DVD auf S.3

POWER-DVD 20 STUNDEN JavaScript-Videos

S&S

JAVA Mag

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 10.2013

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Business Technology

EXKLUSIV FÜR ABONNENTEN

Business Technology als Bonus

W-JAX 13

Alle Infos hier im Heft! ▸83

NASHORN

Der Weg zur polyglotten VM ▸31

©iStockphoto.com/gazza30 ©iStockphoto.com/filo

4 194586 709801 10

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

MEHR THEMEN: Testing mit Arquillian und Arquillian Drone ▸42, 47

GWT 2.5: Technische Evolution und wichtiges Signal ▸58

NoSQL: Neue Herausforderungen, neue Serie ▸76

Anzeige

POWER-DVD

20 Stunden Java-Wissen

Das umfassende JavaScript-Wissen kompakt auf einer DVD! Für Einsteiger, Umsteiger und Experten.

Why Dynamic Languages on the JVM matter

Marcus Lagergren (Keynote JAX 2013)

This keynote explains why language implementations, especially dynamic languages, are more feasible to implement on top of the JVM than ever and how the JVM can execute them with high performance. As an example, we will go into detail of the Nashorn project, Oracle's new JavaScript runtime, part of the JDK as of Java 8.

Nashorn – Implementing Dynamic Languages on the JVM

Marcus Lagergren

There are many implementations of JavaScript, meant to run either on the JVM or standalone as native code. Both approaches have their respective pros and cons. The Oracle Nashorn JavaScript project is based on the former approach. This presentation goes through the performance work that has gone on in Oracle's Nashorn JavaScript project to date in order to make JavaScript-to-bytecode generation for execution on the JVM feasible.

Freuen Sie sich auf viele weitere Konferenzsessions und Interviews von und mit Experten zu JavaScript und vielfältigen Webthemen!


Douglas Crockford


Christian Heilmann


Stefan Tilkov


Gernot Starke


Martin Lippert


Andy Bosch


Christian Groß


Norman Maurer


Papick Taboada


David Zuelke


Sebastian Springer


Jakob Westhoff

AUSSERDEM AUF DVD:

JavaScript Spezial – 100 Seiten Profiwissen rund um JavaScript, Geolocation und Test-driven Development

JavaScript spielt eine immer wichtigere Rolle – nicht nur im bekannten Webumfeld wie auf Websites, im E-Commerce oder bei der Integration von Social-Media-Diensten, sondern auch im Umfeld von Anbietern klassischer Enterprise-Software. JavaScript ermöglicht (Web-)Entwicklern einerseits, Webseiten zu gestalten, die sich wie native Apps auf mobilen Devices anfühlen und in weiten Teilen ein identisches Nutzererlebnis bieten. Andererseits lassen sich Websites auf klassischen Computern und Laptops mit der Technologie dank Drag-and-Drop- und Local-Storage-Funktionalitäten mit demselben Komfort wie Desktopanwendungen bedienen. Das JavaScript Spezial richtet sich daher an erfahrene Entwickler aus allen Sprachen und Bereichen, die sich intensiv mit dieser Technologie beschäftigen möchten. Profitieren Sie von 100 Seiten Expertenwissen!

Auf dem Weg zur polyglotten VM

„Java und JavaScript haben ungefähr so viel gemeinsam wie Ei und Eiffelturm“, sagte einst ein Speaker auf der JAX. Und das, obwohl ironischerweise beides eingetragene Marken der Firma Oracle sind – historisch gewachsen durch die Übernahme von Sun Microsystems (das Patent für die Marke „JavaScript“ findet sich unter http://1.usa.gov/Q3Vi1C).

Auch wenn sich die sprachlichen (und kulturellen) Gemeinsamkeiten von Java und JavaScript stark in Grenzen halten, gibt es heute viele Ausprägungen eines kulturellen Austauschs. Denn viele Java-Entwickler müssen sich mit der Skriptsprache beschäftigen, sei es im Enterprise-Umfeld, bei kleinen Webanwendungen oder Netzwerkgeschichten. JavaScript ist weit verbreitet und hat vor wenigen Jahren eine große Renaissance unter Entwicklern erfahren.

Es spannen sich auch einige Brücken zwischen der Java- und JavaScript-Welt und mit einem solchen Bindeglied beschäftigen wir uns in dieser Ausgabe: Oracle Nashorn ist die neue JVM-basierte JavaScript-Implementierung im JDK 8, hergeleitet von dem Projekt, das dauerhaft von Nashorn ersetzt werden soll: Rhino. Die fünfzehn Jahre alte Rhino-Engine ist nicht mehr zeitgemäß und bietet hauptsächlich Plug-in-Fähigkeit. Nach invokedynamic ist Nashorn der nächste wichtige Schritt in Richtung polyglotte Virtual Machine, ist sogar selbst ein Anwendungsfall für invokedynamic und damit zu Recht ein Thema, das auf das Cover des Java Magazins gehört.

Power-DVD

Um dieses wichtige Thema zu vertiefen, gibt es diesen Monat für alle Leser ein ganz besonderes Extra: auf der beiliegenden DVD finden Sie **20 Stunden Videomaterial** von unseren Konferenzen JAX, W-JAX, MobileTechCon, webinale und IPC rund um das Thema JavaScript – unter anderem auch die Keynote sowie den technischen Talk von Marcus Lagergren zu Nashorn. Marcus ist seit 2011 Mitglied des Java-Language-Teams bei Oracle und kümmert sich dort um dynamische Sprachen auf der JVM. Natürlich sind auch Themen wie Node.js, Eclipse Orion, JavaScript Testing, GWT und MVC-Frameworks Teil der Power-DVD. Und obendrauf gibt es noch das PDF unseres JavaScript-Sonderhefts! Auf Seite 3 finden Sie weitere Informationen.

Noch mehr Geschenke

Ein weiteres Extra wartet exklusiv auf alle Abonnenten des Java Magazins: Ab sofort erhalten Sie vier Mal pro Jahr das **Business Technology Magazin** kostenlos dazu! Das Magazin thematisiert Lean Innovation, Digital Transformation, Architektur sowie Entwicklungs- und Projektmanagementmethoden in der IT-Industrie – Themen also, die wirklich jeden in der Industrie ansprechen. Weder die Grenze zwischen Fachabteilung und IT noch zwischen Entwickler und Architekt oder technologischen Welten soll hier eine Beschränkung sein. Erweitern Sie mit dem Business Technology Magazin Ihren Horizont für eine ganzheitliche IT – wer noch kein Abo hat, kann das auf http://jaxenter.de/java-magazin-abo nachholen!

In diesem Sinne wünsche ich Ihnen viel Spaß bei der Lektüre dieser Ausgabe!

Claudia Fröhling, Redakteurin

@JavaMagazin

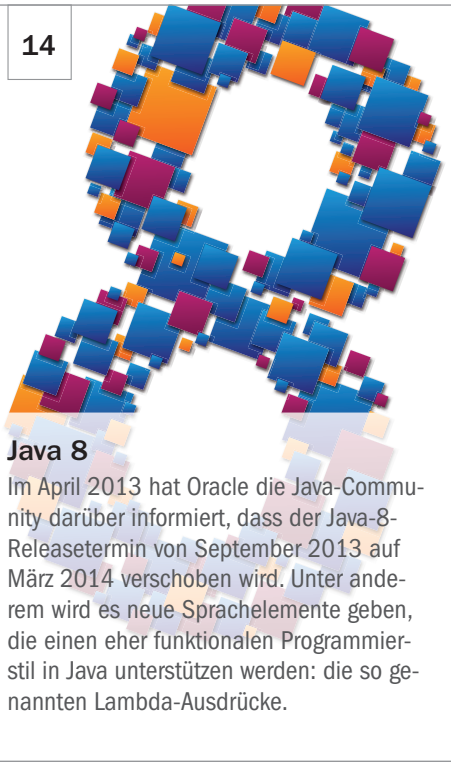
gplus.to/JavaMagazin

PS: Allen Java-8-Fans möchte ich noch einmal unsere **Java Magazin Lambda Tour** ans Herz legen, die gerade anläuft. Zum Zeitpunkt der Drucklegung dieser Ausgabe sind bereits fast alle Karten für die kostenlosen Events vergeben, aber es gibt Wartelisten, in die Sie sich eintragen können. Alle Infos finden Sie unter www.jaxenter.de/lambdatour.

Anzeige

Projekt Nashorn

Seit Mitte Dezember 2012 ist das Projekt Nashorn im OpenJDK angekommen, mit dem Ziel, eine zu hundert Prozent in Java geschriebene, schlanke und performance-optimierte JavaScript-Engine direkt auf der JVM zu implementieren. Den Entwicklern soll Nashorn ermöglichen, unter Berücksichtigung von JSR 223 (Scripting for the Java Platform), JavaScript in Java-Anwendungen einzubinden und eigene JavaScript-Anwendungen zu erstellen, die bisher jrunscript verwendet haben und künftig das neue Command-line-Werkzeug jjs (JavaScript in Java) im JDK 8 benutzen können.



Java 8

Im April 2013 hat Oracle die Java-Community darüber informiert, dass der Java-8-Releasetermin von September 2013 auf März 2014 verschoben wird. Unter anderem wird es neue Sprachelemente geben, die einen eher funktionalen Programmierstil in Java unterstützen werden: die so genannten Lambda-Ausdrücke.

Magazin

Java Core

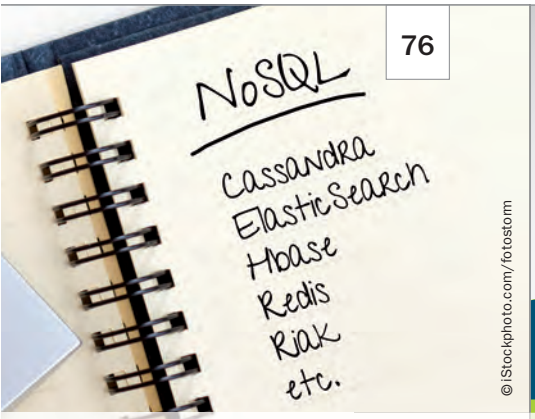
Titelthema

Enterprise

Web

Datenbanken

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



NoSQL
In Zukunft reicht es nicht mehr aus, einfach eine relationale Datenbank zu nutzen. Man muss sich gegebenenfalls für eine NoSQL-Spielart entscheiden – und dann für ein konkretes Produkt. In dieser neuen Serie stellen wir unterschiedliche Datenbanken wie Cassandra oder Neo4j auf den Prüfstand.

Android 4.3
Mangelnde Security ist der Grund Nummer 1, warum iOS in Unternehmen gegenüber (Standard-)Android die Nase vorn hat. Am 24. Juli wurde die Version 4.3 der Android-Plattform, Jelly Bean MR2, vorgestellt. Mit dieser Version hat Google Jelly Bean mit SE Linux aufgebohrt – lohnt sich eine Neubewertung? Ein erster Eindruck der neuen Securityfeatures in Android 4.3.

Android-Tutorial
Kann man eine App in fünf Minuten erstellen? Ein gewagtes Unterfangen, aber solange man nicht auf die Uhr schaut, könnte es klappen. Zeit für eine ganz kurze Zusammenfassung dessen, was wir bisher gemacht haben. Danach gehen wir dem Geheimnis von „R“ auf den Grund. „R“ ist die zentrale Klasse, über die wir Zugriff auf alle Elemente in unserem Projekt erhalten.

Frisch von JAXenter

News, Kommentare und Interviews von www.jaxenter.de

NoSQL-Datenbank Neo4j 2.0 M4 funktioniert nur noch mit Java 7

Für die zweite Major-Version der NoSQL-Datenbank Neo4j ist nun schon der vierte Meilenstein erschienen. Dieser bringt die große Neuerung, dass die Datenbank künftig nur noch mit Java 7 funktioniert. Startet man sie mit Java 6, tritt ein Fehler auf. Als Grund nennen die Macher das Lebensende von Java 6 im Februar 2013 und die inzwischen fehlende Unterstützung vonseiten Oracles.

Abgesehen davon müssen Read Operations jetzt in einer Transaktion verpackt werden. Tut man das nicht, tritt der Fehler *NotInTransactionException* auf. Durch diese neue Anforderung soll die Datenbank ihre Ressourcen deutlich besser nutzen können.

