer die Rechnung freigeben soll (UserTask). Dann folgt die Rechnungsfreigabe an sich (UserTask) – je nach Ergebnis entscheidet die Engine, wo es weiter geht (ExclusiveGateway). Ist die Rechnung freigegeben, dann wird sie in der Buchhaltung erfasst (UserTask) und danach wird die Rechnung vollautomatisch archiviert (ServiceTask). Ein typischer Prozess also, er enthält Aufgaben für Menschen aber auch die vollautomatische Orchestrierung von Services. In dem Modell werden noch zusätzliche Attribute hinterlegt – wie in **Abbildung 3** angedeutet. Diese werden von der Process Engine zur Automatisierung gelesen und sind im Gegensatz zum Rest des Models nicht standardisiert.

Moment, ist ein Standard nicht was Gutes? Natürlich, aber nicht um jeden Preis. Mit diesen so ge-

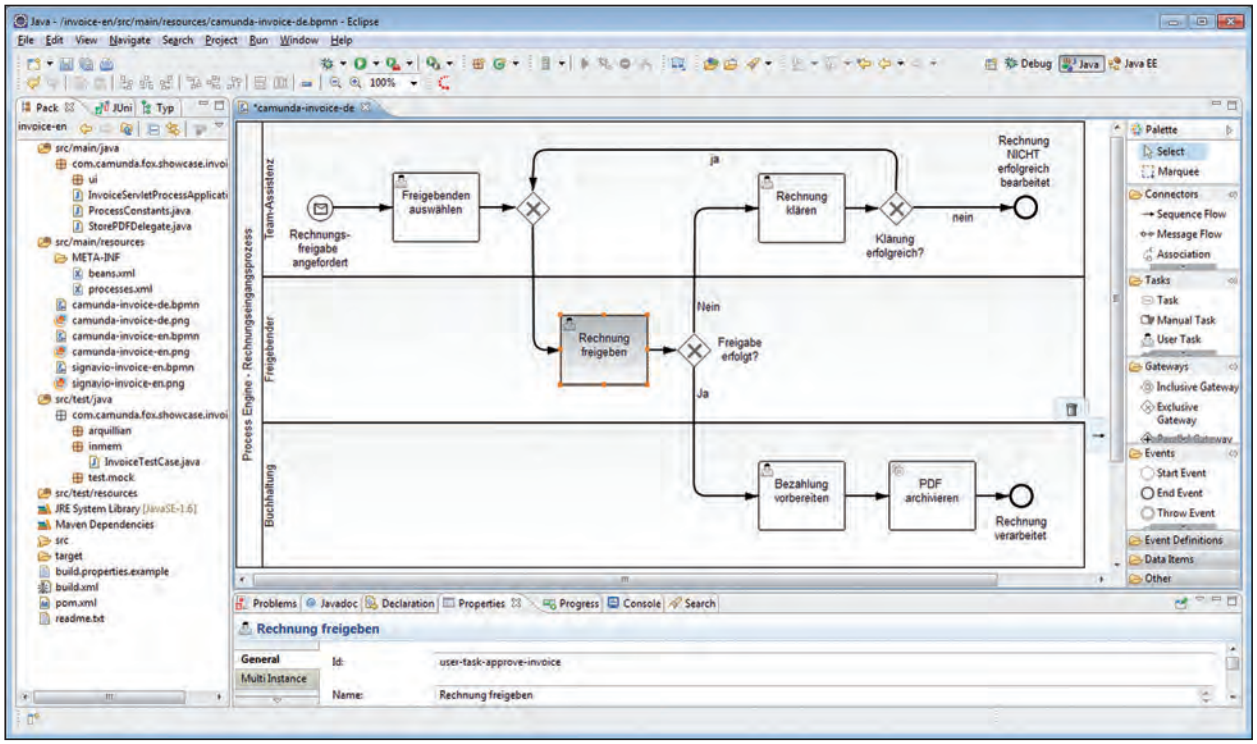


Abb. 2: BPMN-2.0-Prozessmodell im camunda modeler (Eclipse-Plug-in)

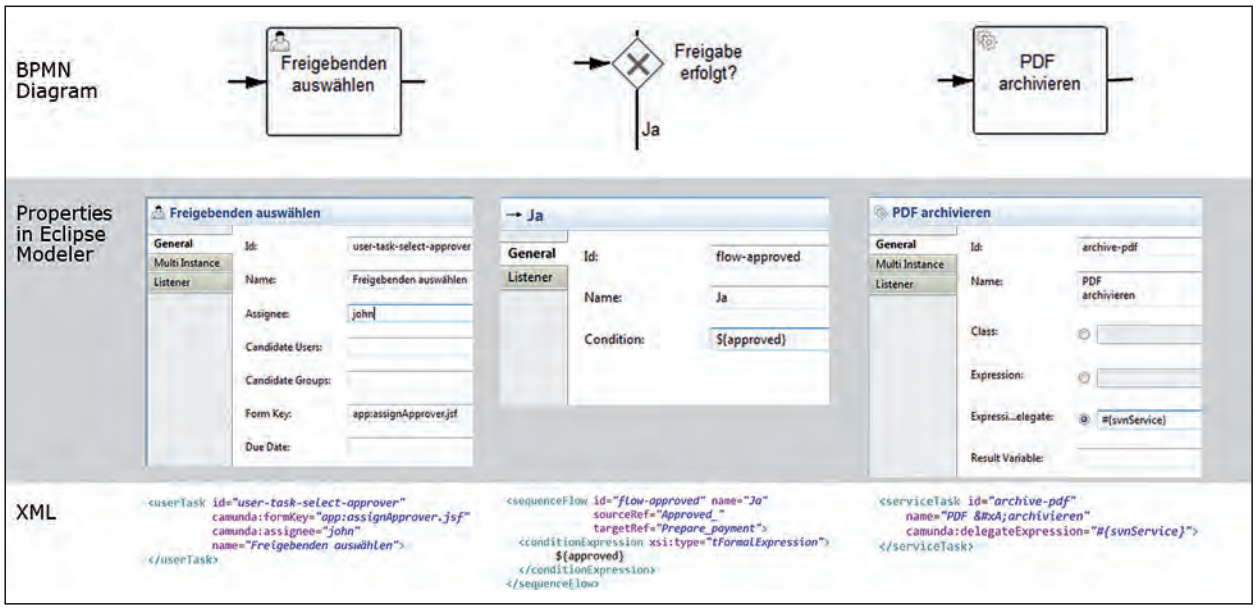


Abb. 3: Attribute für die Ausführung

nannten Extensions – die übrigens vom Standard erlaubt sind – können wir die Engine ganz nah an Java bringen. Das ist zwar nicht mehr hersteller- oder plattformunabhängig, die dadurch reduzierte Komplexität ist uns das aber wert – als Vergleich gibt es in [2] ein Beispiel für den standardisierten Aufruf eines Web Service, dort wird schnell ersichtlich, was ich meine.

Was heißt dann aber Java-nah? Eine Sache ist in **Abbildung 3** zu sehen: Wir benutzen die Java Unified Expression Language (JUEL), die sehr mächtig und bekannt ist – beispielsweise aus JSF. Beim ServiceTask sieht man, dass wir Java-Klassen und Beans direkt referenzieren können, wobei dies sowohl CDI als auch z. B. Spring-Beans sein können – oder natürlich Blueprint, wer es gerne OSGi-oterisch möchte.

Ein zweiter Aspekt ist die Schnittstelle zur Process Engine, hier bietet camunda BPM ein Java-API an, für das wir viel gelobt werden. Listing 1 zeigt einen JUnit Test Case, der einen Durchlauf des Prozesses über das API steuert. Der Test Case fährt im Hintergrund eine Process Engine mit In-Memory-Datenbank hoch, die nur für die Zeit der Testdurchführung lebt. Diese Leichtgewichtigkeit macht es auch ganz einfach, testgetrieben vorzugehen und sollte auch eingefleischte Pro-

cess-Engine-Skeptiker überzeugen. Im User Guide [3] wird noch etwas genauer auf Testing eingegangen – inkl. Verweis auf geeignete Testscopes und Mocking im Kontext von Prozessautomatisierung. Darüber hinaus gibt es unter [4] eine Contribution eines „Fluent Testing API" – die Tests dann noch lesbarer macht, wie in Listing 2 gezeigt.

Im Test Case ist auch zu sehen, wie die Engine mit Daten umgeht: Es können zu jeder Zeit während der Laufzeit Key-Value-Paare geschrieben und gelesen werden. Das können sowohl primitive Datentypen oder serialisierbare Objekte sein. Dabei kann der Serialisierungsmechanismus in der Engine sogar ausgetauscht werden, falls man statt Java-Byte-Arrays lieber XML-Datenstrukturen in der Datenbank möchte. Eine Best Practice ist aber sowieso, möglichst wenige Daten in der Process Engine zu speichern – idealerweise nur Referenzen auf Entitäten, die dann im Fachsystem liegen.

Die Daten (Prozessvariablen) müssen nicht im Prozess deklariert werden, sondern verhalten sich wie eine Map – was den Mechanismus sehr flexibel macht. In der Projektpraxis wird dann meist eine Bean (Java-Klasse) geschrieben, die den Zugriff auf die Variablen regelt und das typsicher, sodass es selten Probleme macht. Das ist allerdings der Grund, dass Daten in der BPMN-Abdeckungsübersicht in **Abbildung 4** als nicht unterstützt aufgeführt sind.

BPMN 2.0

Bisher haben wir lediglich die Spitze des Eisbergs gesehen, was die Mächtigkeit der BPMN angeht. **Abbildung 4** zeigt eine Übersicht aller Symbole mit denjenigen markiert, die von der camunda Process Engine aktuell unterstützt werden. Dabei ist die camunda Engine eine native BPMN-2.0-Prozessmaschine – das BPMN-2.0-XML wird direkt zur Laufzeit interpretiert und die Engine ist auf diese Sprache optimiert. Die Abdeckung der BPMN 2.0 ist dabei sehr gut und weitere Elemente sind bereits auf der Roadmap – für unsere aktuellen Praxisprojekte ist der Stand bereits heute ausreichend.

Oft hört man übrigens das Argument: Die BPMN hat zu viele Symbole und ist zu kompliziert – das können wir aus unserer Erfahrung nicht bestätigen, im Gegenteil. Denn die Symbole sind recht intuitiv, schnell erlernbar und doch sehr aussagekräftig, wodurch sie eine hohe Prozesskomplexität abdecken können, die dann nicht im Code versteckt werden muss. Und bereits heute hat sie eine große Verbreitung erreicht – unser Praxishandbuch BPMN [5] ist zum Beispiel alleine in deutscher Sprache an die 10 000-mal verkauft worden – und fast jeder Wirtschaftsinformatikstudent lernt BPMN auf der Hochschule.

Aber was ist „die Möhre", die wir alle vor der Nase haben? Zum einen hilft uns das Diagramm bereits innerhalb der IT zum Verständnis sowie der Kommunikation. Aber zum anderen hilft es uns eben auch zum viel zitierten „Business-IT-Alignment" – also dem Brückenschlag zur Fachabteilung. In unseren jüngsten Projekten haben wir hier sehr gute Erfahrungen gemacht – sofern sowohl IT als auch Fachbereich wirklich zusammenarbeiten wollen. Hier können sich übrigens auch Entwickler profilieren, indem sie mit geeigneten BPMN-Modellen auf die Fachbereiche zugehen.

In camunda BPM haben wir dann auch das Tool „Cycle" im Gepäck, um den Roundtrip zu beliebigen fachlichen Modelern, die BPMN 2.0 sprechen, zu rea-

Listing 1: Ein JUnit Test Case verwendet das Java-API

```
public class InvoiceTestCase extends ProcessEngineTestCase {

  @Deployment(resources="invoice.bpmn")
  public void testHappyPath() {
    ProcessInstance pi =
                        runtimeService.startProcessInstanceByKey("invoice");

    List<Task> tasks =
          taskService.createTaskQuery().processInstanceId(pi.getId()).list();

    assertEquals(1, tasks.size());
    assertEquals("assignApprover", tasks.get(0).getTaskDefinitionKey());

    Map<String, Object> variables = new HashMap<String, Object>();
    variables.put("approver", "somebody");
    taskService.complete(tasks.get(0).getId(), variables);

    tasks = taskService.createTaskQuery().processInstanceId(pi.getId()).
                                                                  list();
    assertEquals(1, tasks.size());
    assertEquals("approveInvoice", tasks.get(0).getTaskDefinitionKey());
    assertEquals("somebody", tasks.get(0).getAssignee());

    variables = new HashMap<String, Object>();
    variables.put("approved", Boolean.TRUE);
    taskService.complete(tasks.get(0).getId(), variables);

    tasks = taskService.createTaskQuery().processInstanceId(pi.getId()).
                                                                  list();

    assertEquals(1, tasks.size());
    assertEquals("prepareBankTransfer",
                                        tasks.get(0).getTaskDefinitionKey());
    taskService.complete(tasks.get(0).getId());

    assertProcessEnded(pi.getId());
  }
}
```

Anzeige

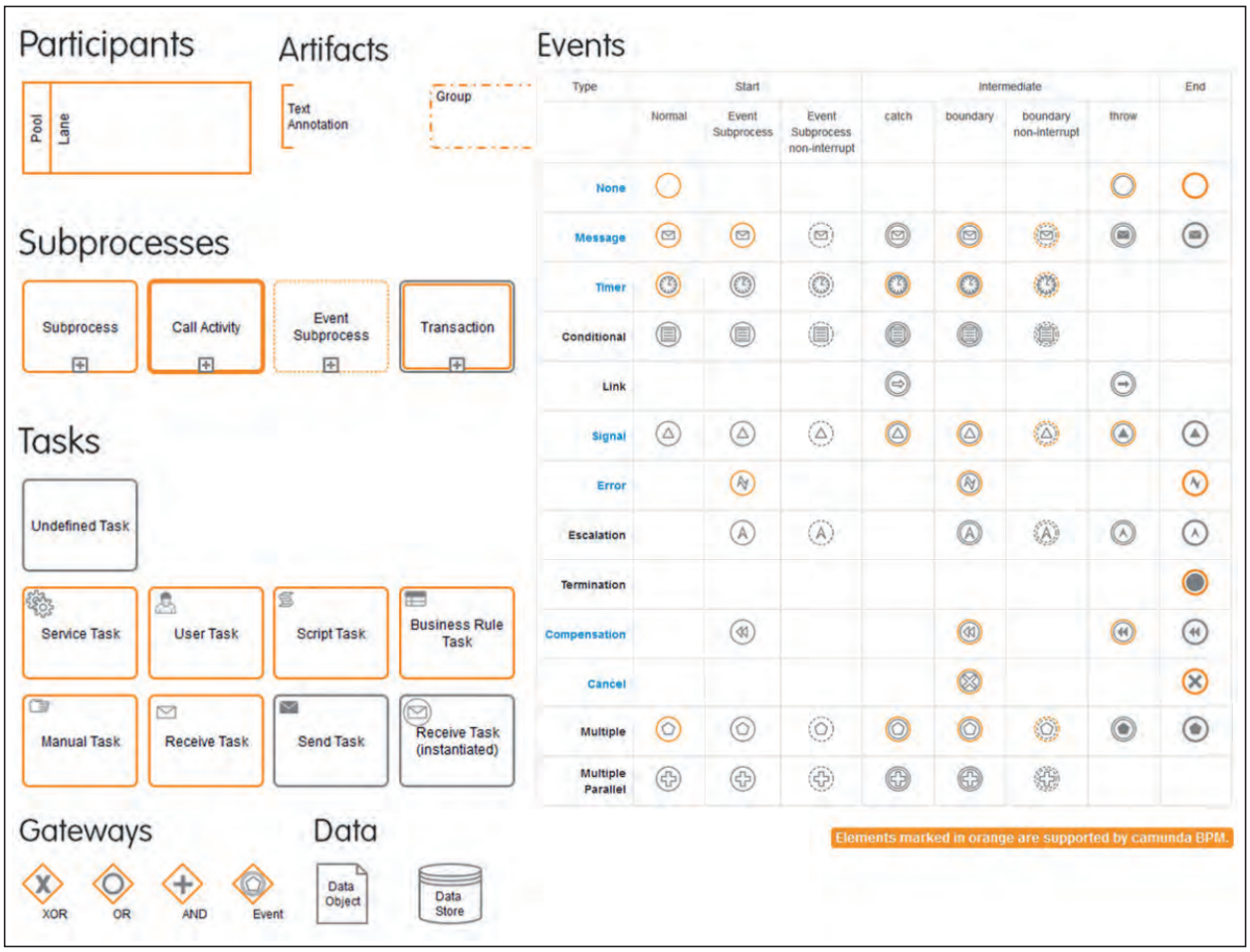


Abb. 4: BPMN-2.0-Übersicht und aktuelle Abdeckung

lisieren – eine genaue Übersicht findet sich in [6]. Da ich das Thema „Roundtrip" in diesem Artikel aus Platzgründen nicht weiter vertiefen kann, empfehle ich dem Leser den Vortrag zu camunda Cycle in [7].

camunda-BPM-Komponenten

An dieser Stelle möchte ich kurz auf die Bestandteile der camunda-BPM-Plattform eingehen – welche in **Abbildung 1** dargestellt waren:

- **Process Engine**: Die native BPMN 2.0 Engine. Dies ist eine einfache Java Library, die in jeder beliebigen Java-Umgebung hochgefahren werden kann und nur Apache MyBatis und eine per JDBC ansprechbare Datenbank benötigt. Von Haus aus unterstützen wir Oracle, DB2, H2, PostgreSQL, MS SQL Server und MySQL. Dies ist jedoch grundsätzlich erweiterbar.
- **CDI- und Spring-Integration**: Wie bereits erwähnt, können Spring und CDI Beans direkt in Prozessen verwendet werden. Andersherum kann auch die Engine als Bean gebunden werden. Für CDI gibt es dabei sogar einen eigenen Scope.
- **Modeler**: Ein BPMN-2.0-Modellierungstool als Eclipse-Plug-in – siehe auch Screenshots in **Abbildung 2** und **3**. Es beherrscht BPMN 2.0 vollständig und zusätzlich die erwähnten Extensions für die camunda Engine.
- **REST-API**: Das in JAX-RS geschriebene REST-API ist auf beliebigen Containern und REST-Implementierungen lauffähig. Die Schnittstelle wird auch für unsere eigenen HTML5-Anwendungen verwendet. Die REST-Schnittstelle kann dabei auch als reine JAR-Abhängigkeit in die eigene Anwendung eingebunden und erweitert (oder limitiert) werden. Details dazu finden sich unter [3].
- **Tasklist**: Eine einfache Aufgabenliste – implementiert in HTML5 – für die Sachbearbeiter im Prozessablauf. Diese stelle ich gleich noch vor.
- **Cockpit**: Eine Anwendung für Betrieb und Monitoring der BPM-Plattform. Da wir ein bereits existierendes Cockpit noch von JSF auf HTML5 umbauen und dabei das Feedback unserer Kunden einarbeiten, ist das Cockpit in der aktuellen camunda-BPM-Version noch etwas rudimentär – wird sich aber schnell weiterentwickeln. Es ist über einen Plug-in-Mechanismus wie in [3] beschrieben erweiterbar, sodass auch eigene Anforderungen an ein Betriebstool umgesetzt und eingehängt werden können. Ziel ist ein Enterprise-taugliches Werkzeug, um die Process Engine im Griff zu halten.
- **Cycle**: Wie bereits erwähnt, setzt Cycle den Roundtrip zwischen dem Modellierungswerkzeug des Entwicklers (normalerweise der camunda Modeler) und dem des „Process Analysts" um.

- **JavaScript**: Wir haben einen vollständigen Renderer für BPMN-2.0-XML im Browser via JavaScript. Das ermöglicht es, Prozessdiagramme zu verwenden, um beliebige Sachverhalte darzustellen – beispielsweise im Monitoring oder der Abarbeitung von Prozessen durch den Sachbearbeiter. Wir haben aber bereits Use Cases in der reinen Dokumentation. So nutzen wir selbst den Renderer für unser BPMN-Tutorial [6]. Geplant ist übrigens auch ein AngularJS-Service als Gegenstück zum REST-API, um noch schneller HTML5-Anwendungen bauen zu können – leider fehlt uns aktuell etwas die Zeit, um es fertigzustellen – aber vielleicht hat ja ein Leser Lust, eine Contribution zu übernehmen?
- **Incubation Space**: Denn in einem eigenen GitHub Repository gibt es Platz für neue Ideen und Contributions – die erst einmal ausprobiert und getestet werden und bei entsprechend positiver Resonanz und Stabilität in camunda BPM übernommen werden. Da könnte morgen also auch schon Ihr Projekt stehen! Nehmen Sie dazu Kontakt übers Forum mit dem Core-Developer-Team auf. Ein leuchtendes Beispiel ist das erwähnte „Fluent Testing API" von unserem Partner Plexiti.

Die gesamte Plattform steht unter Open-Source-Lizenz (der Modeler unter Eclipse Public License und alles andere unter Apache License 2.0) und kann unter www.camunda.org frei heruntergeladen werden. Als fast schon klassisches Open-Source-Geschäftsmodell gibt es bei camunda dann auch eine Enterprise Edition, die den Kunden in Fragen wie Haftung, Stabilität (z. B. die Lieferung von Patches für ältere Versionen) und Support ruhig schlafen lässt.

Eine „Process Application"

Im Beispiel hatten wir einen Prozess erstellt und einen Unit Test geschrieben. Wenn wir die Anwendung live bringen wollen, brauchen wir aber noch eine Oberfläche für die UserTasks und müssen die Anwendung sinnvoll packagen. Nun gibt es verschiedene Möglichkeiten, eine vollständige Anwendung mit Process Engine zu bauen – die Hauptunterscheidung ist:

- **Embedded Process Engine**: Die Anwendung fährt die Engine selbst hoch.
- **Shared Process Engine**: Die Engine wird durch den Container hochgefahren und durch die Anwendungen lediglich benutzt.

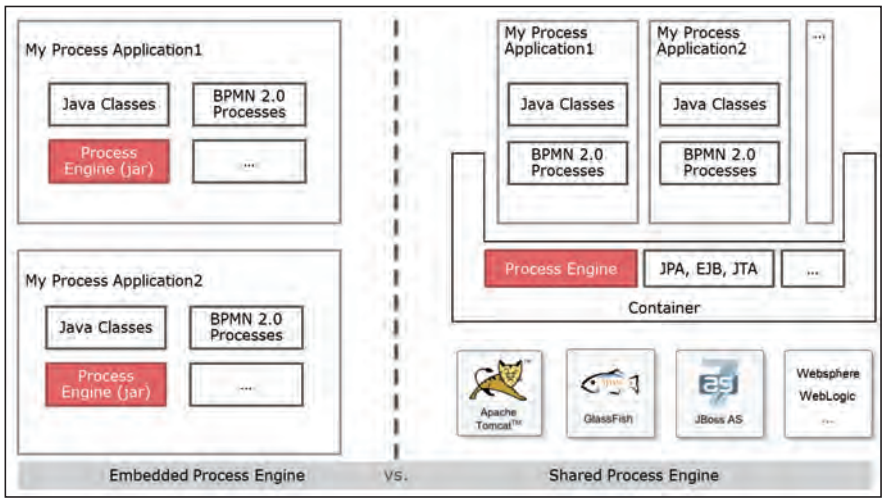


Abb. 5: Embedded vs. Shared Process Engine

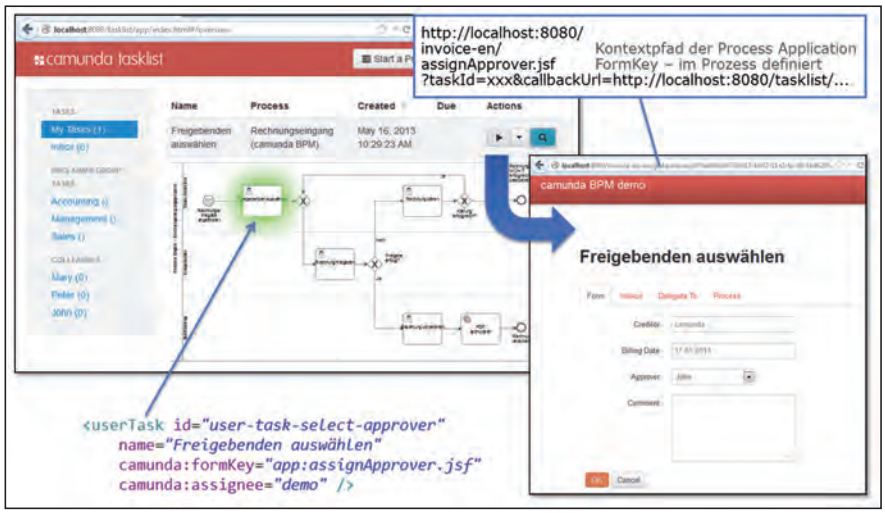


Abb. 6: camunda Tasklist und JSF-Formulare

Abbildung 5 zeigt diesen Unterschied. Als Daumenregel kann man sagen, dass wir die Shared Process Engine bevorzugen – denn dann muss sich der Anwendungsentwickler nicht um die Process Engine kümmern – diese ist Infrastruktur und wird auch vom Container als solche bereitgestellt. Ich vergleiche das gerne mit JPA – dies packt man (zumindest im Java-EE-Kontext) auch nicht mehr selbst ein – es ist einfach da.

Listing 2: Fluent Testing API

```
assertThat(processInstance()).isWaitingAt("review");
assertThat(processInstance().task()).isAssignedTo("piggie");
processInstance().task().claim(USER_STAFF);
processInstance().task().complete("approved", true);
```

Listing 3: Starter-Klasse für ProcessApplication

```
@ProcessApplication("invoice-en")
public class InvoiceServletProcessApplication extends ServletProcessApplication {}
```

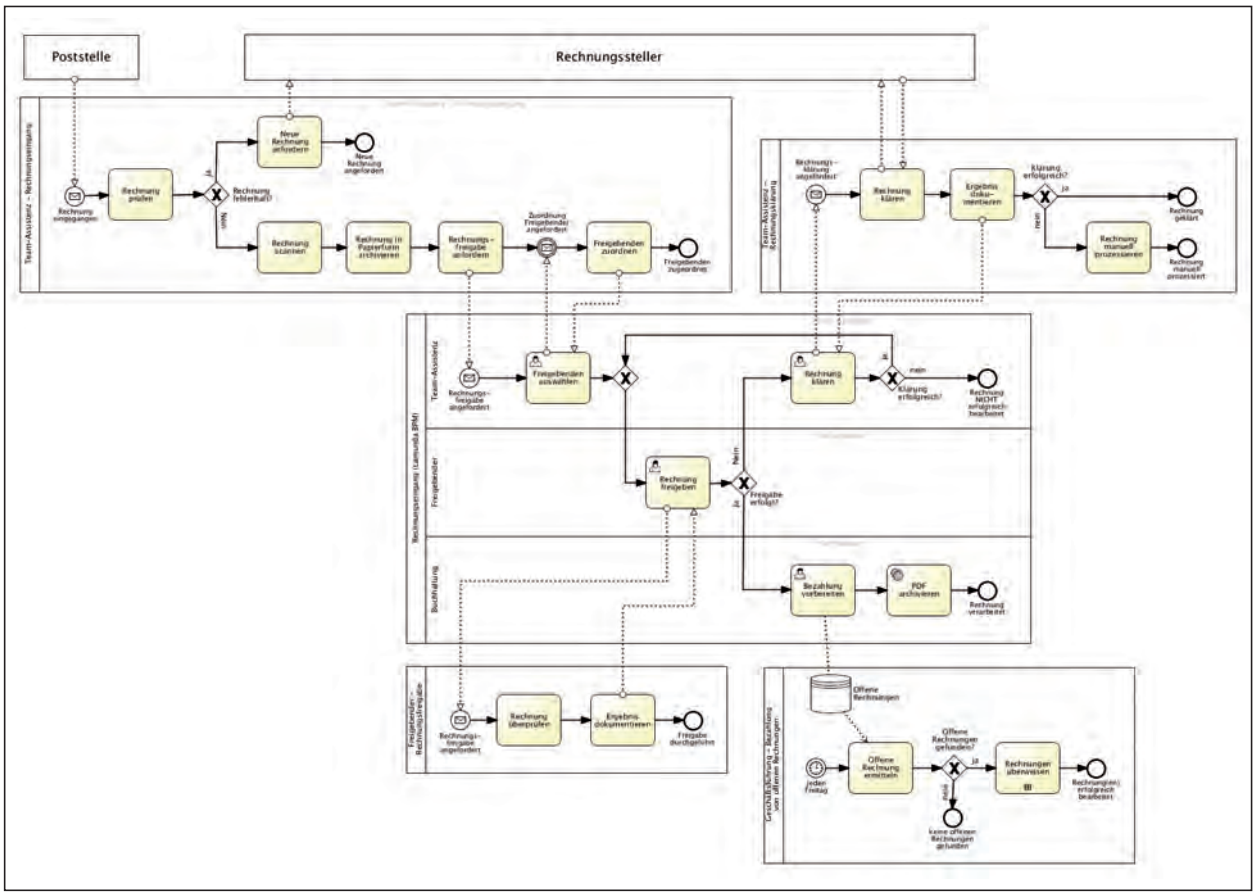


Abb. 7: BPMN-Kollaborationsmodell zeigt mehr als nur den ausführenden Prozess

Eine Shared Process Engine kann in einem Java-EE-6-Container vollständig standardkonform betrieben werden, das bedeutet, dass wir zum Beispiel Thread-Pools vom Container verwenden. In der GlassFish-, WebSphere- und WebLogic-Distribution verwenden wir dafür einen JCA Resource Adapter (RAR). Für den JBoss und Tomcat haben wir sogar spezifisch zugeschnittene Distributionen. Und noch eine kleine Anmerkung mit großer Wirkung am Rande: Natürlich klinken wir die Engine in die JTA-Transaktionsteuerung ein – es können also wie gewohnt Transaktionen verwendet werden – auch Two Phase Commit (XA) ist kein Problem, sofern es der Datenbanktreiber unterstützt.

Auf der Downloadseite des Projekts sind für die Container eigene Distributionen bereitgestellt, in denen die Engine bereits installiert ist. Alternativ kann auch über die Installationsanleitung [3] die Engine in den eigenen Container eingebaut werden. WebSphere- und WebLogic-Distributionen bleiben als einzige unseren Enterprise-Kunden vorbehalten, denn nur Distributionen für Open-Source-Container sind selbst auch frei verfügbar.

Hat man einen so präparierten Container, kann man eine „Process Application“ deployen. Dazu muss einerseits eine Starter-Klasse wie in Listing 3 geschrieben und mit *@ProcessApplication* annotiert werden (ähnlich wie bei JAX-RS) und andererseits ein *META-INF/processes.xml* mit eingepackt werden (ähnlich wie das *beans.xml* bei CDI). Für Details kann ich dem Leser ans Herz legen, den „Get Started Guide“ in [8] unter die Lupe zu nehmen – dort sind auch die Maven-Koordinaten zu finden, um direkt loszulegen – auch mit einer Embedded Engine, wenn gewünscht.

Oberflächen für unseren Prozess

Da die Process Engine über Java oder das REST-API steuerbar ist, kann man die BPM-Oberflächen ganz einfach in die eigene UI-Technologie einbetten, da haben wir in Projekten von JSF über Vaadin, GWT und HTML5 bis zu PHP eigentlich schon alles gesehen. Was man typischerweise baut, ist eine Aufgabenliste und Formulare für diese Aufgaben. Eine mögliche Alternative ist die mitgebrachte „camunda Tasklist“, dann muss man nur noch Formulare für die einzelnen Aufgaben bereitstellen. Dabei kann man die Technologie wieder frei wählen – in der Demoanwendung verwenden wir JSF, wie in **Abbildung 6** dargestellt. Als einfachere Alternative würde es auch reichen, simple HTML-Formulare zu definieren, die dann angezeigt werden. Auch dafür gibt es eine Demo.

Zum Prototyping kann die Tasklist übrigens auch ohne Formulare arbeiten und verwendet dann vollgenerische Formulare, in denen alle Prozessvariablen angezeigt und neue hinzugefügt werden können. Das ist ein sehr spannendes Thema, zu dem gerade im Forum eine Diskussion zwischen Contributoren geführt wird – wir freuen uns hier immer sehr über Feedback oder Beteiligung.

Selbst ausprobieren!

Die gesamte Anwendung ist als Projekt unter [9] zu finden und kann per Maven gebaut und danach direkt in den verschiedenen Containern deployt werden. In **Abbildung 7** ist auch ein so genanntes Kollaborationsmodell der BPMN zu finden, wo zu Dokumentations- und Abstimmungszwecken zusätzlich modelliert ist, was der Mensch im Zusammenspiel mit der Engine so tut – eine sehr spannende Methodik für BPMN-Fortgeschrittene, die unter [5] und [8] weiter ausgeführt ist. Viel Spaß beim Ausprobieren!

Community und Best Practices

Auf den Seiten von www.camunda.org werden wir in Kürze auch Guidelines und Best Practices aus zehn Jahren Projekterfahrung mit BPM veröffentlichen und frei verfügbar machen. Es lohnt sich, öfters mal vorbeizuschauen! So etwas braucht aber natürlich auch Diskussionen und Austausch – daher haben wir die camunda BPM Community ins Leben gerufen und veranstalten regelmäßige Abendveranstaltungen an verschiedenen Orten im deutschsprachigen Raum. Termine sind unter [10] zu finden – schauen Sie doch mal vorbei oder bieten Sie gerne selbst einen Erfahrungsbericht an!

Vision ...

Um die camunda-Vision zu erklären, will ich ganz kurz etwas pathetisch werden: Die Welt automatisiert immer mehr Geschäftsprozesse. Auch wenn gewisse Stimmen den so genannten Taylorismus verteufeln, so fußt doch unser Wohlstand darauf, dass Maschinen unsere Arbeit machen. Automatisierung bedeutet Skalierbarkeit der Geschäftsmodelle. Und große Teile der Geschäftsprozesse bewegen dabei sowieso nur noch Daten, beispielsweise im Banken-, Telekommunikations- oder Medienbereich, aber auch in der Logistik folgt der physische Datenfluss „nur noch“ den Informationen. Damit gibt es sehr viele Kernprozesse, die vollautomatisiert werden können – und natürlich auch werden. Damit spielen BPM und Prozessautomatisierung unserer Ansicht nach eine zentrale Rolle in der Softwarearchitektur der Zukunft.

Und für zentrale Kernprozesse eines Unternehmens – Prozesse, die einen einzigartig machen – eignen sich selten „Zero Code Blackboxes“, die dann doch irgendwann zu unflexibel oder proprietär werden. Wir glauben viel mehr an einen „Best of Breed“-Ansatz, bei denen Komponenten wie benötigt zusammengesteckt werden (wie es bei Cloud-Lösungen übrigens heute schon selbstverständlich wird). Und letztendlich sind die zusammengekauften Suiten großer Hersteller auch nichts anderes – außer, dass sie auf Marketingfolien schön integriert aussehen). Das Java-Ökosystem bietet so viele Möglichkeiten und so viele gute Entwickler, dass hier keine Grenzen gesetzt sind. Mit den angesprochenen Best Practices und Blueprints wollen wir dafür dann auch allen Interessierten helfen, sich im „Best of Breed“-Dschungel zurechtzufinden.

Wir glauben also an „BPM + Java“ mit einer leichtgewichtigen BPM-Plattform, Best of Breed im Java-Kosmos, BPMN 2.0 als gemeinsame Sprache und dem Roundtrip mit fachlichen Tools. Und das Ganze muss natürlich verfügbar sein – also Open Source.

... und Roadmap

Die tagesaktuelle Roadmap kann auf www.camunda.org eingesehen werden. Dabei veröffentlichen wir immer die nächsten zwanzig Epics, die wir mit unserem neunköpfigen Entwicklungsteam angehen. Ansonsten stehen bei uns als größere Baustellen die verbesserte Integration mit Integrationslösungen und ESBs, einfacheres Prototyping, verbesserte generische Formulare sowie die Integration von Case-Management-Ideen an. Zentraler Baustein ist und bleibt aber die Entwicklerfreundlichkeit, Integrierbarkeit und Offenheit der Plattform. Wer dazu Fragen, Anmerkungen oder Vorschläge hat, ist jederzeit im Forum willkommen!

Activiti und camunda BPM

Wir haben von Anfang an die Open Source Process Engine Activiti mit entwickelt und aufgebaut. Unsere konsequente Fokussierung auf BPM und Differenzen in der Zielsetzung des Projekts haben allerdings dazu geführt, dass wir inzwischen das Activiti-Projekt geforkt haben. Details dazu finden Sie in den einschlägigen Blogs. Wichtig für alle Activiti-Kenner ist an dieser Stelle, dass sich camunda BPM unabhängig von Activiti weiterentwickeln wird – die Projekte hängen nicht mehr zusammen.

Bernd Rücker hat vor camunda BPM aktiv an der Entwicklung der Open Source Workflow Engines JBoss jBPM 3 und Activiti mitgearbeitet. Bernd begleitet seit zehn Jahren Kundenprojekte rund um BPM, Process Engines und Java Enterprise. Er ist Autor mehrerer Fachbücher, zahlreicher Zeitschriftenartikel und regelmäßiger Sprecher auf Konferenzen.