Die Query-Sprache Cypher wurde in puncto Syntax und Verhalten angepasst. Man hat Syntaxabnormalitäten beseitigt und die Sprache einheitlicher gestaltet, was besonders Einsteigern zugutekommt. Das sieht man beispielsweise an der neuen einheitlichen Praxis zum Entfernen von Properties. Der neue Cypher-Parser hat eine verbesserte Architektur, die in Bezug auf Effizienz, grammatikalische Korrektheit und Usability überarbeitet wurde.

Neo4j Milestone 2.0.0-M04 steht unter neo4j.org zum Download bereit. Alle Änderungen sind im Handbuch sowie im Changelog nachzulesen.

▶ http://docs.neo4j.org/chunked/2.0.0-M04

Amazon bald mit eigenem OUYA-Killer?

Wie Amazon-Insider an die Newsseite *Game Informer* herangetragen haben sollen, will das große Onlinekaufhaus wohl noch bis Weihnachten zumindest in den USA eine Spielkonsole im Stil der OUYA auf den Markt bringen. Als zeitliches Ziel habe man den „Black Friday" vor Augen, also den Freitag nach dem Erntedankfest, an dem die vierwöchige Phase der Weihnachtseinkäufe beginnt. Die Quellen sollen weiter berichtet haben, dass Amazon die Spiele verkaufen wolle, die bereits auf seiner Plattform verfügbar sind. Jeden Tag soll eine Büro- oder Spiele-App kostenlos verfügbar gemacht werden. Auch werde man für die Konsole einen entsprechenden Controller verkaufen. Eine offizielle Bestätigung seitens Amazon blieb allerdings bisher aus.

Dieser Schritt wäre für Amazon sehr sinnvoll, weil der Konzern dank seines Tablets Kindle Fire schon über Erfahrung im Erstellen eigener Android-Distributionen verfügt. Außerdem steht mit den Amazon Web Services (AWS) die notwendige Infrastruktur bereit, um Spielenetzwerke zu realisieren. Auch der Marktplatz ist schon da, was langfristige Einkünfte garantieren würde. Darüber hinaus hat Amazon wichtige Kontakte zu großen Entwicklern geknüpft, die ihren Beitrag zum Markterfolg einer solchen Konsole leisten könnten. All dies sind Punkte, die eine OUYA eventuell klein aussehen lassen würden, wenn Amazons Plan – so er denn existiert – aufgeht.

▶ http://bit.ly/15htacL

Links und Downloadempfehlungen zu den Artikeln im Heft

- Onlinetutorials zu Arquillian: http://arquillian.org/guides
- Arquillian Drone: http://www.arquillian.org/modules/drone-extension
- Graphvisualisierungen mit D3.js: http://d3js.org
- Project Lambda: http://openjdk.java.net/projects/lambda
- OVal – the object validation framework for Java 5 or later: http://oval.sourceforge.net/userguide.html
- Ehcache: http://ehcache.org
- Langer, Angelika; Kreft, Klaus: „Lambda Tutorial": http://www.AngelikaLanger.com/Lambdas/Lambdas.html
- Batch Application for the Java Platform 1.0: http://jcp.org/en/jsr/detail?id=352
- Android und Bluetooth Low Energy: http://developer.android.com/guide/topics/connectivity/bluetooth-le.html
- Roadmap für GWT auf der Google I/O: https://developers.google.com/events/io/sessions/327833110
- Red5-Flash-Server: https://code.google.com/p/red5

Das schnellste Gradle aller Zeiten: Gradle 1.7 erschienen

Die Verantwortlichen hinter der neuesten Version des bekannten Build-Werkzeugs rühmen sich damit, mit Version 1.7 das schnellste Gradle aller Zeiten geschaffen zu haben. Dabei wurde die Build-Performance deutlich beschleunigt, die Skriptkompilierung soll sogar bis zu 75 Prozent schneller erfolgen.

Aber das waren noch nicht alle Neuerungen, die Gradle 1.7 verspricht. Auch im Dependency Management wurden Verbesserungen vorgenommen. Beispielsweise kann man jetzt einfacher auf Abhängigkeiten aus Bintray verweisen. Die Initialisierungsmechanismen wurden ebenfalls überarbeitet.

Der neue Task Finalizer kann für alle verpflichtenden Funktionen nach der Ausführung eines Tasks verwendet werden. Mit ihm lassen sich unter anderem Ressourcen aufräumen oder Reports erstellen.

Das Plug-in Build Setup vereinfacht die Erstellung eines neuen Gradle-Projekts, und den Gradle-Wrapper kann man ab sofort konfigurieren, ohne eine Extra-Task im Build-Skript hinzuzufügen. Im Umgang mit doppelten Dateien in Kopier- und Archivvorgängen gibt es neue Strategien, TestNG-Parameter sind in Testreports enthalten, und der OSGi- sowie C++-Support wurde verbessert. Dazu kommen 37 Bugfixes.

▶ http://www.gradle.org/downloads

Neues Big-Data-Framework von Microsoft basiert auf Hadoop YARN

Seit Kurzem ist YARN Teil des Hadoop-Projekts und ermöglicht es, mehrere Typen von Jobs in einem einzigen Cluster zu verwalten. Aber einige Typen haben spezielle Anforderungen, was Datenbewegungen, Task Monitoring und die Iteration vorheriger Ergebnisse betrifft. Genau hier soll Microsofts neue Big-Data-Technologie ansetzen, die auf dem neuen YARN-Ressourcen-Manager von Hadoop basiert.

Mithilfe des Frameworks REEF (kurz für Retainable Evaluator Execution Framework) können Nutzer Jobs erstellen, die ihren Status auch nach Abschluss behalten und ihre Daten von praktisch überall ziehen können. Dabei besteht das Framework aus zwei Teilen: Evaluators sind YARN-Container, die REEF Services enthalten, Activities bestehen aus Usercode, der im Evaluator läuft.

Microsoft Technical Fellow und CTO of Information Services Raghu Ramakrishnan kündigte am Montag auf der International Conference for Knowledge Mining and Data Discovery in Chicago an, dass Microsoft sein neues Framework in einem Monat Open Source bereitstellen möchte.

▶ http://bit.ly/1eEPxOZ

Hudson 3.1.0 RC mit neuem Team Concept

Version 3.0 war das erste Major-Release als Eclipse-Projekt. Nun steht mit dem Hudson 3.1.0 Release Candidate schon die nächste Version des Continuous Integration (CI) Servers in den Startlöchern.

Hudson 3.1.0 bringt ein neues Schlüsselfeature im Teamsupport, das den Namen *Team Concept* trägt. Es soll helfen, bei Nutzung desselben Hudson-Konflikte in den Teamaufgaben zu vermeiden. Auch die Skalierbarkeit soll mit der neuesten Version verbessert worden sein.

Alle Entwickler werden nun dazu aufgefordert, den Hudson 3.1.0 RC herunterzuladen, ausgiebig zu testen und anschließend ihr Feedback an die Macher weiterzugeben.

▶ http://bit.ly/14dG1fV

OSGi-Spezifikationen werden transparenter

Den Hauptteil ihrer Arbeitszeit verbringt die OSGi mit der Erstellung von Spezifikationen. Sind diese fertig, werden sie der Öffentlichkeit präsentiert. Das Ergebnis bekam man in der Vergangenheit aber trotz veröffentlichter Early Drafts kaum mit.

Der erste Schritt zur fertigen Spezifikation ist die Sammlung der dazugehörigen Anforderungen im RFP-Dokument, das dann in der passenden Expertengruppe diskutiert wird. Anschließend erfolgt die Entwicklung der Spezifikation im so genannten RFC-Dokument. Beide Dokumentarten sind künftig auf GitHub zu finden. Eine Liste aller aktiven Dokumente finden Sie im OSGi-Blog.

▶ http://bit.ly/1eupVEu

Spring Data Babbage mit erstem Release Candidate

Der Release Train Data Babbage des Spring Frameworks nähert sich mit seinem RC1 der finalen Veröffentlichung. Es handelt sich dabei um ein Bundle aus datenbankrelevanten Spring-Modulen:

- Spring Data Build 1.1
- Spring Data Commons 1.6
- Spring Data JPA 1.4
- Spring Data MongoDB 1.3
- Spring Data Neo4j 2.3
- Spring Data Gemfire
- Spring Data Solr 1.0

Diese befinden sich ihrerseits auch noch im RC-Stadium. Den Releaseinformationen auf GitHub lassen sich für alle Module noch To-dos entnehmen, die bis zum Release auf der Hauskonferenz SpringOne in vier Wochen abgearbeitet werden sollen. Doch schon der Release Candidate bietet etliche Neuerungen. So erlaubt das MongoDB-Modul die Nutzung des Aggregation-Frameworks der NoSQL-Datenbank. Polymorphe Querys lassen sich damit ebenfalls leichter verarbeiten. Beim JPA-Modul lassen sich künftig SpEL-Ausdrücke in händisch definierten Querys nutzen, das Handling von Entitäten wurde hier mit *@IdClass* vereinfacht und Data Binding wurde mit *@TemporalType* ermöglicht. Neo4j lernt *countBy()* und typensichere Query-Ausführung für Repositorys.

Fehler im Release Candidate können im Spring-Jira gemeldet werden.

▶ http://bit.ly/138Th6W

Autor des Monats

Wolfgang Weigend arbeitet als Sen. Leitender Systemberater bei der Oracle Deutschland B.V. & Co. KG. Er beschäftigt sich mit Java-Technologie und Architektur für unternehmensweite Anwendungsentwicklung.

Wie bist du zur Softwareentwicklung gekommen?

Bereits während meiner Schulzeit kam es zu einigen Berührungen mit Commodore, bevor ich die ersten Programme mit Pascal auf X86-kompatiblen PCs schrieb. Während meines Studiums entwickelte ich Software für Industriesteuerungen mit Assembler für die Motorola Prozessorfamilie 68000 und C. Ende der 80er Jahre arbeitete ich als freier Mitarbeiter einer Softwarefirma, die ein Netzwerkbetriebssystem entwickelte und betriebswirtschaftliche Software für den Mittelstand programmierte. Dabei hat mich das Potenzial der Softwareentwicklung fasziniert und nicht mehr losgelassen. In der Folge habe ich mich hauptsächlich mit C beschäftigt – bis endlich Java erfunden wurde.

Was ist für dich der schönste Aspekt in der Softwareentwicklung?

Der technologische Wandel spiegelt sich besonders in den Programmiersprachen und den damit verbundenen Regeln und Mustern wieder. Schöne Aspekte bestehen für mich aus eleganten und durchdachten Programmierlösungen, gepaart mit Lesbarkeit für das gesamte Entwicklungsteam.

Was ist für dich ein weniger schöner Aspekt?

Ständig wiederkehrenden Boilerplate-Code schreiben zu müssen und schlecht lesbaren Code vorzufinden, der den Codereview und die Codeoptimierung verzögert.

Wie und wann bist du auf Java gestoßen?

In meiner Zeit bei Texas Instruments Software schrieb ich immer noch C-Code für unternehmensweite Anwendungen. Durch Diskussionen über neue Programmiersprachen wurde ich erstmals im Jahr 1996 auf die Programmiersprache Java aufmerksam und wechselte daraufhin im Sommer 1997 direkt zu Sun Microsystems. Voller Überzeugung nahm ich bei einem Logistikkunden an einer Vorstudie zum unternehmensweiten Einsatz von Java teil. Bis heute begleite ich bei diesem Kunden zahlreiche Java-Projekte.

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

Die Vielfalt der Java-Landschaft birgt enorme Chancen für die gesamte Entwicklergemeinschaft, gemeinsam mit den großen IT-Herstellern an einer klaren und übersichtlichen Struktur mitzuwirken, von der die Anwender profitieren. Deshalb würde ich den JCP-Öffnungsprozess unter Einbezug aller Beteiligten noch enger mit der Java-Community und den Java User Groups verzahnen und stärkere Anreize aus der Industrie für die Entwickler schaffen, um technische Innovation zu maximieren und Fragmentierung zu minimieren.

Was ist zurzeit dein Lieblingsbuch?

Das Buch „Java Performance“ von Charlie Hunt und Binu John gehört nach wie vor zu meinen Lieblingsbüchern. Gerade weil es zu einem guten Nachschlagewerk geworden ist. Zudem lese ich wieder einmal das Buch „Das SEMCO System“ aus dem Jahre 1993, um den Wandel von Führungs- und Organisationstruktur der Firma Semco mit heutigen Firmen in Brasilien zu vergleichen.

Was machst du in deinem anderen Leben?

Um abzuschalten fahre ich im Urlaub gerne in die Berge. Ganz gleich, ob Wandern oder Skifahren, dort kann ich abschalten und die Zeit mit meinem Sohn genießen. An den Wochenenden fahre ich viel Rad und genieße die Zeit mit Freunden und Familie beim Grillen und schönen Gesprächen. Nicht zuletzt verbringe ich Zeit auf dem Sportplatz, um die Fußballspiele meines Sohnes zu verfolgen.

JUG-Kalender* Neues aus den User Groups

WER?	WAS?	WO?
JUG **Münster**	03.09.2013 – Graphen/NoSQL/Neo4j	http://jug-muenster.de
JUG **Karlsruhe**	04.09.2013 – Neo4j und die wunderbare Welt der Graphen	http://jug-ka.de
JUG **Schweiz**	05.09.2013 – Fahrenheit 451: Agile schadet der Dokumentation	http://jug.ch
JUG **Schweiz**	05.09.2013 – Java Cryptography for Beginners	http://jug.ch
JUG **Stuttgart**	05.09.2013 – Neo4j und die wunderbare Welt der Graphen	http://jugs.de
JUG **Augsburg**	06.09.2013 – Struts Hackathon: Essential Apache Struts 2	http://jug-augsburg.de
JUG **Ostfalen**	12.09.2013 – Abhängigkeiten managen mit Degraph	http://jug-ostfalen.de
JUG **Stuttgart**	16.09.2013 – ObjektForum Agilität und Softwarearchitektur	http://jugs.de
JUG **München**	16.09.2013 – JavaFX 8 – Das neue Java Standard UI Toolkit	http://jugm.de
JUG **Saxony**	19.09.2013 – Java Web Application Security	http://jugsaxony.org
JUG **Karlsruhe**	25.09.2013 – Eine leichtgewichtige BPM/SOA-Infrastruktur mit camunda BPM und Camel	http://jug-ka.de

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

Distributed Version Control with Git

von Lars Vogel

Lars Vogel, bekannt durch zahlreiche Tutorials, Vorträge und Bücher, beweist mit seinem neuesten Werk, dass er nicht nur mit Eclipse und Android bestens vertraut ist. In „Distributed Version Control with Git“ beschreibt er die Grundlagen und die Verwendung von Git. Und das wieder in dem für ihn üblichen leicht les- und nachvollziehbaren Tutorialstil.

Wer die im letzten Jahr veröffentlichte elektronische Version dieses Buchs bereits kennt, wird von der Neuauflage positiv überrascht sein. Während die erste Version das Thema Git noch sehr rudimentär beschrieben hat, ist die neue Version deutlich ausführlicher. Durch diverse Reviews und Anmerkungen von Kollegen und Freunden ist ein Buch entstanden, das sich nicht mehr hinter anderen Werken verstecken muss. Von den Grundlagen einer verteilten Versionskontrolle über die Verwendung des Systems bis hin zu typischen Workflows mit Git ist alles vertreten.

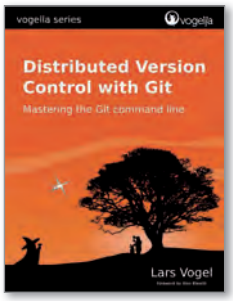
Für Einsteiger und Umsteiger und auch für User, die bereits mit Git gearbeitet, aber nie hundertprozentig verstanden haben, was sie tun, ist dieses Buch sehr zu empfehlen. Vor allem, da Themen ausführlich besprochen werden, die bei der Arbeit mit CVS oder SVN meistens zu Frust geführt haben: Branching und Merging.

In Git wird die Verwendung von Feature Branches fast schon gefordert, und durch die unterschiedlichen Möglichkeiten, die Branches wieder zusammenzuführen, ist dies auch relativ einfach. Vielleicht entsteht dadurch auch an einigen Stellen ein Unverständnis im direkten Vergleich von CVS und SVN zu Git. Dieses Unverständnis versucht der Autor in seinem Buch durch Grafiken, Beispiele und einfache Beschreibungen aufzulösen. Und in den meisten Fällen gelingt ihm das sehr gut. Hin und wieder muss man ein Kapitel zweimal lesen, was allerdings nicht am Autor, sondern eher am mangelnden Verständnis von verteilter Versionskontrolle liegt. Ist dieses Verständnis erst einmal vorhanden, möchte man nicht mehr zurück.

Für jeden, der sich mit Git beschäftigen will, kann dieses Buch uneingeschränkt empfohlen werden. Es trägt dazu bei, das Verständnis für verteilte Versionskontrolle und Git im Speziellen zu stärken. Lediglich der Bereich Troubleshooting fehlt, ist allerdings kaum noch nötig, wenn man die Konzepte erst einmal verstanden hat.

Für Käufer der ersten elektronischen Form wird im Übrigen ein kostenloses Update auf die neue Version angeboten, das man sich auf keinen Fall entgehen lassen sollte.

Dirk Fauth



Lars Vogel

Distributed Version Control with Git

Mastering the Git command line

176 Seiten,
23,53 Euro (Taschenbuch), 6,17 Euro (E-Book)
Vogella, 2013
ISBN 978-3943747065

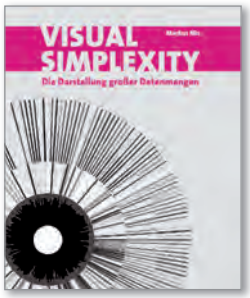
Anzeige

Visual Simplexity

■ *von Markus Nix*

„Daten werden zunehmend zu einem wertvollen Rohstoff – und Datenvisualisierung damit zum notwendigen Prozess, diesen Rohstoff zu veredeln und nutzbar zu machen.“ So startet Visual Simplexity, ein Buch, das in der Geschichte des entwickler.press-Verlags einmalig ist. Weil Print nicht tot ist, weil Daten so wunderschön und atemberaubend sein können, dass man sie sich zuhause am liebsten an die Wand hängen möchte. Ja, dass man mit ihnen ganze Museen füllen könnte.

Datenvisualisierung ist keine neue Wissenschaft, neu ist aber die Fülle an Daten, die unsere Welt täglich produziert. Und es werden immer mehr. Je komplexer das Feld Big Data wird, umso vielfältiger werden auch Datenvisualisierungen. Das war der Ausgangspunkt für Markus Nix, verschiedene Gestalter anzusprechen und in diesen Bildband aufzunehmen. Das Ergebnis sind 240 Seiten voller „bunter Bilder“, aufgeteilt in die Kategorien Mass Data Mess, Net Simplexity, Map Simplexity und Data Emotions. Von der globalen Lebenserwartung im Verhältnis zum Einkommen und der Geschichte der CNN-Website (**Abb. 1**) über die weltweit ansteckendsten Krankheiten (**Abb. 2**) bis hin zu den mit Geo-Tags gespeicherten Bildern auf Flickr und Picasa in New York (**Abb. 3**) oder der Visualisierung al-



Markus Nix

Visual Simplexity

Die Darstellung großer Datenmengen

240 Seiten, 49,90 Euro
entwickler.press, 2013
ISBN 978-3-86802-099-1

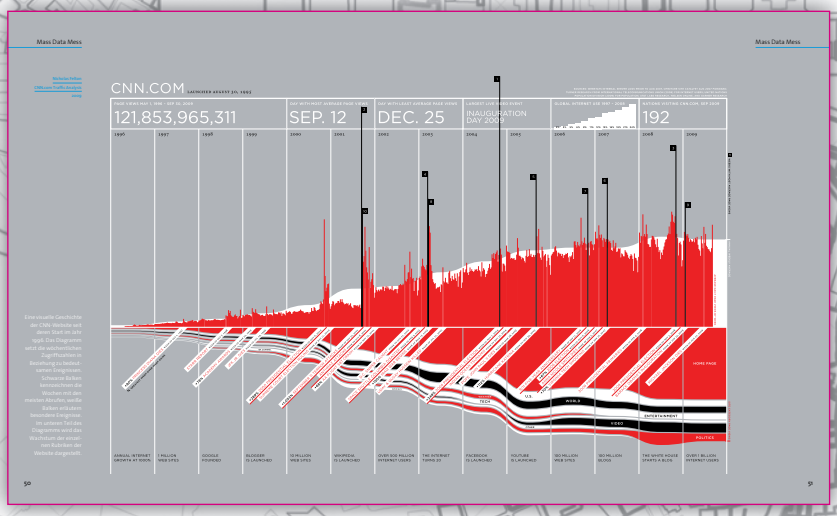


Abb. 1

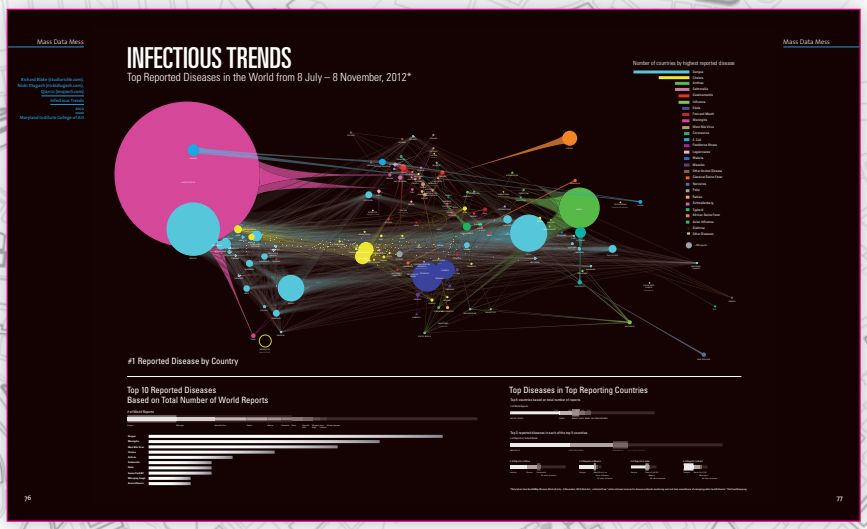


Abb. 2

ler Tweets und Flickr-Daten, die mit Geo-Tags versehen wurden (**Abb. 4**). Wie man sieht, muss eine Rezension über ein Datenvisualisierungsbuch jede Menge Bilder enthalten. Visual Simplexity ist ein Buch, das zum Stöbern einlädt und in vielerlei Hinsicht lehrreich ist. Nicht zuletzt für uns Journalisten, aber auch für Wissenschaftler, Designer, Künstler. Datenvisualisierung ist lehrreich, weil sie erklärt, wie die Dinge wirklich sind.

„Es liegt eine gewisse Arroganz darin zu glauben, dass alles in unserer Welt messbar, erklärbar, darstellbar ist.“ Wenn Daten uns keine wichtigen Informationen vermitteln wollen, dann werden sie zu Kunst. Weil sie beispielsweise den Klang von Beethovens Symphonie No. 5 visualisieren. Der rein künstlerische Teil ist im letzten Kapitel „Data Emotions“ abgebildet und bildet einen schönen Abschluss zu dieser bunten Sammlung von Fakten und Zahlen.

Wir sollten aber gerade vor dem Hintergrund der aktuellen Debatte um NSA und PRISM kritisch hinterfragen, wie sehr der Rohstoff „Daten“ ausgebeutet werden kann. Da regen manche Visualisierungen im Buch auch zum Nachdenken an, wie beispielsweise die von Nicholas Felton, der seit 2005 eine statistische Auswertung des eigenen Lebens vornimmt (Felton Annual Report). Alles wird erfasst: seine Reisen, Mahlzeiten, Schlafgewohnheiten. So weiß ich jetzt zum Beispiel, dass er bei einer Auswahl von Früchten am liebsten zur Mango und bei Fleisch am liebsten zu Schwein greift. Wenn er isst, dann meistens in seinem Apartment in Brooklyn und bevorzugt zusammen mit Olga. Ansonsten war er letztes Jahr in fünf verschiedenen Ländern zu Gast und in den USA bereiste er vierzehn Bundesstaaten. Die könnte ich Ihnen jetzt auch genau aufzählen, aber das würde zu weit führen.