Links & Literatur

[1] https://github.com/camunda/

[2] http://www.bpm-guide.de/2010/12/09/how-to-call-a-webservice-from-bpmn/

[3] http://docs.camunda.org/

[4] https://github.com/camunda/camunda-bpm-fluent-testing

[5] http://www.hanser-fachbuch.de/buch/Praxishandbuch+BPMN+20/9783446429864

[6] http://camunda.org/design/

[7] http://www.bpmnext.com/bpmnext-2013-presentations/model-bpms-roundtripping/

[8] http://camunda.org/get-started/developing-process-applications.html

[9] https://github.com/camunda/camunda-outer-space-demos/tree/master/invoice-en

[10] http://camunda.org/community/forum.html

Anzeige

Anzeige

Testdaten mit Sparx Enterprise Architect

Ein Enterprise-Architekt auf Abwegen

In der modernen Softwareentwicklung sind automatisierte Softwaretests mittlerweile gang und gäbe [1]. Leider ist die Erstellung von Testdaten durch den Entwickler speziell bei komplizierter Fachlichkeit des Problembereichs eine aufwändige und fehleranfällige Aufgabe. Anhand des proprietären Tools „Sparx Enterprise Architect“ [2] wird eine Lösung vorgestellt, Testdaten als Objektinstanzdiagramme zu erstellen. Damit ist es möglich, Testdaten in einer intuitiv verständlichen Form anzulegen und automatisch zu dokumentieren. Die inhaltliche Korrektheit dieser abstrakten Darstellung kann leicht von der Fachseite bestätigt werden. Dank mehrerer Sichten auf die Daten lassen sich auch komplexe Testinstanzen übersichtlich und verständlich darstellen.

von Dominik Kleine

Wie erstellen Sie Ihre Testdaten? Ein häufig genannter Weg ist das Kopieren eines produktiven Datenbestands. Während des Transports oder im Anschluss werden die Daten anonymisiert und in einem Dateiformat oder beispielsweise als Datenbankinstanz den Entwicklern zugänglich gemacht. Die Entwickler suchen anschließend in den Daten nach passenden Konstellationen für einen bestimmten Testfall, was bei einem komplexen System mit zig Tabellen erfahrungsgemäß deutlich länger als die Implementierung des eigentlichen Testfalls dauert. Je nach Ausmaß des Datenqualitätsproblems befinden sich fachlich falsche Daten in der Kopie. Verwendet ein Entwickler diese fehlerhaften Daten als Grundlage seiner Tests, kann dies zu einer falschen Implementierung führen.

Eine andere Alternative ist die manuelle Erstellung der Testdaten durch den Entwickler. Hier werden häufig entweder reine SQL-Insert-Statements erstellt oder POJOs mit Werten gefüllt und z. B. per JPA in eine Datenbank persistiert. Der Vorteil dieses Ansatzes ist die hohe Flexibilität des Entwicklers, genau die Konstellation erstellen zu können, die er für einen Testfall benötigt, ohne lange nach passenden Daten suchen zu müssen. Nachteilig ist hier allerdings, dass dies nur für kleine Ausschnitte eines Problembereichs praktikabel ist. Referenzen über mehrere Entitäten, wie sie z. B. in Telekommunikations- [3] oder Stromnetzen [4] üblich sind, wären nur unübersichtlich abbildbar. Zudem muss der Entwickler die fachliche Bedeutung der Attribute und der Relationen zwischen den erstellten Entitäten beherrschen, damit er nicht anhand falscher Annahmen fehlerhafte Software implementiert.

Korrekte Testdaten von Fachspezialisten

Damit Tests auf einer validen Grundlage entwickelt werden können, müssen die Testdaten (sowohl die Eingabedaten als auch die Daten, gegen die getestet wird) von der Fachseite entweder direkt erstellt oder zumindest geprüft und freigegeben werden. Dazu ist eine Form der Testdatenbeschreibung notwendig, die sich auf die Fachlichkeit konzentriert und von technischen Details (wie POJOs oder Insert-Statements) abstrahiert.

Gleichzeitig darf die Beschreibung nicht von der tatsächlichen Umsetzung der Testdaten abweichen, da die Dokumentation sonst Aufwand verursacht ohne Mehrwert zu bieten. Dies ist nur durch einen automatisierten Prozess zu erreichen, sodass aus einer fachlichen Testdatenbeschreibung Testdaten in einer für Unit Tests verwendbaren Form werden.

Die Lösung: Objektinstanzdiagramme

Anhand des im Folgenden dargestellten Beispiels soll demonstriert werden, wie mithilfe von UML-Objektinstanzdiagrammen die genannten Anforderungen für die Erstellung von Testdaten erfüllt werden können. In **Abbildung 1** ist ein einfaches Klassendiagramm eines fiktiven Krankenhausbetriebs dargestellt. Ein Krankenhaus besteht aus Abteilungen, denen Mitarbeiter zugeordnet sind. Ärzte und Schwestern behandeln Patienten, wobei nur Ärzte Diagnosen für Patienten stellen.

In **Abbildung 2** ist ein Testdatennetz basierend auf dem zuvor beschriebenen Klassendiagramm dargestellt. Dem Krankenhaus „St. Johannisstift“ wurden zwei Abteilungsinstanzen „Chirurgie“ und „Radiologie“ zugewiesen. Es ist direkt ersichtlich, welche Personen in den Abteilungen arbeiten (drei Schwestern, ein Arzt) und wie die Attribute

der Personen belegt sind (Name, Geburtsdatum etc.). Zudem wurde ein Patient angelegt, der zwei Diagnosen an aufeinanderfolgenden Tagen erhalten hat.

Umgang mit großen Testdatenbeständen

Das in **Abbildung 2** dargestellte Netz ist noch sehr überschaubar. Sollen allerdings weitere Abteilungen mit mehr Ärzten und Patienten abgebildet werden, wird ein einzelnes Diagramm unübersichtlich. Die Lösung dafür liefert der Enterprise Architect (kurz EA) von Haus aus mit: mehrere Sichten auf die gleichen Daten.

Um das Konzept zu verdeutlichen, ist in **Abbildung 3** die Paketstruktur der Testdaten dargestellt. In diesem Fall wurde eine Schneidung der Testdaten gewählt, die sich an der Hierarchiestruktur des Klassendiagramms orientiert (z. B. Krankenhaus => Abteilungen => Chirurgie). Innerhalb des jeweiligen Levels können nun ein oder mehrere Diagramme mit Ausschnitten der insgesamt existierenden Testdaten erstellt werden.

Abbildung 4 zeigt einen Ausschnitt der Testdaten bezogen auf einen Patienten und seine Diagnosen. In dem Diagramm werden z. B. die Abteilungen des Krankenhauses nicht dargestellt, obwohl diese natürlich im gesamten Testdatenbestand

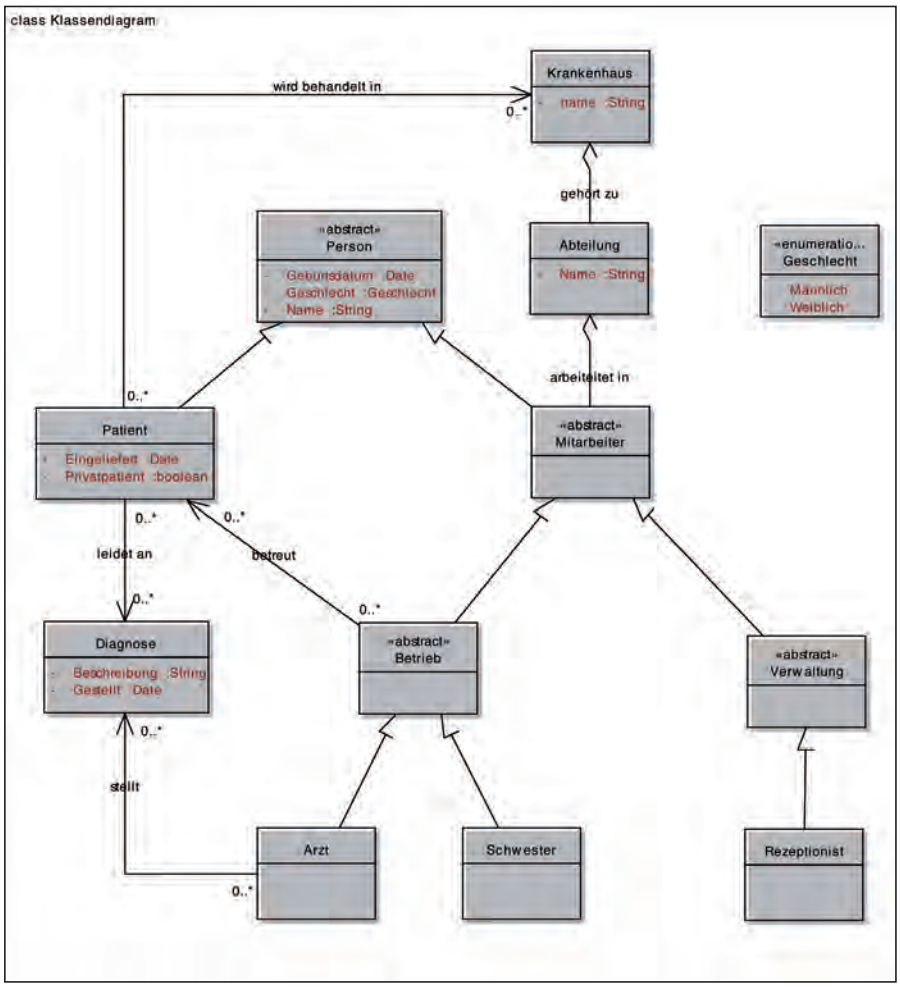


Abb. 1: Beispiel Klassendiagramm Krankenhaus

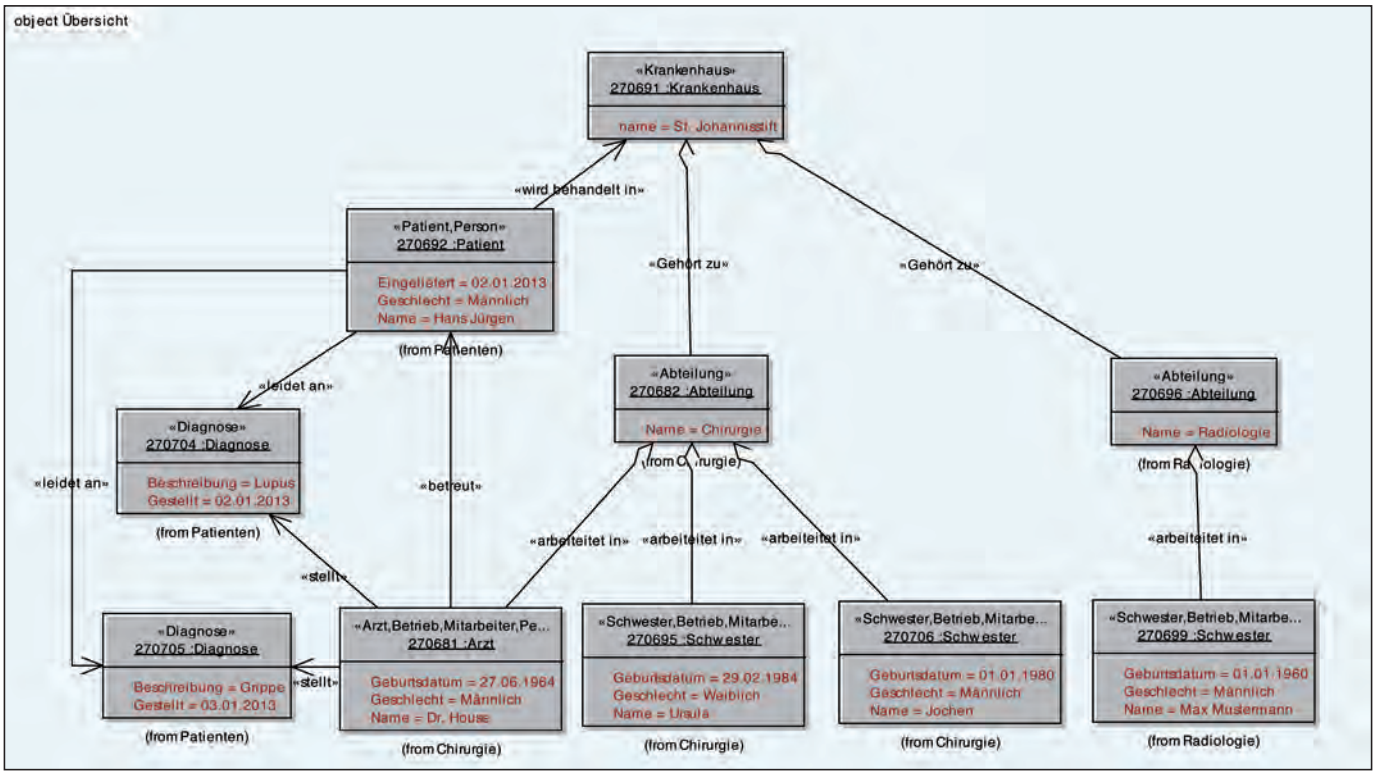


Abb. 2: Testdatennetz als Objektinstanzdiagramm

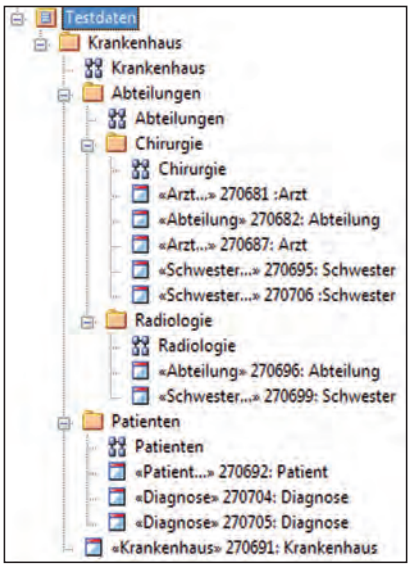


Abb. 3: Testdatenbrowser

vorhanden sind. Durch die farbliche Hervorhebung des Krankenhauses und des Arztes soll verdeutlicht werden, dass die beiden Instanzen noch in anderen Diagrammen vorkommen (wie z. B. in **Abb. 2**).

Anlegen von Testdaten vereinfachen

Ein wichtiger Punkt beim Anlegen der Testdaten ist die Vergabe von Attributwerten. Standardmäßig bietet der EA einen Dialog zur Pflege der sog. „Runstates" (per STRG+SHIFT+R) an. Hier werden die Attribute der dem Objekt zugrunde liegenden Klasse aufgeführt und können mit Werten belegt werden (**Abb. 5**).

Soll ein Element mit anderen bestehenden oder neuen Elementen verbunden werden, kann das im EA über den Quicklinker vorgenommen werden. Der Quicklinker ist ein kleiner Pfeil an der oberen rechten Ecke jedes gewählten UML-Elements, von dem aus neue Elemente mit diesem Element verbunden werden können. In **Abbildung 6** ist das Standardverhalten des EAs bei Verwendung des Quicklinkers dargestellt. Würde also in dem dargestellten Fall der Eintrag „Directed Link" gewählt werden, dann würde der EA ein Element mit dem Namen „Object" auf dem Diagramm erzeugen, das mit dem „Patient"-Element über einen eingehenden gerichteten Pfeil verbunden ist.

Durch die Verwendung eines UML-Profils ist es möglich, die Quicklinker-Definition an den eigenen Bedarf anzupassen. Details zu der Erstellung des Profils finden sich in der Hilfe des EAs (Stichwörter „Create Profiles" und „Quick Linker Example"). Sobald dieses angelegt ist, ermittelt der EA basierend auf den Stereotypen des Quellelements die Liste der Elemente, die über den Quicklinker angelegt werden können (**Abb. 7**).

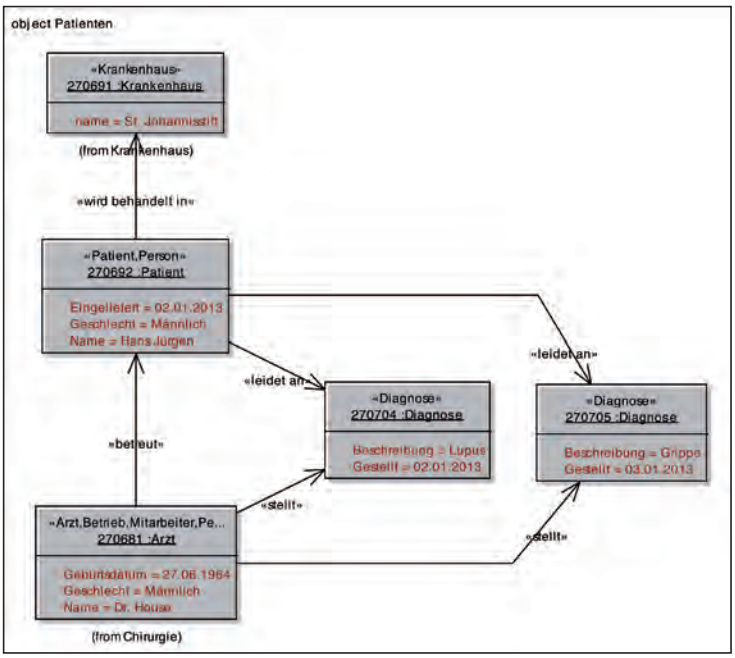


Abb. 4: Teilsicht auf Testdatenbestand

Mithilfe eines dafür zu entwickelnden Plug-ins müssen einige Probleme des EAs beim Umgang mit Objektinstanzen behoben werden. Das neu erstellte Element muss der zugehörigen Klasse zugeordnet werden (siehe EA-Hilfe Element Class | ClassifierID). Erst durch diese Zuordnung können die Attribute über den Runstate-Dialog belegt werden (**Abb. 5**).

Des Weiteren müssen alle Stereotypen anhand der Vererbungsstruktur ermittelt und gesetzt werden. Der EA würde ohne das Plug-in z. B. bei einem neuen „Arzt"-Element nur den Stereotyp „Arzt" setzen. Allerdings erbt ein Arzt auch von „Betrieb" und „Mitarbeiter". Da die Quicklinker nur anhand der gesetzten Stereotypen die Liste der möglichen Relationen zu Zielelementen ermitteln, wäre bei einem „Arzt" nur die Relation zu „Diagnose" möglich. Erst wenn ein „Arzt" auch den Stereotyp „Betrieb" erhält, wird der Quicklinker die Verbindung zu einem „Patienten" anbieten (siehe Klassendiagramm in **Abb. 1**).

Der Name eines Elements wird im Package Browser (**Abb. 3**) dargestellt, wodurch über die eingebaute Suchfunktionen das Element schnell gefunden werden kann. Daher generiert das Plug-in beim Anlegen eines neuen Elements eine eindeutige ID und legt diese als Namen des Elements fest. Das ist besonders hilfreich, wenn in einem Unit Test die ID des fehlschlagenden Elements mit ausgegeben wird. In der hier gezeigten Abbildung wird die ID auf den Wert der *ElementID* (siehe EA-Hilfe Element Class) gesetzt. Für einen anderen Anwendungsfall könnte auch eine komplexere ID-Generierung sinnvoll sein, was bei einer eigenen Implementierung der hier vorgestellten Ideen berücksichtigt werden sollte.

Eine der deutlichsten Vereinfachungen bei der Erstellung von Testdaten kann über die Wiederverwendung von Strukturen als Templates realisiert werden. Wählt ein Anwender mehrere Elemente in einem Diagramm aus, kann er diese über das Plug-in kopieren und als neue Instanzen im gleichen oder einem anderen Diagramm einfügen. Dabei wird u. a. die fachliche ID neu generiert, damit keine eventuell vorhandenen Datenbank-Constraints verletzt werden. So lässt sich z. B. eine komplette Abteilung mit Ärzten und Schwestern als Template speichern und innerhalb von Sekunden ein Krankenhaus mit Abteilungen bevölkern. Damit die Attributwerte nicht immer identisch sind, könnte das Plug-in hier aus einer definierten Menge von Werten beim Einfügen eines Templates die Attribute neu vergeben (z. B. also die Namen der Ärzte und Schwestern).

Export der Testdaten für Unit Tests

In den vorherigen Abschnitten wurden die Idee und die Erstellung von Testdaten mittels Objektinstanzdiagrammen

und eines dafür erstellten Plug-ins beschrieben. Damit lassen sich Instanzbeispiele schon zur Konzeptionsphase von der Fachseite erstellen und in Konzepten verwenden. Der eigentliche Mehrwert der Lösung besteht darin, dass diese übersichtlichen Darstellungen eins zu eins in ein für Unit Tests verwendbares Format gebracht werden können.

Dazu muss ein Exporter entwickelt werden, der je nach Anwendungsfall z. B. SQL-Insert-Statements oder spezifische XMLs o. ä. generiert. Die Ausgabe des Exporters sind also Dateien, die dann von den Unit Tests gelesen und eingespielt werden können.

Sofern der Exporter als Plug-in entwickelt wird, hat dieser Zugriff auf das Element, über das er per Kontextmenü aufgerufen wurde. Der Exporter sollte daher so geschrieben sein, dass er nur die Testdaten aus einem einzelnen gewählten Diagramm exportieren kann. Wenn der Anwender allerdings einen größeren Teil der Daten exportieren will, sollte der Exporter ebenfalls die Möglichkeit haben, die Testdaten aus allen Paketen auf der gleichen oder tieferen Ebene des gewählten Diagramms rekursiv zu exportieren.

Damit die Unit Tests seiteneffektfrei bleiben, empfiehlt es sich, neben den Anweisungen zum Anlegen der Testdaten auch die Anweisungen zum Löschen der Testdaten zu generieren. Am Ende eines Testfalls (z. B. in einer *@After*-Methode bei JUnit) wird die Datei zum Entfernen der zuvor angelegten Testdaten eingespielt.

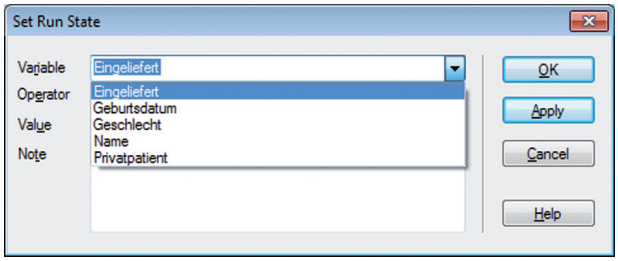


Abb. 5: Attribute eines Patienten

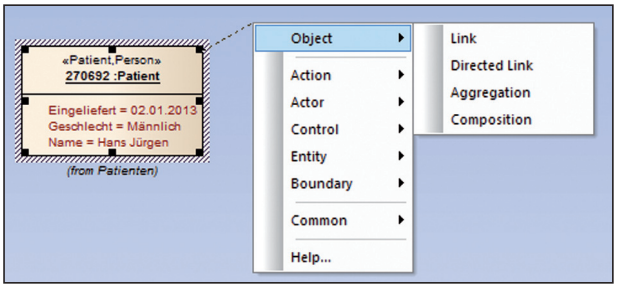


Abb. 6: Quicklinker ohne UML-Profil

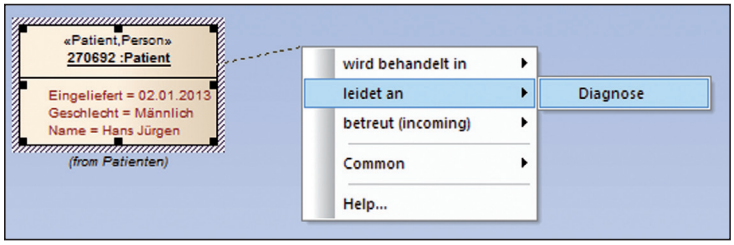


Abb. 7: Quicklinker mit UML-Profil

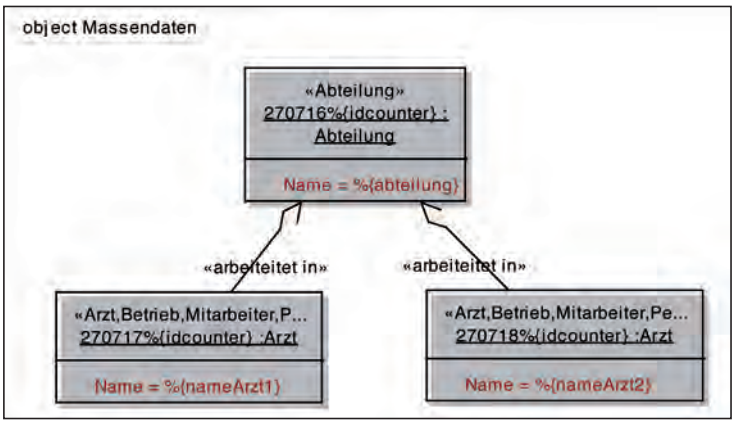


Abb. 8: Testdatennetz für Massendaten

Tabelle 1: Werte für Massendaten

abteilung	nameArzt1	nameArzt2
Chirurgie	Dr. House	Dr. Wilson
Chirurgie	Dr. Foreman	Dr. Taub

Es hat sich als sehr hilfreich herausgestellt, wenn neben den Anweisungen zum Anlegen und Löschen auch die Diagramme als PNG- oder EMF-Bilder exportiert werden (siehe EA-Hilfe Project Class | Put DiagramImageToFile). So kann sich der Entwickler bei der Implementierung der Testfälle oder bei der Korrektur von fehlschlagenden Testfällen die Datenkonstellation ansehen, ohne in den EA wechseln zu müssen.

Testdaten mit Objektinstanzdiagrammen: Qual der Wahl

Es existieren diverse UML-Tools mit unterschiedlichen Vor- und Nachteilen, die häufig auch anhand persönlicher Präferenzen beurteilt werden. Für die Auswahl eines geeigneten Tools zur Testdatenerstellung mit Objektinstanzdiagrammen soll die folgende Liste von Anforderungen als Hilfestellung dienen:

- Das Tool sollte möglichst vollständig per API erweiterbar sein.
- Events wie „ein neues Element wurde in das Diagramm gezogen“ müssen von selbst geschriebenem Code verarbeitet werden können.
- Es muss möglich sein, die erstellten Objektinstanzdiagramme direkt in ein für Unit Tests verwertbares Format zu bringen (z. B. durch einen automatisierten Export in SQL-Insert-Statements).
- Das API sollte eine Möglichkeit bieten, ein Objektinstanzdiagramm als Bild zu exportieren.
- Es müssen mehrere Sichten auf Objektinstanzen möglich sein.
- Die Darstellung der Objektinstanzdiagramme sollte so übersichtlich möglich sein, dass sie sich für die Verwendung in Konzepten eignet (farbliche Hervorhebung, Kommentarelemente etc.).
- Das Tool sollte Back-ups der Objektinstanzdiagramme erstellen können.
- Es sollten auch vorherige Versionen der erstellten Objektinstanzdiagramme abrufbar sein.
- Das Tool sollte die parallele Bearbeitung durch verschiedene Anwender unterstützen.

Der Enterprise Architect erfüllt alle oben genannten Anforderungen, weshalb er für die Umsetzung der Testdatengenerierung mit Objektinstanzdiagrammen gewählt wurde.

Massendaten für Performancetests

Der Exporter lässt sich mit geringem Aufwand dahingehend erweitern, dass er Massendaten für Performancetests generieren kann. Die Idee ist es, in den Runstates der Testdaten nicht ausschließlich konkrete Werte, sondern auch Platzhalter zu verwenden. Statt *Name=Dr. House* könnte dort also *Name=%{name}* stehen. Es bietet sich an, die Platzhalter in Form von OGNL-Statements zu formulieren. Damit können auch Methodenaufrufe beim Export der Testdaten durchgeführt werden (z. B. *Name=%{context('name').replace('ä', 'ae')}*, um ein „ä“ durch ein „ae“ zu ersetzen). In einem Testdatennetz sind dann mehrere Objektinstanzen mit entsprechenden Platzhalten versehen (**Abb. 8**).

Die konkreten Werte für die Platzhalter werden in einer Excel-Tabelle hinterlegt. Während des Exportvorgangs wird über die Zeilen dieser Tabelle iteriert. Jede Spalte der Tabelle hat eine eindeutige Bezeichnung. Wird in einem Platzhalter diese Bezeichnung referenziert, wird der Platzhalter durch den Wert in der Zelle ersetzt. Pro Zeile in der Excel-Tabelle wird genau ein Testdatennetz erstellt (Tabelle 1).

Fazit und Ausblick

Die Erstellung von Testdaten mittels Objektinstanzdiagrammen hat sich in der Praxis bewährt. Mit der vorgestellten Lösung wurden im Projektbetrieb über die Dauer von ca. zwei Jahren Testdaten erstellt, die sonst in der Komplexität nur schwer bis gar nicht direkt (z. B. als Insert-Statements) abbildbar gewesen wären. Damit der Aufwand bei der Testdatenerstellung nicht zu groß wird, muss allerdings für den Enterprise Architect ein Plug-in entwickelt werden.

Ein großer Gewinn bei der vorgestellten Lösung liegt darin, dass die Fachseite sowohl eigene Testdaten erstellen als auch die Korrektheit der von der IT erstellten Testdaten bestätigen kann, da die Darstellung sehr genau und anschaulich ist. Dadurch können Missverständnisse früh vermieden werden, was die Kosten für spätere Fehlerkorrekturen deutlich senkt. Die Möglichkeit, Teile der Testdaten als Templates zu kopieren, sorgt dafür, dass in kürzester Zeit auch größere Testdatennetze erstellt werden können.

Dominik Kleine ist Diplominformatiker und als IT Consultant bei Atos IT Solutions and Services GmbH in Paderborn tätig. Er entwirft und entwickelt schwerpunktmäßig Unternehmensanwendungen für die Java-Plattform. Sein besonderes Interesse gilt effizienter Softwareentwicklung und Implementierung von Big-Data-Lösungen mittels Hadoop.

Links & Literatur

[1] http://de.wikipedia.org/wiki/Modultest

[2] http://www.sparxsystems.de/

[3] http://subs.emis.de/LNI/Proceedings/Proceedings133/gi-proc-133-026.pdf

[4] http://www.nist.gov/smartgrid/upload/InterimSmartGridRoadmapNISTRestructure.pdf

Kolumne: EnterpriseTales

von Lars Röwekamp und Arne Limburg

Kleine Perlen in Java EE 7

Seit dem 29.04. ist Java EE 7 offiziell released. Neben den bekannten, zum Großteil auch hier in der Kolumne besprochenen neuen Spezifikationen wie Batch Processing, WebSockets und JSON und den neuen Versionen der bekannten Spezifikationen JPA, JAX-RS, Servlet, EL, JMS, JSF, EJB, CDI und Bean Validation gibt es auch ein paar kleine, zum Teil sehr interessante Änderungen an bestehenden APIs und eine neue Spezifikation, von der man bisher vielleicht nicht gehört hat. Einen Auszug dieser Änderungen, die in Form von Maintenance-Releases daherkommen und auch besagte neue Spezifikation, bei der es sich um „Concurrency Utilities for Java EE" handelt, möchten wir in dieser Kolumne vorstellen.

Es ist eine bekannte Einschränkung von Java EE, dass im Code manuell keine Threads erzeugt werden dürfen. Dies hängt technisch vor allem damit zusammen, dass nahezu alle Java-EE-Services, wie Transaktionen, Security, CDI Scopes usw. an den aktuellen Thread gebunden sind. All diese Features stehen demzufolge in selbsterzeugten Threads nicht zur Verfügung. Vor Java EE 6 gab es daher praktisch keine Möglichkeit, im Java-EE-Standard selbst asynchrone Tasks auszuführen. Kam man in die Notwendigkeit, dies zu tun, musste man praktisch immer auf proprietäre Lösungen wie z. B. den IBM Workmanager [1] zurückgreifen. Seit Java EE 6 und EJB 3.1 gibt es immerhin die Möglichkeit, mit *@Asynchronous* Methoden asynchron laufen zu lassen [2]. Kontrolle über das Threading bietet diese Möglichkeit allerdings auch nicht. Des Weiteren ist z. B. völlig unspezifiziert, welche CDI-Kontexte in einem asynchronen Methodenaufruf verfügbar sind. Kann man dort z. B. auf eine *@SessionScoped* CDI Bean zugreifen?

Asynchronität in Java EE 7

Von der Masse unbemerkt und etwas im Hype um die anderen neuen Features (oder auch fehlenden Features, wie im Falle der Mandantenfähigkeit und JCache) untergegangen, hat in Java EE 7 ein neuer Standard Einzug erhalten, der genau diese Lücke schließen will: „Concurrency Utilities for Java EE" (siehe JSR 236 [3]). Im Wesentlichen handelt es sich dabei um eine Erweiterung des aus dem Package java.util.concurrent bekannten Java-SE-Features *ExecutorService* [4].

Auch bei der in Java EE 7 zur Verfügung gestellten Klasse *ManagedExecutorService*, die sich via Java EE

Porträt

w-jax Speaker

Lars Röwekamp ist Geschäftsführer der open knowledge GmbH und berät seit mehr als zehn Jahren Kunden in internationalen Projekten rund um das Thema Enterprise Computing.

@mobileLarson

Arne Limburg ist Softwarearchitekt bei der open knowledge GmbH in Oldenburg. Er verfügt über langjährige Erfahrung als Entwickler, Architekt und Consultant im Java-Umfeld und ist auch seit der ersten Stunde im Android-Umfeld aktiv.

@ArneLimburg

Resource Injection oder JNDI-Lookup holen lässt, lassen sich *Runnables* oder *Callables* als Tasks einstellen, die dann im Hintergrund ablaufen.

Der größte Unterschied des *ManagedExecutorServices* zu seinem kleineren Bruder aus dem SE-Umfeld ist die Tatsache, dass alle durch ihn ausgeführten Tasks *container-managed* sind, d. h. alle oben erwähnten Einschränkungen gelten hier nicht. So sind z. B. Transaktionen verfügbar.

Auch die Ausführung von CDI Beans als asynchrone Tasks ist hier klar spezifiziert: Erlaubt ist nur die Ausführung von *@ApplicationScoped* oder *@Dependent* CDI Beans. *@SessionScoped* oder *@RequestScoped* Beans stehen in den Threads nicht zur Verfügung. Jedenfalls weiß man hier, woran man ist.

Ein weiteres Feature hält der *ManagedExecutorService* noch bereit: *ManagedTaskListener*. Diese werden über Lifecycle-Änderungen eines Tasks informiert, der über einen ManagedExcecutorService ausgeführt wird. Lifecycle-Änderungen sind dabei die Übermittlung des Tasks an den TaskExecutorService (*taskSubmitted*), der Beginn der Abarbeitung des Tasks (*taskStarting*), das unerwartete Beenden eines Tasks (*taskAborted*) und die fehlerfreie Beendigung eines Tasks (*taskDone*).

Um einen *ManagedTaskListener* zu registrieren, muss man mit seinem Task entweder das *ManagedTask*-Interface implementieren oder man verwendet eine der zur Verfügung gestellten Factory-Methoden in *ManagedExecutors*, die aus einem *Runnable* oder *Callable* und einem *ManagedTaskListener* einen *ManagedTask* bauen, der dann an den *ManagedExecutorService* übermittelt werden kann.

Zusammenfassend lässt sich sagen, dass der JSR 236 „Concurrency Utilities for Java EE" eine kleine aber feine Ergänzung des Java-EE-Portfolios darstellt.

Weitere Änderungen durch Maintenance-Releases

Neben den „Concurrency Utilities for Java EE" gibt es ein paar weitere kleine Änderungen, die Java EE 7 mit sich bringen wird. Diese kommen nicht in Form neuer Versionen bekannter EE-Technologien (und damit neuer JSRs), sondern in Form von Maintenance-Releases bestehender JSRs. Dies hat den Vorteil (oder Nachteil), dass sie nicht zwingend an das Releasedatum von Java EE 7 gebunden sind und daher auch später fertiggestellt werden können. Das ist auch bei den drei Features der Fall, die wir nun vorstellen werden:

1. Das nächste Maintenance-Release der JTA-Spec [5] wird aller Voraussicht nach die Annotation *@Transactional* enthalten, mit der es möglich sein wird, auch CDI Beans mit Container-managed Transaktionen zu versehen. Bisher ist dieses Feature EJBs vorenthalten.
2. Im nächsten Maintenance-Release der „Commons Annotations" Spec [6] wird es zusätzlich die Annotation *@Priority* geben, die dann allen anderen Spezifikationen zur Verfügung steht, um in irgendeiner Form die Reihenfolge von Klassen festzulegen. In der Interceptors Spec kann damit z. B. die Aufrufreihenfolge der Interceptoren definiert werden.
3. Auch von der Interceptors Spec wird es ein Maintenance-Release geben [7]. Neben besagten Änderungen zur Priorisierung von Interceptoren wird dieses auch die Annotation *@AroundConstruct* enthalten, mit der es dann möglich ist, Interceptoren für Konstruktoraufrufe umzusetzen.

Fazit

Auch abseits der bekannten Neuerungen bietet die neue Java EE 7 Spec ein paar bemerkenswerte Neuerungen. Vor allem mit den „Concurrency Utilities for Java EE" gibt es eine kleine, aber sinnvolle Ergänzung des Java-EE-Portfolios, die sicherlich in Zukunft in einigen Projekten zum Einsatz kommen wird.

Mit *@Transactional* ist es in Zukunft noch besser möglich, bei der Wahl der Komponententechnologie (EJB vs. CDI) nicht nur zwischen Alles (EJB mit Security, Transaktionen, Instance Pooling und Remoting) oder Nichts (CDI) entscheiden zu müssen. Der Standardanwendungsfall, in dem eine einfache Java-Klasse benötigt wird, um die herum eine Transaktionsgrenze aufgespannt wird, kann nun komplett ohne EJB im Java-EE-Standard abgebildet werden.

Aber auch die anderen vorgestellten Neuerungen, wie *@Priority* und *@AroundConstruct* werden sich in der Praxis sicherlich als nützlich erweisen.

Links & Literatur

[1] http://ibm.co/1b7qK4h

[2] http://docs.oracle.com/javaee/6/api/index.html?javax/ejb/Asynchronous.html

[3] http://www.jcp.org/en/jsr/detail?id=236

[4] http://bit.ly/17qRQpX

[5] http://jcp.org/aboutJava/communityprocess/maintenance/jsr907/index5.html

[6] http://jcp.org/aboutJava/communityprocess/maintenance/jsr250/index2.html

[7] http://jcp.org/aboutJava/communityprocess/maintenance/jsr318/index2.html

Ist Solr 4.3 reif für Cloud und agile Entwicklung?

Solr-Power

Eine gut funktionierende Suche ist unverzichtbar für jeden erfolgreichen Onlineshop, sei es Amazon, Etsy oder Zalando. Um eine skalierbare Suche mit vertretbarem Aufwand zu bauen, gab es lange Zeit nur eine Antwort: die Verwendung von Apache Solr als Suchserver im Hintergrund. Solr ist schon lange mehr als nur ein einfacher HTTP-Wrapper um Apache-Lucene-basierte Indizes. Es kann als eine NoSQL-Datenbank verstanden werden, die für die Suche in großen Datenmengen optimiert ist. Solr wird gerne als sekundärer Index für große, relationale Datenbanken eingesetzt, auf denen sich in der Regel schlecht über beliebige Attributkombinationen suchen lässt. Seit 2010 existiert mit ElasticSearch eine würdige Alternative, und glaubt man Vergleichsseiten [1], Blogs [2], [3], [4] und Konferenzbeiträgen, so ist Solr pauschal unterlegen, insbesondere, wenn es um den Einsatz in der Cloud und agiles Entwickeln geht. Konkurrenz belebt aber häufig das Geschäft, und mit Solr 4.3 meldet sich der Platzhirsch eindrucksvoll zurück.

von Martin Breest und Jens Hadlich

Ist Solr reif für die Cloud und agile Entwicklung? Diese Frage mussten wir uns in den letzten Wochen beruflich stellen, da die Suchfunktionalität unserer Plattform, an der verschiedenste Onlineshops und Anwendungen hängen, deutlich erweitert werden soll. Wir haben momentan noch eine ältere Solr-Version (3.1) im Betrieb, die in den letzten Jahren gute Dienste geleistet hat und in einem Master-/Slave-Set-up mit der ständig steigenden Anzahl von Suchanfragen und Vergrößerung des Suchindexes mitwachsen konnte.

Bisher war allerdings nur eine Teilmenge der Entitäten durchsuchbar. In Zukunft werden es aber im Prinzip alle und damit weit über 100 Millionen Items sein. In der Vergangenheit war uns allerdings die Handhabung der Schemata viel zu statisch. Änderungen daran waren in der Praxis eher zeitaufwändig und wurden gerne vermieden. In unserem agilen Umfeld ist dies sehr hinderlich. Im Hinblick auf die bevorstehende Erweiterung ist Einfachheit nunmehr ein K.-o.-Kriterium. Wir wären gerne in der Lage, tägliche Änderungen an unserer Suche ohne Downtimes durchzuführen. Weiterhin sollen von Benutzern neu angelegte oder geänderte Entitäten in Echtzeit verfügbar sein und nicht erst nach 15 Minuten und einer Synchronisierung über unser Master-/Slave-Set-up. Bei Spitzenlast, z. B. im Weihnachtsgeschäft oder bei Marketingaktionen, wollen wir Suchindizes einfach temporär horizontal skalieren können, indem wir mehr Ressourcen zur Verfügung stellen.

Ob Solr in diesem Sinne reif für die Cloud und agile Entwicklung ist, soll anhand folgender Punkte und damit verbundenen Fragen betrachtet werden:

- **Skalierbarkeit**: Wie einfach kann man Suchindizes skalieren, sowohl, was die Indexgröße in GB betrifft als auch die Anzahl der Suchanfragen pro Sekunde? Wie einfach kann man neue Master-/Slave-Set-ups bzw. Shards und Shard-Replikationen aufsetzen? Kann man die Verteilung von Daten auf Shards beeinflussen?
- **Echtzeitfähigkeit**: Wie lange dauert es, bis Änderungen am Suchindex in Suchanfragen sichtbar werden? Ist der Suchindex mit (fast) aktuellen Daten als sekundärer Index zu einer relationalen Datenbank einsetzbar?
- **Administrierbarkeit**: Mit welchem Aufwand ist es verbunden, neue Suchindizes aufzusetzen, neue Replikationen oder Shards anzulegen oder Suchindexkonfigurationen zu verändern? Können Änderungen programmatisch vorgenommen und z. B. für Releases automatisiert werden? Mit welchen Downtimes ist das verbunden?
- **Anpassbarkeit:** Kann ich das Schema des Suchindexes ändern? Kann ich Einfluss auf das Suchranking nehmen, um z. B. Faktoren wie Verkaufszahlen, Klickzahlen oder Alter eines Produkts einzubeziehen? Kann ich Solr auf Codeebene anpassen? Wie einfach lassen sich Suchindexkonfigurationen anpassen, wie zum Beispiel Synonyme, Stoppwörter oder geschützte Begriffe?
- **Entwicklungsgeschwindigkeit**: Wie einfach lassen sich Schemaänderungen durchführen? Wie einfach kann ich die Interna von Solr anpassen? Wie schnell sind Änderungen ausrollbar? Kann ich sie implizit über den neuen Programmcode ausrollen oder müssen Administratoren beim Release in Anspruch genommen werden? Muss ich bei jeder Schemaänderung komplett neu indizieren oder können neue Schemaattribute mit einem „Dark Launch“ online gehen? Muss ich eine zentrale Stelle in meine Plattform einbauen, an der indiziert wird oder kann ich die Indizierung über verschiedenste Module hinweg verteilen?

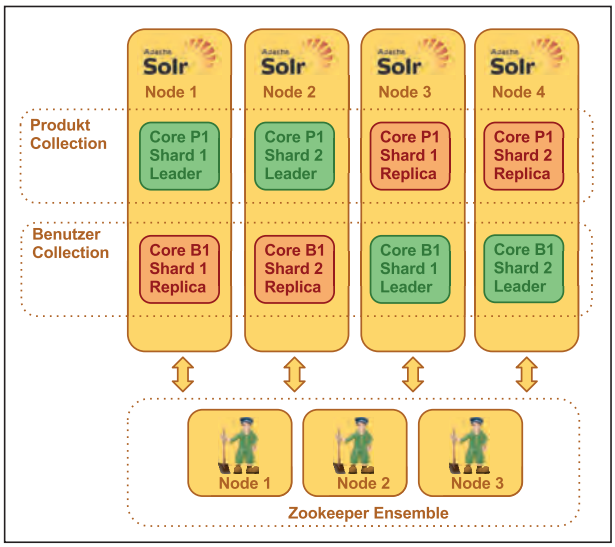


Abb. 1: SolrCloud-Schema mit mehreren Nodes und Collections im ZooKeeper Ensemble

Skalierbarkeit

Solr skaliert bis zu einer bestimmten Indexgröße und ohne Echtzeitanforderung über das schon längere Zeit existierende Master-/Slave-Set-up und mit Replikationsintervallen von z. B. 15 Minuten sehr gut. In diesem Set-up wird üblicherweise auf einen Master geschrieben und von vielen Slaves gelesen. Theoretisch beeinflussen sich die Slaves untereinander nicht, da sie keine geteilte Infrastruktur haben. Ab einer gewissen Indexgröße oder Slave-Anzahl wird allerdings das Datenvolumen für die Replikation zwischen Master und Slaves ein Problem. Diesbezüglich existieren diverse Workarounds, wie z. B. die Verwendung von BitTorrent für die Replikation [5]. Werden die Indizes jedoch zu groß oder soll das indizierte Dokument sofort sichtbar werden, genügt auch ein solches Set-up nicht mehr.

Mit SolrCloud wurde in Solr neben dem Konzept der Cores das Konzept der verteilten Collections eingeführt. Während ein Core genau einen physischen Suchindex verwalten kann, ermöglicht es eine Collection, einen Suchindex in mehrere logische Teile zu zerlegen, auf die die Daten des Suchindexes bzw. die Anfragen dann verteilt werden können. Diese logischen Teile werden Shards genannt. Die tatsächlichen physischen Daten eines Shards werden dann wieder von einem Core verwaltet. Für ein Shard können beliebig viele Kopien angelegt werden, um Single Points of Failure (SPoF) zu vermeiden und im Livebetrieb auch die Last auf einem Shard besser verteilen zu können. Für jeden Shard gibt es immer einen Leader, der diesen Shard verwaltet, und es kann mehrere Replicas geben. Die Cores laufen auf so genannten Nodes, wobei ein Node mehrere Cores verwalten kann. Die Nodes bilden dann den Solr Cluster. Der Cluster wird üblicherweise von einem ZooKeeper Ensemble verwaltet (**Abb. 1**).

Ein aus vier Nodes bestehender Solr Cluster (Port 9000/9001/9002/9003) mit einer ZooKeeper-Instanz (Port 10000) lässt sich mithilfe weniger Skripte und des *start.jar* aus der Solr-Distribution sehr einfach aufsetzen:

```
java -Djetty.port=9000 -Dbootstrap_confdir=./solr/default/conf
      -Dcollection.configName=myconf -DzkRun=localhost:10000 -jar start.jar
java -Djetty.port=9001 -DzkHost=localhost:10000 -jar start.jar
java -Djetty.port=9002 -DzkHost=localhost:10000 -jar start.jar
java -Djetty.port=9003 -DzkHost=localhost:10000 -jar start.jar
```

Neue Collections und Cores anzulegen ist genauso einfach. Dazu kann man die über HTTP bereitgestellte Adminschnittstelle und einen Browser verwenden. Eine neue *products* Collection würde man über folgenden URL anlegen:

```
http://localhost:9000/solr/admin/
 collections?action=CREATE&name=products&numShards=2&replicationFactor=1
```

Will man *shard1* noch eine weitere Replikation gönnen, ruft man den folgenden URL auf:

```
http://localhost:9000/solr/admin/cores?action=CREATE&name=
                        products_replica1&collection=products&shard=shard1
```

Solr wird skaliert, indem diese Prozedur wiederholt wird, bis der Bedarf an Shards und Replicas gedeckt ist. Der Bedarf an physischen Ressourcen kann dabei auch prima mit Maschinen von Cloud-Anbietern wie Amazon gedeckt werden – die SolrCloud in der Cloud. Apropos skalieren: In Solr lässt sich die Verteilung der Daten und Suchanfragen mittels implizitem und explizitem Routing sehr gut steuern.

Implizites Routing ist das Standardverhalten einer Collection und bedeutet, dass Daten bzw. Suchanfragen automatisch auf die Shards verteilt werden. Dazu werden beim Indizieren so genannte *CompositeIds* verwendet, die sich aus einem Präfix und der eigentlichen ID zusammensetzen. Dokumente mit gleichem Präfix werden im selben Shard abgelegt:

```
<add>
  <doc>
    <field name="id">user1!1</field>
    ...
  </doc>
</add>
```

Gibt man nun bei Suchanfragen den Query-Parameter *shard.keys* an, muss Solr nicht alle, sondern nur den einen Shard abfragen:

```
http://localhost:9000/solr/products/select?q=&wt=xml&indent=
                                                true&shard.keys=user1!
```

Beim expliziten Routing konfiguriert man in der *schema.xml* per *_shard_*-Feld konkret den Shard:

```
<field name="_shard_" type="string" indexed="true" stored="true"/>
```

Zu beachten ist, dass nun Cluster und Collection ohne die *numShards*-Option gestartet bzw. angelegt werden

müssen. Beim Hinzufügen wird der Name des Shards dann ebenfalls angegeben:

```
<add>
  <doc>
    <field name="id">1</field>
    ...
    <field name="_shard_">shard1</field>
  </doc>
</add>
```

Analog dazu wird bei Anfragen ein Shard per Query-Parameter *shards* angegeben:

```
http://localhost:9000/solr/products/select?q=*%3A*&wt=xml&indent=
                                                    true&shards=shard1
```

Explizites Routing bietet somit maximale Kontrolle darüber, auf welche Shards Daten indiziert und welche für Suchanfragen verwendet werden sollen. Die SolrCloud nutzt für Replikation und zum Abspeichern der Daten ein *optimistic locking*. Damit das korrekt funktioniert, muss in der *schema.xml* ein *_version_*-Feld vorhanden sein:

```
<field name="_version_" type="long" indexed="true" stored="true"/>
```

Zudem muss das Transaction Logging in der *solrconfig.xml* konfiguriert werden:

```
<updateHandler class="solr.DirectUpdateHandler2">
  <updateLog>
    <str name="dir">${solr.data.dir:}</str>
  </updateLog>
  ...
</updateHandler>
```

Damit Replikation und Echtzeitfähigkeit richtig funktionieren, müssen zudem noch ein paar Request Handler in der *solrconfig.xml* definiert werden:

```
<requestHandler name="/replication" class="solr.ReplicationHandler"
                                                    startup="lazy" />

<requestHandler name="/get" class="solr.RealTimeGetHandler">
  <lst name="defaults">
    <str name="omitHeader">true</str>
  </lst>
</requestHandler>

<requestHandler name="/admin/" class="solr.admin.AdminHandlers" />
```

Es gibt da natürlich auch noch ein paar offene Punkte. Nutzt man implizites Routing, kann man nicht einfach die Anzahl der Shards einer Collection verändern, z. B. von 2 auf 3. Dieses Feature ist wohl aber schon auf der Solr-Roadmap. Des Weiteren funktionieren nicht alle Suchkomponenten im verteilten Modus.

Echtzeitfähigkeit

In der aktuellen Version ist Solr quasi echtzeitfähig, indem es das NRT-Feature (Near Real Time) des darunter liegenden Apache-Lucene-Frameworks verwendet. Neben den bekannten Hard Commits und dem dazugehörigen Overhead für das Abspeichern der Daten auf Platte und das Anlegen eines neuen Searchers gibt es jetzt auch so genannte *Soft Commits*. Diese erzeugen deutlich weniger Overhead, werden jedoch bis zum nächsten Hard Commit nicht auf Platte geschrieben. Bei einem Crash der JVM gehen sie verloren (keine On-Disk-Garantie).

Gut ist auch, dass Solr jetzt zeitgesteuert serverseitig Hard und Soft Commits ausführen kann. Das erspart einem das regelmäßige Verschicken dieser Commits aus der Applikation heraus. Die Dokumente, die über ein Soft Commit committet wurden, sind dann je nach Einstellung kurz danach in Suchergebnissen sichtbar. Dazu muss man in der Konfiguration *solrconfig.xml* des Cores die *autoSoftCommit*-Option anstellen:

```
<updateHandler class="solr.DirectUpdateHandler2">
  ...
  <autoCommit>
    <maxTime>10000</maxTime>
    <openSearcher>false</openSearcher>
  </autoCommit>
  <-- soft commits for near real time -->
  <autoSoftCommit>
    <maxTime>1000</maxTime>
  </autoSoftCommit>
</updateHandler>
```

Wichtig ist, dass die *schema.xml* ein *_version_*-Feld enthält, und dass in der *solrconfig.xml* Request Handler und Transaction Logging, wie oben bereits beschrieben, konfiguriert wurde.

Administrierbarkeit

Bei der Administrierbarkeit von Solr hat sich vieles verbessert. Es gibt einfach benutzbare Admin-APIs, über die Collections und Cores verwaltet werden können [6], [7]. Ein Core wird folgendermaßen anlegt bzw. gelöscht:

```
http://localhost:9000/solr/admin/cores?action=CREATE&name=
                        products_replica1&collection=products&shard=shard1
http://localhost:9000/solr/admin/cores?action=UNLOAD&core=
                                                    products_replica1
```

Es können auch Cores nach Konfigurationsänderungen neu geladen, ausgetauscht oder mit Daten eines anderen Cores initialisiert werden. Zudem kann man den Zustand der verschiedenen Cores abfragen, was für Monitoring-Zwecke sehr nützlich ist:

```
http://localhost:9000/solr/admin/cores?wt=json
http://localhost:9000/solr/products/admin/system?wt=json
```

Zudem gibt es auch neue Admin-APIs, die Zugriff auf den von ZooKeeper verwalteten Cluster-Zustand erlauben:

```
http://localhost:9000/solr/zookeeper?wt=json
```

Braucht man weitere Information über den Cluster-Zustand, kann man auch die ZooKeeper-APIs verwenden. Setzt man das API richtig ein, können so Downtimes bei Releases vermieden werden, selbst wenn der Suchindex nach Schemaänderungen oder Änderungen an der Similarity-Funktion komplett neu aufgebaut werden müsste.

Core-Konfigurationsdateien (*schema.xml*, *solrconfig.xml*, Bibliotheken mit eigenen Anpassungen) werden bei der SolrCloud in einem zentralen Repository in ZooKeeper abgelegt. In diesem Repository können unterschiedliche Core-Konfigurationen hinterlegt werden, wobei es eine Standardkonfiguration gibt. Neue Collections und Cores können dann unter Verwendung dieser Konfiguration angelegt und gestartet werden. Die verwalteten Konfigurationsdateien lassen sich auch einfach mittels ZooKeeper-APIs und des in der Solr-Distribution vorhandenen ZooKeeper-Clients aktualisieren. Damit allerdings eine neue Konfiguration von einem bereits laufenden Core übernommen wird, muss dieser bzw. die Collection durchgestartet werden:

```
java -classpath lib\* org.apache.solr.cloud.ZkCLI -cmd upconfig -zkhost
localhost:10000  -confdir instance1/solr/default/conf -confname
                                        myconf -solrhome instance1/solr

http://localhost:9000/solr/admin/collections?action=RELOAD&name=
                                                              products
```

Das funktioniert z. B. für Schemaänderungen sehr gut. Trotzdem kann es zu kurzen Downtimes kommen. Neue Nodes lassen sich, wie bereits weiter oben beschrieben, sehr einfach zu einem Solr-Cluster hinzufügen, wie hier zusätzlich auf Port 9004:

```
java -Djetty.port=9004 -DzkHost=localhost:10000 -jar start.jar
```

Über das Admin-API können dann neue Cores, neue Shards und Replicas für existierende Shards angelegt werden. Dadurch ist man sehr flexibel, sowohl, was das Allozieren zusätzlicher Ressourcen, als auch das Verlagern von Suchanfrage- und Indizierungs-Traffic auf diese neuen Ressourcen durch das Anlegen bzw. Löschen von Shard Leadern und Shard Replicas anbelangt. Dies kann weitestgehend ohne Downtimes durchgeführt werden:

```
http://localhost:9004/solr/admin/cores?action=CREATE&name=
                    products_replica1&collection=products&shard=shard1
http://localhost:9000/solr/admin/cores?action=UNLOAD&name=
                                                    products_replica1
```

Was die Administrierbarkeit anbelangt, hat sich in der aktuellen Solr-Version und mit den Solr-Cloud-Features einiges verbessert. Die meisten Änderungen können jetzt programmatisch und automatisiert über einen Installer beim Release oder über die Applikation selbst bei Bedarf vorgenommen werden. Wenn beispielsweise die Zahl der Anfragen einen Schwellwert übersteigt, können einfach weitere Nodes im Amazon-Rechenzentrum hochgefahren und Shard-Replikationen auf diese Knoten verlegt werden. Es ist trotzdem nach wie vor etwas umständlich, Konfigurationsänderungen ohne Downtimes auszurollen. Gerade wenn man z. B. öfter die Dateien für Synonyme, geschützte Wörter oder Stoppwörter verändern muss, würde man sich feingranularere, REST-basierte APIs nicht nur an dieser Stelle wünschen.

Anpassbarkeit

Solr ist schon immer recht gut an die eigenen Bedürfnisse anpassbar. Durch die Änderungen in Apache Lucene 4.0 hat sich hier noch einmal einiges verbessert [8].

Ein Schema wird in Solr grundsätzlich mittels *schema.xml* beschrieben. Bei Änderungen verändert man diese Datei immer offline. Erst danach wird live ausgerollt. Dies war in älteren Solr-Versionen im einfachsten Fall mit kurzen Downtimes beim Durchstarten des Cores verbunden und im schlimmsten Fall mit dem Durchstarten des Nodes oder dem kompletten Neuindizieren aller Dokumente im veränderten Core. Um Downtimes zu vermeiden, gab und gibt es diverse Workarounds unter Verwendung der Merge-Index-Tools [9].

In der aktuellen Version und mit dem bereitgestellten Admin-APIs für Solr und ZooKeeper lässt sich das alles besser automatisieren. Über die Klasse *org.apache.solr.cloud.ZkCLI* lassen sich, wie weiter oben bereits beschrieben, sehr einfach Schemaänderungen in das zentrale ZooKeeper Repository hochladen. Damit die Schemaänderungen live gehen, muss man noch die betreffende Collection bzw. die einzelnen Cores über das bereits beschriebene *RELOAD*-Kommando durchstarten. Dies kann zu kurzen Downtimes führen. Ob der Index danach anstandslos funktioniert, hängt natürlich von den vorgenommenen Änderungen ab. Solange man aber z. B. nur neue Felder zum Index hinzufügt, funktioniert das Ganze reibungslos.

Um Schemaänderungen ganz zu umgehen, kann man jetzt dynamische Felder (Dynamic Fields) verwenden. Dabei handelt es sich um abstrakte Felddefinitionen, denen ein Name vergeben wird, der einem Pattern entspricht. Ein paar beispielhafte Felddefinitionen könnten wie folgt aussehen:

```
<dynamicField name="*_t" type="text_en" indexed="true" stored="true"/>
<dynamicField name="*_tt" type="text_en" indexed="true" stored="true"
                                                        multiValued="true"/>
<dynamicField name="*_t_de" type="text_de" indexed="true"
                                                        stored="true"/>
<dynamicField name="*_tt_de" type="text_de" indexed="true" stored="true"
                                                        multiValued="true"/>
```

Hält man sich nun beim Indizieren neuer Dokumente an diese Patterns, kann man im Prinzip beliebige neue

Felder in den Index aufnehmen, ohne erst umständlich das Schema verändern zu müssen. Das setzt natürlich ein entsprechend weitsichtig geplantes Schema voraus:

```
<add>
 <doc>
  <field name="id">1</field>
  ...
  <field name="name_t_de">gelber Sportwagen</field>
  <field name="labels_tt_de">Sportwagen</field>
  <field name="labels_tt_de">Auto</field>
  <field name="labels_tt_de">Fahrzeug</field>
 </doc>
</add>
```

Einfache Ranking-Anpassungen lassen sich recht einfach und ohne Konfigurationsänderungen durchführen, etwa, indem man Document- und Field-Boost-Werte verwendet [10]:

```
<add>
 <doc boost="2.5">
  <field name="id">1</field>
  ...
  <field name="name_t_de" boost="2.0">gelber Sportwagen</field>
 </doc>
</add>
```

Will man das Ranking zur Suchanfragezeit manipulieren, kann man z. B. Boost-Annotationen im Dismax Handler bzw. Function Queries verwenden [11]:

```
http://localhost:9000/solr/products/select?defType=dismax&qf=
                                name_t_de&q=Sportwagen&bf=sqrt(popularity)
```

Wem das nicht reicht, der kann jetzt auch die für einen Suchindex verwendete Similarity-Funktion anpassen. Dazu muss eine Similarity-Klasse implementiert werden, die dann den Scoring-Algorithmus entsprechend beeinflusst [8]. Die Similarity-Klasse kann in der *schema.xml* sowohl für den kompletten Suchindex als auch für einzelne Felder angepasst werden:

```
<similarity class="similarity.CustomSimilarity"/>
```

Die Klasse selbst muss in einer JAR-Datei verpackt im Dateisystem abgelegt und in der *solrconfig.xml* eingetragen werden:

```
<lib dir="myextensions" regex=".*\.jar" />
```

Die Änderungen treten natürlich wieder erst in Kraft, nachdem die betreffenden Collections bzw. einzelnen Cores über die *RELOAD*-Aktion durchgestartet wurden. Das kann entsprechend kurze Downtimes nach sich ziehen, und das geänderte JAR muss davor auf alle Nodes verteilt werden. Bei Änderungen an der Similarity-Funktion ist grundsätzlich Vorsicht geboten, da eine volle Neuindizierung notwendig wird. Um das Ranking manuell zu beeinflussen, kann alternativ auf das Query-Elevation-Plug-in [12] zurückgriffen werden.

Anpassungen an weiteren Konfigurationsdateien wie z. B. *synonyms.txt*, *stopwords.txt*, *protwords.txt*, *elevate.xml* etc. werden analog zum Schema offline vorgenommen, mittels ZooKeeper-Client hochgeladen und per *RELOAD* in Kraft gesetzt.

Auch die Suchanfrage- und Datenindizierungsverarbeitung lässt sich nach eigenen Wünschen modifizieren. Dazu muss man nur die entsprechenden Klassen, basierend auf den Klassen des Solr- bzw. Lucene-API, implementieren und in der *schema.xml* bzw. *solrconfig.xml* eintragen. So lassen sich relativ schnell eigene Suchkomponenten für die optimierte Gruppierung von Dokumenten oder für eine besser von außen konfigurierbare Query-Elevation schreiben. Die eigenen Klassen müssen wieder in einer JAR-Datei verpackt im Dateisystem liegen und in der *solrconfig.xml* über die *lib*-Anweisung bekannt gegeben werden. Das Durchstarten der Cores ist natürlich auch wieder notwendig.

Entwicklungsgeschwindigkeit

Die einfache Administrierbarkeit wirkt sich natürlich auch positiv auf Entwickler aus. Schließlich wird während der Entwicklung der Einfachheit halber häufig nur eine Instanz verwendet, mit der Folge, dass im Cluster der Testumgebung oder, noch schlimmer, live „irgendetwas“ nicht richtig funktioniert. Mit Solr ist dies kein Problem; man ist somit viel näher am Zielsystem.

Solr lässt sich, wie bereits dargestellt, sehr gut an die eigenen Bedürfnisse anpassen. Wie schnell man neue Features live bekommt, hängt aber davon ab, auf welcher Ebene man Anpassungen vornehmen muss. Gut ist, dass der komplette Releaseprozess jetzt automatisierbar und im günstigsten Fall nur mit sehr kurzen Downtimes verbunden ist.

Benutzt man die vorgestellten dynamischen Felder, um weitestgehend ohne Schemaänderungen auszukommen und arbeitet mit Document Boost, Field Boost und Function Queries, um das Suchranking zu manipulieren, lassen sich Konfigurationsänderungen vermeiden.

Komplizierter wird es, wenn man eigene Anpassungen auf Codeebene vorgenommen hat, z. B. eine eigene Similarity-Funktion, und es regelmäßig Änderungen gibt. Dasselbe gilt, wenn man Änderungen an Konfigurationsdateien, wie z. B. den Dateien zur Konfiguration von Stoppwörtern, Synonymen oder geschützten Wörtern, regelmäßig ausrollen will. Dieser Prozess lässt sich unter Verwendung des ZooKeeper-Clients und des Admin-API ganz gut automatisieren – sowohl über einen Installationsprozess beim Release als auch aus einer Clientanwendung heraus. Die damit zum Teil verbundenen Downtimes sind aber sehr störend.

Warum man z. B. wegen einer Änderung eines Synonyms alle Cores der Collection durchstarten muss, ist eigentlich unverständlich. Dasselbe gilt für Änderungen an eigenen Bibliotheken. Hier würde man sich feingranulare-

re REST-APIs wünschen. Warum sich die laufenden Cores die Änderungen an der von ihnen verwendeten ZooKeeper-Konfiguration bei einer Änderung nicht automatisch ziehen können und sich entsprechend smart aktualisieren, ist auch ein Punkt, der noch verbessert werden könnte.

Eine große Verbesserung ist die Unterstützung partieller Updates. Dadurch lassen sich Daten bzw. einzelne Felder gezielt aktualisieren. Früher brauchte man eine zentrale Stelle, an der diese zu kompletten Dokumenten zusammengebaut wurden. Im Sinne der Separation of Concerns [13] können nun verschiedene Teilaspekte eines zu indizierenden Dokuments von verschiedenen Subsystemen verwaltet werden, z. B. Kerndaten, Labels, Preise oder zusätzliche Daten für die Berechnung des Rankings, die dann bei Änderungen nur für Updates ihrer Teildaten im Suchindex sorgen.

Damit partielle Updates funktionieren, muss im Solr-Schema ein *_version_*-Feld definiert sein. Zudem müssen alle Felder als *stored* konfiguriert sein. Das führt dazu, dass der Suchindex ggf. ein wenig größer wird:

```
<field name="id" type="string" indexed="true" stored="true" required="true"
                                                    multiValued="false" />
<field name="price" type="string" indexed="true" stored="true"
                                                    multiValued="false"/>
<field name="labels" type="string" indexed="true" stored="true"
                                                    multiValued="true"/>
<field name="_version_" type="long" indexed="true" stored="true"/>
```

Hat man diese Einstellungen vorgenommen, kann man unter Angabe einer Unique-ID und unter Verwendung des *update*-Attributs, das auf *add*, *set* oder *inc* stehen kann, entsprechend partielle Updates an Solr verschicken:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="price" update="set">0.00</field>
    <field name="labels" update="add">1</field>
  </doc>
</add>
```

Schade ist, dass man Werte auf *Multivalued*-Felder nur komplett setzen oder selektiv hinzufügen, aber nicht selektiv entfernen kann.

Fazit

Was sich im Bereich Skalierbarkeit mit den neuen Features der SolrCloud getan hat, wirkt sehr leichtgängig und überzeugend. Zusätzlich zum bekannten Master-/Slave-Set-up hat man damit ein weiteres Werkzeug, um die Suche zu skalieren. Durch die gut funktionierende Echtzeitfähigkeit kann die neue Solr-Version punkten.

Auch bezüglich der Administrierbarkeit gibt es eigentlich wenige Lücken. Fast alles ist programmatisch machbar und lässt sich einfach automatisieren. Störend ist, dass man auch für kleine Änderungen an Dateien für Synonyme, geschützte Wörter oder Stoppwörter umständlich die Konfiguration hochladen und anschließend die Collection durchstarten muss. Hier wären z. B. weitere feingranularere REST-APIs wünschenswert.

Die Anpassbarkeit von Solr auch im Code hat sich weiter verbessert. Man kann jetzt eigene Similarity-Funktionen schreiben und unter Verwendung dynamischer Felder Schemaänderungen umgehen. Die Änderungen live zu bekommen, wirkt noch schwerfällig. Hier sollte in den nächsten Versionen nachgebessert werden.

Mit der aktuellen Version von Solr lässt sich einfach und zügig entwickeln. Viele Änderungen können direkt live gehen. Wenn man aber z. B. regelmäßig die Liste der Synonyme verändert oder Änderungen an eigenen Bibliotheken vornimmt, kommt man ums Ausrollen der Konfiguration nicht herum, der Prozess wird schwerfällig. Die damit teilweise verbundenen Downtimes bzw. die Workarounds, die notwendig sind, um diese zu umgehen, wirken sich auf die Entwicklungsgeschwindigkeit negativ aus.

Als Fazit lässt sich also sagen, dass Solr reif für die Cloud und bedingt für agile Entwicklung ist. Wer sich an der Schwerfälligkeit gerade im Konfigurationsbereich stört, sollte einen Blick auf Alternativen wie ElasticSearch [14] oder RiakSearch [15] werfen.

Martin Breest ist Senior Software Architect bei Spreadshirt in Leipzig. Er beschäftigt sich seit mehr als zehn Jahren mit der Entwicklung von skalierbaren Systemen auf der Java-Plattform.

Jens Hadlich ist Software Architect und Team Lead bei Spreadshirt in Leipzig. Er beschäftigt sich seit über zehn Jahren mit dem Entwurf und der Entwicklung skalierbarer Systeme, überwiegend auf der Java-Plattform.

Links & Literatur

[1] http://solr-vs-elasticsearch.com/

[2] http://backstage.soundcloud.com/tag/elastic-search/

[3] http://bit.ly/PrBFu3

[4] http://java.dzone.com/articles/klout-search-powered

[5] http://etsy.me/wMyPjl

[6] http://wiki.apache.org/solr/SolrCloud

[7] http://wiki.apache.org/solr/CoreAdmin

[8] Schindler, Uwe: „Apache Lucene 4.0 – Neue Features im Überblick“, in Java Magazin 12.2012

[9] http://wiki.apache.org/solr/MergingSolrIndexes

[10] http://wiki.apache.org/solr/UpdateXmlMessages

[11] http://wiki.apache.org/solr/SolrRelevancyFAQ

[12] http://wiki.apache.org/solr/QueryElevationComponent

[13] http://en.wikipedia.org/wiki/Separation_of_concerns

[14] http://www.elasticsearch.org/

[15] http://docs.basho.com/riak/latest/tutorials/querying/Riak-Search/

[16] Meder, Christian: „Solr unter Strom – Suchmagie für Applikationsentwickler“, in Java Magazin 12.2011

[17] Kenworthy, Andrew; Meder, Christian: „Big Data und Suche – Auf der Suche nach dem heiligen Gral“, in Java Magazin 7.2012

SynchronizeFX: Remote JavaFX Property Binding

In sync

Mit JavaFX 2 steht Java-Entwicklern eine moderne Technologie zur Entwicklung von reichhaltigen und interaktiven Benutzeroberflächen zur Verfügung: Sie bringt einerseits viele interessante Konzepte wie deklarative Gestaltung, Animationen und Gestenerkennung mit, führt andererseits aber auch Properties und Bindings auf Datenebene ein. Das hier vorgestellte Framework SynchronizeFX greift die letzten beiden Konzepte auf. Es ermöglicht die Implementierung von verteilten JavaFX-Anwendungen, die sich ein Datenmodell teilen und dieses visualisieren. Der Mechanismus eignet sich nicht nur zur Umsetzung von Echtzeit-Multiplayer-Spielen, sondern lässt sich auch im Umfeld von Geschäftsanwendungen einsetzen.

von Raik Bieniek, Manuel Mauky, Alexander Casall, Vincent Tietz und Michael Thiele

Wir wollen zunächst den fachlichen Anwendungsfall aus dem Tagesgeschäft der Autoren betrachten, um erste Einsatzszenarien der Technologie zu umreißen. Aufgrund der wachsenden Bedeutung agiler Vorgehensmodelle in der Softwareentwicklung arbeitet die Saxonia Systems AG seit 2011 an einer Integrationsplattform zur Unterstützung dieser Arbeitsprozesse, insbesondere in verteilten Entwicklungsszenarien. Das Gesamtkonzept basiert, neben audiovisuellen Konferenzsystemen, auf einem Toolstack zur Förderung der Prozesstransparenz und Kommunikation der Teammitglieder untereinander. Dieser Werkzeugkasten enthält unter anderem ein digitales Scrum Board. Ähnlich dem analogen Vorbild werden Aufgaben auf einfache Art und Weise visualisiert, wodurch der Projektfortschritt auf einen Blick erkennbar ist. Über die Zustandsänderung von Aufgaben hinaus bieten sich Möglichkeiten der Skalierung, Filterung und der Darstellung von Detailinformationen, die in analogen Boards fehlen.

Um die haptischen und visuellen Eigenschaften klassischer Boards zu bewahren, wurde das digitale Scrum Board mit JavaFX umgesetzt, da es die Implementierung einer reichhaltigen Benutzeroberfläche (GUI) mit Multi-Touch-Interaktion ermöglichte. Dabei überzeugten nicht nur das Styling mit CSS und die integrierte Gestenerkennung, sondern auch die Möglichkeit der komfortablen Implementierung von Animationen und die Nutzbarkeit einer umfangreichen Komponentenbibliothek. Allerdings sind in den verteilten Entwicklungsszenarien mehrere Scrum Boards vorgesehen, deren Zustände untereinander synchronisiert werden müssen (**Abb. 1**). JavaFX liefert mit Properties und Bindings zwar nützliche Konzepte, um Abhängigkeiten zwischen Werten abzubilden und umzusetzen, doch sind diese auf eine JVM begrenzt. Um die Synchronisation der einzelnen Instanzen des Scrum Boards zu realisieren, war es erforderlich, Zustandswechsel über Rechnergrenzen hinweg zu implementieren. Bevor auf das daraus entstandene Framework SynchronizeFX genauer eingegangen wird, betrachten wir die hierfür relevanten Grundlagen von JavaFX: Properties, Bindings und Listener.

Properties und Bindings

Neben den Gestaltungsmöglichkeiten von GUIs führt JavaFX Properties und Bindings ein. Dabei handelt es

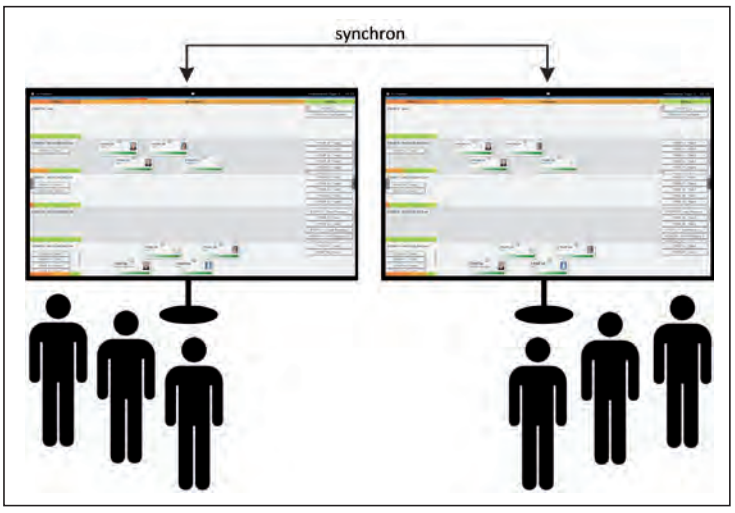


Abb. 1: Synchronisation der Scrum Boards zwischen mehreren Standorten

Das Scrum Board bei YouTube

Auf YouTube gibt es einen Einblick in die Funktionsweise [1] und Synchronisierung [2] des Scrum Boards. Die Videos sind ein gutes Beispiel für eine komplexe JavaFX-Anwendung.

sich um eine konzeptionelle Weiterentwicklung der klassischen Datentypen und des JavaBeans-Standards, die dem Entwickler einen eventgetriebenen Programmierstil zur Verfügung stellen. Dadurch kann mit geringem Aufwand auf Änderungen von Werten reagiert und es können direkte logische Verbindungen zwischen Werten aufgebaut werden.

Hierzu führt JavaFX Wrapper-Klassen ein, die herkömmliche Datentypen umschließen und dem Entwickler neue Möglichkeiten der Benutzung bieten. Diese Klassen existieren sowohl für alle flachen Datentypen als auch für Strings, Listen und allgemein Objekte. Die Benennung erfolgt stets nach dem Muster *<Datentyp>Property*, folglich *IntegerProperty*, *StringProperty*, *ListProperty* und *ObjectProperty*.

Properties umschließen immer einen normalen Wert vom jeweiligen Typ. Um direkt an den umschlossenen Wert zu gelangen, können die Methoden *get()* sowie *set()* benutzt werden. Listing 1 zeigt die Initialisierung und Benutzung einer String Property. *SimpleStringProperty* ist, wie der Name vermuten lässt, eine Implementierung des *StringProperty*-Interface, die JavaFX standardmäßig mitliefert. Äquivalent sind in JavaFX für alle Property-Typen so genannte *Simple**-Implementierungen vorhanden. Diese reichen als Standardimplementierung für die meisten Zwecke aus.

In Listing 1 ist zu erkennen, dass der JavaBeans-Standard für die Klasse *Person* eingehalten wird, da auf den Vornamen nur über *Getter* und *Setter* zugegriffen werden kann. Natürlich muss die verwendete Property auch sichtbar sein, damit deren Vorteile direkt genutzt werden können. Dazu gibt Oracles Namenskonvention vor, dass dies über Methoden nach dem Muster *<Feldname>Property()* geschehen soll.

Listing 1