Fazit: Warum das Buch so begeistert? Wegen der visuellen Präferenz des Menschen, die uns schon in die Wiege gelegt ist. Wir wollen Daten verstehen und mit anderen darüber diskutieren. In Zukunft wird das wichtiger und auch interessanter, wie der letzte Satz des Bildbands klarmacht: *„Nie zuvor hatte Statistik mehr Sex-Appeal“!*

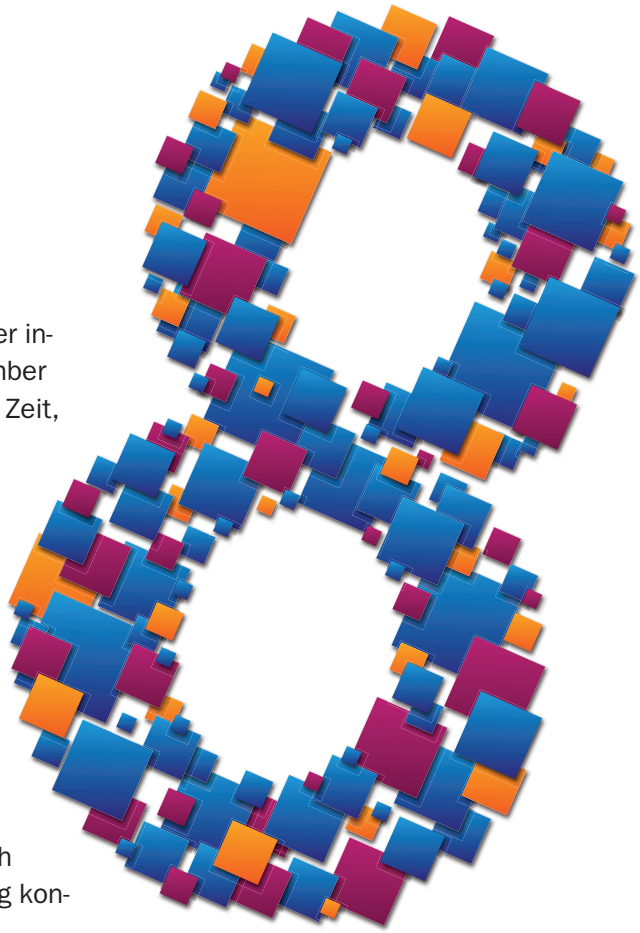
Claudia Fröhling

Abb. 3

Abb. 4

Funktionale Programmierung in Java

Effective Java: Im Zeichen der 8



Im April 2013 hat Oracle die Java-Community darüber informiert, dass der Java-8-Releasetermin von September 2013 auf März 2014 verschoben wird. Das gibt uns Zeit, uns auf die neuen Sprachmittel und die neuen JDK-Abstraktionen von Java 8 in Ruhe vorzubereiten. Mit diesem Beitrag beginnen wir deshalb eine Serie von Beiträgen zu den Neuerungen in Java 8. Unter anderem wird es neue Sprachelemente geben, die einen eher funktionalen Programmierstil in Java unterstützen werden. Dabei geht es um die so genannten Lambda-Ausdrücke. Ehe wir uns diese jedoch in einem der nachfolgenden Beiträge genauer ansehen, wollen wir uns zunächst damit befassen, was funktionale Programmierung generell ausmacht, wo man funktionale Ansätze praktisch anwenden kann und wie funktionale Programmierung konkret in Java aussehen wird.

von Klaus Kreft und Angelika Langer

w-jax Speaker

Als Java-Entwickler beschreiben wir, wie Objekte aussehen, wenn wir eine Klasse definieren, d. h. wir legen fest, welche Daten den Zustand eines Objekts beschreiben und welche Methoden die Fähigkeiten eines Objekts ausmachen. Wir erzeugen Objekte, wenn wir von den Klassen Instanzen bilden. Wir verändern Objekte beispielsweise, wenn wir die Felder ändern oder Methoden aufrufen, die dies tun. Wir reichen Objekte herum, wenn wir sie z. B. als Argumente an Methoden übergeben. Mit Objekten sind wir als Java-Entwickler bestens vertraut.

In funktionalen Sprachen (wie zum Beispiel Erlang oder Haskell) stehen nicht Objekte, sondern Funktionen im Vordergrund. Funktionen ähneln Methoden; sie repräsentieren ausführbare Funktionalität. Sowohl Methoden als auch Funktionen werden aufgerufen und ausgeführt. Funktionen in funktionalen Sprachen werden darüber hinaus herumgereicht. Man übergibt sie beispielsweise als Argumente an Operationen; diese Operationen können dann die übergebenen Funktionen in einem bestimmten Kontext aufrufen. Funktionen können auch als Return-Wert einer Operation zurückgegeben werden. Das heißt, in funktionalen Sprachen werden Funktionen herumgereicht wie Objekte in objektorientierten Sprachen. Dieses Prinzip des Herumreichens von Funktionen wird auch als „Code-as-Data“ bezeichnet.

Funktionen werden aber nicht nur übergeben und aufgerufen, sondern auch kombiniert und verkettet oder

manipuliert und verändert. Es gibt z. B. das so genannte *Currying* (benannt nach Haskell Brooks Curry), bei dem aus einer Funktion mit mehreren Argumenten durch Argument-Binding eine Funktion mit einem Argument gemacht wird. In einer reinen funktionalen Sprache (*pure functional language*) haben die Funktionen nicht einmal Seiteneffekte. Das heißt insbesondere, dass Funktionen keine Daten verändern, sondern bestenfalls neue Daten erzeugen.

Soviel zur Theorie. Was fängt man damit in der Praxis an? Kann man funktionale Prinzipien in Java überhaupt gebrauchen? Zur Illustration wollen wir uns ein Idiom ansehen, bei dem Funktionen eine wesentliche Rolle spielen und das auch in Java recht nützlich sein kann. Es geht um das *Execute-Around-Method*-Pattern.

Das Execute-Around-Method-Pattern

Bei dem Execute-Around-Method-Pattern [1], [2] geht es darum, strukturell ähnlichen Code so zu zerlegen, dass die immer wiederkehrende, identische Struktur herausgelöst und in eine Hilfsmethode ausgelagert wird. Der Teil, der in dieser Struktur variiert, wird an die Hilfsmethode übergeben und in dieser Hilfsmethode, eingebettet in die Struktur, an der richtigen Stelle aufgerufen. Beispiele für solche wiederkehrenden Strukturen gibt es viele:

Verwendung von Ressourcen: Wenn eine Ressource verwendet wird, dann ergibt sich oft eine wiederkehrende Struktur, nämlich:

acquire resource
use resource
release resource

Das Anfordern und Freigeben der Ressource ist oft identisch, aber die Benutzung dazwischen variiert. Ein Beispiel für solche Ressourcen sind die expliziten Locks wie zum Beispiel das *ReentrantLock*. Hier ist die immer gleiche Struktur, die sich bei der Benutzung von expliziten Locks ergibt:

```
lock.lock();
try {
  .. critical region ...
} finally {
  lock.unlock();
}
```

Das Anfordern und Freigeben des Locks ist immer gleich, nur die Anweisungen dazwischen variieren.

Exception Handling: Wenn Operationen aus einem bestimmten Framework verwendet werden, dann wer-

Anzeige

fen sie oft die gleichen Exceptions, die immer gleich behandelt werden:

```
try {
  ... invoke operations ...
} catch (ExceptionType_1 e1) { ... }
catch (ExceptionType_2 e2) { ... }
catch (ExceptionType_3 e3) { ... }
```

Die *catch*-Klauseln sind immer gleich, aber die im *try*-Block aufgerufenen Operationen sind unterschiedlich.

Iterierung: Es wird ein Iterator angefordert und aufs jeweils nächste Element in einer Sequenz weitergeschaltet, bis das letzte Element erreicht ist:

```
Iterator iter = seq.iterator();
while (iter.hasNext()) {
  Object elem = iter.next();
  ... use element ...
}
```

Die Handhabung des Iterators ist immer gleich; lediglich die Verwendung des jeweiligen Elements variiert.

Beim Execute-Around-Method-Pattern wird der gemeinsame, wiederkehrende Teil in eine Hilfsmethode ausgelagert. Der veränderliche Teil wird als Argument an die Hilfsmethode übergeben. Wir wollen dies am Beispiel der Iteration demonstrieren. Wir definieren eine Hilfsmethode *forEach*:

```
public class Utilities {
  public static <E> void forEach(Iterable<E> seq, Consumer<E> block) {
    Iterator<E> iter = seq.iterator();
    while (iter.hasNext()) {
      E elem = iter.next();
      block.accept(elem);
    }
  }
}
```

Die Hilfsmethode *forEach* enthält den strukturell wiederkehrenden Teil, nämlich das Anfordern des Iterators, die Abfrage auf das Ende der Sequenz und das Weiterschalten des Iterators auf das jeweils nächste Element. Die Verwendung des Elements ist der sich unterscheidende Teil. Er wird von außen an die Hilfsmethode übergeben und in der Hilfsmethode an entsprechender Stelle aufgerufen. Die Hilfsmethode *forEach* bekommt deshalb als Argumente die Sequenz der Elemente, auf der man iterieren will, und die Funktionalität, die während der Iterierung auf jedes Element in der Sequenz angewandt werden soll. Für die Beschreibung der Funktionalität, die auf jedes Element angewandt wird, definieren wir ein Interface *Consumer*:

```
public interface Consumer<T> {
  void accept(T t);
}
```

Dieses Interface gibt es tatsächlich in Java 8 im Package *java.util.function*. Mit dieser Zerlegung in die Hilfsmethode mit dem strukturell identischen Teil und das Interface mit dem variierenden Teil brauchen wir die Iterierung nicht mehr redundant hinschreiben. Hier ist ein Benutzungsbeispiel. Wir wollen alle Elemente aus einer Liste von Zahlen ausgeben. Herkömmlich sieht es so aus:

```
List<Integer> numbers = new ArrayList<>();
... populate list ...
Iterator iter = numbers.iterator();
while (iter.hasNext()) {
  Integer elem = iter.next();
  System.out.println(elem);
}
```

Gemäß Execute-Around-Method-Pattern sieht es so aus:

```
List<Integer> numbers = new ArrayList<>();
... populate list ...
Utilities.forEach(numbers, new Consumer<Integer>() {
  public void accept(Integer elem) {
    System.out.println(elem);
  }
});
```

Nun mag man sich fragen, was an der Execute-Around-Method-Version besser sein soll als an der guten alten Iterierung per Schleife. Mit klassischen Java-Mitteln, so wie sie uns in Java 7 zur Verfügung stehen, ist nichts gewonnen. Man muss eine Implementierung des *Consumer*-Interface definieren, um den Consumer an die Hilfsmethode *forEach* zu übergeben, und das ist selbst unter Verwendung von anonymen inneren Klassen noch recht umständlich. Genau diese syntaktische Umständlichkeit wird in Java 8 mit den Lambda-Ausdrücken verschwinden [3], [4]. In Java 8 mit einem Lambda-Ausdruck sieht es viel eleganter aus:

```
List<Integer> numbers = new ArrayList<>();
... populate list ...
Utilities.forEach(numbers, e -> System.out.println(e));
```

Es geht auch noch eleganter mithilfe von Methodenreferenzen – einem weiteren neuen Sprachmittel in Java 8. So sieht es dann in Java 8 mit einer Methodenreferenz aus:

```
List<Integer> numbers = new ArrayList<>();
... populate list ...
Utilities.forEach(numbers, System::println);
```

Auf die Syntax von Lambda-Ausdrücken wie *e -> System.out.println(e)* und Methodenreferenzen wie *System.out::println* wollen wir in diesem Beitrag nicht näher eingehen. Das besprechen wir im nächsten Artikel der Serie im Detail. Aber auch ohne große Erläuterung kann man intuitiv verstehen, dass der Lambda-Ausdruck so

Anzeige

etwas Ähnliches wie eine Funktion ist. Er nimmt ein Argument mit Namen *e*, dessen Typ sich der Compiler selbst überlegen kann. Augenscheinlich soll es ein Integer aus der Liste sein. Dieses Argument *e* wird per *System.out.println*-Methode ausgegeben. Auf diese Weise wird die *forEach*-Methode alle Zahlen in der Liste *numbers* mit *println* auf *System.out* ausgeben.

Die Methodenreferenz ist auch selbsterklärend. *System.out::println* ist die *println*-Methode von *System.out*. In der *forEach*-Methode sollen also alle Elemente mit *println* nach *System.out* ausgegeben werden.

Im JDK 8 wird es solche Hilfsmethoden wie *forEach* geben. Sie sind dann aber nicht in irgendwelchen Utility-Klassen definiert, sondern die Collections selbst sind erweitert worden. In Java 8 hat jede Collection aus dem *java.util*-Package eine *forEach*-Methode, und zwar erbt sie diese von ihrem Superinterface *Iterable*. Das *Iterable*-Interface ist für Java 8 erweitert worden und sieht in Java 8 so aus:

```
public interface Iterable<T> {
  Iterator<T> iterator();

  default void forEach(Consumer<? super T> action) {
    for (T t : this) {
      action.accept(t);
    }
  }
}
```

Das *Iterable*-Interface hat zusätzlich zur *iterator*-Methode, die es schon immer hatte, eine *forEach*-Methode bekommen. Um die existierenden Interfaces im JDK wie oben gezeigt erweitern zu können, hat man mit Java 8 die so genannten *Default*-Methoden erfunden. Darauf werden wir in einem der Folgebeiträge genauer eingehen. Hier nur ganz kurz: Normalerweise kann man ein Interface nicht problemlos erweitern. Wenn man Methoden hinzufügt, dann müssen alle abgeleiteten Klassen diese Methode implementieren. Andernfalls gibt es Fehlermeldungen bei der Kompilierung. Die Default-Methoden sind nun Methoden, die eine Implementierung haben. Das heißt, sie sind nicht abstrakt und müssen von den abgeleiteten Klassen auch nicht implementiert werden. Alle Klassen, die keine Implementierung für die neue zusätzliche Methode haben, erben einfach die Default-Implementierung aus dem Interface. Auf diese Weise kann man existierende Interfaces erweitern, ohne die abgeleiteten Klassen ändern zu müssen. Die *forEach*-Methode im *Iterable*-Interface ist eine solche Default-Methode. Sie hat eine Implementierung. Sie verwendet in der Implementierung die *for-each*-Schleife, die es seit Java 5 gibt und die intern einen Iterator verwendet. Die *forEach*-Methode im Interface *Iterable* entspricht der *forEach*-Methode aus unserer Utilities-Klasse, mit dem kleinen Unterschied, dass sie das erste Argument nicht braucht, weil sie als nicht statische Methode der Collection ohnehin über die *this*-Referenz auf die Collection zugreifen kann. Das Beispiel von oben sieht in Java 8 letztendlich so aus:

```
List<Integer> numbers = new ArrayList<>();
... populate list ...
numbers.forEach(System.out::println);
```

Die herkömmliche Art der Iterierung mit einem expliziten Iterator bezeichnet man im Übrigen als externe Iterierung, im Gegensatz zur internen Iterierung in der *forEach*-Methode einer Collection [5]. Bei der externen Iterierung wird der Iterator an den externen Benutzer einer Collection gegeben und der Benutzer bestimmt, wie er den Iterator verwendet, um alle Elemente der Sequenz zu besuchen. Bei der internen Iterierung bestimmt die Collection selbst, wie sie in ihrer *forEach*-Methode alle Elemente besucht. Das kann sie mit einem Iterator machen, so wie wir es im Beispiel gesehen haben. Sie kann es aber auch ganz anders machen, zum Beispiel parallel mit vielen Threads statt sequenziell mit nur einem Thread.

Genau die parallele Ausführung von Operationen wie *forEach* ist der wesentliche Grund dafür, dass man die Sprache um Lambda-Ausdrücke erweitert hat. Eines der Ziele in Java 8 ist die bessere Unterstützung von Parallelverarbeitung. Deshalb wird es neue Abstraktionen im JDK-Collection-Framework geben, nämlich so genannte Streams. Diese Streams haben Operationen wie *forEach* (oder auch *sort*, *filter* etc.) mit interner Iterierung, die wahlweise sequenziell oder parallel ausgeführt werden können. Die Streams und ihr umfangreiches API werden wir uns in einem der nachfolgenden Beiträge im Detail ansehen.

Funktionale Programmierung in Java

In dem oben geschilderten Beispiel der internen Iterierung bzw. des Execute-Around-Method-Patterns sieht man typische Elemente der funktionalen Programmierung. Beispielsweise sieht man das „Code-as-Data"-Prinzip: Die *forEach*-Methode benötigt als Argument eine Funktion, die auf alle Elemente der Collection angewandt werden soll. Es wird zwar streng genommen ein Objekt als Argument übergeben, aber dieses Objekt repräsentiert Funktionalität. Das einzige, was an dem Objekt interessant ist, ist die eine Methode, die es mitbringt und die auf alle Elemente der Sequenz angewandt werden soll. In diesem Sinne ist das Consumer-Argument der *forEach*-Methode eine Funktion. In Java 7 muss dafür umständlich eine anonyme innere Klasse definiert werden. In Java 8 mit den Lambda-Ausdrücken und Methodenreferenzen sieht die Funktionalität optisch und syntaktisch so aus, wie man sich eine Funktion vorstellt.

Allerdings gehen die Spracherweiterungen in Java 8 nicht so weit, dass aus Java nun eine funktionale Sprache wird. Viele Dinge, die es in funktionalen Sprachen gibt, wird es in Java nicht geben. Genauer gesagt: Es wird sie zunächst einmal nicht geben. Es ist nicht ausgeschlossen, dass es in Java 9, 10 oder später weiterge-

hende Unterstützung für funktionale Programmierung in Java geben wird. Die Sprachdesigner haben darauf geachtet, dass der Weg für zukünftige Erweiterungen nicht verbaut wird. Aber für Java 8 wurde versucht, die Spracherweiterung erst einmal minimalistisch zu gestalten und nur das hinzuzufügen, was für die Weiterentwicklung des JDK zwingend erforderlich ist.

Beispielsweise hat Java keine Funktionstypen. Die Sprachdesigner haben es bewusst vermieden, das Typsystem von Java umzukrempeln und um eine völlig neue Kategorie von Typ zu erweitern. Stattdessen hat man sich überlegt, wie der Compiler es bewerkstelligen kann, als Typ für Lambda-Ausdrücke und Methodenreferenzen ganz „normale" Typen (d. h. Klassen oder Interfaces) zu verwenden.

Target Typing und SAM Types: Sehen wir uns die Einbettung von Lambda-Ausdrücken und Methodenreferenzen ins Java-Typsystem anhand unseres Beispiels an. Noch einmal das Beispiel von oben:

```
List<Integer> numbers = new ArrayList<>();
... populate list ...
numbers.forEach(System.out::println);
```

Der Compiler geht prinzipiell so vor: Er schaut sich den Kontext an, in dem ein Lambda-Ausdruck oder eine Methodenreferenz steht, überlegt, welcher Typ von Objekt an dieser Stelle benötigt wird, und deduziert daraus den Typ für den Lambda-Ausdruck oder die Methodenreferenz.

In unserem Beispiel findet er die Methodenreferenz *System.out::println* als Argument im Aufruf der *forEach*-Methode. Der Compiler schaut sich also den deklarierten Argumenttyp der *forEach*-Methode an. Weil es die *forEach*-Methode einer *List<Integer>* ist, stellt der Compiler fest, dass für den Methodenaufruf ein Objekt vom Typ *Consumer<Integer>* benötigt wird. *Consumer<Integer>* ist ein Interface mit einer einzigen abstrakten Methode, nämlich der *accept*-Methode. Nun prüft der Compiler, ob die Signatur der *accept*-Methode kompatibel zur Methodenreferenz *System.out::println* ist. Die *accept*-Methode von *Consumer<Integer>* nimmt ein Argument vom Typ Integer, gibt *void* zurück und wirft keine *checked*-Exceptions. Das passt zu unserer Methodenreferenz *System.out::println*. Die *println*-Methode ist überladen und unter all den vielen *println*-Varianten gibt es eine, die ein Argument vom Typ *Integer* nimmt, *void* zurückgibt und keine *checked*-Exceptions wirft. Das heißt, die *accept*-Methode aus dem *Consumer<Integer>*-Interface hat dieselbe Signatur wie die Methodenreferenz *System.out::println*. Der Compiler schließt daraus, dass die Referenz *System.out::println* in diesem Kontext vom Typ *Consumer<Integer>* ist.

Diesen Prozess der Deduktion des Typs eines Lambda-Ausdrucks oder einer Methodenreferenz wird als *Target Typing* bezeichnet, weil dabei der Zieltyp (*Target Type*) für den Ausdruck oder die Referenz aus dem Kontext

Anzeige

ermittelt wird. Für Lambda-Ausdrücke funktioniert das Target Typing ganz analog.

Interfaces wie *Consumer* mit einer einzigen abstrakten Methode heißen übrigens *Functional Interface Types* (oder auch *SAM* Types, wobei *SAM* für *Single Abstract Method* steht). Die *SAM*-Typen spielen beim Target Typing eine wesentliche Rolle. Sie sind nämlich die einzigen Typen, die als Zieltypen in Frage kommen.

Über diesen Trick mit den *SAM*-Typen und der Deduktion eines kontextabhängigen Zieltyps konnte es vermieden werden, gravierend in das Typsystem von Java einzugreifen. Deshalb gibt es in Java – anders als in funktionalen Sprachen – keine spezielle Kategorie von Typen, mit denen man Funktionen oder Funktionssignaturen beschreiben könnte.

Seiteneffekte: Das Fehlen von echten Funktionstypen ist aber nur eine Eigenart, die funktionale Programmierung in Java von funktionalen Sprachen unterscheidet. In reinen funktionalen Sprachen sind die Funktionen stets frei von Seiteneffekten. Insbesondere modifiziert eine reine Funktion keine Daten, sondern produziert ein Ergebnis. Das ist in Java natürlich anders. Es gibt in Java gar keine Möglichkeit, eine Funktion daran zu hindern, Felder oder Variablen zu modifizieren.

In unserem Beispiel haben unsere Lambda-Ausdrücke und Methodenreferenzen zwar nichts modifiziert, aber einen Seiteneffekt, nämlich die Ausgabe auf *System.out*, haben sie dennoch produziert. Für diese Funktionen macht es einen Unterschied, ob sie mehrfach aufgerufen werden oder in welcher Reihenfolge sie aufgerufen werden, denn es hat Einfluss auf die Ausgabe. Bei einer reinen Funktion, die keinerlei Seiteneffekte hat, wäre es völlig egal, wie oft und in welcher Reihenfolge sie ausgeführt wird. Eine reine Funktion wäre beispielsweise folgender Lambda-Ausdruck:

```
IntPredicate isEven = (int i) -> { return i%2==0; };
```

Dabei ist *IntPredicate* ein SAM-Typ aus dem Package *java.util.function* mit einer einzigen abstrakten Methode, die so aussieht: *boolean test(int value)*.

Dieser Lambda-Ausdruck *(int i) -> { return i%2==0; }* nimmt einen *int*-Wert und liefert *true* zurück, wenn es eine gerade Zahl ist, andernfalls *false*. Hier wird überhaupt kein Seiteneffekt ausgelöst. Es wird einfach nur ein Wert genommen und ein boolesches Ergebnis zurückgeliefert. Diese Funktion kann aufgerufen werden, so oft man will und in jeder beliebigen Reihenfolge. Es macht überhaupt keinen Unterschied. Hier zum Kontrast ein Lambda-Ausdruck, der Modifikationen macht:

```
List<Point> points = new ArrayList<>();
... populate list ...
points.forEach(p -> { p.x = 0; });
```

Hier werden alle Elemente der Sequenz modifiziert; es sind Point-Objekte, deren x-Koordinate in dem Lambda-Ausdruck geändert wird. Das ist im Sinne der funktionalen Programmierung schlechter Stil, kann aber in Java nicht verhindert werden. Wenn das Argument einer Funktion eine Referenz auf ein veränderliches Objekt ist, dann kann die Funktion Modifikationen machen. Java hat einfach keine Sprachmittel, um solche Modifikationen zu verhindern.

Solche Lambda-Ausdrücke sind u. U. problematisch. Wir werden in nachfolgenden Beiträgen erläutern, warum. Aber bereits hier sollte schon klar sein, dass man mit modifizierenden Lambda-Ausdrücken leicht Fehler machen kann. Hier ist eine solche Fehlersituation:

```
List<Point> points = new ArrayList<>();
... populate list ...
points.forEach(p -> points.add(new Point(0,p.y)));
```

In dem Lambda-Ausdruck werden während der Iterierung neue Elemente in die Collection eingefügt. Das scheitert zur Laufzeit mit einer *ConcurrentModificationException*.