```
public class Person implements Serializable {
  private StringProperty vorname = new SimpleStringProperty();

  public void setVorname(String neuerVorname){
    this.vorname.set(neuerVorname);
  }
  public String getVorname(){
    return vorname.get();
  }
  public StringProperty vornameProperty(){
    return vorname;
  }
  // ...
}
```

Listing 2

```
final Person p = new Person();
    p.vornameProperty().addListener(new ChangeListener<String>() {
      public void changed(final ObservableValue<? extends String> observable, final String
                                                                                  oldValue,
      final String newValue) {
        System.out.println("Vorname geändert von " + oldValue + " auf " + newValue);
        if (newValue.equals("DotNet")) {
          p.setVorname(oldValue == null ? "Java" : oldValue);
        }
      }
    });
    p.setVorname("DotNet");
  //Vorname geändert von null auf DotNet
  //Vorname geändert von DotNet auf Java
```

Listing 3

```
DoubleProperty a = new SimpleDoubleProperty();
    DoubleProperty b = new SimpleDoubleProperty();
    a.bind(b);
    b.set(1.3);
    System.out.println("a=" + a.get()); // -> a=1.3
    a.set(2.4); //java.lang.RuntimeException: A bound value cannot be set.
```

Listing 4

```
Person p = new Person();
    TextField vornameInput = new TextField();
    vornameInput.textProperty().bindBidirectional(p.vornameProperty());
```

Listener

Properties bieten unter anderem die Möglichkeit, Listener zu registrieren, die über Änderungen an der Property informieren. In Listing 2 wird eine Instanz von *Person* angelegt und ein Listener auf die darin enthaltene String Property gesetzt, der Änderungen auf der Konsole ausgibt. Dazu wird eine anonyme Klasse *ChangeListener* implementiert, deren *changed()*-Methode den alten und den neuen Wert der Property übergeben bekommt. Diese Werte können vom Entwickler verwendet werden, um etwa eine Validierung des neuen Werts vorzunehmen und bei Fehlschlag den alten wiederherzustellen.

Bindings

Properties können mithilfe von Bindings Änderungen von einem zu mehreren gebundenen Properties propagieren. Dabei sind sowohl unidirektionale als auch bidirektionale Bindings möglich. Ein unidirektionales Binding ist in Listing 3 zu sehen. In diesem Fall werden die Double Properties *a* und *b* angelegt und *a* an den Wert von *b* gebunden. Wird anschließend der Wert von *b* verändert, ändert sich automatisch auch der Wert von *a*. Der umgekehrte Fall gilt jedoch nicht.

Wichtig ist in diesem Zusammenhang auch der Umstand, dass der Wert einer unidirektional gebundenen Property nicht mehr per Hand gesetzt werden kann. Der Versuch wird mit einer Runtime Exception quittiert.

Unidirektionale Bindings können mit der Methode *unbind()* nachträglich wieder gelöst werden.

Im Gegensatz dazu werden Änderungen bei bidirektionalen Bindings in beide Richtungen weitergegeben, weshalb die Einschränkung bezüglich manuellen Änderns von gebundenen Properties hier nicht existiert.

Da JavaFX die meisten Eigenschaften seiner GUI-Elemente ebenfalls über Properties zur Verfügung stellt, können Bindings auf sehr wirksame Art und Weise genutzt werden. In vielen Fällen ist es möglich, umfangreiche GUI-Logik zum Auslesen und Setzen von Formularfeldern und das Abgleichen mit Modellwerten komplett einzusparen. Stattdessen kann man, wie in Listing 4 abgebildet, einfach die Modellwerte bidirektional an das entsprechende Formularfeld binden. Gibt der Nutzer einen Wert in das Textfeld ein, ist dieser automatisch und unmittelbar im Modell gesetzt. Ändert sich der Wert im Modell, z. B. ausgelöst durch eine Änderung im Backend, wird automatisch das Textfeld in der GUI geändert.

Darüber hinaus lassen sich mit JavaFX wesentlich komplexere Bindings implementieren. In Listing 5 sind einige Beispiele unter Nutzung des Fluent API von JavaFX dargestellt. Ändert sich der Vor- oder Nachname, wird automatisch das Label *name* aktualisiert. Im zweiten Beispiel wird die Fläche neu berechnet, sobald sich die Grundmaße des Vierecks ändern – aber nur, falls dieser Wert angefordert wird. JavaFX führt automatisch Buch darüber, welche Properties von welchen anderen Properties abhängen und berechnet den neuen Wert *lazy*, d. h. nur, wenn dieser benötigt wird. Neben dem Fluent API können zudem gänzlich generische Bindings entwickelt werden, auf die an dieser Stelle aber nicht weiter eingegangen wird.

SynchronizeFX

Richtig eingesetzt, können Property Bindings ein mächtiges Instrument bei der Entwicklung von Anwendungen sein, da mit ihnen viel Boilerplate-Code zur Synchronisation von Werten eingespart wird. Ein Pattern auf dieser Basis ist das *Presentation Model* (siehe Kasten: „Presentation Model“). Hierbei ist es notwendig,

Listing 5

```
Person p = new Person();
    Label name = new Label();
    name.textProperty().bind(p.vornameProperty().concat(" ").concat(p.nachnameProperty());

    public class Viereck {
      private final DoubleProperty breite = new SimpleDoubleProperty();
      private final DoubleProperty hoehe = new SimpleDoubleProperty();
      private final DoubleProperty flaeche = new SimpleDoubleProperty();
      public Rectangle() {
        flaeche.bind(breite.multiply(hoehe));
      }
      // ...
    }
```

Listing 6

```
public class PresentationModel {
  private final DoubleProperty sliderValue = new SimpleDoubleProperty();
  public DoubleProperty sliderValueProperty() {
    return sliderValue;
  }
  // Getter und Setter
}
```

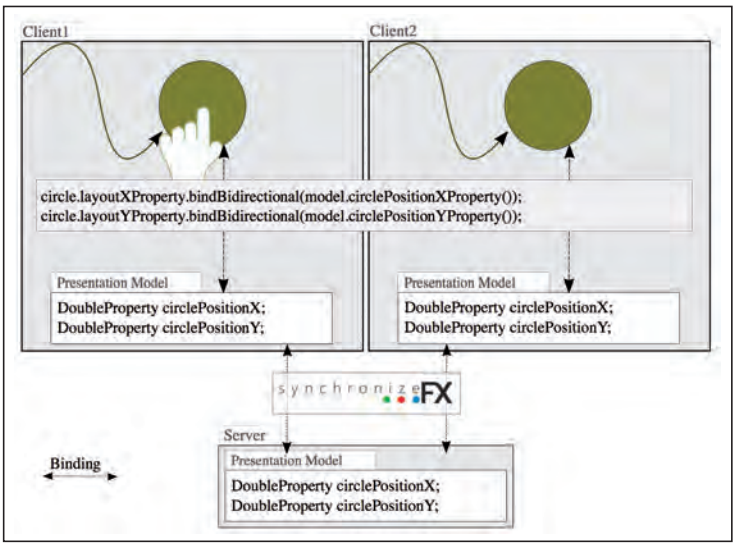


Abb. 3: Prinzip der Synchronisation von Informationen anhand von Positionsdaten

Abb. 4: Schematische Darstellung der Schichtenarchitektur von SynchronizeFX

Daten zwischen einzelnen Schichten zu synchronisieren. In JavaFX kann man dies über Properties vom Domänenmodell bis zur View und entsprechenden Bindings erreichen. Das GUI ist damit immer aktuell zum Domänenmodell.

Das Framework SynchronizeFX führt diesen Ansatz weiter und ermöglicht Property Bindings über JVM- und sogar Rechnergrenzen hinweg. Dies erlaubt Client-Server-Architekturen, bei denen das *Presentation Model* zwischen Server und Client synchronisiert wird. Dabei können aber auch Änderungen an einem Client über den Server hin zu anderen Clients propagiert und somit eine Synchronisation der Darstellung auf allen beteiligten Clients erreicht werden (**Abb. 3**).

Gegenwärtig unterstützt SynchronizeFX genau diese Herangehensweise. Der Server stellt initial ein zentrales Modell zur Verfügung. Clients, die sich nachfolgend beim Server registrieren, erhalten den aktuellen Zustand des Servermodells. Anschließend werden sie über sämtliche Änderungen auf dem Laufenden gehalten. Wird auf einem der Clients eine Änderung an dem Modell vorgenommen, wird diese zunächst zum Server und von dort aus anschließend an alle anderen verbundenen Clients propagiert.

SynchronizeFX – Beispiel

Um den Einsatz von SynchronizeFX zu demonstrieren, soll ein einfacher JavaFX Slider verwendet werden, dessen Wert auf beliebig vielen Clients synchron gehalten wird. Der vollständige Code des Beispiels ist im GitHub-Repository [4] von SynchronizeFX enthalten und soll hier nur in Auszügen gezeigt werden.

Zunächst wird ein *Presentation Model* definiert, das den relevanten UI-Zustand abbildet und später über die Clients synchronisiert wird. In unserem Beispiel ist das Modell eine einzelne Klasse, die eine Double Property *sliderValue* enthält.

Der Server besteht aus einer einzelnen Klasse mit einer einfachen *main()*-Methode (Listing 7). Zunächst wird eine Instanz des Modells angelegt. Anschließend wird der *SynchronizeFxServer* mithilfe eines Builders erzeugt. Der übergebene *ServerCallback* dient lediglich der Fehlerbehandlung bei Netzwerkproblemen und ermöglicht eine Eskalation.

Das Listing 8 zeigt die *start()*-Methode unserer JavaFX-Applikation. Die instanziierte *View* enthält die Anzeigekomponenten für den Slider und die dazugehörigen Labels. Der Wert des Sliders wird als Double Property *sliderValue* nach außen gegeben.

Auf Clientseite wird nun mithilfe des *SynchronizeFxBuilders* eine *SynchronizeFxClient*-Instanz erzeugt. Auch hier muss ein Callback-Interface implementiert werden, bei dem auch auf Clientseite auf Fehler reagiert werden kann. Dabei signalisiert die *onError()*-Methode Fehler in der Netzwerkübertragung. Die *onServerDisconnect()*-Methode wird, wie der Name vermuten lässt, aufgerufen, sobald der Client vom Server getrennt wurde.

Wichtiger ist jedoch die *modelReady()*-Methode. Diese wird aktiv, sobald das Servermodell initial komplett zum Client übertragen wurde. Diese Methode stellt

Presentation Model

Das Pattern von Martin Fowler [3] beschreibt eine View, die einzig für die Anzeige verantwortlich ist (**Abb. 2**). Was wann angezeigt werden soll, ist im Presentation Model definiert. Dieses hat Zugriff auf das Domänenmodell und ergänzt es um View-spezifische Informationen. Zwischen diesen Schichten gibt es einen Synchronisationsmechanismus wie z. B. Property Binding.

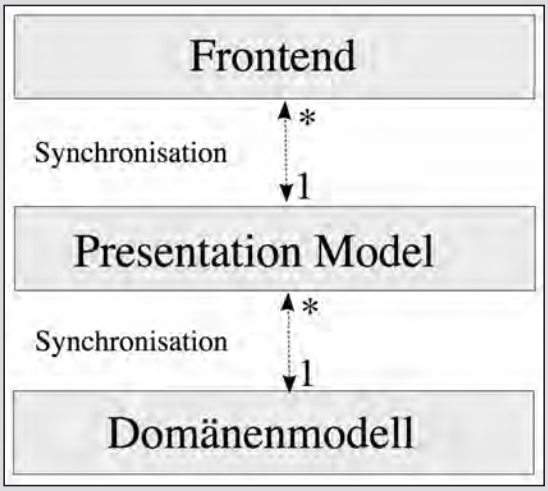


Abb. 2: Das Presentation Model

somit den Einstiegspunkt dar, ab dem das Modell zur Verfügung steht und alle darauf basierenden Bindings im Client aufgebaut werden können. In unserem Fall binden wir die Value Property des Sliders bidirektional an die *sliderValue* Property im Modell.

Jetzt kann das Beispiel ausgeführt werden. Nachdem der Server gestartet wurde, können sich beliebig viele Clients verbinden. Wenn auf einem der Clients der Slider mit der Maus bewegt wird, erscheint diese Bewegung unmittelbar auf allen anderen Clients. Im Beispiel ist die Serveradresse mit „localhost" konstant definiert. Dadurch läuft das Beispiel lediglich auf einem lokalen Computer. Hier könnte jedoch auch eine IP-Adresse des Servers eingetragen werden, um es im Netzwerk ausführen zu können.

Architektur

Die Architektur von SynchronizeFX besteht aus drei Schichten. Dies sind die Metamodellschicht, die Topologieschicht sowie die Netzwerk- und Serialisierungsschicht.

Abbildung 5 zeigt die Kommunikation zwischen den Schichten, die im Einzelnen folgende Aufgaben haben:

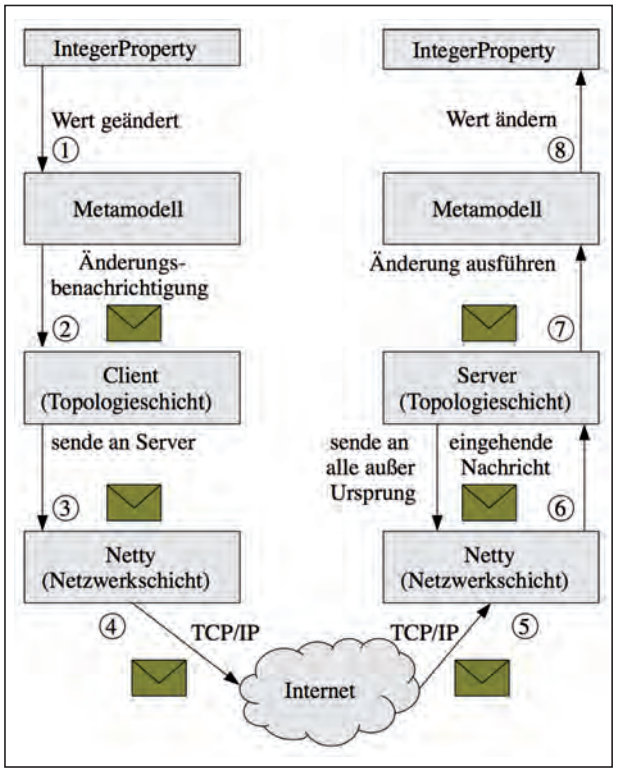


Abb. 5: Kommunikation zwischen den Schichten

Die Metamodellschicht sorgt dafür, dass sämtliche Änderungen an Properties im *Presentation Model* des Benutzers erkannt werden. SynchronizeFX erwartet als Parameter das Wurzelelement des Objektgraphen des *Presentation Models*, über das sämtliche andere Teile des Modells direkt oder indirekt referenziert sein müssen. Mithilfe von Reflection wird anschließend der Objektgraph traversiert, und auf alle Properties, die im *Presentation Model* existieren, werden spezielle Listener registriert. Erkennt einer der Listener eine Änderung, wird eine Nachricht erzeugt (1), die diese Änderung beschreibt. Eine solche Nachricht kann beispielsweise das Hinzufügen eines Elements zu einer Liste oder das Ändern des Werts einer Integer Property beschreiben.

Listing 7

```
public static void main(final String... args) {
  System.out.println("starting server");
  final PresentationModel model = new PresentationModel();
  final SynchronizeFxServer syncFxServer =
  SynchronizeFxBuilder.create().server().model(model)
  .callback(new ServerCallback() {
    @Override
    public void onError(final SynchronizeFXException exception) {
      System.out.println("Server Error:" + exception.getLocalizedMessage());
    }
  }).build();
  syncFxServer.start();
  // ...
}
```

Listing 8

```
@Override
  public void start(final Stage stage) throws Exception {
    stage.setTitle("SynchronizeFX Example Client");
    final View view = new View();
    stage.setScene(new Scene(view, 400, 200));

    client = SynchronizeFxBuilder.create().client().address("localhost")
    .callback(new ClientCallback() {
      @Override
      public void modelReady(final Object object) {
        model = (Model) object;
        view.sliderValueProperty().bindBidirectional(model.
                                                    sliderValueProperty());
      }
      @Override
      public void onError(final SynchronizeFXException exception) {
        System.out.println("Client Error: " + exception.getLocalizedMessage());
      }
      @Override
      public void onServerDisconnect() {
        System.out.println("Server disconnected");
      }
    }).build();
    client.connect();
    stage.show();
  }
```

Dieses Vorgehen hat den Vorteil, dass Reflection nur benötigt wird, um alle Properties im *Presentation Model* zu finden. Für den häufigeren Fall die Änderung einzelner Werte in Properties, wird sie nicht benötigt. Das hat Vorteile bezüglich der Performance und ermöglicht beispielsweise ein flüssiges Synchronisieren von Positionsdaten von GUI-Elementen. Nachteilig ist, dass nur Property-Felder in Java-Objekten synchron gehalten werden können.

Die Nachrichten werden anschließend an die Topologieschicht übergeben (2). Auf der anderen Übertragungsseite nimmt wiederum die Metamodellschicht diese Nachrichten von der Topologieschicht entgegen (7) und führt die Änderungen am *Presentation Model* des Benutzers aus (8).

Die Aufgabe der Topologieschicht ist es, zu entscheiden, an welche Stellen Nachrichten weitergeleitet werden müssen. Gegenwärtig existiert eine Client-Server-Implementierung für diese Schicht. Es wäre aber auch möglich, eine Peer-To-Peer-Variante zu entwickeln.

Die Netzwerk- und Serialisierungsschicht sorgt dafür, dass Nachrichten über JVM-Grenzen hinweg übertragen werden können. Dazu müssen Nachrichten, die in den darüber liegenden Schichten als Java-Objekte ausgetauscht werden, in der Regel zu Byte-Arrays oder Byte-Streams serialisiert und entsprechend auch deserialisiert werden. In der aktuellen Standardimplementierung findet die Serialisierung mithilfe des Frameworks Kryo [5] statt. Die so erzeugten Nachrichtenobjekte werden in der Standardimplementierung über TCP ausgetauscht ((4) und (5)), wofür Netty [6] verwendet wird.

Es existiert eine zweite Implementierung, die Nachrichten über WebSocket austauscht. Dazu wird serverseitig das WebSocket-API von Apache Tomcat [7] verwendet. Das hat den Vorteil, dass die Verschlüsslung und Benutzerauthentifizierung durch HTTP, die auf dem WebSocket aufbaut, bereits gegeben ist.

Abgrenzung

Das Projekt OpenDolphin [8], das im Java Magazin 5.2013 vorgestellt wurde, erlaubt ebenfalls einen Abgleich von Benutzeroberflächen. OpenDolphin unterstützt neben JavaFX weitere Frontends, wie zum Beispiel JavaScript-Clients. Der signifikanteste Unterschied der Frameworks ist das clientseitige API. Während OpenDolphin ein auf Key-Value basierendes Zugriffskonzept verwendet, um das *Presentation Model* zu lesen und zu manipulieren, erreicht SynchronizeFX durch das Arbeiten mit Properties eine Synchronisierung. Dies vereinfacht das Binden eines JavaFX GUI an die bereitgestellten Werte. Auch ermöglicht das reflektive Scannen des Objektgraphen eine sehr einfache Instanziierung des *Presentation Models*, bei dem man sich nach dem Set-up die Frage nach der Synchronisierung nicht mehr stellen muss. Durch das Bereitstellen des *Presentation Models* über JavaFX Properties kann aber im Moment nur ein JavaFX GUI ohne große Modifikationen genutzt werden.

Ausblick

SynchronizeFX ist zunächst als Teil des zu Beginn beschriebenen Scrum Boards entstanden. Anschließend wurde es Stück für Stück als eigenständiges Framework extrahiert und Open Source veröffentlicht. Gegenwärtig befindet sich das Framework im Betazustand. Es wird seit einiger Zeit erfolgreich für die visuelle Echtzeitsynchronisation der Scrum-Board-Software eingesetzt. Es gibt noch einige, bisher nicht implementierte, nützliche Erweiterungen des Frameworks. So wäre ein Mechanismus wünschenswert, der es einem Client erlaubt, Änderungen nur für einen bestimmten Teil des *Presentation Models* zu erhalten. Auch die Implementierung verschiedener Strategien zur Umsetzung konkurrierender Zugriffe ist denkbar.

Das Framework steht über GitHub unter der LGPL zur Verfügung. Wir freuen uns über Feedback, Bugreports und natürlich interessierte Entwickler, die sich an dem Framework beteiligen möchten [9].

Raik Bieniek arbeitet als Werkstudent bei der Saxonia Systems AG und ist eingeschriebener Studierender an der Hochschule Zittau/Görlitz.

Michael Thiele arbeitet seit April 2012 für die Saxonia Systems AG und entwickelt mit JavaFX interaktive Applikationen für Touchscreens. Sein Interesse gilt eigentlich dem Backend-Bereich und der Programmiersprache Scala; durch JavaFX hat er den Spaß an GUI-Entwicklung wiederentdeckt.

Manuel Mauky hat bis 2010 an der Hochschule Zittau/Görlitz Informatik studiert und arbeitet seitdem bei der Saxonia Systems AG in Görlitz, wo er sich vor allem mit Java-EE- und Frontend-Entwicklung befasst. Privat beschäftigt er sich mit JavaScript, vor allem abseits des Browsers.

Alexander Casall ist ein GUI-affiner Softwareentwickler, der leidenschaftlich Apps für iOS entwickelt und seinen technologischen Fokus im Java-Umfeld auf JavaFX gelegt hat. Er hält Vorträge über JavaFX auf Konferenzen (JAX, User Groups) und schreibt der Technologie großes Potenzial zu.

Vincent Tietz studierte an der TU Dresden Medieninformatik und ist seit 2008 als Softwareentwickler für die Saxonia Systems AG im Bereich Java- und Webentwicklung tätig. Seit 2009 arbeitet er in Kooperation mit dem Lehrstuhl für Multimediatechnik der TU Dresden an seiner Dissertation zu modellgetriebener Entwicklung von prozessorientierten Web-Mashups.

Links & Literatur

[1] http://youtu.be/uM07WWBgjsY

[2] http://youtu.be/eaBN6TcubmY

[3] http://martinfowler.com/eaaDev/PresentationModel.html

[4] https://github.com/saxsys/SynchronizeFX

[5] https://code.google.com/p/kryo/

[6] http://netty.io/

[7] http://tomcat.apache.org

[8] http://open-dolphin.org

[9] https://github.com/saxsys/SynchronizeFX

Anzeige

Ab JDK 1.8 wird JavaFX um eine Dimension reicher

Die JavaFX-Baumschule

JavaFX bekommt ab JDK 1.8 ein verbessertes 3D-API, das es relativ einfach macht, sich virtuelle 3-D-Welten zu erschaffen, vielleicht auch die eigene Applikation etwas aufzupeppen oder einfach nur, um etwas zu experimentieren. Wir wollen im Folgenden ein Programm entwerfen, das baumähnliche Strukturen auf den Bildschirm zeichnet – keine abstrakten, sondern tatsächliche, organische Bäume.

von Robert Ladstätter

Als Implementierungssprache wird Scala verwendet. Diese Programmiersprache bedarf wohl keiner weiteren Vorstellung im Java Magazin (Lesetipp). JavaFX als offizieller Nachfolger von Swing sollte dem interessierten Java-Entwickler ebenso präsent sein.

Die grundlegenden Designentscheidungen von JavaFX machen es möglich, in wenigen Zeilen Code schon ein recht komplexes Verhalten zu modellieren. Da JavaFX im Wesentlichen auf einem Scene-Graph basiert (es gibt Knoten mit Eltern-Kind-Beziehungen), ist der Schritt, ein 3D-API für JavaFX einzuführen, naheliegend. Um den im Artikel beschriebenen Sourcecode selbst ausprobieren zu können, muss man einen Early Access Build vom JDK 8 installieren [1].

Der vollständige Sourcecode zu diesem Artikel befindet sich auf meiner GitHub-Seite [2], sodass man sich das Abtippen sparen kann. In **Abbildung 1** sieht man das Ergebnis des Programms, und unter [3] gibt es dazu ein kurzes Video.

Lesetipp

Über den aktuellen Stand von Scala informiert der große Schwerpunkt in der Ausgabe 6.2013 des Java Magazins: www.jaxenter.de/magazines/Java-Magazin-613

Los geht's

Das Programm zeigt also im besten Fall eine Struktur, die einem Baum nicht unähnlich ist. Sicherlich nicht perfekt, aber gut genug, um sich mit ein paar Konzepten des 3D-API vertraut zu machen. Des Weiteren ist

Abb. 1: Screenshots des Programms

die Fingerübung auch ein nettes Beispiel für einen rekursiven Algorithmus, Pattern Matching, Implicits und Traits, sodass es von dieser Seite gesehen auch seinen Reiz hat. In Listing 1 sind die neuen Klassen aufgelistet, die ab JDK 1.8 dabei sind und von dem Beispiel benötigt werden.

Es gibt also neue Shapes, die man verwenden kann – *Cylinder* und *Sphere* leiten von einer abstrakten Klasse *Shape3D* ab. Wie in **Abbildung 2** ersichtlich, gibt es noch weitere Shapes wie *MeshView* oder *Box*.

Die Klassen tun genau das, was sie tun sollen: Sie stellen Formen im 3-D-Raum dar, die man über Transformationen in der 3-D-Welt platzieren, skalieren und drehen kann. Die *Material*-Klasse weist, wie in der 3-D-Programmierung üblich, einem Shape ein Aussehen zu, *PhongMaterial* ist eine Spezialisierung davon. Mit diesen Klassen ist es z. B. auch möglich, Shapes eine Textur, oder einfacher, eine Farbe zuzuweisen.

Mithilfe der Klasse *PointLight* kann man Licht in die Szenerie bringen. Neben *PointLight* gibt es noch die *AmbientLight*-Klasse, die eine Szene unspezifisch beleuchtet. Damit die Lichter aktiv werden, müssen sie dem Scene-Graph zugeordnet werden.

Das *Types*-Objekt dient zur Definition der *implicit* Konversion zwischen der *Point3D*-Klasse und der *SPoint3D*-Klasse. Letztere nützt die Möglichkeit unter Scala aus, Methodennamen auch mal „*“ oder „/“ zu

Listing 1

```
import javafx.scene.PointLight
import javafx.scene.paint.Material
import javafx.scene.paint.PhongMaterial
import javafx.scene.shape.Cylinder
import javafx.scene.shape.Shape3D
import javafx.scene.shape.Sphere
```

Listing 2

```
object Types {

  implicit def toSPoint3D(point3D: Point3D): SPoint3D = {
    SPoint3D(point3D.getX, point3D.getY, point3D.getZ)
  }
  implicit def toPoint3D(spoint3D: SPoint3D): Point3D = spoint3D.toPoint3D
}

case class SPoint3D(x: Double, y: Double, z: Double) {

  def -(that: SPoint3D) = SPoint3D(that.x - x, that.y - y, that.z - z)
  def +(that: SPoint3D) = SPoint3D(x + that.x, y + that.y, that.z + z)
  def *(factor: Double) = SPoint3D(factor * x, factor * y, factor * z)
  def /(l: Double) = if (l != 0) SPoint3D(x / l, y / l, z / l) else sys.error("div.0")
  def length = scala.math.sqrt(x * x + y * y + z * z)
  def displace(f: Double) =
    SPoint3D(x + (Random.nextDouble - 0.5) * f, y + (Random.nextDouble - 0.5) * f,
                                                +z + (Random.nextDouble - 0.5) * f)

  def onedir = this / length
  def normal: SPoint3D = SPoint3D(-y, x, z)
  def midpoint(other: SPoint3D) = SPoint3D((other.x + x) / 2, (other.y + y) / 2,
                                                              (other.z + z) / 2)
  def spin(phi: Double): SPoint3D =
    SPoint3D((x * cos(phi)) - (y * sin(phi)), (x * sin(phi)) + (y * cos(phi)), z
      + (if (nextBoolean) nextDouble else (-nextDouble)))

  def degrees: (Double, Double) = (if (x != 0) scala.math.atan(y / x) * 180 /
                                                          scala.math.Pi else 90,
    if (x != 0) scala.math.atan(z / x) * 180 / scala.math.Pi else 90)

  def toPoint3D = new Point3D(x, y, z)
  override def toString: String = s"[$x/$y/$z]"
}
```

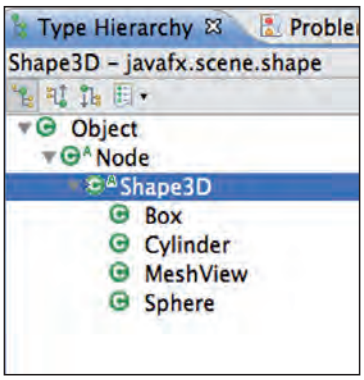


Abb. 2: Screenshot des Typs „Shape3D“

nennen, was man auch in die Schublade „operator overloading“ einordnen könnte. (Der Scala-Compiler konvertiert diese Symbole dann zu gültigen Java Identifiern wie z. B. *$plus$*. Dies bleibt jedoch ein Detail des Kompiliervorgangs und muss uns nicht weiter kümmern.)

Die Einführung der *SPoint-3D*-Klasse dient vor allem der Lesbarkeit des Programms. Neben den typischen Operationen wie +/-/* gibt es noch spezielle, für die Zwecke des Programms nützliche Methoden wie *displace* oder *midpoint*. *displace* verschiebt einen Punkt um einen gewissen randomisierten Wert, *midpoint* rechnet sich den Mittelpunkt zwischen zwei Punkten aus.

Listing 3

```
trait ShapeUtils {
  import Types._
  def mkMidPointReplacement(source: SPoint3D,
  dest: SPoint3D, displace: Double, curDetail: Double): List[(SPoint3D, SPoint3D)] = {
    if (displace < curDetail) {
      List((source, dest))
    } else {
      val displacedCenter = source.midpoint(dest).displace(displace)
      mkMidPointReplacement(source, displacedCenter, displace / 2, curDetail) ++
      mkMidPointReplacement(displacedCenter, dest, displace / 2, curDetail)
    }
  }

  def mkCylinder(source: SPoint3D, dest: SPoint3D, radius: Double, material: Material):
                                                                        Cylinder = {
    val rotation = (dest - source)
    val height = (dest - source).length
    val c = new Cylinder(radius, height)
    c.setTranslateX(source.x)
    c.setTranslateY(source.y)
    c.setTranslateZ(source.z)
    c.setRotationAxis(rotation)
    c.setRotate(rotation.degrees._1)
    c.setMaterial(material)
    c
  }

  def mkSphere(p: SPoint3D, radius: Double, material: Material): Sphere = {
    val c = new Sphere()
    c.setTranslateX(p.x)
    c.setTranslateY(p.y)
    c.setTranslateZ(p.z)
    c.setRadius(radius)
    c.setMaterial(material)
    c
  }

}
```

In den *ShapeUtils* aus Listing 3 sieht man auch gleich den Vorteil der Einführung der *SPoint3D*-Klasse, da nun der Midpoint-Algorithmus sehr leicht lesbar ist. Im Wesentlichen geht es darum, zwischen zwei Punkten die geometrische Mitte zu finden, diese zufällig zu verschieben und zwischen den beiden Ursprungspunkten und dem neuen Mittelpunkt dieselbe Operation anzuwenden, bis ein gewisser Schwellwert erreicht ist.

Listing 4

```
trait JfxUtils {
  def mkEventHandler[E <: Event](f: E => Unit) = new EventHandler[E] {
                                                        def handle(e: E) = f(e) }
}

trait Spinner extends JfxUtils {

  var anchorX: Double = _
  var anchorY: Double = _
  var anchorAngle: Double = _

  def initSpinner(scene: Scene, drawingArea: Node): Unit = {
    scene.setOnMousePressed(mkEventHandler(event => {
      anchorX = event.getSceneX()
      anchorAngle = drawingArea.getRotate()
    }))

    scene.setOnMouseDragged(mkEventHandler(event => {
      drawingArea.setRotate(anchorAngle + anchorX - event.getSceneX())
    }))
  }

}
```

Listing 5

```
object PlantSomeTrees {

  def main(args: Array[String]): Unit = {
    Application.launch(classOf[PlantSomeTrees], args: _*)
  }

}

class PlantSomeTrees extends javafx.application.Application with
                                                    ShapeUtils with Spinner {

  ...

  override def start(stage: Stage): Unit = {
    ...
  }

}
```

Durch den Import des *Types*-Objekts kann man ein *SPoint3D*-Objekt anscheinend auch in der *mkCylinder*-Methode dem Zylinder als Rotationsachse zuordnen, jedoch wird vorher noch eine „implizite Konversion" in ein *Point3D*-Objekt durchgeführt. Mit diesem Trick ist es also möglich, bestehende APIs mit eigenen Klassen zu füttern. Entsprechende Konversionen werden im Hintergrund ausgeführt.

Wer sich in die Richtung schlauer machen möchte, dem sei auf jeden Fall „Programming in Scala, 2nd Edition" [4] (bzw. für besonders schwere Fälle das Buch „Scala in Depth" [5]) empfohlen.

Listing 4 enthält zwei Traits, die für den täglichen JavaFX-Gebrauch sehr nützlich sind. Im *JfxUtils* wird eine Hilfsfunktion definiert, die einen *EventHandler* erzeugt, der in seiner *handle*-Methode die ihm übergebene Funktion ausführt. Dieses Muster tritt immer wieder auf und kann in Sprachen, die Funktionen als Parameter übergeben können, schön zentral zusammengezogen werden. Im nachfolgenden *Spinner* Trait wird die *mkEventHandler*-Methode auch sofort benützt. Hier kann man schön sehen, wie kurz und knapp man solche Sachen in Scala formulieren kann, wo Java doch etwas mehr „plappert". Aber mit JDK 8 wird sich auch in

Listing 6

```
class PlantSomeTrees extends javafx.application.Application with ShapeUtils
                                                              with Spinner {

  // Konstanten
  val canvasWidth = 800
  val canvasHeight = 600
  val treeDepth = 4
  val trunkWidth = 20
  val trunkColor = Color.BURLYWOOD
  val curDetail = 2
  val displace = 40
  val treeSize = 400
  val (minDegree, maxDegree) = (Pi / 4, Pi / 2)
  val growingSpeed = 96

  ...

  sealed trait ATree
  case class Branch(source: SPoint3D, dest: SPoint3D, material: Material,
                                        width: Int, ord: Int) extends ATree
  case class SubTree(center: ATree, left: ATree, right: ATree) extends ATree

  def mkRandomTree(root: Branch): ATree = {

    def mkRandTree(tree: ATree): ATree =
    tree match {
      case Branch(start, dest, material, width, ord) => {
        ord match {
          case 0 => tree
          case _ => {
            val dir = (start - dest).onedir
            val length = (start - dest).length

            val l1 = length * (1 - Random.nextDouble * 0.7)
            val l2 = length * (1 - Random.nextDouble * 0.7)

            // startpoint of left and right branch
            val midRoot = start + dir * length

            mkRandTree(SubTree(
            Branch(start, start + (dir * length), treeMaterials(ord), width,
                                                    ord - 1), // trunk
              Branch(midRoot, midRoot + (dir.spin(mkRandDegree) * l1),
                  treeMaterials(ord - 1), if (width > 1) width / 2 else 1, ord - 1),
                                                              // leftbranch
              Branch(midRoot, midRoot + (dir.spin(-mkRandDegree) * l2),
                 treeMaterials(ord - 1), if (width > 1) width / 2 else 1, ord - 1)))
                                                              // rightbranch
          }
        }
      }
      case SubTree(center, left, right) => SubTree(mkRandTree(center),
                                        mkRandTree(left), mkRandTree(right))

    }

    mkRandTree(root)
  }

  def traverse(tree: ATree): List[Shape3D] = {
    tree match {
      case Branch(start, dest, m, width, ord) => {
        val points = mkMidPointReplacement(start, dest, displace, curDetail)
        width match {
          case 1 => {
            points.map {
              case (start, dest) => {
                mkCylinder(start, dest, if (width > 0) width else 1, m)
              }
            }
          }
          case _ => points.map {
            case (start, dest) => {
              mkCylinder(start, dest, width, m)
            }
          }
        }
      }
      case SubTree(center, left, right) => traverse(center) ++ traverse(left)
                                                        ++ traverse(right)
    }
  }

  ...