Zusammenfassung und Ausblick

Java 8 wird neue Sprachmittel haben, die in gewissem Umfang funktionale Programmierung in Java unterstützen. Die betreffenden Sprachmittel sind Lambda-Ausdrücke und Methodenreferenzen. Wir haben uns in diesem Beitrag das Execute-Around-Method-Pattern sowie die interne Iterierung als Spezialfälle davon angesehen. Beide sind Idiome, die von den neuen Sprachmitteln profitieren. Mit Lambda-Ausdrücken und Methodenreferenzen sind sie wesentlich einfacher zu benutzen. In Java 8 werden die Collections interne Iterierung unterstützen. Genau für diese Erweiterungen der Collections im JDK sind die neuen Sprachmittel entwickelt worden. Im nächsten Beitrag sehen wir uns die Lambda-Ausdrücke und Methodenreferenzen genauer an.

Angelika Langer arbeitet selbstständig als Trainer mit einem eigenen Curriculum von Java- und C++-Kursen. Kontakt: http://www.AngelikaLanger.com.

Klaus Kreft arbeitet selbstständig als Consultant und Trainer. Kontakt: http://www.AngelikaLanger.com.

Links & Literatur

[1] Buschmann, Frank; Henney, Kevlin; Schmidt, Douglas C.: „Pattern-Oriented Software Architecture“, Wiley, 2007

[2] Rising, Linda: „The Pattern Almanac“, Addison-Wesley, 2000

[3] Project Lambda: http://openjdk.java.net/projects/lambda

[4] Langer, Angelika; Kreft, Klaus: „Lambda Tutorial“: http://www.AngelikaLanger.com/Lambdas/Lambdas.html

[5] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: „Design Patterns“, Addison-Wesley, 1994

Design by Contract, aber mit Prüfung zur Compile-Zeit

Das Ende der Toleranz

Es geht hier nicht um die Lebensweise von Menschen, sondern um Programmfehler, die nicht mehr toleriert werden sollen. In Java sind Datentypen ein Weg zu sicheren Programmen, aber die Möglichkeiten sind begrenzt. Mit dem hier beschriebenen Codegenerator ist erheblich mehr möglich und ein Hauch von Haskell oder Scala kommt auf.

von Heiner Kücker

Das Prinzip „Design by Contract“ beruht auf einem Vertrag, der beim Aufruf einer Methode einzuhalten ist. Dafür gibt es den JSR-303 Bean Validation und Frameworks wie Oval [1]. Am Beispiel einer Vorbedingung kann man dies auch mit einfachem Java realisieren (Listing 1). Leider werden die Contracts nur zur Laufzeit geprüft. Nur durch einen Test können sie zur Wirkung gebracht werden (außer zur Echtlaufzeit, das wäre zu spät). Der Compiler weiß nichts von den Contracts, er kann nicht prüfen, ob diese beim Aufruf eingehalten werden.

Der Java-Compiler

In Java kann man keine eigenen primitiven Typen anlegen, also kann man den Java-Compiler nur über Klassen zur Mithilfe bewegen. Ein Array-Index darf nicht kleiner 0 sein. Bauen wir uns also eine kleine Klasse, die diese Bedingung absichert (Listing 2). Im Konstruktor wird die Einhaltung des Contracts geprüft, es ist nicht möglich, ein ungültiges Objekt zu erzeugen.

Unser aufgerufener Code sieht nun wie folgt aus:

```
public X get(IntGtOrEqZero index) {
  ... weiterer Code ...
}
```

An der aufrufenden Codestelle tritt jetzt ein Compilerfehler auf. Diesen korrigieren wir durch Eintragen unseres Typs und die dadurch auftretenden Compilerfehler wiederum, bis der *int*-Wert literal hingeschrieben dasteht oder aus einer unsicheren Quelle (Eingabefeld, Datei, Datenbank) kommt. Hier rufen wir den Konstruktor auf. Falls der Wert die Bedingung verletzt, wird die *IllegalArgumentException* geworfen. Dies tritt nun wesentlich näher an der Verursacherstelle (an welcher der Wert entsteht/auftaucht) auf und nicht erst beim Zugriff auf das Array. Dumm nur, wenn das Constraint-Objekt (*IntGtOrEqZero*) null ist, dies muss durch entsprechende Sorgfalt (niemals mit null belegen) oder einen ergänzenden Checker ausgeschlossen werden.

Der Constraint-Code-Generator setzt generierte Java-Klassen ein, um die Datenflussanalyse des Java-Compilers auszunutzen. Alle weiteren Features sind davon beeinflusst. Alternativ könnte man den Code mithilfe eines AST-Parsers analysieren und die Constraints über Annotationen oder qualifizierte Kommentare (Javadoc) ausdrücken.

Die Macht sei mit dir

Das Java-Typsystem ist eine Art Schaltersystem. Man kann durch Vererbung und das Implementieren von Interfaces immer weitere Eigenschaften (Rechte) hinzuschalten, aber keine verbieten (Listing 3). *B* kann nicht durch das Java-Typsystem ausgeschlossen werden.

Listing 1

```
public X get(int index){
  if (index<0){
    throw new IllegalArgumentException(
    "index lesser zero: " +
    index );
  }
  ... weiterer Code ...
}
```

Artikelserie

Teil 1: Der Constraint-Code-Generator

Teil 2: Includes, Excludes, Perfomance und Historie

Mein Codegenerator erzeugt Constraint-Java-Klassen aus Prädikaten, die mit AND, OR, NOT sowie XOR verknüpft werden können. Durch die expressive Power der booleschen Ausdrücke kann man alle fachlichen Aspekte für den Compiler prüfbar ausdrücken.

Hierzu ein ergänzendes Beispiel, eine Firma hat vier Vertriebsregionen, Nord, Süd, Ost und West. Rabatte werden je nach Artikelgruppe, Vertriebsregion und dem Umsatz des Kunden vergeben. Wie soll man im Java-Typsystem ausdrücken, dass in einer Vertriebsregion ein bestimmter Artikel nicht vertrieben werden darf oder im Gegensatz dazu einen besonderen Rabatt erhält, um den Umsatz anzukurbeln? Mit Expressions ist dies kein Problem.

Prädikate

Der Codegenerator beruht auf Prädikaten. Die Prädikatsklasse implementiert die abstrakte Oberklasse aller Prädikate mit einer *testImpl*-Methode, welche ein boolesches Ergebnis zurückgibt (siehe Code am Ende des Absatzes). Es gibt keine Spezialsyntax in Kommentaren oder Annotationen, die Prädikate sind in Java geschrieben. Dadurch sind alle sprachlichen Möglichkeiten von Java nutzbar, egal ob es sich um Null-Prüfungen, Wertevergleiche oder Mindest-Array-Größen handelt. Die Prädikate können mit den Operanden *AND*, *OR*, *NOT* sowie *XOR* verknüpft werden, wobei *XOR* nicht auf zwei Operanden beschränkt ist, sondern beliebig viele Operanden mit *XOR* verknüpft werden dürfen.

```
abstract public class PrimitivPredicate<CT>{
  abstract public boolean testImpl(CT contextObj);
}
```

Das Kontextobjekt

Der *testImpl*-Methode der Prädikate wird ein über Generics festgelegter Parameter übergeben, der die zu prüfenden Werte/Objekte enthält. Der Typ des Kontextobjekts kann eine primitive Klasse wie *Integer* oder *String*, aber auch eine komplexe fachliche Klasse sein. Falls ein Prädikat mehrere verknüpfte Objekte/Werte prüfen soll, müssen diese in diesem einen Kontextobjekt verpackt werden, die Verwendung von Prädikaten mit mehreren Parametern ist nicht möglich. Wichtig ist die Unveränderlichkeit der Kontextobjekte und ihrer Member, weil eine einmal erfolgte Prüfung sonst korrumpiert werden könnte.

Generierte Constraint-Java-Klassen

Die Constraints des Codegenerators werden als boolesche Expressions definiert, die aus Prädikaten bestehen. Aus der jeweiligen Constraint-Expression werden Java-

Listing 2

```
public class IntGtOrEqZero {
  public final int value;
  public IntGtOrEqZero(final int value ) {
    if (value<0) {
      throw new IllegalArgumentException(
      "value lesser zero: " +
      value );
    }
  }
}
```

Listing 3

```
class A {
}

interface B {
}

void myMethod(A aButNotB) {
  if (aButNotB instanceof B)
    throw new IllegalArgumentException(
    "B not allowed");
  }
  ...
}
```

Prädikate als Atome der Constraints

Der Aufbau der Constraints aus Prädikaten ist alternativlos und auch ein Glücksgriff, lösen sich dadurch doch gleich mehrere Probleme. In einem Prädikat könnte folgende Prüfung erfolgen:

```
x % 4 == 0
```

Es ist offensichtlich, dass in diesem Fall auch immer

```
x % 2 == 0
```

erfüllt ist. Nun könnte man das einem Compiler beibringen, 2 ist eine Primzahl und ein ganzzahliger Teiler von 4. Aber was ist, wenn man mit Wurzeln rechnet, soll das der Compiler auch kennen? Wenn man als Bedingung eine selbstgeschriebene Methode verwendet, wird es für den Compiler geradezu unmöglich, diese zu verstehen.

Durch das Prädikat muss der Codegenerator nur wissen, ob das Prädikat erfüllt wird oder nicht. Außerdem kann man dem Codegenerator mitteilen, ob ein anderes Prädikat stets erfüllt ist, wenn das aktuelle Prädikat erfüllt ist (einfache Implikation [2]) oder ob sich zwei Prädikate wiedersprechen (Ausschluss der Konjunktion [2]).

Weiter ist es wichtig, dass ein Prädikat zur Laufzeit zur Verfügung steht und geprüft werden kann:

```
if ( predicate.test( ctxObj ) ) {
  ... sicherer Block ...
}
```

Im Block des *if* ist die Einhaltung des Prädikats abgesichert, außerhalb nicht.

Klassen generiert. Der Name der Java-Klasse wird anhand der zugrunde liegenden Expression gebildet:

```
and( new A() , new B() )
```

wird zu

```
ANDB_A_B_ANDE
```

wobei *ANDB* und *ANDE* für Beginn und Ende eines *AND* stehen. Ähnlich ist es beim *OR* und *XOR*. *NOT_* ist das Präfix für negierte Prädikate oder Klammerabschnitte.

Klassennamen von Constraints, die nur aus einem einzigen Prädikat bestehen, bekommen den Postfix Constraint, damit Constraint und Prädikat nicht den gleichen Java-Klassennamen haben, was zur Verwirrung beim Import führen könnte. So wird zum Beispiel Prädikat A zu AConstraint.

Die generierten Constraint-Java-Klassen sind final und haben keine Oberklasse (außer Object, kann man schließlich nicht vermeiden). Sonstige Vererbung/Implementierung oder die Nutzung der Java-Typkompatibilität zur Oberklasse/Interface ist nicht vorgesehen.

Jede generierte Constraint-Java-Klasse besitzt wiederum eine *test*-Methode zum Prüfen der Einhaltung des Constraints und einen Konstruktor, in welchem eine eventuelle Constraint-Verletzung mit einer *IllegalArgumentException* abgewehrt wird.

Evolution der Typsicherheit

In einer Anwendung existiert ein Status, zum Beispiel Auftragsstatus. Da dieser in der Datenbank gespeichert wird, ist sein Typ wahrscheinlich *String*, eventuell auch *int*. Eine erste Verbesserung ist die Verwendung eines Enum. In der Persistenzschicht muss dafür gesorgt werden, dass die Umwandlung vom Datenbankstring zum Enum und umgekehrt erfolgt. Leider kann man dem Compiler nicht sagen, dass bei einem Methodenaufruf nur ein bestimmter Status erlaubt ist, alle Status haben den gleichen Typ. Um dies zu lösen, könnte man statt Enum eine abstrakte Oberklasse mit finalen inneren abgeleiteten Klassen als Singletons verwenden, so hat jeder Status eine eigene Klasse. Will man jetzt aber bei einem Methodenaufruf nur den Status A oder B erlauben, so muss man beiden ein gemeinsames Interface geben und dieses als Parametertyp verwenden. Soll jeder Status außer C erlaubt sein, wird es schon ziemlich haarig. Ein weiteres Problem ist, wenn eine Methode einen Auftragsdatensatz (Entität) mit einem bestimmten Status erfordert. Kaum anzunehmen, dass jemand eine Java-Klasse *AuftragsEntityInStatusA* anlegt. Mit Constraint-Expressions ist dies alles mühelos ausdrückbar.

Anzeige

Kondensieren von Expressions

Das Prinzip der Typsicherheit in Scala oder Haskell beruht vereinfacht gesagt darauf, dass Expressions, die prinzipiell auch zur Laufzeit abgearbeitet werden könnten, bereits zur Compile-Zeit abgearbeitet werden. Dafür müssen die Informationen bereits zur Compile-Zeit vorliegen und die Laufzeitsprache kann nicht verwendet werden, weil der Compiler die Expressions abarbeitet. Eine Alternative ist die Prüfung zum Zeitpunkt der Initialisierung, des Ladens der entsprechenden Klasse oder zum Zeitpunkt der erstmaligen Benutzung. Dadurch steht die Laufzeitsprache zum Prüfen zur Verfügung und der Code muss nicht erst durch einen vollständigen Test ins Leben gerufen werden. So kann man zum Beispiel einen Zustandsautomaten darauf prüfen, dass jeder außer dem finalen Status verlassen werden kann. Der Compiler kann natürlich nicht die Situation zur Laufzeit eines Programms kennen. Nehmen wir als Beispiel einen Plotter. Es gibt einen horizontalen und einen vertikalen Antriebsmotor. Jedem Antriebsmotor kann ein Befehl zum vorwärts- oder rückwärtsfahren gegeben werden. Wird einem Antriebsmotor ein Fahrbefehl gegeben, der über den Fahrbereich hinaus führt, wird eine Exception geworfen. Wenn nun der aufgerufene Code dem aufrufenden Code sagt: „Bevor du mir einen Fahrbefehl gibst, musst du geprüft haben, ob noch Platz in der angegebenen Richtung ist“, und der aufrufende Code muss dies in einem Constraint verpacken, dann ist das Risiko, das Prüfen zu vergessen, wesentlich geringer. Ohne diese Information bekommt der Aufrufer die Exception präsentiert und sagt zum aufgerufenen Code: „Hättest du doch gleich sagen können, dass ich diese Bedingung prüfen muss!“. Mit dem Constraint kann der aufgerufene Code dem aufrufenden Code dies von vornherein mitteilen: „Gut, dass wir drüber gesprochen haben.“. Constraints verbessern also die Kommunikation im Code.

Quellcodeentfernung

Hier handelt es sich um ein von mir verwendetes empirisches Maß für die Entfernung abhängiger Codestellen. Diese können wenige Zeilen entfernt sein, sodass sie auf einen Bildschirm passen, sie können sich aber auch in unterschiedlichen Dateien oder Verzeichnissen befinden. Wenn kein Quellcode vorhanden ist (JAR-Datei), kann man vielleicht von einer unendlichen Quellcodeentfernung sprechen. Je weiter zwei abhängige Codestellen entfernt sind, desto schwerer ist der meist nicht explizit ausformulierte Vertrag zwischen ihnen zu entdecken. Je größer die Quellcodeentfernung ist, desto unwahrscheinlicher ist auch, dass ein Test diesen Vertrag absichert. Eine geringe Quellcodeentfernung nützt aber auch nichts, wenn sich niemand den betroffenen Bereich ansieht, weil eine entfernte Codeänderung den Vertrag bricht, der Code aber trotzdem compilierbar bleibt. Hierbei ist die Entfernung der geänderten Codestelle vom betroffenen Bereich maßgeblich.

Der Codegenerator

Nachdem wir unsere Prädikate definiert haben, können wir unseren konkreten Codegenerator von der Klasse *AbstractConstraintCodeGenerator* ableiten (Code unter [3]). Ein paar Festlegungen wie Packages, Pfade und Member-Namen sind erforderlich. Durch das Double-Brace-Initialization-Pattern bilden wir einen Initialisierungs-Body, in welchem *add*-, *and*-, *or*-, *not*- und *xor*-Methoden zur Verfügung stehen. Mit diesen Methoden können wir unsere Prädikate mit den zur Verfügung stehenden Methoden *AND*, *OR*, *NOT* sowie *XOR* verknüpfen und dem Codegenerator bekannt machen. Diese Methoden arbeiten mit Java 5 Varags, benötigen also keine Array- oder Collection-Parameter, um beliebig viele Prädikate zu verknüpfen. Der Codegenerator wird mit einer einfachen *main*-Methode aufgerufen und generiert direkt in das Source-Verzeichnis des jeweiligen Projekts oder in ein spezielles Verzeichnis. Pfade nochmal kontrollieren, Verzeichnis für die zu generierenden Klassen anlegen, den Parameter *deleteUnusedConstraintJavaFiles* erst mal auf *false* setzen, auch eine Datensicherung kann nicht schaden, und schon kann das Generieren losgehen.

Die Codegenerator-Suite

Außer in den winzigsten Beispielen werden in einer Applikation mehrere unterschiedliche Kontextobjekte benötigt. Für jede Kontextobjektklasse wird ein eigener Codegenerator als Singleton angelegt. Diese Codegeneratoren werden in einer Suite zusammengefasst und gemeinsam gestartet. Die Singleton-Eigenschaft ist wichtig, da sich die verschiedenen Codegeneratoren gegenseitig über gewünschte Ziel-Constraints für Konvertierungen von einem Kontextobjekt zum anderen informieren. An den Prädikaten wird über ein spezielles Interface *ToOtherContextObjTypeConvertablePredicate* festgelegt, ob und in welches Zielprädikat ein Prädikat konvertiert werden kann. Besteht eine Expression vollständig aus konvertierbaren Prädikaten, kann die gesamte Expression konvertiert werden.

Constraints im Einsatz

Wie werden die generierten Constraints nun verwendet? Der natürliche Lebensraum der generierten Java-Constraint-Klassen ist der *Methoden*-Parameter. Der Kopf der Methoden ist der Ort, an dem die aufrufende Seite das Constraint einhalten muss und sich die aufgerufene Seite auf das Constraint verlassen kann. Demzufolge kann sich der Aufrufer darauf verlassen, dass die aufgerufene Methode mit dem Constraint klar kommt, also den gesamten erlaubten Wertebereich verarbeiten kann, während sich die aufgerufene Methode darauf verlassen kann, nur erlaubte Werte übergeben zu bekommen. Eine Methode gliedert nicht nur den Quellcode, sondern verkleinert auch den vom Compiler zu analysierenden Zustandsraum. Das explizite Ausformulieren eines Constraints an einer Methodengrenze vereinfacht dem Compiler oder dem jeweiligen anderen statischen Checker das

Analysieren des Codes. Da die Methodendeklaration/-definition und der Methodenaufruf im Quellcode immer mehr oder weniger weit entfernt sind, wirkt hier die Sicherheit der statischen Prüfung.

Sicherer und unsicherer Bereich

Schützen wir jetzt eine Methode mit einem *int*-Parameter dahingehend, dass der Wert des *int*-Parameters nicht negativ sein darf (siehe Code im Abschnitt „Der Java-Compiler"). Die Einführung dieses Constraints wirkt viral, in jeder weiteren aufrufenden Schicht muss jetzt das Constraint statt des eventuell negativen *int*-Werts übergeben werden. Doch irgendwo bekommen wir einen Wert, den wir nicht unter Kontrolle haben, vom Benutzer, der Datenbank, Netzwerk oder Dateisystem. Diesen Wert kennt der Compiler nicht. Zum Erwerben des Constraints müssen wir den Konstruktor der Constraint-Klasse aufrufen, eine Constraint-Instanz erzeugen. Der Konstruktor bekommt das Kontextobjekt, zum Beispiel ein Integer, übergeben. Nach dem erfolgreichen Aufruf des Konstruktors befinden wir uns im sicheren Bereich, falls der übergebene Wert ungültig ist, erhalten wir eine *IllegalArgumentException*. Vorher befinden wir uns im unsicheren Bereich.

Problem mit null

Die generierten Constraints sind Klassen, in Java kann man nun mal keine eigenen primitiven Typen anlegen. Da die Werte von Klassen Objekte sind, könnte man das Constraint einfach durch das Übergeben von *null* erfüllen. Dies muss man durch Disziplin, einen ergänzenden Checker oder einen ergänzenden Test verhindern.

Constraint-Kompatibilität

Nehmen wir ein Constraint *and(new A() , new B())* an. Dieses ist ohne Verletzung einer Bedingung umwandelbar in *new A()* beziehungsweise *new B()*. Dies stellt der Codegenerator ohne weiteres Zutun fest und erzeugt die entsprechenden Konvertierungsmethoden *convertToAConstraint()* und *convertToBConstraint()*. Die Voraussetzung für die Kompatibilität ist die Erfüllung der einfachen Implikation, in der Literatur meist als Doppelpfeil (Fat Arrow) dargestellt:

- (A and B) => A
- (A and B) => B
- A => (A or B)
- B => (A or B)

Die Java-Typkompatibilität zur Oberklasse/Interface wird wegen des nicht ausdrückbaren *NOT* nicht verwendet, die Konvertierungsmethoden sind die einzige Möglichkeit der Constraint-Kompatibilität.

Spezialisierung von Constraints

Haben wir in einer Klasse Methoden mit spezielleren Constraints, die wiederum Methoden mit allgemeineren Constraints aufrufen, reichen die oben beschriebenen

Der natürliche Lebensraum der Java-Constraint-Klassen ist der Methoden-Parameter.

Konvertierungsmethoden aus. Es kann aber vorkommen, dass in einer Methode über eine *if-else*-Kaskade oder *switch-case* ein Wertebereiche bzw. Rechte/Rollen des ursprünglichen Constraints/Werts spezialisiert werden muss, die aufrufende Methode wurde sozusagen mehrfach verwendet. Im Constraint-Code-Generator gibt es dafür die SwitchDefinition, die das Generieren einer abstrakten nicht statischen inneren Klasse in der Constraint-Klasse auslöst. Die Benutzung einer abstrakten Klasse mit einer zu implementierenden Methode für jeden Zweig sichert ab, dass kein Zweig bei der Implementierung vergessen wird. Dies löst das Problem, dass man bei einem Java-*switch-case* oder einer *if-else*-Kaskade nicht über den Compiler absichern kann, dass kein Zweig vergessen wird. Die generierte abstrakte innere Klasse wird mit

```
constraint.new XxxSwitch(){
  ... zu implementierende Methodenrümpfe ...
}
```

angelegt, in einer üblichen IDE wird das erforderliche Gerüst automatisch erzeugt.

Die booleschen Ausprägungen der einzelnen Zweige dürfen sich bezüglich ihrer Wertebelegung (Model) nicht überlappen (müssen disjoint sein), damit nicht zufällig ein vor einem anderen Zweig geprüfter Zweig gewinnt. Dabei werden die definierten Includes und Excludes (siehe weiter unten) zur Vereinfachung des neuen spezialisierten Constraints benutzt.

Falls nicht für jede boolesche Ausprägung des ursprünglichen Constraints ein gültiger Zweig existiert, wird eine zu implementierende (abstrakte) *caseDefault*-Methode erzeugt. Für diese wird kein Constraint-Parameter erzeugt.

Das verfluchte if

Ohne *if* oder ein äquivalentes Konstrukt wäre eine Programmiersprache nicht Turing-vollständig, könnte nicht funktionieren. Leider entzieht sich das *if* meist der statischen Prüfung, die bei Klassen und Methodensignaturen möglich ist. Ein *if* befindet sich im dynamischen Teil des Codes und kann nur über einen Test ins Leben gerufen werden.

Ein Problem beim *if* ist die meist unausgesprochene (nicht explizite) Abhängigkeit des Funktionierens von der Reihenfolge der *if-elseif*-Zweige. Ursache ist die fehlende Prüfung auf gegenseitigen Ausschluss der Bedingungen bereits zur Compile-Zeit.

Ein weiteres Problem kann das fehlende finale *else* eines *if* bzw. einer *if-elseif*-Kaskade sein. Darf dieser *else*-Zweig nicht vorkommen, ist es ein Standardfall oder wurde er einfach vergessen? Ich werfe meist explizit eine Exception, wenn der finale *else*-Zweig nie durchlaufen werden darf.