}
```

diese Richtung einiges tun [6]. Der *Spinner* Trait merkt sich im Wesentlichen die Differenz der Mauspositionen, wenn man die Maustaste gedrückt lässt, und wendet diese Differenz dann auf die Rotation der Zeichenfläche – in unserem Fall den Baum – an. Nun kommen wir zum Hauptprogramm, das die bisher eingeführten Klassen und Traits miteinander verbindet.

In Listing 5 sehen wir nun, wie man mit Scala JavaFX-Applikationen startet. Man muss hier etwas aufpassen und die „richtige" Applikation instanziieren, da auch Scala einen *Application* Trait kennt und der bei fehlendem Import dann vom Scala-Compiler genommen wird – ein kleiner Stolperstein mit bösen Folgen!

In Listing 6 findet man die Definition von diversen Konstanten. Hier lohnt es sich, ein wenig zu experimentieren. Je nach Auswahl der Parameter entstehen die unterschiedlichsten Bäume – hier kann man sehr schnell das Ende der Kaffeepause übersehen. Der Kern des Programms basiert auf einer zufälligen Variation von diesen Parametern, also hat der Algorithmus nichts mit richtigen Pflanzen und deren Charakteristiken zu tun. Dem Leser sei weiterführende Literatur ans Herz gelegt, einen guten Einstieg gibt [7].

Der Kern des Programms besteht aus den zwei Funktionen *mkRandomTree* und *traverse*. Erstere baut eine Baumstruktur auf, zweitere verarbeitet sodann die Datenstruktur und verwandelt die *abstrakte* Darstellung in eine *konkrete*, auf JavaFX basierende Liste von Zeichenbefehlen. Diese Trennung macht es leichter, auch für andere Zielplattformen (wie wär's mit WebGL?) die geeignete Befehlsfolge zu generieren. In beiden Funktionen sieht man, wie man durch Pattern Matching und Rekursion solche Baumstrukturen sehr leicht verarbeiten kann.

Die *traverse*-Funktion beschreibt auch schön in ihrer Signatur, was von ihr zu erwarten ist: Wir bekommen eine Liste von 3-D-Shapes zurück. Diese Liste wird schließlich dem Scene-Graph hinzugefügt.

Dies sieht man in Listing 7, wo der Variable die Liste der zu zeichnenden Shapes zugeordnet wird. Diese Liste wiederum wird in einer Timeline Element für Element abgearbeitet und dem Scene-Graph hinzugefügt, was einen netten grafischen Effekt („der Baum wächst") ergibt.

Listing 7

```
class PlantSomeTrees extends javafx.application.Application with
ShapeUtils with Spinner {

  ....

  def growTree(drawingArea: Group, start: SPoint3D) = {
    val dest = start + SPoint3D(0, treeSize, 0)
    val broot = Branch(dest, start, treeMaterials(treeDepth), trunkWidth,
treeDepth)
    var cylinders2Paint = traverse(mkRandomTree(broot)).toList

    val tree = new Group()

    drawingArea.getChildren.add(tree)
    val growTimeline = new Timeline
    growTimeline.setRate(growingSpeed)
    growTimeline.setCycleCount(Animation.INDEFINITE)
    growTimeline.getKeyFrames().add(
    new KeyFrame(Duration.seconds(1), mkEventHandler((event:
ActionEvent) => {
      if (!cylinders2Paint.isEmpty) {
        val (hd :: tail) = cylinders2Paint
        tree.getChildren.add(hd)
        cylinders2Paint = tail
      } else {
        growTimeline.stop
      }
    })))
    growTimeline.play()
  }

  ...
}
```

Fazit

Wie man nun sieht, gibt es in der Behandlung von JavaFX-3-D-Klassen keinen großen Unterschied zu der Art, wie man mit 2-D-Klassen umgeht. Man kann z. B. auch auf 3-D-Objekten Event Handler registrieren oder Properties binden. Es liegt jedoch in der Natur der Sache, dass man mit einer zusätzlichen Dimension eine ganze Reihe neuer Lösungen suchen muss: Wo schaut meine Kamera hin? Wo ist das Licht? usw. Das alles wurde im JavaFX-3D-API jedoch gut gelöst.

Dipl.-Ing. Robert Ladstätter ist Softwareentwickler aus Graz.

Links & Literatur

[1] http://jdk8.java.net/download.html

[2] http://github.com/rladstaetter/plant-some-trees-with-javafx/

[3] http://youtu.be/UlirlK3Tzrg

[4] http://www.artima.com/shop/programming_in_scala_2ed

[5] http://www.manning.com/suereth/

[6] http://www.techempower.com/blog/2013/03/26/everything-about-java-8/

[7] http://www.computerpflanzen.de

3-D-Programmierung

Besonderheiten und Möglichkeiten

Processing ist eine auf Java basierende Open-Source-Programmiersprache samt Entwicklungsumgebung. Im ersten Artikel dieser Serie wurden Grundlagen und Ursprünge dieses Visual-Java-Tools erläutert und ein paar Fingerübungen mit der 2-D-Engine gemacht. Nun möchten wir einem aktuellen Trend in der Filmindustrie folgen und überprüfen, was Processing im 3-D-Bereich zu bieten hat.

von Stefan Siprell

Bisher haben wir immer den Standard 2-D-Rendermode verwendet. Dieser wird implizit beim Initialisieren des Bildschirms gesetzt:

```
size(800, 600)
```

Daher ist der Schritt bei der Ausgabe von 3-D die explizite Benennung des Modus. Hierzu gibt es eine Reihe von Modi (Tabelle 1).

Der Artikel wird sich ausschließlich mit P3D beschäftigen, da hier die Vorteile von Processing unterstrichen werden sollen – OpenGL bietet lediglich direkten Zugriff auf den Java-OpenGL-Wrapper an. Außerdem gibt es keine Garantien, dass der Modus auch weiterhin durch Processing unterstützt wird. So wurde zum Beispiel schon jetzt die Möglichkeit des Java-Applet-Exports für OpenGL eingeschränkt.

Der 3-D-Raum leitet sich direkt vom 2-D-Raum ab. Vom Ursprung nach unten zeigt die Y-Achse, die X-Achse verläuft vom Ursprung nach rechts, und die Z-Achse kommt aus dem Ursprung auf den Betrachter zu.

Die Kamera schaut auf die Mitte des Bildschirms mit z = 0 und hat ihren Ursprung ebenfalls in der Mitte des Bildschirms (**Abb. 1**), besitzt aber einen z-Wert, der ungefähr der Bildschirmhöhe entspricht (die genaue Formel lautet: (Höhe/2)/(tan π/6)).

Genug Theorie, wir wollen anfangen, eine – zugegebenermaßen primitive, aber nachvollziehbare – Darstellung einer beliebigen Großstadt mit Skyline interaktiv darzustellen.

Aufbau der Szene

Zunächst benötigen wir einen Himmel, den wir uns mit *drawSky()* generieren. Hier finden wir eine neue Methode mit dem Namen *sphere()* zum Zeichnen einer Kugel. Im ersten Artikel haben wir bereits das 2-D-Gegenstück *ellipse()* besprochen. *totallength* ist die Kantenlänge unserer Stadt. Hier benutzen wir sie, um eine Kugel um die gesamte Stadt zu zeichnen. *fill()* und *stroke()* sind ebenfalls schon bekannt und dienen dazu, den Himmel blau zu zeichnen. Um den Boden bzw. die Straße zu zeichnen, nutzen wir *drawGround()*. Auch hier sind die Operationen *pushMatrix()*, *translate()* und *popMatrix()* schon bekannt. Neu ist die *box()*-Methode, mit der man Quader zeichnen kann. In der Methode zeichnen wir einen flachen Quader mit einer Höhe von 0 (**Abb. 2**).

Die dreidimensionalen Körper können mit *fill()* farblich gefüllt werden. Um die Flächen mit Bitmaps, also mit Bildern, zu überziehen, muss man in der Regel auf die schon vorgestellten Vertexes zurückgreifen [1]. Im Rahmen des Artikels möchten wir aber eine externe Bibliothek einbinden, um Körper mit Texturen zu versehen.

Modus	Beschreibung
P2D	Default-2-D-Renderer (nutzt intern OpenGL)
P3D	Default-3-D-Renderer (nutzt intern OpenGL)
OPENGL	Bietet direkten Zugriff auf das JOGL-API
PDF	Vector 2-D-Shaped in PDF-Dateien

Tabelle 1: Render-Methoden

Artikelserie

Teil 1: Einführung ins Processing, Nutzen der 2-D-Rendering-Engine
Teil 2: Nutzen der 3-D-Rendering-Engine mit Kamerafahrten
Teil 3: Computer Vision und Augmented Reality mit Processing
Teil 4: Professionelle Datenvisualisierung mit Java

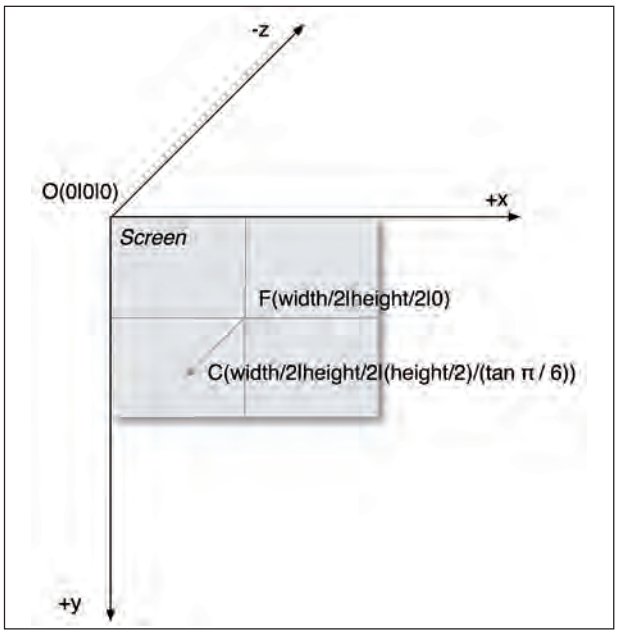


Abb. 1: Dimensionen im 3-D-Raum, der Ursprung liegt in O, die Kamera liegt auf dem Punkt C und zeigt auf F

Standardbibliothek	Beschreibung
dxf	Export von Modellen in das DXF-Format
lwjgl	Unterstützt bei der Erstellung von Spielen
minim	Integriert Audiofunktionen
net	Zugriff auf diverse Netzwerkroutinen
pdf	Export der Ausgabe in das PDF-Format
serial	Zugriff auf die virtuellen Serial Ports via USB
video	Erlaubt das Abspielen von Videos, Lesen der Webcam etc.

Tabelle 2: Standardbibliotheken

Methode	Beschreibung
camera()	Angabe der Position, Blickpunkt und Ausrichtung der Kamera
ortho()	Rendern mit einer orthographischen Perspektive
perspective()	„normale“ Projektion
frustrum()	Weitere Perspektivendarstellung, Quadrate werden als Pyramidenstümpfe gerendert
printCamera()	Ausgabe von Debug-Informationen für die Kamera
printProjection()	Ausgabe von Perspektiveninformationen

Tabelle 3: Methoden für Kamera- und Perspektivensteuerung

Methode	Beschreibung
mouseMoved()	Maus bewegt sich über den Ausgabebildschirm
mouseClicked()	Eine der Maustasten wurde gedrückt und losgelassen
mouseDragged ()	Die Maus wird mit gedrückter Taste bewegt
mousePressed ()	Eine der Maustasten wurde gedrückt
mouseReleased ()	Eine der Maustasten wurde losgelassen
keyPressed ()	Auf der Tastatur wurde eine Taste gedrückt
keyReleased()	Eine der Tasten auf der Tastatur wurde losgelassen
keyTyped	Es erfolgte eine Eingabe auf der Tastatur

Tabelle 4: Event-Handler-Methoden für Tastatur- und Mauseingaben

Fremde Bibliotheken einbinden

Bibliotheken lassen sich sehr einfach einbinden, indem man in der Processing-IDE unter dem Menüpunkt Sketch | Import Library eine der schon vorinstallierten Bibliotheken auswählt. Diese können Tabelle 2 entnommen werden.

Neue Bibliotheken lassen sich unter Sketch | Import Library | Add Library... suchen und installieren. Nachdem man nach *Shapes3D* gesucht, die entsprechende Bibliothek installiert und eingebunden hat, wird man feststellen, dass die entsprechenden Import-Statements am Anfang der Sketch eingebunden werden. Ferner ist es möglich, lauffertige Beispiele für die jeweiligen Bibliotheken anzeigen zu lassen. Hierzu kann man unter File | Examples die entsprechenden Sketches öffnen und ausprobieren. Bibliotheken lassen sich auch per Hand in das *Processing/libraries*-Verzeichnis im User Home kopieren.

Nach der Vorarbeit kann die *initBlocks()*-Methode geschrieben werden. Hierzu wird für jeden Block im Raster, analog zu einer amerikanischen Stadt (**Abb. 3**), eine Box (eine der *Shapes3D*-Klassen) instanziiert. Die Grundfläche entspricht der des Blocks, und die Höhe wird zufällig aus einer Zahl zwischen 120 und 250 gesetzt. Auf den Seiten und der Decke der Box projizieren wir jeweils eine zufällig ausgewählte Textur und speichern das Ergebnis in einem zweidimensionalen Array (**Abb. 4**). Die erste Dimension des Arrays gibt die horizontale Position und die zweite Dimension die vertikale Position im Raster wieder.

In dem Beispiel haben wir nur eine Box benutzt. Es gibt aber noch weitere Formen, wie etwa dreidimensionale Ellipsen, Spiralen, Röhren, Kegel und vieles mehr. Falls man einen objektorientierten Zugriff bevorzugt oder viel mit Texturen arbeitet, sollte man *Shapes3D* definitiv anschauen [2].

Lights, Camera, Action!

Wir haben nun eine ganz nette Szene an Hochhäusern zusammengestellt. Es fehlt nur noch die Kamera, um das alles aufzunehmen. Processing stellt hier einige Methoden zur Verfügung, um die Kamera zu steuern (Tabelle 3).

Man kann mit den Funktionen *beginCamera()* und *endCamera()* eine Klammer über Transformationen bilden, um die Kamera zu drehen oder Ähnliches. Allerdings ist das alles etwas mühselig, sodass wir für schöne Kamerafahrten eine zweite Bibliothek einbinden werden: OCD – Obsessive Camera Direction. Nach dem Suchen und dem Einbinden der Bibliothek kann man sich eine Kamera instanziieren. Auch hier (im Code in der *setup()*-Methode) gibt man Position, Blickpunkt und Ausrichtung an und kann das Objekt später manipulieren. Da wir die Kameraausrichtung und -position ändern werden, müssen wir die Kameraeinstellungen regelmäßig an die Render Engine melden, was mittels *camera.feed()* in der *lightsAndCamera()*-Methode geschieht.

Die Kamera werden wir mit der Maus verschieben. Horizontale Bewegungen drehen die Kamera um die

Mitte der Stadt, während mit vertikalen Bewegungen die Kamera hinein- und hinausfahren wird. Hierzu nutzen wir eine der Input-Methoden von Processing – in der Java-Welt wäre das ein Event Handler. Die in Tabelle 4 aufgelisteten Methoden stehen zur Verfügung.

Innerhalb der *mouseMoved()*-Methode können wir nun mit *mouseX* und *mouseY* die aktuelle Mausposition abrufen. Mit *pmouseX* und *pmouseY* kann uns Processing auch mitteilen, wo sich die Maus im vorherigen Frame (also zum Zeitpunkt der vorherigen Ausgabe) befand. Mit der Bewegung der Maus zwischen den Frames berechnen wir einfach die Parameter für die jeweiligen Kameramethoden.

Die OCD-Kamera verhält sich viel mehr wie eine echte Filmkamera als die originale Processing-Kamera. Viele der Bewegungen eines professionellen Kameramanns sind abrufbar. So kann man die Kamera mit *truck()*, *boom()* und *dolly()* linear verschieben, mit *tilt()*, *pan()* und *roll()* um diverse Achsen drehen oder auch mit *arc()* und *circle()* Kreisbewegungen ausführen. Auf der Homepage der Bibliothek [3] sind alle Methoden gut dokumentiert und mit Beispielen versehen.

Die schönsten Szenen und tollsten Kamerafahrten sind nutzlos, wenn es kein Licht gibt. Ohne weitere Einstellungen wird Processing ein ambientes weißes Licht zur Verfügung stellen. Das Licht hat weder Quelle noch Richtung, sodass alle Flächen gleich ausgeleuchtet und keine Schatten geworfen werden. Damit kann man zwar sehen, wie die Szene aufgebaut ist. Aber spannend ist das nicht.

In der *lightsAndCamera()*-Methode wird das ambiente Licht auf einen Grauwert eingestellt, um sehr dunkle und unnatürliche Schatten zu reduzieren. Mit *lightFalloff()* wird für alle nachfolgend definierten Lichtquellen definiert, wie das Licht über die Entfernung nachlässt. Die Methode akzeptiert drei Parameter für einen konstanten, linearen und quadratischen Abfall. Anschließend gibt es eine punktförmige Lichtquelle, die radial vom Mittelpunkt ausstrahlt. Diese wird mit dem Aufruf *pointLight()* definiert. Die ersten Parameter bestimmen die Farbe und die letzten Parameter die Koordinaten der Lichtquelle (**Abb. 5**).

Eine kleine Auflistung der möglichen Lichtquellen und der entsprechenden Methoden finden Sie in Tabelle 5.

Damit man auch als Laie erahnen kann, wo die Lichtquelle steht, legen wir unter die Lichtquelle eine weiße Kugel. In diesem Beispiel ausgelassen sind die Möglichkeiten, die Reflektionen der Lichtquellen an den Objekten einzustellen. Man kann damit zum Beispiel bei Stillleben tolle Effekte erzielen. Mit *lightSpecular()* kann man die Farbe des Lichts einstellen, das in eine konkrete Richtung an einem Objekt reflektiert wird. Die diffusen Reflektionen des Lichts werden weiterhin über die Lichtfarbe eingestellt. Mit den Methoden *ambient()*, *emissive()*, *shininess()* und *specular()* kann man die optischen Materialeigenschaften einstellen. Beispiele findet man direkt in der Processing IDE unter File | Examples ... und dort unter Basics | Lighting.

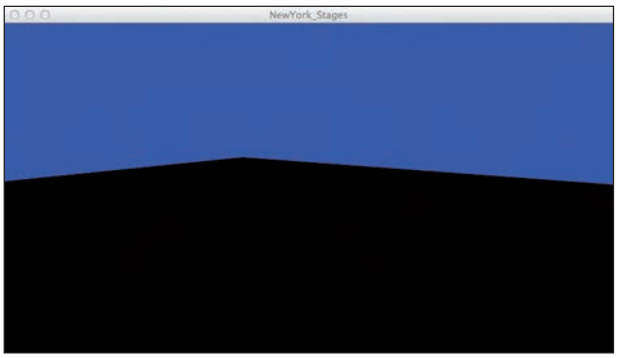


Abb. 2: Die Himmelszene (blau), um eine Bodenplatte erweitert

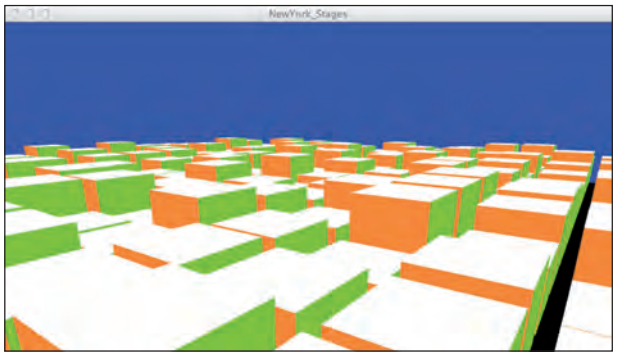


Abb. 3: Gebäude werden nun hinzugefügt, haben aber noch keine Textur

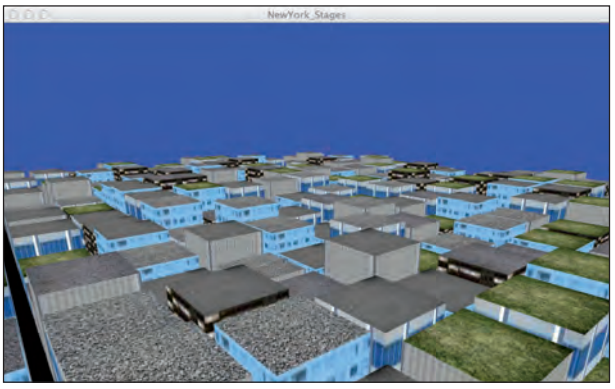


Abb. 4: Gebäude haben nun auch Texturen

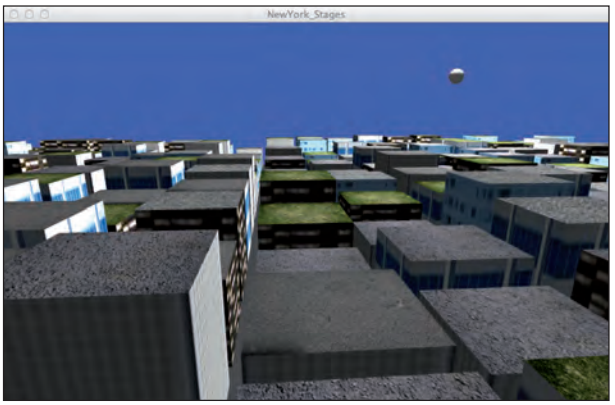


Abb. 5: Finale Szenen mit neuen Lichtquellen und der Kugel zur Kennzeichnung der Lichtquelle

Methode	Beschreibung
ambientLight ()	Licht ohne Richtung und Quelle
directionalLight ()	Licht mit spezifischer Richtung, aber ohne Quelle
pointLight ()	Radiale Lichtquelle mit spezifischem Ursprung
spotLight ()	Licht mit spezifischer Richtung und Quelle

Tabelle 5: Methoden zur Gestaltung der Lichtquellen

Zusammenfassung

Leider habe ich einen kleinen Wermutstropfen, der dem aufmerksamen Leser sicherlich aufgefallen ist. Normalerweise kann man mit der Methode *loadImage()* Bilder ohne Pfadangabe laden, wenn diese zuvor mit Sketch | Add File installiert wurden. Leider hat die aktuellste Processing-Version (2.0b9) auf OS X einen Bug, sodass ich den kompletten Pfad angeben musste. Daher die Konstruktion mit der Methode *ldPrefixImg* und der Konstanten *PREFIX*.

Auch mit dem Workaround konnten wir mit weniger als 100 Zeilen Code, zwei neuen Bibliotheken und ein paar Texturen eine komplette Szene erstellen und ein paar Einblicke in die 3-D-Fähigkeiten von Processing gewinnen (Listing 1). Im nächsten Artikel möchten wir noch tiefer in 3-D einsteigen und eine Augmented-Reality-Anwendung erstellen. Bis dahin kann ich Ihnen nur nochmals die Processing-Reference-Seite [4] ans Herz legen, da sind alle erwähnten Methoden nochmals im Detail erklärt.

Stefan Siprell (stefan.siprell@exxeta.com) ist Teamleiter bei der Exxeta AG in Karlsruhe. Seine fachlichen und technologischen Schwerpunkte liegen im agilen Software Engineering. In der Freizeit beschäftigt er sich mit Arduino, Processing und Co.

Links & Literatur

[1] http://processing.org/reference/texture_html

[2] http://www.lagers.org.uk/s3d4p/

[3] http://gdsstudios.com/processing/libraries/ocd/reference

[4] http://processing.org/reference/

Listing 1: Die Anwendung

```
import shapes3d.utils.*;
import shapes3d.*;
import damkjer.ocd.*;

int blockLength = 100;
int streetWidth = 20;

int blocksTotal = 20;
int totalLength  =  blockLength * blocksTotal + streetWidth * (blocksTotal - 1);
Box[][] blocks = new Box[blocksTotal][blocksTotal];
private static final String PREFIX = "/Users/stefansiprell/Pictures/";
PImage[] sides = new PImage[]{ldPrefixImg("/side01.jpeg"), ldPrefixImg
            ("side02.jpeg"),ldPrefixImg("side03.jpeg"),ldPrefixImg("side04.jpeg")};
PImage[] tops = new PImage[]{ldPrefixImg("/top01.jpeg"), ldPrefixImg
              ("top02.jpeg"),ldPrefixImg("top03.jpeg"),ldPrefixImg("top04.jpeg")};
Camera camera;

PImage ldPrefixImg(String name){
  return loadImage(PREFIX+name);}

void setup(){
  size(800, 600, P3D);
  initBlocks();
  camera = new Camera(this, 0, -100, -300, 0, 0, 0, 0, -1, 0); }

void initBlocks(){
 for(int horizontal = 0; horizontal < blocksTotal; horizontal ++){
   for(int vertical = 0; vertical < blocksTotal; vertical ++){
       blocks[horizontal][vertical] = new Box(this, blockLength, random(120, 250),
                                                                  blockLength);
       blocks[horizontal][vertical].setTexture(sides[int(random(0,4))], Box.FRONT |
                                            Box.BACK | Box.LEFT | Box.RIGHT);
       blocks[horizontal][vertical].setTexture(tops[int(random(0,4))], Box.TOP);
       blocks[horizontal][vertical].drawMode(Shape3D.TEXTURE);

   }}}

void lightsAndCamera(){
 camera.feed();
 ambientLight(90,90,90);
 pushMatrix();
  translate(-totalLength/4, -300, +totalLength/4);
  noStroke();
  fill(255);
  shininess(1.0);
  sphere(20);
 popMatrix();
 lightFalloff(0.5, 0, 0.0);
 pointLight(250, 250, 255, -totalLength/4, -420, +totalLength/4); }

void draw(){
 lightsAndCamera();
 drawGround();
 for(int horizontal = 0; horizontal < blocksTotal; horizontal ++){
   for(int vertical = 0; vertical < blocksTotal; vertical ++){
       drawBlock(horizontal, vertical);
   }}
 drawSky();}

void drawBlock(int horizontal, int vertical){
  int xOffset = (blockLength + streetWidth) * horizontal - totalLength / 2;
  int zOffset = (blockLength + streetWidth) * vertical - totalLength / 2;
  pushMatrix();
    fill(200);
    stroke(255);
    translate(xOffset, 0, zOffset);
    blocks[horizontal][vertical].draw();
  popMatrix();}

void drawGround(){
  pushMatrix();
    fill(10);
    noStroke();
    box(totalLength, 0, totalLength);
  popMatrix();}

void drawSky(){
  fill(color(0,0,255));
  noStroke();
  sphere(totalLength*(1/sqrt(2)));}

void mouseMoved() {
  camera.circle(radians(mouseX - pmouseX));
  camera.dolly(mouseY - pmouseY);}
```

Modellbahn powered by Java EE

Java on Tracks

Wenn man sich professionell mit der Entwicklung von Software befasst und dadurch Tastatur und Monitor tagsüber (und manchmal auch nachts) ständige Begleiter sind, nutzt man IT dann auch in der Freizeit? Man muss nicht, aber man kann – und im hier beschriebenen Fall liegt die Verbindung auch recht nahe: Ich baue und betreibe schon seit einigen Jahren eine Modellbahn im Keller. Die Betonung liegt hier auf „Modell" als Abgrenzung zu den Spielzeugeisenbahnen, die zumindest in meiner Kinder- und Jugendzeit einige Kinderzimmer schmückten. Um bei der Modellbahn einen vorbildähnlichen Betrieb zu ermöglichen, ist sie mit einer Mehrzugsteuerung ausgestattet, die auch ein Computerinterface aufweist.

von Dirk Weil

Natürlich kann man entsprechende Steuerungssoftware fertig erwerben, aber das ist ja langweilig. Also haben wir uns bei GEDOPLAN für eines unserer Code Camps die Aufgabe gestellt, die Modellbahn mit einer Java-EE-Anwendung zu steuern. Zunächst musste ein Projektname gefunden werden. Die Wahl fiel auf *V5T11*, was *Visual Train Control* bedeutet – analog zur bekannten Abkürzung *I18n* für *Internationalization*. In anderer Reihenfolge erhält man *VT 11.5*, was einen Triebzug bezeichnet, der sicher die Herzen vieler Modellbahner höher schlagen lässt.

Aufgabenstellung

Die vollständige Automatisierung des Anlagenbetriebs soll durch *V5T11* nicht erreicht werden, schließlich möchte der Modellbahner seine Züge meist selbst aktiv steuern. Vielmehr soll *V5T11* die Aufgaben lösen, die beim großen Vorbild im Stellwerk angesiedelt sind: Visualisierung des Gleisplans, Anzeige von Gleisbelegungen sowie Reservieren und Freigeben von Fahrstraßen inklusive der dazu nötigen Weichen- und Signalstellungen.

Für unser Code Camp war es darüber hinaus erwünscht, möglichst viele Teile der Plattform Java EE 6 sinnvoll einzusetzen. Dadurch finden sich in *V5T11* neben der auf CDI basierenden Geschäftslogik ein Ressourcenadapter zur Anbindung der Mehrzugsteuerung, eine Webanwendung zur Administration und zum Monitoring, EJBs für die Versorgung eines Remote-Stellwerk-Clients sowie REST-Web-Services für die Fahrzeugsteuerung mit einer Android-App. Im Folgenden soll auf einige Details von *V5T11* ein wenig Licht geworfen werden.

Systemaufbau

Zur Steuerung der Fahrzeuge wird eine Mehrzugsteuerung eingesetzt. Anders als bei der Spieleisenbahn benötigt man dann nicht mehrere Stromkreise, auf denen die Loks unabhängig voneinander fahren können. Vielmehr werden alle Fahrzeuge mit der gleichen Dauerspannung versorgt, auf die ein Digitalsignal für Geschwindigkeit, Richtung, Licht etc. aufgeprägt ist. Decoder in den Fahrzeugen setzen die so kodierten Befehle für die Fahrzeuge entsprechend um. In ähnlicher Weise werden Weichen und Signale angesteuert sowie umgekehrt die Belegung von Gleisabschnitten durch Stromfühler gemessen und an eine Zentrale gemeldet. Loks, Weichen, Signale und Gleisabschnitte sind somit an einen Kommandobus angeschlossen. Bei meiner Anlage kommt eine Mehrzugsteuerung des Systems Selectrix zum Einsatz, für das Komponenten von diversen Herstellern erhältlich sind (**Abb. 1**).

Mit den beschriebenen Hardware- und Softwarekomponenten ergibt sich der in **Abbildung 2** gezeigte schematische Systemaufbau für eine kleine Demonstrationsanlage, die während unseres Code Camps zum Einsatz kam (**Abb. 3**).

Selectrix-Adapter

Die Kommunikation mit der Mehrzugsteuerung geschieht über ein einfaches serielles Protokoll: Die am Kommandobus angeschlossenen Geräte haben eine Adresse im Bereich 1 … 127. Die Steuerinformationen für eine Lok lassen sich in 8 Bit ausdrücken. An einen Funktionsdecoder sind bis zu acht Weichen oder Signale angeschlossen. Ebenso viele Gleisabschnitte lassen sich mit einem Besetztmelder überwachen. Somit reichen also zur Kommunikation mit der Steuerung jeweils Einzelbytes für Adressen und Daten.

Abb. 1: Komponenten der Mehrzugsteuerung

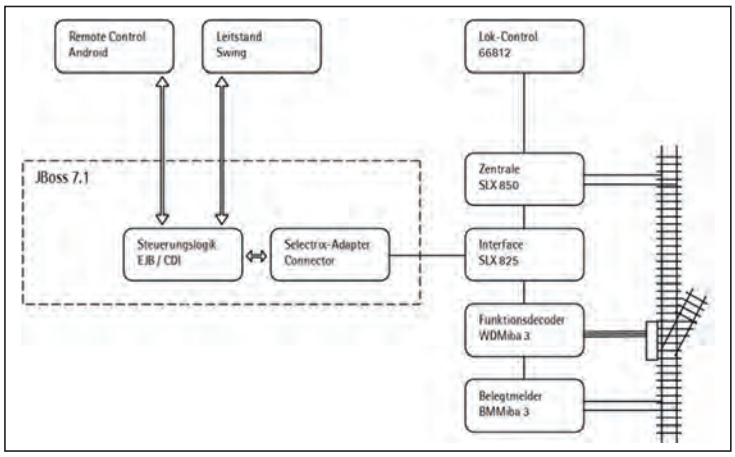


Abb. 2: Systemaufbau

Technisch wird der Datenaustausch mit der Mehrzugsteuerung über einen Ressourcenadapter nach JCA 1.6 abgewickelt. Er stellt der restlichen Anwendung das Interface *SelectrixConnection* zur Verfügung:

```
public interface SelectrixConnection extends AutoCloseable
{
  public int getValue(int address);
  public void setValue(int address, int value);
  ...
```

Die Mehrzugsteuerung kann Änderungen auch selbsttätig melden. Daher verarbeitet der Adapter auch Inbound Messages des Typs *SelectrixMessage*:

```
public class SelectrixMessage implements Serializable
{
  private int address;
  private int value;
...
```

Der Ressourcenadapter benötigt für die Kommunikation eine serielle Schnittstelle (für die Jüngeren unter uns: So etwas hatte früher oft einen 9- oder 25-poligen so genannten Sub-D-Stecker und ist heute bspw. per USB-Seriell-Adapter zu bekommen). Leider enthält die Java Runtime von sich aus keine Unterstützung für serielle Schnittstellen. In *V5T11* kommt daher die frei verfügbare Bibliothek *RXTX* [1] zum Einsatz. Sie enthält ein leicht einsetzbares API zum Zugriff auf die Schnittstellen und einen nativen Anteil in Form einer DLL bzw. Shared Library. Dieser native Anteil ist allerdings nicht ganz unproblematisch: Zum einen kann nativer Programmcode bei Auftreten eines Fehlers die vollständige Java-Anwendung zu Fall bringen. Als Teil einer Java-EE-Anwendung könnte also der gesamte Server betroffen sein. Man muss abwägen, ob man das akzeptieren kann. Wir haben es für unsere nicht lebenswichtige Anwendung getan. Zum anderen kann man eine Native Library nur einmal laden. Bei einem Redeployment des Adapters versucht *RXTX* dies jedoch erneut zu tun, was dann fehlschlägt.

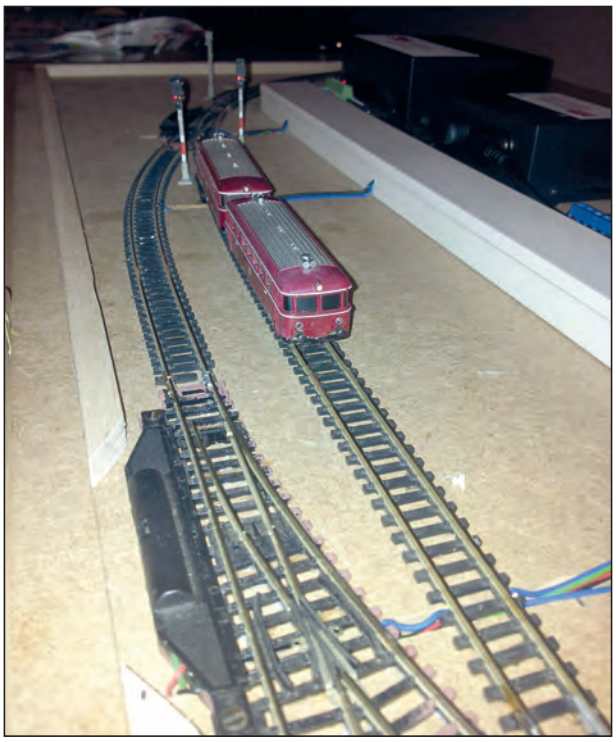


Abb. 3: Demonstrationsanlage

Abhilfe schafft die Auslagerung von *RXTX* aus der Anwendung heraus in den Server. JBoss basiert seit der Version 7 auf einem Modulsystem, das es erlaubt, Bibliotheken wie *RXTX* als Modul zur Verfügung zu stellen. Dazu muss die Bibliothek – hier *rxtx-2.2-20081207.jar* – mit einem proprietären Modul-Descriptor *module.xml* in einem Verzeichnis unterhalb von *jboss-as-7.1.x.Final/modules* abgelegt werden (**Abb. 4**). Der Descriptor gibt dem neuen Modul einen Namen, führt die enthaltenen Bibliotheken auf und könnte darüber hinaus noch Abhängigkeiten zu weiteren Modulen deklarieren. In unserem recht einfachen Fall sieht er so aus:

```
<module xmlns="urn:jboss:module:1.1" name="org.rxtx.rxtx">
  <resources>
    <resource-root path="rxtx-2.2-20081207.jar"/>
  </resources>
</module>
```

Der Ressourcenadapter kann dann das neue Modul mithilfe des proprietären Deployment Descriptors *jboss-deployment-structure.xml* (in *META-INF*) referenzieren:

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.rxtx.rxtx" />
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

JBoss lädt das Modul dann bei Bedarf einmalig. Ein Redeployment des Adapters ist nun unproblematisch.

Betriebssteuerung

Dieser Teil enthält die eigentliche Fachlogik des Systems: Eine umfangreiche baumartige Objektstruktur (**Abb. 5**) repräsentiert die steuer- und überwachbaren Elemente der Modellbahn (Loks, Weichen, Signale, Gleisabschnitte) sowie die dazu nötigen technischen Bausteine der Mehrzugsteuerung (Zentrale, Funktionsdecoder, Besetztmelder). Dieser Objektbaum wird aus einer XML-basierten Konfigurationsdatei erstellt und mithilfe des Ressourcenadapters ständig mit dem Zustand der Anlage synchronisiert, und zwar *inbound* durch entsprechende Meldungen des Adapters sowie *outbound* durch Observer, die Statusänderungen der Objekte als Events verarbeiten.

Zur Verankerung dieser Statusverwaltung bietet CDI sehr elegante Möglichkeiten an: Die Bereitstellung des beschriebenen Objekts übernimmt ein Producer, der – mittels *@ApplicationScoped* als Singleton ausgeprägt – für die einmalige Initialisierung des Objekts in seiner *@PostConstruct*-Lifecycle-Methode JAXB verwendet (Listing 1).

Die Inbound Messages des Adapters werden durch eine Message-driven Bean verarbeitet:

```
@MessageDriven(messageListenerInterface = SelectrixMessageListener.class)
public class PropagateSelectrixMessageMdb implements
                                              SelectrixMessageListener
{
  @Inject
  Steuerung       steuerung;
  public void onMessage(SelectrixMessage message)
  {
    this.steuerung.onMessage(message);
```

Für die umgekehrte Richtung feuern die Objekte CDI Events, die dann durch eine Observer-Methode an den Adapter weitergeleitet werden (Listing 2).

Diese lose Kopplung ohne explizite Registrierung des Observers an der Event Source macht es möglich, dass das Statusobjekt auch außerhalb des Servers in der Clientanwendung genutzt werden kann.

Serviceangebot für Clients

Die zentrale Geschäftslogik von *V5T11* wird von unterschiedlichsten Clients genutzt: Das Stellwerk greift als Remote-Client von Ferne zu, eine Android-App nutzt REST-Services zur Steuerung von Fahrzeugen, und eine Webanwendung auf Basis von JSF dient dem administrativen Zugriff auf die Steuerung sowie dem rudimentären Monitoring. Um dies möglichst redundanzfrei zu ermöglichen, ist die Betriebssteuerung durch einige Boundaries umgeben, die als Kontaktpunkt für die erwähnten Zugänge dienen (**Abb. 6**).

Als Beispiel wird so die Funktionalität „Weiche stellen“ als EJB für den Remote-Zugriff bereitgestellt:

```
@Stateless
public class SteuerungRemoteServiceBean implements
                                              SteuerungRemoteService
{
```

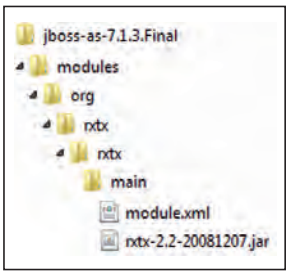


Abb. 4: Ablage eines Moduls in JBoss 7.1.x

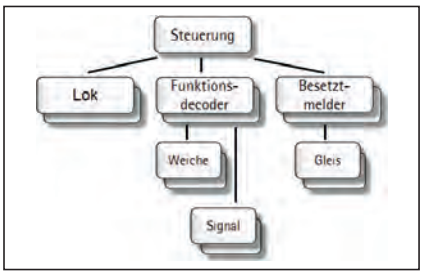


Abb. 5: Steuerungsobjekt als Repräsentation des Systemzustands

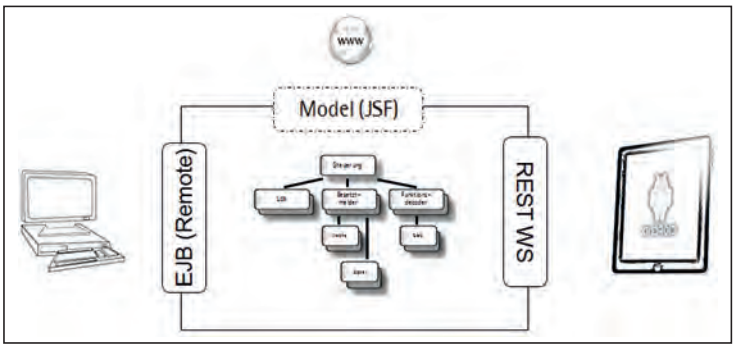


Abb. 6: Boundaries für gemeinsame Geschäftslogik

Listing 1

```
@ApplicationScoped
public class SteuerungProducer
{
  @Produces @Dependent
  private Steuerung steuerung;
  @PostConstruct
  private void init()
  {
    this.steuerung = XmlConverter.fromXml(Steuerung.class, ...);
    ...
  }
```

Listing 2

```
public class Steuerung
{
  ...
  public void setWert(int adresse, int wert)
  {
    ...
    beanManager.fireEvent(new SelectrixMessage(adresse, wert),
                          new AnnotationLiteral<Outbound>(){});
  }
}

@ApplicationScoped
public class SelectrixGateway
{
  public void setValue(@Observes @Outbound SelectrixMessage selectrixMessage)
  {
```

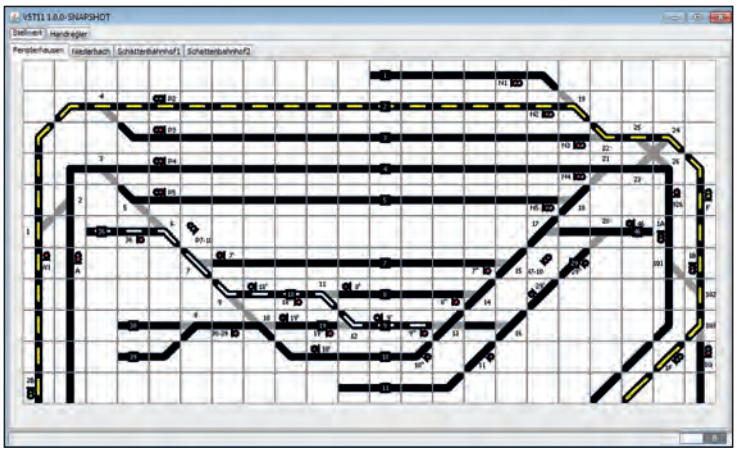


Abb. 7: Stellwerk

Abb. 8: Android-basierte Remote Control

```
@Inject
Steuerung steuerung;
@Override
public void setWeicheStellung(String bereich,
                              String name, Stellung stellung)
{
  Weiche weiche = this.steuerung.getWeiche(bereich,
                                           name);
  weiche.setStellung(stellung);
```

Die gleiche Funktionalität als REST-Web-Service zeigt Listing 3.

Analog lässt sich auch ein Presentation Model für eine JSF-basierte Webanwendung bereitstellen. Durch die saubere Schichtung der Anwendung ist es nun also problemlos möglich, verschiedene Clients bzw. Präsentationen auf die gleiche Geschäftslogik zu setzen.

Stellwerk

Die Stellwerksanwendung wurde als Remote Client auf Basis von Swing entwickelt. Sie verwendet das gleiche Statusobjekt, das schon in der serverseitigen Betriebssteuerung Anwendung fand, um den Zustand der Anlage zu visualisieren und interaktiv Änderungen zu initiieren: Weichen und Signale stellen, Fahrstraßen reservieren bzw. freigeben (**Abb. 7**).

Die Statusaktualisierung vom Server zum Client geschieht mittels JMS. Dazu werden Änderungen auf der Serverseite in Form von *SelectrixMessage*-Objekten in eine Queue eingetragen, die vom Client asynchron verarbeitet wird.

Remote Control

Zur Steuerung der Fahrzeuge verwenden Mehrzugsteuerungen so genannte Remote Controls, mit denen sich mehr oder weniger bequem Fahrzeuge auswählen und in ihrer Geschwindigkeit, Richtung und Beleuchtung steuern lassen. Ältere Geräte sind schnurgebunden, was den Bewegungsradius des Modellbahners etwas einschränkt. *V5T11* ermöglicht die Nutzung von Android-Geräten für diesen Zweck. Die entsprechende Anwendung ist allerdings bislang noch sehr rudimentär ausgebildet, was dem verfügbaren Zeitrahmen unseres Code Camps geschuldet ist. Sie nutzt die oben erwähnten REST-basierten Services zur Steuerung der Loks (**Abb. 8**).

Zusammenfassung

V5T11 – und damit auch dieser Artikel – hat nicht das Ziel der Erstellung einer vollständigen und umfassenden Modellbahnsteuerung. Vielmehr wollten wir zeigen, dass die sich in einem so maschinennahen Arbeitsbereich stellenden Anforderungen mit Standardmitteln der Plattform Java EE gelöst werden können. Das ist insbesondere durch die Nutzung von CDI als zentralem Dreh- und Angelpunkt für die Geschäftslogik recht elegant gelungen. Dieser Artikel konnte nur Schlaglichter auf die beschriebene Anwendung werfen. Wenn Sie einen tieferen Blick in den Code werfen wollen, können Sie das auf GitHub tun [2].

Listing 3

```
@Path("steuerung/weiche")
public class WeicheWebservice
{
  @Inject
  private Steuerung steuerung;
  @Path("{bereich}/{name}/stellung")
  @POST
  public Response setStellung(@PathParam("bereich") String bereich,
                              @PathParam("name") String name,
                              @FormParam("stellung") Stellung stellung)
  {
    Weiche weiche = this.steuerung.getWeiche(bereich, name);
    weiche.setStellung(stellung);
```

Dirk Weil ist seit 1998 als Berater im Bereich Java tätig. Als Geschäftsführer der GEDOPLAN GmbH in Bielefeld [3] ist er für die Konzeption und Realisierung von Informationssystemen auf Basis von Java EE verantwortlich. Seine langjährige Erfahrung in der Entwicklung anspruchsvoller Unternehmenslösungen macht ihn zu einem kompetenten Ansprechpartner und anerkannten Experten auf dem Gebiet Java EE. Er ist Autor des Fachbuchs „Java EE 6 – Enterprise-Anwendungsentwicklung leicht gemacht“ [4], schreibt Fachartikel, hält Vorträge und leitet Seminare und Workshops aus einem eigenen Java-Curriculum.

Links

[1] http://rxtx.qbang.org

[2] https://github.com/dirkweil/v5t11

[3] http://www.gedoplan.de

[4] http://entwickler.de/press/Java-EE-6

Pragmatisches Integrationstesten

Orchesterprobe

Das Leben eines Entwicklers könnte so einfach sein, wenn mit Unit Tests zur Verifikation der Geschäftslogik das Testen schon erledigt wäre. Doch wenn sichergestellt werden soll, dass beispielsweise eine Anwendung ohne Fehler deployt werden kann, dass Datenbankabfragen zu den gewünschten Ergebnissen führen oder die Interprozesskommunikation über RESTful Services funktioniert, dann sind Integrationstests angesagt. Diesem mannigfaltigen Thema widmet sich der vorliegende dritte und letzte Teil dieser Serie.

von Kai Spichale

Im ersten Teil zum Thema Testwissen für Entwickler wurden Best Practices für Unit Tests vorgestellt. Eine wichtige Empfehlung war, die Funktionsweise der Software mit den vorgestellten Assertions-Frameworks in vielen einfachen, feingranularen und schnellen Tests zu überprüfen. Der zweite Teil konzentrierte sich auf das Testen von Objekten mit Abhängigkeiten zu anderen Objekten, die deren Testen erschweren. Zur Isolation der getesteten Objekte und zur Überprüfung von Objektinteraktionen wurden verschiedene Typen von Testdoubles vorgestellt. Im Mittelpunkt des dritten Teils stehen Integrationstests und ihre Automatisierung.

Integrationstests überprüfen Komponenten in Kombination miteinander. Diese Tests können sich auf die interne Funktionsweise eines Systems beziehen oder auf mehrere Systeme, die zusammen eine Aufgabe erfüllen. Angenommen, wir hätten Code zur Serialisierung geschrieben und ihn mit Unit Tests getestet, ohne das Dateisystem zu benutzen. Wie wollen wir dann wissen, ob das Laden und Speichern der Daten auf Festplatte tatsächlich funktioniert? Vielleicht haben wir ein *flush* oder ein *close* vergessen. End-to-End-Tests überprüfen alle Komponenten eines Systems. Diese Tests gehen durch alle Schichten vom Browser über Webserver bis zur Datenbank inklusive sonstiger beteiligter Systeme. Akzeptanztests werden von User Stories abgeleitet und sollen sicherstellen, dass die funktionalen Benutzeranforderungen umgesetzt sind. All diesen Tests ist gemein, dass sie in der Regel langsamer und grobgranularer als Unit Tests sind. Falls ein Fehler auftritt, ist es aufgrund der vielen beteiligten Komponenten oder Systeme meist schwieriger, die Ursache zu identifizieren. Wie können wir trotzdem Integrationstests pragmatisch einsetzen, um unsere Produktivität zu erhöhen?

Im Folgenden soll es insbesondere darum gehen, wie Integrationstests mit Frameworks wie JUnit durchgeführt werden können. Es werden mit Cargo [1] und Arquillian [2] zwei Technologien zum Deployen der zu testenden Software in einen Applikationsserver vorgestellt. Integrationstests ohne Deployment, die beispielsweise mit Spring möglich sind, werden ebenfalls betrachtet.

Teile und herrsche!

Da Integrationstests im Allgemeinen langsamer und komplexer sind als Unit Tests, von denen hunderte in einer Sekunde auf einem Entwicklerrechner ausgeführt werden können, sollten Unit und Integrationstests separat ausgeführt werden. Die Integrationstests werden nach den Unit Tests gestartet, sofern diese erfolgreich waren. Durch diese Trennung kann die Feedbackschleife um wertvolle Minuten oder gar Stunden verkürzt werden. Zur Ausführung von Unit Tests nutzt Maven das bekannte Surefire-Plug-in, das in der Testphase des Maven-Build-Lifecycles standardmäßig alle Klassen, deren Namen mit „Test“ beginnen oder enden, ausführt. Für die Integrationstests eignet sich das Maven-Failsafe-Plug-in, denn zur Ausführung der Integrationstests werden in der Regel gleich vier Phasen eines Builds benötigt. In der Pre-Integration-Testphase wird die Testumgebung vorbereitet. In der anschließenden Integration-Testphase erfolgt die eigentliche Ausführung der Tests durch das Failsafe-Plug-in. Die

Artikelserie

Teil 1: Eigenschaften von TDD und BDD, Best Practices für Unit Tests
Teil 2: Teststile
Teil 3: Integrationstests: Test- und Build-Automatisierung

Listing 1

```
<properties>
<catalina.home.dir>target/tomcat</catalina.home.dir>
  <catalina.base.dir>target/tomcat-instance</catalina.base.dir>
</properties>
...
<plugin>
<groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
<executions>
   <execution>
     <id>start-container</id>
     <phase>pre-integration-test</phase>
     <goals>
       <goal>start</goal>
     </goals>
   </execution>
   <execution>
     <id>stop-container</id>
     <phase>post-integration-test</phase>
     <goals>
       <goal>stop</goal>
     </goals>
   </execution>
   </executions>
   <configuration>
     <wait>false</wait>
     <container>
       <containerId>tomcat7x</containerId>
       <home>${catalina.home.dir}</home>
     </container>
     <configuration>
       <type>standalone</type>
       <home>${catalina.base.dir}</home>
     </configuration>
     <deployables> // deployable artifacts
     </deployables>
   </configuration>
 </plugin>
```

Listing 2

```
@RunWith(Arquillian.class)
public class VehicleFleetIT {

@EJB VehicleFleet vehicleService;

@Deployment public static JavaArchive createDeployment() {
   return ShrinkWrap.create(JavaArchive.class)
       .addClass(VehicleFleet.class).addClass(Vehicle.class)
       .addAsResource("META-INF/persistence.xml");
}

@Test public void createNewVehicle() {
   Vehicle vehicle = createTestVehicle();
   vehicleService.add(vehicle);
   List<Vehicle> availableVehicles = vehicleService.getAvailableVehicles();
   assertThat(availableVehicles).containsOnly(vehicle);
}
}
```

Post-Integration-Testphase dient dazu, die Testumgebung zu stoppen. In der Verify-Phase werden schließlich die Testergebnisse des Failsafe-Plug-ins überprüft. Falls ein Testfehler auftrat, wird erst in dieser Phase der Build abgebrochen, sodass in jedem Fall die Testumgebung sauber beendet werden kann. Das Failsafe-Plug-in erkennt die Integrationstests standardmäßig an den Namenszusätzen „IT“ und „ITCase“.

In einem Multi-Module-Projekt führt der Maven-Reactor den kompletten Build-Lifecycle für jedes Modul in der Reihenfolge des spezifizierten Build-Plans aus. Das bedeutet, dass nicht erst alle Module die Testphase durchlaufen, bevor die Package-Phase beginnt. Möchte man also nicht auf das Ergebnis der Integrationstests von Modul A warten, bevor die Unit Tests von Modul B beginnen, so muss man die Testausführung auf mehrere Builds verteilen. Das Build-Pipeline-Plug-in für den Build-Server Jenkins ist ein geeignetes Werkzeug für diese Aufgabe. Wenn beispielsweise der Build für die Unit Tests stabil ist, kann ein nachgelagertes (downstream) Projekt für die Integrationstests getriggert werden. Jenkins setzt jedoch nicht zwingend den Erfolg oder die Stabilität aller vorgelagerten (upstream) Projekte voraus, um ein nachgelagertes Projekt zu bauen. Die Entwickler des Build-Pipeline-Plug-ins hatten nicht nur einfache Build-Promotions im Sinn, sondern wollten mit diesem Plug-in ganze Deployment-Pipelines aufbauen und überwachen können. Diese Pipelines können für kontinuierliche Integrationen und Deployments bis in die Produktivumgebung genutzt werden.

Auf die Teststrecke

Für das Deployment der zu testenden Anwendung in einen Container wie Tomcat, GlassFish oder JBoss eignet sich das Framework Cargo. Es kann durch das Cargo-Maven-2-Plug-in in den Build-Prozess integriert werden. Wie in Listing 1 dargestellt ist, wird der ausgewählte Container in der Pre-Integration-Testphase gestartet und in der Post-Integration-Testphase gestoppt. Die angegebenen Deployables können entweder „hot“ in den laufenden Container oder „static“ beim Start des Containers deployt werden. In Listing 1 erfolgt das Deployment vor dem Start des Containers. Dass wir nicht in einen JBoss 7, sondern in Tomcat 7 deployen wollen, teilen wir dem Plug-in durch die Container-ID mit. Ein Container kann in verschiedenen Konstellationen genutzt werden. Der Konfigurationstyp *standalone* zeigt an, dass die Integrationstests gegen einen Container ausgeführt werden, den Cargo eigens für diesen Zweck erzeugt. Im gezeigten Beispiel erzeugt Cargo auf Basis der Tomcat-Installation in *${catalina.home.dir}* eine neue Tomcat-Instanz in *${catalina.base.dir}*, die nach dem Test gelöscht werden kann. *${catalina.home.dir}* und *${catalina.base.dir}* sind benutzerdefinierte Maven-Properties. Mit dem Konfigurationstyp *existing* kann ein bereits installierter und konfigurierter Container verwendet werden. Ein einfacher Container wie Jetty kann sogar *embedded* genutzt werden.

In Listing 1 greifen wir auf eine existierende Tomcat-Installation zurück, um eine neue Serverinstanz zu erzeugen. Doch wie kommt der Server dorthin? Der Build sollte portabel sein, aber ins Versionsverwaltungssystem zum Rest des Quellcodes sollte der Container auch nicht. Um dieses Problem zu lösen, könnte der Container als Maven-Dependency hinzugefügt werden. Mit dem Maven-Dependency-Plug-in kann diese Abhängigkeit im gewünschten Verzeichnis entpackt werden. Alternativ bietet Cargo die Möglichkeit, in der Container-Konfiguration einen <*zipUrlInstaller*> anzugeben.

Gleich die komplette Testumgebung in einer separaten virtuellen Maschine könnte mit Vagrant [3] erzeugt werden. Mit dem Vagrant-Jenkins-Plug-in kann eine virtuelle Maschine gestartet und provisioniert werden. Das bedeutet, dass auf der virtuellen Maschine zunächst nur ein Betriebssystem installiert ist. Auf der erzeugten VM werden dann automatisch ein Applikationsserver, eine Datenbank oder andere notwendige Programme installiert. So können auch komplexe Umgebungen erzeugt werden – und später in Produktion identisch aufgebaut werden.

Schnelles Feedback

Cargo ist ein nützliches Werkzeug zum Ansteuern eines Containers und zur Automatisierung des Deployments von Artefakten wie WAR- und EAR-Dateien. Insbesondere für End-to-End-Tests eignet sich dieser Ansatz. Es ist jedoch zu bedenken, dass eine Anwendung typischerweise aus mehreren Archiven besteht, deren Erzeugung zeitintensiv sein kann. Auch das Deployment der Anwendung kostet Zeit. Das ist vor allem dann ärgerlich, wenn nur ein kleiner Teil der Anwendung für den jeweiligen Test notwendig ist. Zusätzlich kann die mangelnde IDE-Integration die Erstellung und Wartung von Integrationstests erschweren. Aus den genannten Gründen wollen wir auch den Einsatz von Arquillian vorstellen.

Arquillian ist ein Framework für leichtgewichtiges Integrationstesten. Zur Ausführung von Integrationstest in JUnit-Tests bietet es einen entsprechenden Testrunner. Für TestNG-basierte Tests existiert eine Testbasisklasse. Ein großer Vorteil von Arquillian ist, dass nur jene Klassen in den Container deployt werden müssen, die für den Test relevant sind. Durch diesen leichtgewichtigen Ansatz kann die Feedbackschleife drastisch verkürzt werden. Das heißt, ein Entwickler benötigt weniger Zeit, um einen Integrationstest auszuführen.

Arquillian startet den Applikationsserver, deployt die zu testenden Klassen in den Container, führt die Tests aus und fährt schließlich den Applikationsserver wieder herunter. Dies erinnert stark an Cargo. Doch der entscheidende Unterschied besteht darin, dass die Abwicklung dieses Lebenszyklus nicht an eine spezifische Build-Technologie wie Maven gebunden ist. Ein Integrationstest wie in Listing 2, der auf JUnit basiert,

Im Unterschied zu Cargo ist Arquillian nicht an eine spezifische Build-Technologie gebunden.

kann deswegen auch direkt in der IDE ausgeführt werden. Die Kommunikation zwischen Arquillian und dem jeweiligen unterstützten Container erfolgt über ein containerspezifisches Plug-in, das beispielsweise als Maven-Testabhängigkeit in den Klassenpfad des Tests eingefügt wird. Durch Scannen des Klassenpfades wird der gewünschte Zielcontainer von Arquillian automatisch erkannt.

Wie Cargo kann Arquillian einen Container in drei unterschiedlichen Varianten ansteuern: In der „remote"-Variante existiert bereits ein Container, der in einer externen JVM ausgeführt wird. Der Lebenszyklus des Containers wird in diesem Fall nicht von Arquillian gesteuert. In der „managed"-Variante kümmert sich Arquillian um das Starten und Steuern des Containers. Konfigurationen für den Container werden in der *arquillian.xml* gepflegt. In der dritten Variante wird der Container „embedded" ausgeführt und vollständig von Arquillian kontrolliert.

Wie schreibt man einen Integrationstest mit Arquillian? Betrachten wir dazu das Beispiel in Listing 2. Der JUnit-Test erhält durch die Annotation *@RunWith* den Testrunner *Arquillian.class*. Die mit *@EJB* markierte Session Bean *VehicleFleet* wird in den Test injiziert. Auch CDI-Objekte, die die EJBs mit einschließen, können referenziert werden. Das für den Test zu deployende Archiv wird in der Methode *createDeployment* erzeugt. Die Methode ist aus diesem Grund mit *@Deployment* markiert. Das Java-Archiv wird mithilfe von *ShrinkWrap* programmatisch erzeugt. Neben JAR-Archiven können auch WAR-, EAR-, SAR- und RAR-Archive erzeugt werden.

Die Referenzen auf EJBs und andere CDI-Objekte werden durch *Enricher* aufgelöst. Der Enricher der *@Resource*-Annotation unterstützt den Zugriff auf alle Java-EE-Ressourcen, die via JNDI erreichbar sind. Das Injizieren von Feldern mit dem Enricher der *@EJB*-Annotation haben wir bereits in Listing 2 gezeigt. Fügt man ins Deployment-Archiv eine *beans.xml* hinzu, so können mit *@Inject* CDI-Objekte injiziert werden. Damit alle Ressourcen des Containers im Test referenziert werden können, wird standardmäßig auch der Test mit in den Container deployt. Alternativ können Tests im so genannten *Client Mode* auch außerhalb des Containers ausgeführt werden, um eine Anwendung über ihre externen Schnittstellen zu testen.

JPA-Test

Das Beispiel in Listing 2 verwendet indirekt JPA, weswegen neben der Entity Class *Vehicle* auch eine *per-*

sistence.xml in das Archiv eingefügt wird. Falls der JPA-Provider innerhalb eines Applikationsservers wie JBoss oder GlassFish verwendet wird, muss der *EntityManager* nicht selbst gemanagt werden. Zum Test des O/R Mappings und der Queries braucht man jedoch nicht zwingend einen Applikationsserver, denn mithilfe eines unmanaged *EntityManagers* kann die Persistenzschicht auch ohne Applikationsserver getestet werden. Die Transaktionsgrenzen müssen jedoch dann explizit im Test gesetzt werden, denn deklarative Transaktionen werden nicht unterstützt. Für das Testen der Persistenzschicht könnte eine angepasste *persistence.xml* für eine leichtgewichtige Datenbank mit In-Memory-Modus wie Derby, H2 oder HSQLDB verwendet werden. Die Datenbank kann als Testabhängigkeit über ein Maven-Repository ohne Installation bereitgestellt werden. Da für den Test keine JTA-Transaktionen benutzt werden können, muss für die Persistence Unit als Transaktionstyp *RESOURCE_LOCAL* angegeben werden.

> Schon eine kleine Änderung an der Serverkonfiguration kann Deployment-Fehler verursachen.

Mit einem solchen pragmatischen Testsetup ist es möglich, die Persistenzschicht isoliert vom Rest der Anwendung zu testen. Die Tests können direkt in einer IDE ausgeführt werden und notfalls kann man sie auch debuggen. Sie können jedoch die Tests auf der Produktivdatenbank nicht ersetzen, denn die Strategie zur Generierung der IDs, der Datenbankzeichensatz oder so einfache Dinge wie die Maximallänge von Bezeichnern der Datenbankobjekte könnten unterschiedlich sein.

Integrationstests mit Spring

Zum Testen einer Persistenzschicht mit managed *EntityManager* und automatischer Transaktionsverwaltung eignet sich hervorragend das Spring Framework [4]. In Teil 2 hatten wir bereits gezeigt, wie mithilfe des *SpringJUnit4ClassRunner* in einem JUnit-Test ein Spring-Applikationskontext gestartet werden kann. TestNG wird ebenfalls unterstützt. Neben der Transaktionsverwaltung ist Dependency Injection für Test-Fixtures ein großer Vorteil dieses Ansatzes. Vor Start der Tests werden die mit *@Autowired*, *@Resource* und *@Inject* annotierten Felder der Testklasse injiziert. Der Applikationskontext wird, sofern nicht anders definiert, durch den *GenericXMLContextLoader* geladen. Die dazugehörigen Spring-XML-Konfigurationsdateien werden mit der Annotation *@ContextConfiguration* angegeben. Alternativ kann der Context auch programmatisch mithilfe des *AnnotationConfigContextLoader*s definiert werden. Mithilfe von mit *@BeforeTransaction* und *@AfterTransaction* annotierten Methoden kann der Datenbankzustand vor dem Beginn und nach Abschluss einer Transaktion überprüft werden. Mit *@Rollback* kann das Festschreiben bzw. das Zurücksetzen des veränderten Datenbankzustandes gesteuert werden. Seit Spring 3.1 werden auch Profile unterstützt [5]. So können beispielsweise unterschiedliche Data Sources in speziellen Profilen für Test, Entwicklung und Produktionsbetrieb definiert werden. Die Aktivierung der Profile erfolgt durch die Annotation *@ActiveProfiles*.

Das Spring-*TestContext*-Framework ist jedoch nicht auf Repositories beschränkt und kann ebenfalls zum Test anderer Spring-Beans wie Components und Services verwendet werden. Tests in Kombination mit Spring MVC werden ebenfalls unterstützt.

Zusammenfassung

Nur durch kontinuierliches Testen außerhalb und innerhalb eines Containers kann die Korrektheit des Verhaltens einer Anwendung mit definiertem Zustand überprüft werden. Schon eine kleine Änderung an der Serverkonfiguration könnte Deployment-Fehler verursachen. Da moderne Web- und Applikationsserver mit Skripten oder über ein API angesteuert werden können, ist es möglich, diese für Tests zu konfigurieren und entsprechend Definitionen, die gemeinsam mit dem Code abgelegt werden, zu starten. Mit dem Cargo-Framework können Java-EE-Container gesteuert werden, um Anwendungen für die Durchführung von Integrationstests zu deployen. Arquillian ist eine komplementäre Technologie, die es erlaubt, direkt aus dem Test heraus ohne Abhängigkeiten zum Build-System ein Deployment-Archiv zu erzeugen und zu deployen. Integrationstests müssen aber nicht immer in einem Container ausgeführt werden, wie wir mit JPA und Spring gezeigt haben. Also gibt es viele Möglichkeiten, auch bei Integrationstests sinnvoll zu automatisieren.

Kai Spichale arbeitet als Senior Software Engineer beim IT-Dienstleistungs- und Beratungsunternehmen adesso AG. Seine Schwerpunkte sind Java EE, Spring und NoSQL.

Links & Literatur

[1] http://cargo.codehaus.org

[2] http://arquillian.org/

[3] http://www.vagrantup.com/

[4] http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/testing.html

[5] http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/new-in-3.1.html

Was es in Sachen Google-Produkte Neues zu berichten gibt

Google I/O 2013

Zur Google I/O zu gehen, heißt erst einmal Schlange stehen! Es fängt beim Ergattern der Tickets an, dann, um in das Gebäude zu kommen, an der Rolltreppe, für das Essen, für den Kaffee und schließlich um einen Platz im Saal für die Keynote, die Ähnlichkeit mit einem Popkonzert hat. Die Menge johlt und wartet auf den Star. In diesem Fall sind die Stars aber weniger die Menschen, sondern eher die zu erwartenden Neuerungen und Meldungen, die hier alle aus erster Hand erhalten wollen.

von Oliver Zeigermann

6 000 Leute sind vor Ort, 40 000 nehmen über spezielle Google-I/O-Extended-Events in der ganzen Welt teil und 1 000 000 folgen zumindest der Keynote über YouTube. Obwohl in der Keynote nichts über Google Glass berichtet wurde, machte der Begriff „Glasshole" – als abfälliger Name für jemanden, der ein „Google Glass" trägt – hier die Runde. Amüsant. Anders als in den Jahren zuvor gab es eine einzige lange Keynote, die im Vergleich zu früheren I/Os wenig technisch, sondern eher eine Marketingveranstaltung für neue Produktfeatures war. Auch im Bereich der Sessions gab es weniger generelle technische Themen und mehr über Googles Produkte.

Android

Android als das zurzeit erfolgreichste mobile Betriebssystem hat bis heute 900 000 000 Aktivierungen. Das sind 500 000 000 neue Aktivierungen seit der letzten Google I/O 2012 (400 000 000 Aktivierungen) und 800 000 000 neue Aktivierungen seit 2011 (100 000 000 Aktivierungen).

Die spannendste technische Neuerung ist das neue Android Studio (siehe auch den Artikel von Dominik Helleberg auf Seite 108). Es basiert auf der freien Community Edition von IntelliJ und macht einen sehr guten Eindruck. Google wird zwar das Eclipse-Plug-in weiterhin unterstützen, vertraut für eine bessere Integration aber eher auf IntelliJ. Ein Grund für den Umstieg war, dass der Standard-Build-Prozess nun auf Gradle aufsetzt, das eine gute IntelliJ-Anbindung bietet. Damit beruhen Development und Integration Builds auf derselben Technik.

Das eindrucksvollste Feature des Android Studios ist ein Echtzeitrendering der Layouts auf unterschiedlichen Device-Größen und mit unterschiedlichen Sprachen – auf Wunsch sogar mehrere gleichzeitig. So kann man sein Layout sehr schnell testen und kommt mit einem Klick von den gerenderten Layouts an die entsprechenden Codestellen. Dies funktioniert sogar in beide Richtungen und in Echtzeit, ohne dass ein Simulator gestartet werden müsste. Ebenso ist das Tooling für Ressourcen sehr eindrucksvoll.

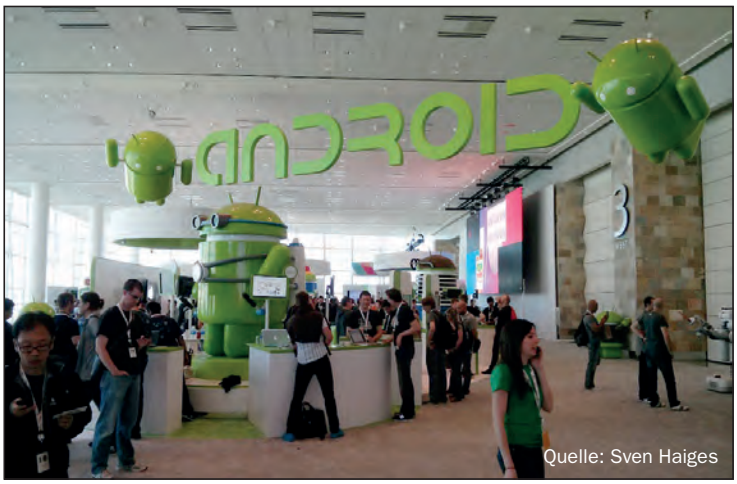
Für das Ausrollen der Apps über den Play Store kann man ein Beta-Testing über Staging realisieren. Dabei wird über Google+-Gruppen und Anteile in Prozenten angegeben, wer das neue Release als Erster bekommt, und kann über Google+ sofort privates Feedback erhalten.

AngularJS

AngularJS ist in diesem Jahr mit einer Session und sogar mit einem speziellen Event vor dem offiziellen Beginn der Google I/O vertreten. Bezüglich der verschiedenen Frameworks zum Erstellen von Webapplikationen wird es durch AngularJS etwas unübersichtlich bei Google, da diese Technik in Konkurrenz zu Dart und GWT steht. Ideen über Integrationen der Techniken existieren, aber eine Orientierung wird dadurch nicht einfacher.

AngularJS ist ein MVC-Framework, das komplett im Browser abläuft. Die drei Hauptpunkte bei AngularJS sind die drei Ds: Data Binding, Dependency Injection

Quelle: Sven Haiges



Quelle: Sven Haiges

und Directives. Durch ein Data Binding in zwei Richtungen können sehr leicht Daten in ein Modell und wieder zurück abgebildet werden, ohne Code zu schreiben. Für echte Logik braucht man natürlich immer noch Code, in diesem Fall kann man in Controllern eine einfache Form der Dependency Injection nutzen. Vielleicht die wichtigste Technik von AngularJS sind Directives. Sie erfüllen einen ähnlichen Zweck wie Custom Elements in Web Components, nur gibt es sie jetzt schon in einer einfachen Form.

Web Components

Spannende Neuerungen gibt es im Umfeld von HTML5. Mit den so genannten Web Components kommt ein Satz unterschiedlicher Technologien. Zuerst einmal gibt es Templates, die im Wesentlichen HTML-Fragmente sind. Sie werden bereits in eine DOM-Struktur geparst, aber noch nicht dargestellt. Zur Darstellung werden sie über ihre ID herausgesucht, instantiiert, eventuelle Lücken werden gefüllt und schließlich wird die Templateinstanz dargestellt.

Das Shadow DOM ist eine ausformulierte Darstellung von komplexeren DOM-Knoten. So kann z. B. ein Input-Tag je nach Typ aus unterschiedlichen *div*-Elementen aufgebaut werden. Der Chrome-Browser tut genau dies, und wenn man das in den Developer-Tools aktiviert, kann man sich genau dieses Shadow DOM ansehen. Anwendungsentwickler werden nun dieselben Möglichkeiten wie die Browserhersteller bekommen und können ebenfalls solche HTML-Komponenten bauen. Dabei befindet man sich innerhalb eines Blocks und kann so z. B. auch lokale CSS Styles anwenden.

Über Custom Elements kann man nun genau solche Komponenten bauen. Die Definition erfolgt entweder über ein spezielles *element*-Tag in HTML oder über ein JavaScript-API. Bei der Definition gibt es einen Templateteil, der das Aussehen der Komponente bestimmt, und einen JavaScript-Teil, der z. B. angibt, was beim Laden des Elements passieren soll. Custom Elements können entweder über ihr Markup oder wie alle anderen DOM-Elemente über *document.createElement* erzeugt werden. Bei der Definition kann man auf bereits bestehende Elemente zurückgreifen und z. B. von einem Button oder einem Input-Feld erben.

Über das *link*-Tag mit dem Attribut *rel=" import"* wird es möglich sein, HTML-Blöcke in andere HTML-Dokumente einzubinden. Das Eingebundene kann einfaches HTML sein, allerdings auch Templates oder Custom Components.

Vieles davon läuft bereits in Chrome und Firefox, HTML-Imports sind jedoch bisher noch in keinem Browser umgesetzt. Ob und wann die anderen Browser nachziehen, ist unklar.

Dart

Lars Bak, der ursprüngliche Entwickler von Googles V8 JavaScript Engine, entwickelt nun für Google die Programmiersprache Dart und die dazugehörige VM. Schon im letzten Jahr war er mit seinen Kollegen bei der Google I/O, der Lärm um Dart ist in diesem Jahr aber ein bisschen abgeebbt und nun eher bei AngularJS zu finden.

Das Dart-Team führt zwei grundlegende Probleme an, die mit der neuen Sprache gelöst werden sollen: Zum einen ist JavaScript zu kaputt, um es noch reparieren zu können, zum anderen kann JavaScript nicht so gut optimiert werden wie eine Sprache mit deklarierten, statischen Typen.

Dart als neue Sprache mit optionaler Typinformation soll beides besser machen. Dabei wird die Ausführung in zwei Phasen geteilt: Deklaration von Typen und dann


Quelle: Sven Haiges


Quelle: Sven Haiges

Ausführungen. Sobald ein Programm ausgeführt wird, kann ein Typ nicht mehr verändert werden. Dies gibt momentan einen Performanzvorteil von Faktor 2 gegenüber JavaScript-Code, der in V8 ausgeführt wird. Dies allerdings nur, wenn der Code nativ ausgeführt wird. Dies ist zurzeit in der Chromium-Variante Dartium möglich. Und die Dart-Entwickler gehen davon aus, dass mehr Performanz zu spannenden neuen Programmen führen wird. Vielleicht im Spielebereich? Dart kann aber auch nach JavaScript übersetzt werden und läuft dann etwas langsamer als von Hand geschriebener Code in jedem Browser.

Fraglich ist, ob die Trennung und Kompilierung mit typischen JavaScript-Bibliotheken funktionieren wird und ob die gewonnene Performanz nicht auch durch Kompilierung in Formate wie asm.js möglich wäre. Spannend ist auch, ob jemals ein anderer Browserhersteller Dart implementieren wird.

GWT

Anders als AngularJS und Dart hat GWT in diesem Jahr keinen eigenen Stand und auch keine Office Hours, um mit den Entwicklern Kontakt aufnehmen zu können. Es gab aber immerhin zwei Sessions zu GWT, eine davon behandelte dessen Zukunft.

GWT wurde bekanntlich vor einem Jahr von Google in die Hände eines Steuerungskomitees übergeben. Seitdem hat sich der Code nach GitHub bewegt und die 2.5.1-Version, die in erster Linie einige Bugs behoben hatte, wurde releast. Nun steht die Roadmap für die Version 2.6 bzw. 3.0. Hier soll es bessere Geschwindigkeit und bessere Integration mit JavaScript geben und die wichtigsten Bugs sollten geschlossen werden. Java 7 wird unterstützt werden, und es gibt dafür sogar schon einen Patch. Sobald Java 8 fertig ist, soll auch dies unterstützt werden.

Mit der Version 3.0 wird die Unterstützung für alte IE-Versionen aufgekündigt. GWT 2.6 wird Ende 2013 erscheinen, zur Google I/O 2014 soll GWT 3.0 fertig sein. Die neue Entwicklerseite ist nun unter http://www.gwtproject.org/ zu finden.

Die vielleicht wichtigste Neuerung bei GWT könnte der Einstieg des mgwt-Entwicklers Daniel Kurka in das GWT-Team sein. Hier dürfen wir auf Neuerungen im mobilen Bereich hoffen. Dabei wird spannend sein, ob es die Entwickler schaffen werden, mit der Geschwindigkeit mitzuhalten, mit der neue Funktionen in aktuelle Browser auf mobilen Geräten Einzug halten.

Oliver Zeigermann ist Softwareentwicklungsenthusiast, der sich besonders für Single Page Applications (SPAs) in Java und JavaScript interessiert. Er lebt in Hamburg und arbeitet als selbstständiger Consultant, Trainer, Coach und Softwareentwickler.

The new Guy in town

Android Studio

Das war schon eine Überraschung. Als ich während der Google I/O Keynote auf die Ankündigung einer neuen Android-Version (vergeblich) wartete, kam überraschend Tor Norbye vom Android-Tools-Team auf die Bühne, um „die neue Android IDE: Android Studio“ vorzustellen. Android Studio? IntelliJ? Gradle Build System? Und was wird aus Ant und ADT? Wir versuchen hier, die bisher bekannten Infos zu sichten und einen ersten Blick auf „den Neuen“ zu werfen.

von Dominik Helleberg

Doch zunächst zu den Fakten: Android Studio wurde auf der Google I/O 2013 vorgestellt und steht aktuell in der Version 0.1 „I/O Preview AI-130.677228“ zum freien Download [1] bereit. Daran ist schon zu erkennen, dass es sich um eine sehr frühe Version der IDE handelt und das Tool dementsprechend noch einige „unrunde Ecken“ mit sich bringt.

„Hello Android Studio“

Nach der Installation von Android Studio präsentiert sich die IDE wie in **Abbildung 1** zu sehen. Spätestens hier wird deutlich, dass Android Studio nicht auf Eclipse basiert, sondern auf IntelliJ aus dem Hause JetBrains (siehe dazu auch das Interview mit Dmitry Jemerov von JetBrains auf Seite 113). Das Konzept ähnelt dem bekannten ADT-Bundle, denn auch hier wird eine vorkonfigurierte IDE heruntergeladen und mit allen benötigten Plug-ins und Einstellungen inklusive Android SDK geliefert. Dies vereinfacht die Installation und benötigt keinerlei zusätzliche Konfiguration, um mit der Entwicklung zu beginnen.

Im Detail haben wir es offensichtlich mit einer angepassten IntelliJ-Variante auf Basis der neuen Version 13 zu tun. Die Version wurde um verschiedene Plug-ins erweitert, um Android-spezifische Features umzusetzen. Mehrfach wurde während der Google I/O betont, dass es sich hier nicht um einen Branch der IntelliJ IDE handelt, sondern lediglich Hooks und Plug-ins verwendet

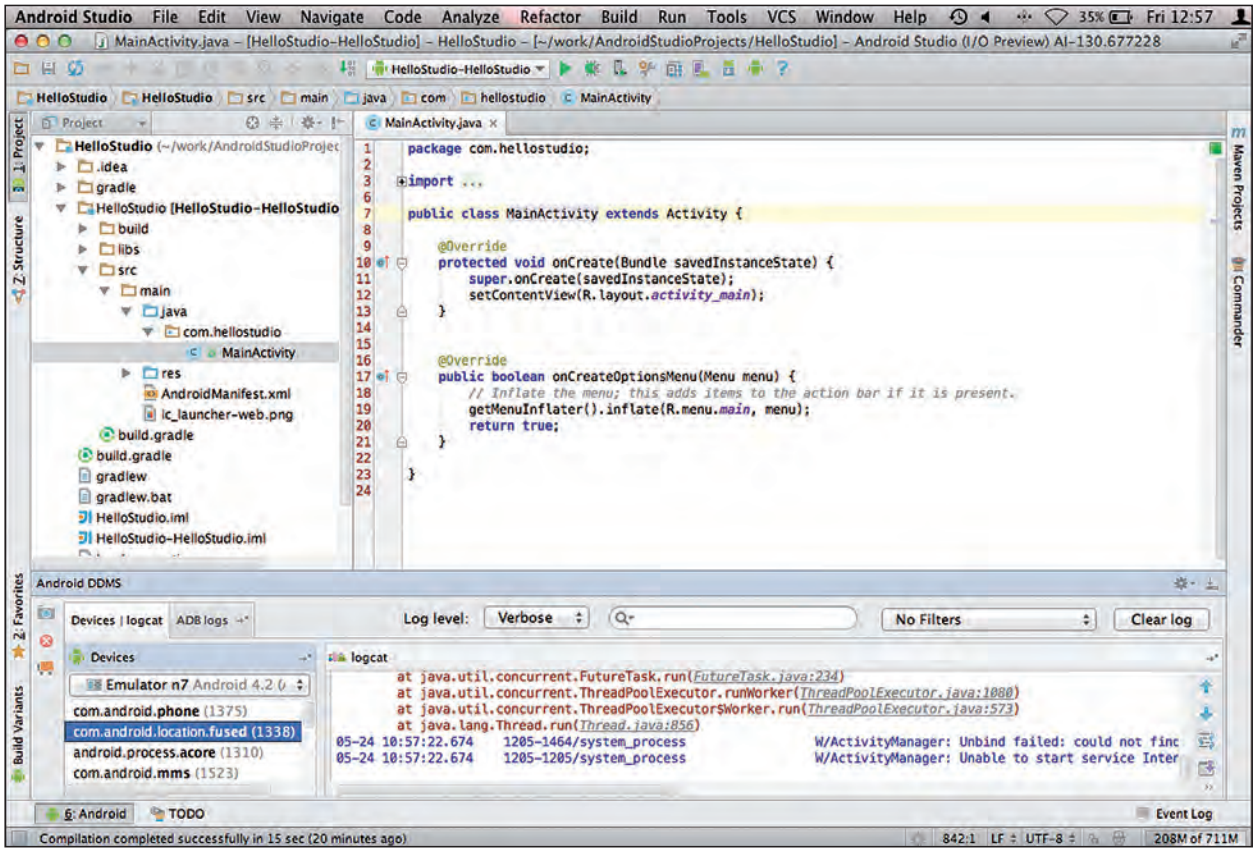


Abb. 1: Android Studio

wurden, um die Android-Funktionen hinzuzufügen. Das Prinzip ähnelt auch hier dem des ADT-Plug-ins für Eclipse. So gibt es zum Beispiel einen Designer für die Android-Layoutdateien, der dem bekannten Designer unter Eclipse sehr ähnelt. In Sachen Java-Sourcecode gibt es vergleichbare Features wie Lint-Integration und Android-spezifisches Code-Refactoring, z. B. „extract String resource“. Generell erscheinen die beiden Plug-ins (Eclipse und Android Studio) sehr ähnlich, vom „new Android Project Wizard“ über den „Layout Editor“ oder dem „Launch Editor“-Dialog gibt es keine größeren Unterschiede im GUI und den gewählten Icons.

Der Funktionsumfang der zugrunde liegenden IntelliJ-Version entspricht dem der freien Community Edition [2], so sind Features wie z. B. Versionskontrolle oder Codeanalyse enthalten, andere wie z. B. Java-EE-Support fehlen.

Auf den (wirklich) allerersten Blick finden sich gar nicht so große Unterschiede zwischen dem Android Studio und einem mit ADT-Plug-ins erweiterten Eclipse. Aber das täuscht, wie in den folgenden Abschnitten beschrieben wird.

Projektstruktur und Build-System

Legt man mit Android Studio ein neues Projekt an, fällt sofort die geänderte Verzeichnisstruktur auf. Ein Beispiel findet man im Kasten „Mehr Verzeichnisse -> mehr Struktur?“. Zusätzlich finden sich diverse neue Dateien, die man aus den bisherigen Projekten nicht kannte. So

Mehr Verzeichnisse -> mehr Struktur?

Der Umstieg auf das neue Gradle-basierte Build-System bringt u. a. neue Konventionen für Verzeichnisstrukturen mit sich, die wir anhand von einem einfachen Beispiel beschreiben. Legt man ein neues Projekt über den Wizard im Android Studio an, werden folgende Strukturen generiert (nicht vollständig):

```
.gradle/
.idea/
gradle/
HelloWorldProjekt/
    build/
    libs/
    src/
      main/
        java/
        res/
```

Das neue Layout ist deutlich tiefer geschachtelt und berücksichtigt dabei zum einen ein Multi-Projekt-Layout (neben dem HelloWorld-Projekt können andere Projekte wie z. B. Library-Projekte angelegt werden) sowie mehrere Source- und Ressourcenverzeichnisse. Unterhalb von *src* können parallel zu *main* quasi beliebig viele weitere Ordner wie z. B. *beta* oder *paid* angelegt werden. Diese enthalten dann wieder Java- und Ressourcenordner, um spezielle Varianten der Applikation bauen zu können.

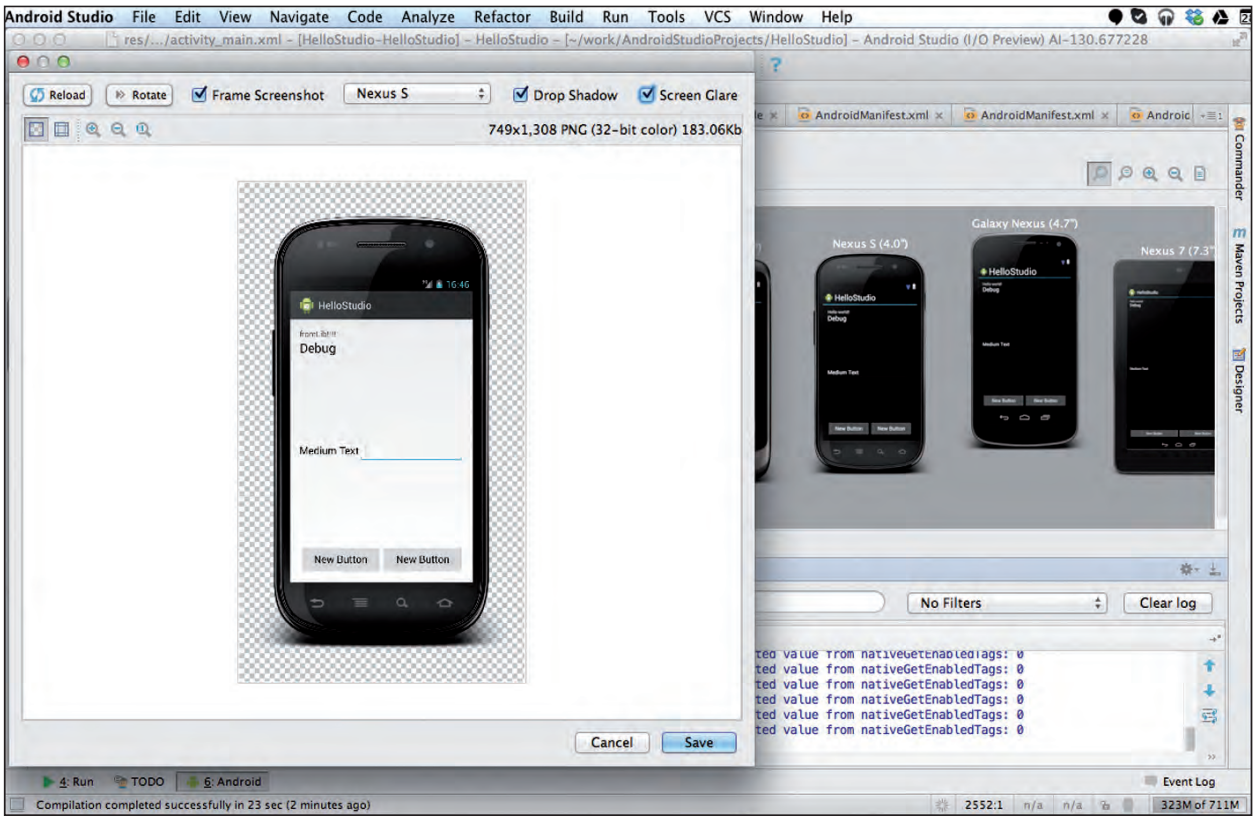


Abb. 2: Screenshots/Multi-Konfiguration-Preview

findet man einige *.iml*- sowie diverse *.gradle*-Dateien. Die *.iml*-Dateien sind IntelliJ-spezifisch, die *.gradle*-Dateien zeugen von einer weiteren großen Änderung: dem Umstieg auf das neue Android-Build-System. Das neue System ist fester Bestandteil von Android Studio, oder um es klar zu formulieren: Ein Umstieg auf Android Studio erfordert auch den Einsatz des neuen Android-Build-Systems basierend auf Gradle. Dadurch wird die Umstellung größer als auf den ersten Blick angenommen, schließlich migriert man nicht nur seine im täglichen Einsatz befindliche IDE, sondern auch die darunter liegende Infrastruktur.

Das neue Build-System verspricht, viele Schmerzen der bisherigen Lösung zu beheben, und eine detaillierte Beschreibung würde einen eigenen Artikel erfordern. Wichtig ist zu verstehen, dass man sich beim Wechsel auf Android Studio auch auf den Weg der Gradle-Integration begibt, und auch, wenn dies mit vielen schönen Features versehen ist (siehe Kasten: „Gradle Wrapper"), arbeitet man hier mit einer Version 0.4, die noch einige Probleme mit sich bringt (siehe Absatz: „Features und Bugs").

Einer der vielen Gründe für ein neues Build-System sind die immer wieder auftretenden Probleme mit dem ADT-Plug-in und dem Ant-Build-System. Im Prinzip handelt es sich hier um zwei getrennte Build-Systeme, und somit sind die daraus entstehenden Artefakte nicht zwingend identisch. So kann es z. B. regelmäßig zu Problemen auf dem CI-Server mit dem Ant-Build-System kommen, die unter Eclipse mit ADT nicht auftraten. Android Studio soll nun einzig und allein Gradle verwenden und somit auf ein zentrales Build-System bauen. Um dies zu realisieren, ist eine tiefe Integration zwischen der IDE und Gradle notwendig. Erste Ansätze davon sind bereits zu erkennen, so zeigt das Android Studio zum Beispiel die verschiedenen Module und Build Types an, die in den *build.gradle*-Dateien konfiguriert werden. Wird das Build- oder Run-Kommando ausgeführt, startet Android Studio im Hintergrund einen Gradle Build. Das hat den Vorteil, dass die Artefakte exakt identisch

Gradle Wrapper

Der Gradle Wrapper ist ein sehr praktisches Konzept, um Schmerzen rund um die Konfiguration und Versionierung des eigentlichen Build-Systems zu minimieren. Für reproduzierbare Builds ist es wichtig, die Umgebung, die die Artefakte generiert, ebenfalls zu versionieren und einfach einzurichten. Der Gradle Wrapper sorgt dafür, dass auf der lokalen Maschine das richtige Gradle vorhanden ist und einfach installiert werden kann. Dazu wird ein Verzeichnis *gradle/* sowie die Datei *gradlew* bzw. *gradlew.bat* angelegt. Diese sorgen dafür, dass auf einer Maschine mit installiertem Java die notwendige Gradle-Version inkl. Plug-ins initialisiert wird und sich somit der Installationsaufwand verringert.
Für Android Builds muss zurzeit leider noch das Android SDK manuell installiert werden. Details zum Gradle Wrapper findet man auch in der offiziellen Gradle-Dokumentationsseite [3].

sein sollen, unabhängig davon, ob mit oder ohne IDE gebaut wurde. Allerdings ist hier noch einiges zu verbessern (verständlich bei einer Version 0.1 der IDE), so verwendet die IDE zurzeit noch nicht die in der *build.gradle*-Datei definierten Abhängigkeiten zu lokalen JARs, diese müssen nach wie vor manuell in der IDE nachgezogen werden, aber das grundlegende Prinzip ist vielversprechend.

Features und Bugs

Was bietet Android Studio im Verglich zu Eclipse und ADT 22? Wie von einer Version 0.1 nicht anders zu erwarten, fehlt es noch an einigen Features. Die wichtigsten Basisfunktionen sind allerdings vorhanden. Das Editieren von Java- und Ressourcendateien funktioniert gut und ohne Probleme, die Generierung der R-Datei ist ebenso vorhanden wie die Validierung der XML-Dateien und die Verknüpfung zwischen den Ressourcen und der Java-Welt. Hier geht Android Studio noch einen Schritt weiter als die ADT-Integration: Werden Strings aus Ressourcen in der Java-Welt verwendet, versucht Android Studio, diese Strings direkt im Java-Code anzuzeigen. Ein schöner Effekt, von dem sich allerdings erst zeigen muss, wie gut er in der Praxis wirklich hilft. Generell hat man offensichtlich bei der Implementierung der Features etwas mehr Wert auf Optik und Usability gelegt. So zeigt z. B. die Multi-Konfiguration-Preview im Android Studio auf Wunsch auch schon die jeweiligen Geräte (z. B. ein Nexus 4) an, dasselbe gilt für die Screenshot-Funktion (**Abb. 2**).

Der Layout Editor wirkt für Grid-Layouts aufgeräumter, bietet aber nicht so umfangreiche Refactoring-Funktionen wie die ADT-Variante. So funktioniert z. B. das Umbenennen von View-IDs innerhalb von Ressour-

Migration bestehender Projekte

Wer sein existierendes Projekt mit Android Studio bearbeiten möchte, kann die neue *export*-Funktion in Eclipse verwenden. Diese versucht auf Basis des aktuellen Projekt-Set-ups, eine *build.gradle*-Datei zu generieren. Dazu öffnet man zunächst das Projekt in Eclipse und wählt File | Export | Android | Generate Gradle build files. Im Anschluss daran startet man Android Studio und wählt Import Projekt | Import project from external model | Gradle, Use local gradle distribution, Gradle home -> Auf eine lokale Kopie von Gradle 1.6 zeigen, den Donwload findet man hier [4].
Leider funktioniert die generierte *gradle*-Datei nicht immer sofort, teilweise müssen Pfade angepasst werden oder es fehlen Referenzen zu Bibliotheken. In diesen Fällen ist die *build.gradle*-Datei zu überprüfen und ggf. anzupassen.

cendateien, allerdings werden die Referenzen in der Java-Welt nicht berücksichtigt. Scheinbar werden auch alte View-IDs nicht aus der R-Datei entfernt, was vermutlich einer der zahlreichen Bugs ist.

Aus der DDMS-Perspektive in Eclipse sind bisher nur Basisfunktionen im Android Studio zu finden. Sie bestehen aus einem LogCat-Fenster und einem einfachen Device-Management. Erweiterte Funktionen wie Traceview, Hierarchy Viewer oder Allocation Tracker sind nicht integriert. Es gibt allerdings einen Button, um die Standalone-DDMS-Variante zu starten.

Android Studio bringt sein eigenes Android SDK mit. Hat man bereits ein Android SDK installiert, kann dies zu interessanten Effekten führen. So wurden auf unserer Installation die AVDs des bereits vorhandenen SDKs im Android Studio angezeigt; der Versuch, diese zu starten, ergab dann allerdings eine Fehlermeldung.

Kleinere Probleme gibt es auch bei der Arbeit mit Android-Library-Projekten. Legt man zum Beispiel ein neues Library-Projekt im Android Studio über die Funktion „New Module“ an, fehlt im generierten Android-Manifest der „package“-Eintrag. Dies kann durch manuelles Hinzufügen behoben werden.

Wie geht's weiter?

Neben der Euphorie um so viel neue Technologie mischen sich auch einige Fragen, die wir versuchen, auf Basis der aktuellen Fakten (kurz nach der Google I/O) zu beantworten:

- **Soll ich meine Projekte von Eclipse auf Android Studio migrieren?** Das hängt letztendlich von der persönlichen Präferenz ab. Ob man IntelliJ oder Eclipse bevorzugt, ist eine individuelle Entscheidung. Android Studio hat noch zahlreiche Bugs, die Arbeit damit kann also holprig sein. Dafür belohnt es mit guter Performance und dem ein oder anderen kleinen Extrafeature. Ein zusätzliches Risiko besteht allerdings in der Migration auf Gradle, die unter Android Studio vorausgesetzt wird.
- **Was passiert mit den ADT unter Eclipse?** Sie werden offiziell weiter „supported“. Was das genau bedeutet, kann niemand vorhersagen, denkbar ist aber, dass alle zukünftigen Android-Versionen und -APIs unterstützt werden, und eine Gradle-Anbindung über das bereits verfügbare Eclipse-Gradle-Plug-in realisiert werden wird.
- **Was wird aus dem Ant-basierten Build-System?** Laut Xavier Ducrohet – seines Zeichens Android Tools Lead – wird das Ant-System demnächst als „deprecated“ markiert und nicht weiterentwickelt [5].
- **Wird Android Studio Teil von IntelliJ 13?** Nein, Android Studio ist eine getrennte Variante basierend auf IntelliJ 13 [6].
- **Kann ich Features der IntelliJ Ultimate Edition in Android Studio verwenden?** Nein, die Empfehlung in diesem Fall ist mit der Ultimate Edition zu arbeiten. Diese bietet (in der Version 13) nahezu dieselben Features zur Android-Entwicklung wie Android Studio [6].

Fazit

Es kommt Bewegung in den Android-Tool-Bereich. Das neue Build-System war schon länger bekannt, aber die Ankündigung von Android Studio auf der Google I/O hat dann doch überrascht. Mit den aktuellen Beta- bzw. Alphaversionen kann man sich schon heute ein Bild davon machen, wie die nahe Zukunft der Android-Entwicklungsumgebung aussehen wird: Die vollwertige Referenzimplementierung (Android Studio) wird auf IntelliJ aufsetzen und eine enge Verzahnung mit dem Android-Plug-in für Gradle mit sich bringen. Eclipse wird über ADT weiterhin unterstützt, aber wahrscheinlich nicht mit so viel Aufmerksamkeit vom Android-Tools-Team bedacht werden wie die IntelliJ Integration. Über kurz oder lang sollte man sich vom Ant-basierten Build-System verabschieden.

Spannend bleibt, wie lange es dauert, bis das neue Build-System und Android Studio einen stabilen Status erreichen und ohne größere Schmerzen für produktive Projekte eingesetzt werden können. Die Frage, ob IntelliJ oder Eclipse (oder NetBeans) die „bessere“ IDE darstellt, kann und darf jeder für sich selbst beantworten und sich dabei vielleicht an die schon nostalgisch anmutenden Diskussionen zurückerinnern, ob denn nun Emacs, oder vi oder Vim oder XEmacs oder nano der „bessere“ Editor ist.

Dominik Helleberg ist bei der inovex GmbH für die Entwicklung von mobilen Applikationen zuständig. Neben diversen Projekten im JME-, Android- und Mobile-Web-Umfeld hat er den JCP und das W3C bei der Definition von Standards für mobile Laufzeitumgebungen unterstützt.

Links & Literatur

[1] http://developer.android.com/sdk/installing/studio.html

[2] http://www.jetbrains.com/idea/features/index.html

[3] http://www.gradle.org/docs/current/userguide/gradle_wrapper.html

[4] http://www.gradle.org/downloads

[5] https://www.youtube.com/watch?feature=player_embedded&v=LCJAgPkpmR0

[6] http://blogs.jetbrains.com/idea/2013/05/intellij-idea-and-android-studio-faq/

Dmitry Jemerov von JetBrains zu Googles neuem Android Studio, das auf IntelliJ IDEA basiert

„Apps von Anfang bis Ende entwickeln“

Java Magazin: Google hat kürzlich Android Studio bekannt gegeben, eine Kooperation mit JetBrains. Es geht um eine IDE für Android-Entwickler, die auf IntelliJ IDEA basiert. Kannst du uns bitte erzählen, wie es zu dieser Zusammenarbeit kam?

Dmitry Jemerov: Diese Idee entstand bei Google. Sie sind an uns herangetreten mit dem Vorschlag, gemeinsam zu diskutieren, wie man die Vorteile von IntelliJ nutzen kann, um ein besseres Erlebnis für Android-Entwickler zu schaffen. Wir haben dann vorgeschlagen, eine Partnerschaft aufzubauen, und zwar um die Open-Source-Version von IntelliJ IDEA herum. Nach einigem Hin und Her hat Google dem zugestimmt, und beide Parteien sind soweit sehr glücklich mit dem Verlauf der Dinge.

JM: Android Studio ist also ein Google-Produkt, inwiefern seid ihr in die Entwicklung noch involviert?

Jemerov: Im Wesentlichen hat Google die Betreuung des Android-Plug-ins von uns übernommen. Wir investieren immer noch einen gewissen Anteil an Entwicklungsaufwand, um sicherzustellen, dass die Bedürfnisse unserer Bestandsnutzer abgedeckt werden, aber der Großteil der neuen Funktionalitäten wird vom Google-Team entwickelt. Wir stellen Google außerdem die Plattformerweiterungen zur Verfügung, die sie brauchen, um ihr gewünschtes Feature-Set für die Android-Tools zu implementieren.

JM: Wo liegt der Unterschied zwischen der neuen IDE und den Android-Features in IntelliJ IDEA?

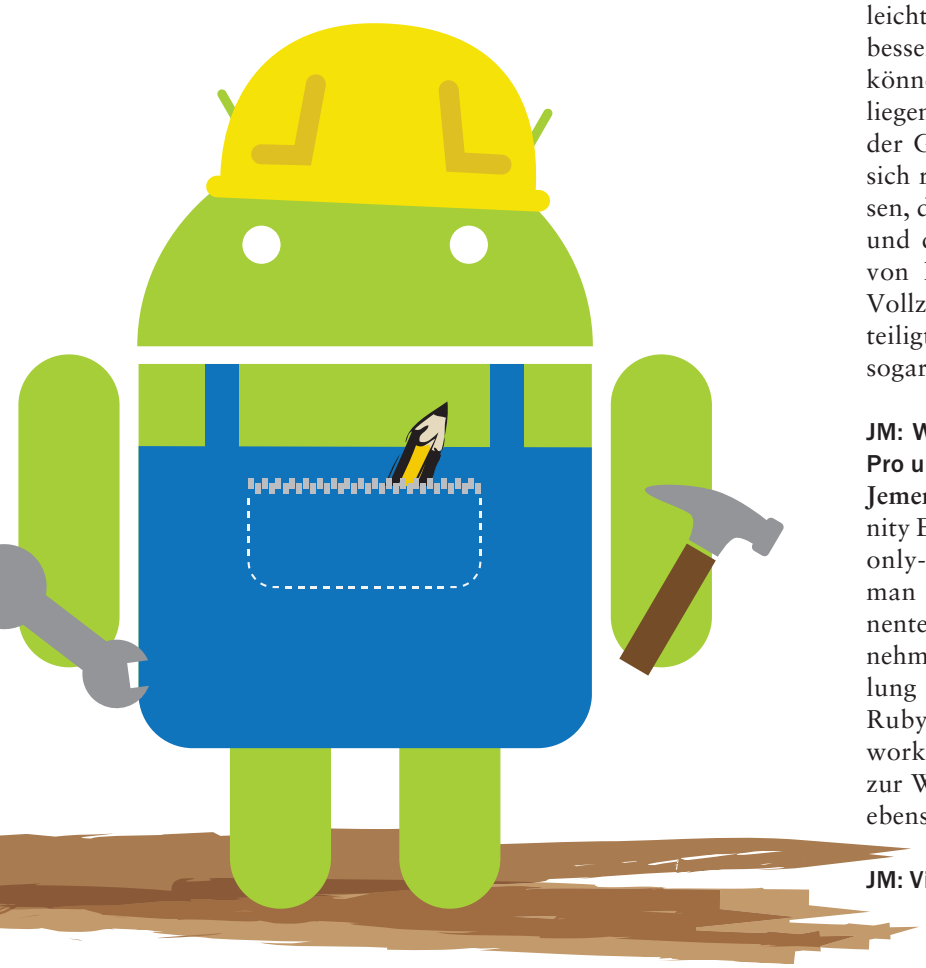
Jemerov: Der größte Unterschied ist, dass Android Studio sich komplett auf Android und das Gradle-Build-System konzentriert. Gradle wird erforderlich sein, um Projekte bauen zu können. Außerdem wird es in Android Studio einige Features geben, die die Einrichtung der Entwicklungsumgebung vereinfachen

Seit seinem Start bei JetBrains 2003 hat Dmitry Jemerov viele Aufgaben im Unternehmen innegehabt und war an der Entwicklung der meisten Java-IDEs von JetBrains beteiligt, genau wie an der Entwicklung der Programmiersprache Kotlin. In seiner aktuellen Rolle als CTO des Unternehmens ist Dmitry verantwortlich für externe Partnerschaften, diese Arbeit führte auch zur Entstehung von Android Studio. Er leitet des Weiteren die Entwicklung von PyCharm und WebStorm, der JetBrains IDE für die Python- und Web-/JavaScript-Entwicklung.

Android Studio stellt ein kompettes Set von Tools bereit, um Android-Anwendungen von Anfang bis Ende zu entwickeln.

sollen, zum Beispiel wird es ein gepacktes und automatisch konfiguriertes Android SDK geben. IntelliJ auf der anderen Seite wird weiterhin eine breite Palette von Projekttypen und verschiedenen Build-Systemen wie Gradle, Maven, Ant und das in IntelliJ IDEA eingebaute Build-System unterstützen.

JM: Was sind denn deiner Meinung nach die wichtigsten Features von Android Studio?
Jemerov: Ich kann es zwar schwer mit anderen IDEs wie Eclipse vergleichen, da ich mit Eclipse nicht so vertraut bin, aber ich kann sagen, dass Android Studio ein komplettes Set von Tools bereitstellt, um Android-Anwendungen von Anfang bis Ende zu entwickeln. Um ein neues Projekt zu starten, steht dem Entwickler ein neuer intuitiver Wizard zur Verfügung, der detaillierte Beschreibungen zu allen Auswahlmöglichkeiten bietet, die man zu Beginn treffen muss. Für die App-Entwicklung kann man einen guten visuellen Designer und ein mächtiges Set von Codevervollständigungstools nutzen, um Java-Code zu schreiben. Eine hohe Anzahl von gepackten Lint-Checks, das sind Werkzeuge für die statische Codeanalyse, stellt sicher, dass die Anwendung die Qualitätsstandards einhält, die heutzutage von Android-Apps erwartet werden. Und schließlich sorgt das Gradle-Build-System dafür, dass die finale *.apk*-Datei für die Anwendung erstellt wird, samt der Unterstützung für Debug/Release, Free/Paid und alle weiteren Eventualitäten, die die App braucht.



JM: IntelliJ IDEA wurde vor vier Jahren Open Source gestellt. Das war offensichtlich der Wendepunkt für Unternehmen wie Google, ihr System auf das von IDEA aufzubauen, und davon wiederum profitiert die Codebasis von IntelliJ IDEA. Kannst du uns ein bisschen was von euren Erfahrungen der letzten vier Jahre berichten, von den Vorteilen der Open-Source-Welt?
Jemerov: Der größte Vorteil der Community Edition war, dass die Entwicklung von Plug-ins extrem erleichtert wurde. Plug-in-Entwickler verstehen jetzt viel besser, wie der Code der Plattform funktioniert, und können ihre Plug-ins in Verbindung mit der darunter liegenden Plattform debuggen. Ich würde sagen, dass der Grad an Communitymitwirkung am Produkt an sich recht gering ist, ganz einfach weil die Leute wissen, dass JetBrains ein kommerzielles Unternehmen ist und davon ausgehen, dass die Produktverbesserung von Leuten gemacht/übernommen wird, die dafür Vollzeit bezahlt werden. Dennoch ist die Zahl der Beteiligten stetig gestiegen, und ein paar davon waren sogar substanzielle Features für das Produkt.

JM: Wie wird sich Android Studio von IntelliJ IDEA 13 Pro unterscheiden?
Jemerov: Android Studio oder IntelliJ IDEA Community Edition sind die richtige Wahl, wenn man Mobile-only-Android-Anwendungen bauen möchte. Wenn man Anwendungen mit einer serverseitigen Komponente bauen will, sollte man IntelliJ IDEA Ultimate nehmen, denn es unterstützt die serverseitige Entwicklung in verschiedenen Sprachen wie Java, Python, Ruby oder PHP und mit verschiedenen Webframeworks. Außerdem beinhaltet Ultimate die Möglichkeit zur Webentwicklung, namentlich mit JavaScript/CSS, ebenso wie SQL und Database-Support.

JM: Vielen Dank für das Gespräch!

Android-Anwendungen mit optischer Zeichenerkennung

Android lernt lesen

Visitenkarten abfotografieren und direkt als Kontakt speichern, ein Schild mit der Smartphonekamera aufnehmen und direkt in eine andere Sprache übersetzen – in vielen Situationen kann optische Zeichenerkennung das Leben erleichtern. Mit den Tesseract-Android-Tools kann Optical Character Recognition (OCR) problemlos im eigenen Projekt realisiert werden. Erfahren Sie, wie man Google Tesseract offline in der eigenen Anwendung nutzt.

von Mathias Gebbe

Die optische Zeichenerkennung bietet im mobilen Sektor vielseitige Anwendungsmöglichkeiten. Ein großer Vorteil hierbei ist, dass die meisten Smartphones heutzutage von Haus aus mit einer hochauflösenden Kamera ausgerüstet sind. So lassen sich mit Android-Apps zum Beispiel Visitenkarten abfotografieren, die dann direkt als Kontakt gespeichert werden. Oder das gerade aufgenommene Schild wird automatisch in eine andere Sprache übersetzt. Interessant wird es für den Entwickler, die eigene, bereits bestehende Anwendung durch das neue Feature der Zeichenerkennung aufzupeppen. Eine App zur Prüfung der Lotto-Gewinnzahlen wird durch die automatische Ziffernerkennung attraktiver als die Konkurrenzprodukte, da die mühsame und zeitaufwändige manuelle Eingabe des Lottoscheins wegfällt. Das Los wird nun einfach abfotografiert, und die eingelesenen Gewinnzahlen werden automatisch verglichen. Dabei muss die Programmlogik nicht verändert werden. Lediglich eine Erweiterung des Einlesevorgangs ist vonnöten. Allerdings gilt zu berücksichtigen, dass keine Texterkennung perfekt funktioniert. So ist eine Kontrolle und Korrektur durch den Endanwender erforderlich, da die Ergebnisse der automatischen Zeichenerkennung stark von den äußeren Umweltfaktoren abhängen, unter denen das Bild aufgenommen wurde. Wenn sich schon auf dem Foto keine brauchbare Information befindet, können auch von der Software keine sinnvollen Zeichen extrahiert werden. Dieser Artikel gibt einen kurzen und einfachen Einstieg in die Android-Entwicklung mit Tesseract [1] anhand eines kleinen Eclipse-Projekts. Dort wird die Zeichenerkennung auf ein vorher mit der Kamera aufgenommenes Bild angewandt. Nach der Verarbeitung wird ein editierbares Textfeld mit dem automatisch erkannten Text befüllt.

Tesser-wer?

Tesseract gilt als eine der genauesten frei verfügbaren OCR Engines auf dem Markt [2] und steht unter der freien Apache-Lizenz. Die Engine dient der optischen Zeichenerkennung und besitzt keine grafische Oberfläche. Das Programm wird direkt über die Kommandozeile gesteuert und benötigt für die Erkennung von Zeichen keinen Internetzugriff. Durch Trainingsdaten ist Tesseract in der Lage, neue Sprachdaten zu generieren. Diese Sprachdaten liegen bereits in über sechzig verschiedenen Sprachen vor – darunter Chinesisch, Arabisch und Hebräisch. Wer will, kann aber auch selbst Sprachdaten erzeugen. Ursprünglich wurde Tesseract von Hewlett-Packard in den Jahren 1985 bis 1995 als proprietäre Software entwickelt. Nachdem HP sich aus dem OCR-Geschäft zurückgezogen und die Weiterentwicklung weitestgehend eingestellt hatte, wurde das Projekt im Jahr 2005 an die Universität von Nevada in Las Vegas (UNLV) übergeben. Da der ehemalige Hauptentwickler Ray Smith zu diesem Zeitpunkt bereits bei Google arbeitete, nahm Google sich des Projekts an, überarbeitete den Quellcode und veröffentlichte die Software noch im gleichen Jahr unter der freien Apache-Lizenz. Tesseract wird unter anderem für die bekannte Google-Dienstleistung Google Books eingesetzt, die das in Büchern gespeicherte Wissen der Welt digitalisiert. Die in C++ geschriebene Software stellt ein umfangreiches API [3] bereit. Durch dieses wird es möglich, Tesseract direkt aus einer Android-Applikation

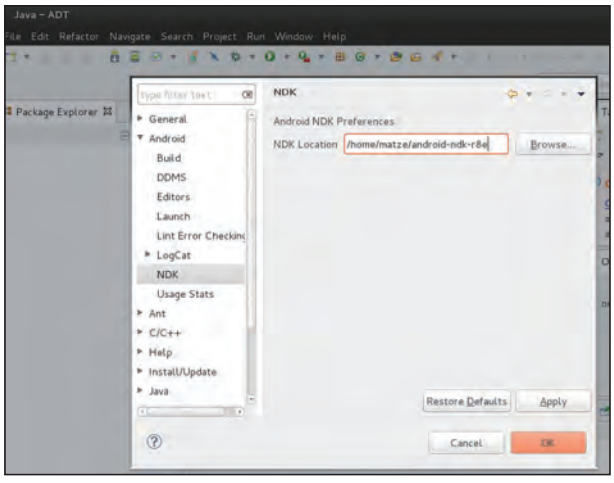


Abb. 1: Eclipse NDK

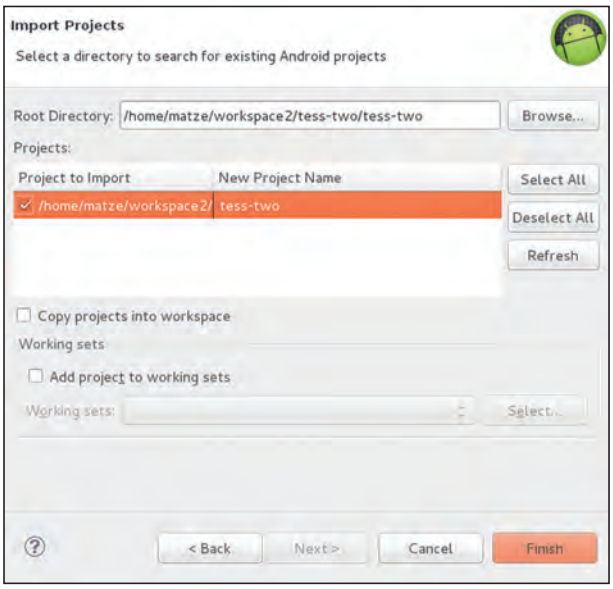


Abb. 2: Import des Projektverzeichnisses

zu verwenden. Mit dem Android Native Development Kit [4] (Android NDK) können Teile einer App in nativen Codesprachen wie C oder C++ implementiert werden. Mit den Tesseract-Android-Tools stellt Google ein fertiges Eclipse Library Project bereit. Die Android-Bibliothek wiederum ermöglicht den Zugriff auf das nativ kompilierte Tesseract-API. Da zusätzlich zum klassischen Android SDK das Android NDK benötigt wird, ist in dem Beispielprojekt dazu das aktuelle *android-ndk-r8e* installiert. Vorausgesetzt, es ist eine Eclipse IDE mit den Android-Developer-Tools [5] auf dem System vorhanden, kann die native Tesseract-Bibliothek nach dem Download mit dem Befehl *ndk-build* händisch erzeugt werden. In diesem Artikel wird der GitHub-Fork *tess-two* [6] der Tesseract-Android-Tools des Autors Robert Theis genutzt. In dem Fork sind bereits alle benötigten Komponenten vorhanden, und das Werkzeug wurde um einige zusätzliche Funktionen wie z. B. die freie Bildverarbeitungsbibliothek Leptonica Image erweitert. *Tess-two* greift auf das aktuelle *Tesseract v3.02.02* zurück und bringt dieses beim Download mit. Die Engine kann ab der Android-Version 2.2 (API-Level 8) und höher eingesetzt werden.

Listing 1: Bibliothek erzeugen

```
# tess-two Tesseract-Android-Bibliothek erzeugen
# Klonen des aktuellen Repositorys mit git
cd <workspace-verzeichnis>
git clone https://github.com/rmtheis/tess-two.git
cd tess-two/tess-two
# Kompilieren der Bibliothek
ndk-build -j8
```

Listing 2: „MainActivity.java onCreate()"-Methode – Auszug

```
    // Die benötigten Verzeichnisse auf der SD-Karte bei Bedarf anlegen
    String[] paths = new String[] { app_path + "/" };
  for (String path : paths) {
  File dir = new File(path);
  if (!dir.exists()) {
    // Erzeugt die Verzeichnisse falls nicht vorhanden
    if (!dir.mkdirs()) {
      Log.v(TAG, "ERROR: " + path + " failed");
      return;
    } else {
      Log.v(TAG, "Created directory " + path);
    }
  }
}
// Existenz der Sprachdaten lang.traineddata (hier lang=deu prüfen)
// http://code.google.com/p/tesseract-ocr/downloads/list
if ((new File(app_path + "/tessdata/" + lang + ".traineddata")).exists()) {
  Log.v(TAG, "found " + lang + " traineddata");
}else{
  Log.e(TAG, "traineddata not found");
}
```

Android NDK vorbereiten

Nachdem das aktuelle Android NDK heruntergeladen und entpackt wurde, wird es in Eclipse unter Window | Preferecnes | Android | Ndk eingebunden (**Abb. 1**).

Zusätzlich sollte das NDK-Verzeichnis den Umgebungsvariablen hinzugefügt werden. Unter Linux wird dazu die *Path*-Variable erweitert:

```
export PATH=$PATH:/home/$USER/android-ndk-r8e/ >> /home/$USER/.bashrc
```

Los geht's!

Zuerst wird das *tess-two* Git Repository in den Workspace von Eclipse geklont oder direkt von der Webseite https://github.com/rmtheis/tess-two heruntergeladen. In dem Repository befindet sich in dem Unterordner *tess-two* ein Android-Bibliotheksprojekt. In diesem Verzeichnis kann der Befehl *ndk-build* ausgeführt werden, damit die Bibliothek für alle gängigen Prozessorarchitekturen erzeugt wird (Listing 1).

Ist der *ndk-build* erfolgreich abgeschlossen, sind in dem automatisch erzeugten Unterverzeichnis *./lib* die Bibliotheken *libtess.so* (Tesseract) und *liblept.so* (Leptonica) für die verschiedenen Prozessorarchitekturen (ARM, X86 usw.) zu finden. Das gesamte Projektverzeichnis kann jetzt in Eclipse unter File | Import | Android | Existing Android Code Into Workspace | Next | Browse importiert werden (**Abb. 2**).

Nach dem Import des Projekts sollten die Abhängigkeiten und Eigenschaften sicherheitshalber neu gesetzt werden:

- Rechtsklick auf das Projekt, Android Tools | Fix Project Properties
- Rechtsklick auf das Projekt, Properties | Android | Is Library Checkbox auswählen

Die Tesseract-Werkzeuge stehen jetzt bereit. Lediglich die Sprachdaten der jeweiligen Zielsprache werden noch benötigt.

Die Beispielanwendung

Bei der Beispielanwendung [7] handelt es sich um eine einfache Applikation mit einer Activity, die demonstriert, wie der manuelle Eingabeprozess in einem Textfeld *EditText* (**Abb. 3** oben) durch die Zeichenerkennung ersetzt werden kann. Die Beispiel-App besteht darüber hinaus nur aus einem Button (**Abb. 3** unten), der die Aufnahme mit der Kamera startet, und einer *ImageView* (**Abb. 3** mittig), um das aufgenommene Bild anzuzeigen.

Damit die Android-Anwendung auf die Kamera und die Speicherkarte zugreifen darf, sind erweiterte Berechtigungen in die *AndroidManifest.xml* einzutragen:

```
// Die Berechtigungen werden zusätzlich benötigt. Falls nur Bilder von der
// SD-Karte oder dem Internet geladen werden, kann auf die Berechtigung für
// die Kamera verzichtet werden.
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.
                                        WRITE_EXTERNAL_STORAGE" />
```

Abb. 3: Beispielanwendung (Quelle: http://www.oxygen-icons.org)

Abb. 4: Sprachdaten werden im Verzeichnis „tessdata“ benötigt

Die Sprachdaten können direkt in die Anwendung eingebunden oder, wie hier, der Einfachheit halber auf dem SD-Kartenspeicher des Smartphones abgelegt werden. Die Daten sind direkt von der offiziellen Tesseract-Homepage [6] zu beziehen und werden jeweils durch ein länderspezifisches Kürzel wie zum Beispiel *deu.traineddata* für Deutschland oder *eng.traineddata* für England gekennzeichnet (**Abb. 4**). Die Sprachdaten werden von der Beispielanwendung in dem Verzeichnis *tesssdata* benötigt und müssen manuell dorthin kopiert werden.

Damit das Laden der Daten auch auf anderen Android-Geräten gelingt, wird die folgende Umgebungsvariable genutzt:

```
public static final String app_path = Environment.
                    getExternalStorageDirectory().toString() + "/tesseract-ocr";
```

Die Trainingsdaten werden bei der Initialisierung des *TessBaseAPI()* geladen. In Listing 2 sind die Initialisierungen der *onCreate()*-Methode erläutert.

Listing 3: „startCameraActivity()“: Bild mit Android aufnehmen

```
// image_path = app_path + "/ocr.png";
protected void startCameraActivity()
{
  File file = new File( image_path );
  Uri outputFileUri = Uri.fromFile( file );
  // Bild aufnehmen und danach die MainActivity aufrufen
  Intent intent = new Intent(android.provider.MediaStore.
                                          ACTION_IMAGE_CAPTURE );
  intent.putExtra( MediaStore.EXTRA_OUTPUT, outputFileUri );
     startActivityForResult( intent, 0 );
}
// Die Rückgabe des Image-Capture-Events abfangen
// Dazu wird die lokale Methode onActivityResult überschrieben
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                                    Intent data) {
  Log.v(TAG, "resultCode: " + resultCode);
  if (resultCode == -1) {
    // Bild wurde aufgenommen. Die Methode onPhoto() übernimmt die
    // eigentliche Zeichenerkennung durch Tesseract
    onPhoto();
  } else {
    Log.v(TAG, "no picture captured");
  }
}
```

Ein Bild aufnehmen

Da keine Bildverarbeitung oder -manipulation während der Vorschau oder Aufnahme selbst durchgeführt wird, kann der Standard-Intent *ACTION_IMAGE_CAPTURE* zur Bildaufnahme genutzt werden. Der Intent wird in der Beispielanwendung mit einem Klick auf den Button gestartet, woraufhin dieser die Methode *onPhoto()* aufruft. Diese beginnt dann mit der eigentlichen Zeichenerkennung (Listing 3).

Die Tesseract-Bibliothek einbinden

Damit die Applikation das Tesseract-API nutzen kann, muss das durch den *ndk-build* erzeugte Library-Projekt eingebunden werden. In Eclipse wird dies erreicht durch: Rechtsklick auf das Projekt, Properties | Android | Add ... Die IDE sollte das Verzeichnis *tess-two* automatisch vorschlagen und erkennen, da es vorher als Bibliothek gekennzeichnet wurde. Konnte die Bibliothek erfolgreich eingebunden werden, wird dies durch einen grünen Haken an der Referenz gekennzeichnet (**Abb. 5**).

Mit dem folgenden Befehl wird das Tesseract-API in den Programmcode importiert:

```
import com.googlecode.tesseract.android.TessBaseAPI;
```

Listing 4: „onPhoto()“-Methode: Bildverarbeitungsteil

```
// Bild um ¼ verkleinern
BitmapFactory.Options opts = new BitmapFactory.Options();
opts.inSampleSize = 4;
Bitmap bitmap = BitmapFactory.decodeFile(image_path,opts);
// Tesseract braucht ARGB_8888 jeder Pixel in 4 Byte gespeichert
// Aus http://developer.android.com:
// This configuration is very flexible and offers the best
// quality. It should be used whenever possible.
// https://github.com/rmtheis/android-ocr/

bitmap = bitmap.copy(Bitmap.Config.ARGB_8888, true);

// Bildausrichtung herausfinden
// http://stackoverflow.com/questions/4517634/
try {
  ExifInterface exif = new ExifInterface(image_path);
  int orientation = exif.getAttributeInt(
  ExifInterface.TAG_ORIENTATION,
  ExifInterface.ORIENTATION_NORMAL);
  Log.v(TAG, "Orientation: " + orientation);
  int rotate = 0;
  switch (orientation) {
    case ExifInterface.ORIENTATION_ROTATE_90:
    rotate = 90;
    break;
    case ExifInterface.ORIENTATION_ROTATE_180:
    rotate = 180;
    break;
    case ExifInterface.ORIENTATION_ROTATE_270:
    rotate = 270;
    break;
  }
  Log.v(TAG, "rotation: " + rotate);
  //Bild drehen
  //http://stackoverflow.com/questions/8608734/
  if (rotate != 0) {
    Matrix matrix = new Matrix();
    matrix.postRotate(rotate);
    bitmap = Bitmap.createBitmap(bitmap, 0, 0,
    bitmap.getWidth(), bitmap.getHeight(), matrix, false);
  }
} catch (IOException e) {
  Log.e(TAG, "can not rotate the image: " + e.toString());
}
// Bild in der ImageView der Anwendung anzeigen
image.setImageBitmap( bitmap );
```

OCR vorbereiten und durchführen

Bevor die Zeichenerkennung auf das gespeicherte Bild angewandt werden kann, muss das aufgenommene Bild noch korrekt ausgerichtet bzw. rotiert werden. Darüber hinaus erwartet Tesseract die Bitmap in ARGB_8888, damit jedes Pixel in genau 4 Byte gespeichert wird. Die Rückgabe des Kamera-Intents ruft durch die überschriebene *onActivityResult*-Methode die Funktion *onPhoto()* auf. Hier wird zu Beginn die Bildverarbeitung durchgeführt (Listing 4). Im weiteren Verlauf startet die Initialisierung von Tesseract und die Zeichenerkennung beginnt.

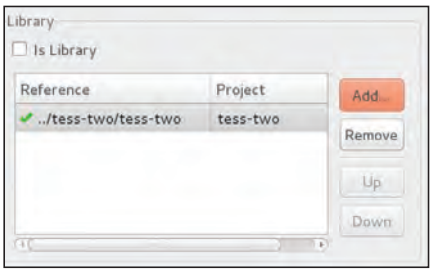


Abb. 5: Bibliothek erfolgreich eingebunden

Die Bitmap ist nun korrekt ausgerichtet und für die Zeichenerkennung bereit. Die eigentliche Magie Tesseracts auf das Bild anzuwenden, fällt dabei vergleichsweise kurz aus. Mit der Methode *getUTF8Text();* liefert Tesseract automatisch alle auf dem Bild erkannten UTF8-Zeichen als String zurück. Um Fehler vorzubeugen, werden danach alle Sonderzeichen und überflüssige Leerzeichen aus dem Rückgabe-String entfernt. Sollen in der späteren Applikation beispielsweise nur Zahlen erkannt werden, kann vor der Zeichenerkennung eine so genannte *Black-* oder *Whitelist* angelegt werden. Alle Zeichen der Blacklist werden ignoriert oder nur die Zeichen auf der Whitelist zugelassen. Die Initialisierung und Durchführung der Zeichenerkennung ist in Listing 5 erläutert.

Fazit

Das Ergebnis der Beispielanwendung ist **Abbildung 6** zu entnehmen. Im Beispieltext „Java Magazin ist super 42 ist die Antwort automatisch erkannter Text“ wurden lediglich die Zeichen „i“ als „1“ interpretiert, womit Tesseract ein wirklich tolles Ergebnis liefert. Da der Textanteil im Verhältnis zur gesamten Bildgröße eher klein ausfällt, kann es dadurch aber zu weiteren Fehlinterpretationen der Software kommen. So ist das Ergebnis der Zeichenerkennung, je nach Qualität der Aufnahme und Zeichenquelle, mal besser, schlechter – oder aber unbrauchbar. Für den professionellen Einsatz ist es also wichtig, den Anwender über diese Fehleranfälligkeit zu informieren. Um Fehlinterpretationen vorbeugen zu können, helfen vorher definierte fixe Bildbereiche, in denen ausschließlich bestimmte Zeichen erwartet werden. Störfaktoren wie Bildrauschen und unsaubere Farbübergänge können durch eine vorgeschaltete Bildverarbeitung mit Gauß-Frequenzfilter oder Schwarzweiß-Wandlung reduziert oder sogar komplett entfernt werden. Wer will, kann sogar selbst Sprachdaten erzeugen, die dann direkt auf das jeweilige Szenario der Anwendung zugeschnitten sind. Alles in allem ist Tesseract eine tolle freie Software, die, wenn richtig konfiguriert, vor allem im mobilen Sektor sehr viel Spaß macht, solide Ergebnisse liefert und sich dabei sehr leicht in bestehende Projekte integrieren lässt.

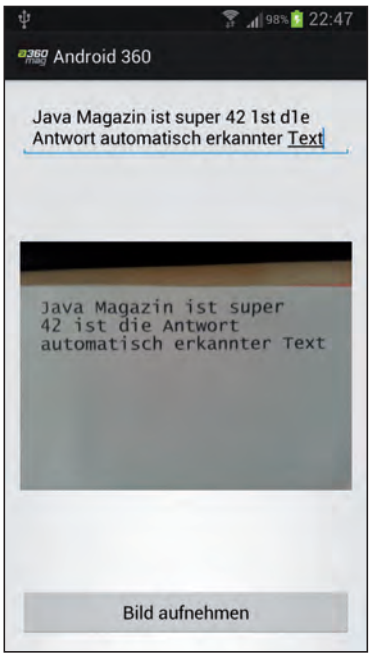


Abb. 6: Ergebnis der Beispielanwendung; lediglich die Zeichen „i“ wurden als „l“ interpretiert

Listing 5: „onPhoto()“-Methode: OCR durchführen

```
// Tesseract initialisieren
TessBaseAPI baseApi = new TessBaseAPI();
// Debug-Ausgabe im logcat
baseApi.setDebug(true)

// deutsche Sprachdaten laden String lang = "deu"
baseApi.init(app_path, lang);

// um z. B. nur Zahlen zu erkennen oder um die Zeichen abcABC zu
//ignorieren:
//baseApi.setVariable(TessBaseAPI.VAR_CHAR_WHITELIST,"1234567890");
//baseApi.setVariable(TessBaseAPI.VAR_CHAR_BLACKLIST, "abcABC");

// Bild laden und OCR starten
baseApi.setImage(bitmap);
String ocr = baseApi.getUTF8Text();
baseApi.end();

Log.v(TAG, "OCR-TEXT: " + ocr);
// alle Sonderzeichen rauswerfen
ocr = ocr.replaceAll("[^a-zA-Z0-9]+", " ");
// Textfeld EditText edit_text mit dem automatisch erkannten Text füllen
edit_text.setText(ocr);
```

Mathias Gebbe (27) studiert Informatik – Verteilte und mobile Anwendungen im Masterstudiengang an der Hochschule Osnabrück. Nebenbei arbeitet er bei der auf Open Source spezialisierten Intevation GmbH (www.intevation.org) in Osnabrück.

Links & Literatur

[1] tesseract-ocr: http://code.google.com/p/tesseract-ocr/

[2] http://archive09.linux.com/articles/57222

[3] C++ API für Tesseract: http://code.google.com/p/tesseract-ocr/source/browse/trunk/api/baseapi.h

[4] Android NDK: http://developer.android.com/tools/sdk/ndk/index.html

[5] Eclipse IDE mit ADT (Android Developer Tools): http://developer.android.com/sdk/index.html

[6] Tesseract-Sprachdaten: tesseract-ocr-3.02.deu.tar.gz: http://code.google.com/p/tesseract-ocr/downloads/list

[7] https://github.com/matzegebbe/Android360

Wie kann ich mit meiner App Geld verdienen?

In-App Billing reloaded!

Generell gibt es da zwei Varianten: Entweder ich schalte in meiner App Werbung oder ich bringe meine Nutzer dazu, für meine App zu bezahlen. Letzteres ist gar nicht so einfach. Das Standardmodell, dass ich meine App in einer kostenlosen „Lite"-Version und einer kostenpflichtigen „Pro"-Version anbiete, ist zwar einfach zu realisieren, führt aber dank hoher Hürde (Benutzer muss erneut in den Play Store und die App suchen und installieren) selten zum Erfolg. Vielversprechender ist da schon das In-App Billing, bei dem der Benutzer einfach direkt in der App durch Bestätigung eines Knopfs den Kauf der Pro-Version durchführen kann. Leider ist diese Variante für den Entwickler deutlich komplexer und schwieriger umzusetzen. Grund genug, das neue In-App-Billing-API näher zu beleuchten.

von Lars Röwekamp und Arne Limburg 

In-App Billing wurde zum ersten Mal im März 2011 veröffentlicht [1]. Seit diesem Zeitpunkt gibt es die Möglichkeit, aus der eigenen App heraus über den Google Play Store Erweiterungen zu verkaufen. Seither hat sich viel getan. Die verbreitetste Version des In-App-Billing-API ist aktuell die Version 2, seit Dezember letzten Jahres ist aber auch Release 3 verfügbar. Wie lässt sich In-App Billing in die eigene App integrieren? Wo liegen die Unterschiede zwischen Version 2 und 3 und lohnt sich ein Umstieg? Diese Kolumne wird Antworten auf diese Fragen liefern.

Komplexe Architektur in Version 2

Die Version 2 des In-App-Billing-API ist komplett asynchron und kommuniziert über Broadcast Intents mit der Applikation. Dieser Ansatz wirkt zwar auf den ersten Blick elegant und sinnvoll, weil es sich bei dem Billing-Aufruf ja um einen Remote-Call handelt, der sich direkt mit Googles Server für die Kaufabwicklung verbindet. In der Praxis erweist sich diese Architektur aber als äußerst unpraktisch, weil so viele Komponenten implementiert und registriert werden müssen, um In-App Billing zu realisieren [2].

Die Komponente in Android, die für das In-App Billing in der Version 2 zuständig ist, ist der *IMarketBillingService*, an den man sich wie in Android gewohnt, binden kann (Listing 1).

Der Service besteht im Wesentlichen nur aus der Methode *sendBillingRequest*, die sowohl als Parameter ein Bundle erwartet, als auch als Ergebnis ein Bundle liefert. Das Ergebnis-Bundle enthält allerdings nur einen Statuscode. Die tatsächliche Antwort erfolgt, wie bereits erwähnt, asynchron über ein Broadcast, sodass auch noch ein Broadcast Receiver implementiert und registriert werden muss. Damit nicht genug: Die Regeln zur Implementierung eines Broadcast Receivers besagen, dass dieser nur sehr kurz laufen darf, damit er nicht vom Android-System beendet wird. Daher sollte die tatsächliche Abwicklung des In-App Billing nicht im Broadcast Receiver erfolgen, sondern in einem separaten Service, der dann auch noch implementiert werden muss. Bei dem hier beschriebenen Aufwand handelt es sich nur um den Aufwand, der betrieben werden muss, um das In-App Billing abzuwickeln.

Zusätzlich sollte man bei jedem Start der App überprüfen, welche Features der Benutzer tatsächlich gekauft hat. Dieser Aufruf ist erneut ein Remote-Call, der erstens ähnlich kompliziert zu implementieren ist und

zweitens auch noch so lange dauert, dass man ihn eigentlich nicht bei jedem Start ausführen möchte. Die Alternative ist, die Informationen über die Käufe des Benutzers in der App zu speichern. Wenn sich diese Informationen allerdings in der App befinden, birgt das das Risiko, gehackt zu werden und dass Informationen manipuliert werden. Um das zu verhindern, müssten die Informationen zusätzlich verschlüsselt werden.

Alles in allem lässt sich sagen, dass mit der Version 2 des In-App-Billing-API viel Aufwand zur Integration betrieben werden muss. Werfen wir einen Blick auf Version 3.

Alles neu macht die 3

Der größte Vorteil der Version 3 ist die Umstellung auf synchrone Kommunikation [3]. Hier ist es möglich, das In-App Billing einfach mit *startIntentSenderForResult* zu starten (Listing 2). Das zugehörige Bundle samt *PendingIntent* erhält man vom *IInAppBillingService*, der den *IMarketBillingService* ersetzt.

Die Antwort erfolgt dann direkt über den Aufruf von *onActivityResult* und enthält das Ergebnis im JSON-Format (Listing 3).

Noch einfacher ist die Abfrage bereits getätigter Käufe. Hierzu kann erneut der *IInAppBillingService* verwendet werden. Der benötigte Aufruf ist hier ein ganz normaler Aufruf der Methode *getPurchases*. Möglich wird dieser synchrone Aufruf, weil der Google Play Store die Remote-Abfragen cached und daher nicht immer eine Netzwerkverbindung aufgebaut werden muss.

Das API bietet weitere Möglichkeiten, wie eine Abfrage verfügbarer Käufe oder die Möglichkeit, gekaufte Dinge zu „konsumieren“.

Mit der Version 3 lässt sich In-App Billing mit ein paar Zeilen Code realisieren.

Fazit

Die Version 3 des In-App-Billing-API ist ein großer Schritt vorwärts. Vor allem die Message-getriebenen asynchronen Aufrufe der Version 2, die dem Entwickler die Umsetzung einer umfangreichen Infrastruktur zum Verarbeiten der Kaufabwicklung abverlangt hat, wird niemand vermissen. Mit der Version 3 lässt sich In-App Billing mit ein paar Zeilen Code realisieren, auch weil dem Entwickler das komplizierte lokale Cachen der gekauften Features vom Play Store abgenommen wird.

Listing 1

```
private boolean bindToMarketBillingService() {
  String action = "com.android.vending.billing.MarketBillingService.BIND"
        return bindService(
              new Intent(action),
              serviceConnection,
              Context.BIND_AUTO_CREATE);
  }
```

Listing 2

```
Bundle buyIntentBundle = mInAppBillingService.getBuyIntent(
      3, getPackageName(), itemId, "inapp", developerPayload);
PendingIntent intent = buyIntentBundle.getParcelable("BUY_INTENT");
startIntentSenderForResult(
      intent.getIntentSender(), MY_ITEM_ID, new Intent(), 0, 0, 0);
```

Listing 3

```
protected void onActivityResult(int requestCode, int resultCode,
                                                Intent data) {
  if (requestCode == MY_ITEM_ID && resultCode == RESULT_OK) {
    String purchaseData = data.getStringExtra("INAPP_PURCHASE_DATA");
    // parse JSON-String here
  }
}
```

Lars Röwekamp ist Geschäftsführer der open knowledge GmbH und berät seit mehr als zehn Jahren Kunden in internationalen Projekten rund um das Thema Enterprise Computing.

@mobileLarson

Arne Limburg ist Softwarearchitekt bei der open knowledge GmbH in Oldenburg. Er verfügt über langjährige Erfahrung als Entwickler, Architekt und Consultant im Java-Umfeld und ist auch seit der ersten Stunde im Android-Umfeld aktiv.

@ArneLimburg

Links & Literatur

[1] http://android-developers.blogspot.de/2011/03/in-app-billing-launched-on-android.html

[2] http://developer.android.com/google/play/billing/v2/api.html

[3] http://developer.android.com/google/play/billing/api.html

Vorschau auf die Ausgabe 9.2013

Google Android

Es ist ziemlich genau drei Jahre her, dass wir Android auf das Cover des Java Magazins gebracht haben. „Wie hebt sich Android von Java ME ab?“, war damals die Frage, die ganz schön zeigt, wie viel sich in den letzten Jahren beim Betriebssystem Android getan hat. Mittlerweile hat es eine eigene Rubrik hier im Heft, als Java-Entwickler kommt man kaum noch daran vorbei. Doch auch wenn Android rein nach Zahlen den Markt gegenüber iOS dominiert, hängt der Erfolg doch entscheidend davon ab, wann Entwickler primär für Android bzw. nur noch Android entwickeln. „Android First“, dahin müssen wir kommen, und um den Einstieg für Sie zu erleichtern, haben wir nächsten Monat viele wertvolle Artikel für die Android-Entwicklung: Tutorial, Tools, Tipps und Tricks. Was kann dann noch schiefgehen?

Aus redaktionellen Gründen können sich Themen kurzfristig ändern.

Die nächste Ausgabe erscheint am 7. August 2013

Querschau

eclipse MAGAZIN — Ausgabe 4.2013 | www.eclipse-magazin.de

- **Jetty 9: Zukunft der Webprotokolle heute**
- **Thinking of U and I and E: Vaaclipse bringt die Eclipse 4 Workbench ins Web**
- **Eclipse à la carte: Dynamische Menüeinträge in Eclipse 4**

entwickler magazin — Ausgabe 4.2013 | www.entwickler.de

- **Reality Check: Professionelle Service- und Softwaretests mit Real Application Testing**
- **Google Hacking: Finden, was nicht gefunden werden soll**
- **Neue Ansätze für Tracing: Trace-basiertes Debugging von Multi-Core-Systemen**

MOBILE TECHNOLOGY — Ausgabe 2.2013 | www.mobiletechmag.de

- **Firefox OS: Mozillas Betriebssystem bringt HTML5 auf Smartphones**
- **Zwei Fliegen mit einer Klappe: Native Cross-Plattform-Apps mit Tabris**
- **Sencha Touch im Einsatz: Non-native Apps mal anders**

Inserenten

Verlag:
Software & Support Media GmbH

Anschrift der Redaktion:
Java Magazin
Software & Support Media GmbH
Darmstädter Landstraße 108
D-60598 Frankfurt am Main
Tel. +49 (0) 69 630089-0
Fax. +49 (0) 69 630089-89
redaktion@javamagazin.de
www.javamagazin.de

Chefredakteur: Sebastian Meyen
Redaktion: Claudia Fröhling, Corinna Kern, Diana Kupfer
Chefin vom Dienst/Leitung Schlussredaktion: Nicole Bechtel
Schlussredaktion: Jennifer Diener, Frauke Pesch, Lisa Pychlau
Leitung Grafik & Produktion: Jens Mainz
Layout, Titel: Tobias Dorn, Flora Feher, Dominique Kalbassi, Laura Keßler, Nadja Kesser, Maria Rudi, Petra Rüth, Franziska Sponer

Autoren dieser Ausgabe:
Tobias Bayer, Raik Bieniek, Andy Bosch, Martin Breest, Alexander Casall, Niko Eder, Mathias Gebbe, Christian Grobmeier, Gerrit Grunwald, Jens Hadlich, Steffen Heinzl, Dominik Helleberg, Dominik Kleine, Robert Ladstätter, Arne Limburg, Boris von Loesch, Dr. Benno Luthiger, Manuel Mauky, Michael Müller, Florian Pirchner, Lars Röwekamp, Bernd Rücker, Benjamin Schmeling, Peter Siemen, Stefan Siprell, Kai Spichale, Michael Thiele, Vincent Tietz, Ramon Wartala, Dirk Weil, Matthias Weßendorf, Oliver Zeigermann

Anzeigenverkauf:
Software & Support Media GmbH
Patrik Baumann
Tel. +49 (0) 69 630089-20
Fax. +49 (0) 69 630089-89
pbaumann@sandsmedia.com

Es gilt die Anzeigenpreisliste Mediadaten 2013

Pressevertrieb:
DPV Network
Tel.+49 (0) 40 378456261
www.dpv-network.de

Druck: PVA Landau
ISSN: 1619-795X

Abonnement und Betreuung:
Leserservice Java Magazin
65341 Eltville
Tel.: +49 (0) 6123 9238-239
Fax: +49 (0) 6123 9238-244
javamagazin@vuservice.de

Abonnementpreise der Zeitschrift:

Inland:	12 Ausgaben	€ 118,80
Europ. Ausland:	12 Ausgaben	€ 134,80
Studentenpreis (Inland)	12 Ausgaben	€ 95,00
Studentenpreis (Ausland):	12 Ausgaben	€ 105,30

Einzelverkaufspreis:

Deutschland:	€ 9,80
Österreich:	€ 10,80
Schweiz:	sFr 19,50
Luxemburg:	€ 11,15

Erscheinungsweise: monatlich