Tun wir mal so, als hätten wir die Aufgabe, einen *if*-Zweig statisch zu prüfen. Erst mal ist das *if* ein reiner Zeichensalat, den kein Programm analysieren kann. Grob kann man das *if* in die Bedingung und den Body unterteilen. Es wird schon etwas klarer. In der Bedingung, also zwischen den runden Klammern, kann nun Mathematisches, Fachliches oder anderes Geheimnisvolles stehen. Wenn nun die Bedingung aus Prädikaten besteht, die durch boolesche Operatoren verknüpft sind, kriegen wir unser Prüfprogramm eventuell noch zum Laufen, mit Boolean-Expressions können wir umgehen. Was in den Prädikaten passiert, ist nicht im Scope unseres Prüfprogamms (Kasten: „Prädikate als Atome der Constraints"). Wenn wir nun statt des *if* den Constraint-Switch verwenden, bekommen wir einiges geschenkt. Die einzelnen Zweige des ursprünglichen Constraints müssen disjoint sein, die Reihenfolge spielt keine Rolle mehr. Wenn keine zu implementierende *caseDefault*-Methode generiert wurde, gibt es auch keine Möglichkeit, den *else*-Zweig zu vergessen. Den einzelnen Zweigmethoden wird ein spezialisiertes Constraint übergeben, auf welches sie sich 100-prozentig verlassen können.

Nun bleibt noch der Body unseres *if* zu analysieren. Wenn er aus einem einzigen Methodenaufruf besteht, der einen bestimmten Vertrag einhält, kann unser Prüfprogramm auch diesen Teil checken.

Ein Rezept zur Absicherung eines vorhandenen Programms ist das schrittweise Ersetzen aller *if*-Verzweigungen durch ConstraintSwitches. Dann geht dem Entwickler bei einer Änderung mit Sicherheit kein *if*-Verzweiger durch die Lappen, der aufgrund der Änderung angepasst werden muss.

Im zweiten Teil dieser Serie beschäftigen wir uns unter anderem mit Includes und Excludes und werfen einen Blick auf Historie und Zukunft des Codegenerators.

Heiner Kücker ist als Elektroingenieur ein typischer Nebeneinsteiger in der IT-Branche und arbeitet seit dem Jahr 2000 mit Java. Neben der Fehlervermeidung in Java-Programmen interessiert er sich für moderne Programmiertechniken wie künstliche Intelligenz und funktionale Programmierung.

Links & Literatur

[1] OVal – the object validation framework for Java™ 5 or later: http://oval.sourceforge.net/userguide.html

[2] Wikipedia-Seite „Aussagenlogik": http://de.wikipedia.org/wiki/Aussagenlogik

[3] www.heinerkuecker.de/ConstraintCodeGenerator.html

ConstraintSwitch oder Polymorphie

Der ConstraintSwitch ist die Umsetzung fachlicher Kategorien (Prädikate) in frei kombinierbarer Verknüpfung (Expressions) als statisches strukturelles Bedingungsgerüst (keine Daten, kein Verhalten) einer Applikation. Eine Alternative ist die Verwendung von Polymorphie als Ersatz für *if*- oder *switch*-Kaskaden mit Constraints für Hierarchien (Vererbung-/Interface-Implementierung) und Verhalten (Zwang zur Methodenimplementierung). Vererbung ist eine Realisierungsmöglichkeit des von der Clean-Code-Bewegung propagierten *open close principle* (neben Strategie und Observer, noch unübersichtlicher). Ich habe es mit Polymorphie versucht und es entstanden ähnliche parallele Hierarchien mit Querverknüpfungen (ZentraleXxPage, FilialeXxPage – ZentraleXxAction, FilialeXxAction). Irgendwo, meist in einer Factory, ist doch mindestens ein *if*-Verteiler notwendig, die Vererbungs-Polymorphie leidet unter der Dominanz einer Hierarchie und einem starren Strukturgefängnis. Komposition statt Vererbung mildert dieses Problem nur ab (verkleinert den Scope mehrerer Hierarchien). In der Realität eignet sich die Polymorphie nur für technische (wiederverwendbarer Kern) und kaum für fachliche Belange, höchstens triviale Übungsbeispiele. Irgendwo taucht ein *instanceof-if* oder eine *UnsupportedOperationException* auf. Mancher Modellierer/Entwickler wird sich über diese Codestellen geärgert haben und glaubte, beim nächsten Projekt klappt es besser. Es wird nicht besser klappen, weil es nicht möglich ist.

Ich habe fachliche Hierarchien gesehen, die durchaus sinnvoll und berechtigt waren, aber das Liskovsche Substitutionsprinzip verletzten, das eine weitere strukturelle Fessel ist. Besonders wiedersinnig ist die Hierarchie in XML-Schema, welche eine Hierarchie von Datenbehältern ist, die dann von einem Framework wie JAXB auf eine Vererbungshierarchie abgebildet wird, welche ursprünglich als Verhaltenshierarchie vorgesehen ist. Die Strukturierung der Bedingungen eines fachlichen Modells als Expressions aus Prädikaten ist hier eine echte Befreiung. Zusammengehörige Prädikate können dabei durchaus eine gemeinsame Oberklasse zur Redundanzverminderung und Gliederung haben. Aus den Datenbehältern (Entitäten) werden die Kontextobjekte der Prädikate. Natürlich muss dieses Gerüst aus Datenbehältern und Bedingungen (Constraint-Expression) mit Verhalten gefüllt werden, was in dieser sicheren Struktur kein Problem ist. Änderungen in den Bedingungen führen zu Compilerfehlern an allen betroffenen Codestellen. Statt unsicherer unkontrollierter Erweiterbarkeit (fachlich ist unkontrollierbare Erweiterbarkeit oft unerwünscht, man denke an das Verbot, einen Hund in der Mikrowelle zu trocknen) ziehen Änderungen an den Bedingungen Compilerfehler an allen betroffenen Codestellen nach sich, die sicher gegen Vergessen und Flüchtigkeitsfehler angepasst werden können. Insofern sind Constraints nicht nur ein Thema für den Codierer, sondern auch ein Architektur- und Fachthema.

Anzeige

Anzeige

Anzeige

Anzeige

Nashorn ist die neue JVM-basierte JavaScript-Implementierung im JDK 8

DER WEG ZUR POLYGLOTTEN VM

Seit Mitte Dezember 2012 ist das Projekt Nashorn im OpenJDK angekommen, mit dem Ziel, eine zu hundert Prozent in Java geschriebene, schlanke und performanceoptimierte JavaScript-Engine direkt auf der JVM zu implementieren. Den Entwicklern soll Nashorn ermöglichen, unter Berücksichtigung von JSR 223 (Scripting for the Java Platform), JavaScript in Java-Anwendungen einzubinden und eigene JavaScript-Anwendungen zu erstellen, die bisher *jrunscript* verwendet haben und künftig das neue Command-line-Werkzeug *jjs* (JavaScript in Java) im JDK 8 benutzen können.

von Marcus Lagergren und Wolfgang Weigend

W-JAX Speaker

© iStockphoto.com/gazza30 © iStockphoto.com/filo

Das Nashorn-Projektdesign nutzt den Vorteil einer komplett neuen Codebasis und fokussiert sich auf moderne Technologien für native JVMs mittels Method Handles und invokedynamic-APIs, wie im JSR 292 (Supporting Dynamically Typed Languages on the Java Platform) beschrieben.

Motivation für das Projekt Nashorn

Die fünfzehn Jahre alte Rhino-Engine ist nicht mehr zeitgemäß und bietet hauptsächlich Plug-in-Fähigkeit über das Package *javax.script*. Das bisherige Scripting-API besteht aus Interfaces und Klassen zur Definition der Java-Scripting-Engine und liefert ein Framework zur Verwendung dieser Schnittstellen und Klassen innerhalb von Java-Anwendungen. Für Rhino bleibt trotz seines Alters und seiner Schwächen das einzige Argument übrig, als Bindeglied den Zugriff zur großen und umfassenden JDK-Bibliothek herzustellen. Die Motivation für Nashorn, als Bestandteil von JDK 8, liegt in der Unterstützung von JavaScript, die Java-/JavaScript-Integration und die JVM für dynamisch typisierte Skriptsprachen attraktiv zu machen. Dies beruht auf den bereits in Java SE 7 eingeführten invokedynamic-Instruktionen, bei denen es darum geht, den Bytecode *invokedynamic* zu verwenden, um Methodenaufrufe von Skriptsprachen in der JVM erheblich zu beschleunigen. Das Package *java.lang.invoke* enthält dynamische Sprachunterstützung, die direkt von den Java-Core-Klassenbibliotheken und der VM bereitgestellt wird. JavaScript oder andere JVM-Sprachen können von den Vorteilen der darunterliegenden JVM, inklusive Memory-Management, Codeoptimierung und den Grundeigenschaften vom JDK profitieren, wie in **Abbildung 1** dargestellt.

Die Nashorn-Engine ist hundert Prozent ECMAScript-kompatibel und hat alle notwendigen Kompatibilitätstests für ECMAScript 5.1 bestanden. Auch die nächste Version ECMAScript 6.0 wird bereits vom Entwicklungsteam genau verfolgt, um eine konforme Implementierung der Skriptsprache mit Unterstützung aller Typen, Werte, Objekte, Funktionen, Programmsyntax und Semantik in Nashorn sicherzustellen.

Nashorn im Leistungsvergleich

Der direkte Vergleich von Nashorn und Rhino, ermittelt durch die Octane-Benchmark-Suite, einer Micro-Benchmark-Suite im Single-Thread-Modus, zeigt auf einem iMac mit OS X und Intel-Quad-Core-Prozessor i7 (3.4 GHz) eine deutliche Leistungsverbesserung der Nashorn-Engine (**Abb. 2**). Die Darstellung vom Graph wurde dabei auf die Rhino-Leistungsfähigkeit normalisiert, und es ist erkennbar, dass sich die Leistungsklasse von Nashorn in einer anderen Größenordnung befindet. Dennoch sollten die Benchmark-Ergebnisse nicht überbewertet werden, weil der Octane-Benchmark nicht den optimalen Leistungsvergleich zur Bestimmung der JavaScript-Geschwindigkeit liefert, sondern hauptsächlich die Bytecode-Ausführungsgeschwindigkeit ermittelt wird. Jedoch sind einige Testläufe interessant, beispielsweise Splay, weil dabei ein großer Anteil der Leistung vom GC-Durchsatz innerhalb der JVM hergeleitet wurde und die Ergebnisse im Vergleich zur Google-V8-JavaScript-Engine gleichwertig sind. Die vielversprechenden Leistungsmerkmale von Nashorn sind unter anderem invokedynamic zu verdanken und zeigen auf, dass sich die

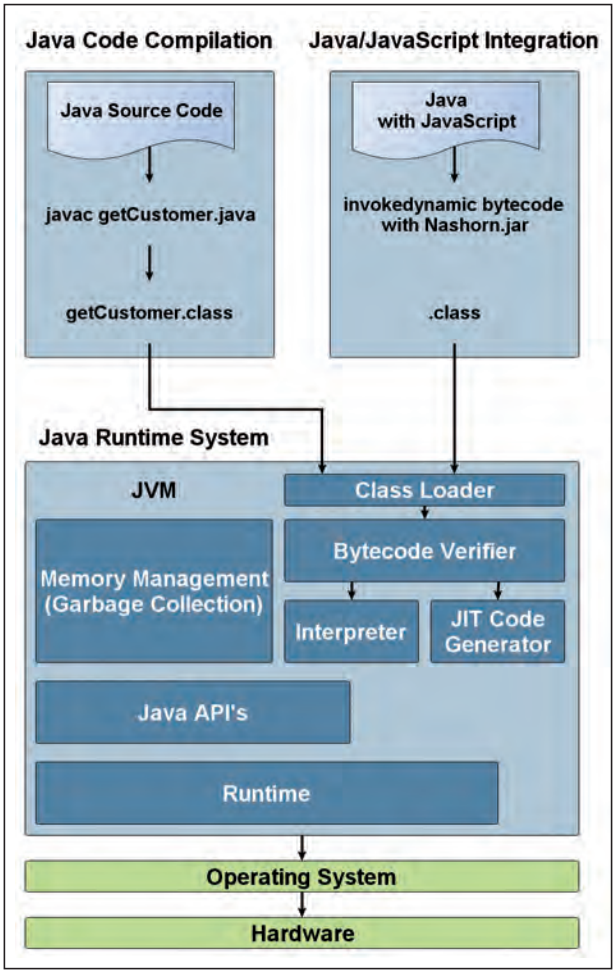


Abb. 1: Java-Bytecode in der JVM

Mehr zum Thema

Interessieren Sie sich für polyglotte Programmierung? Dann sollten Sie auch unseren Heftschwerpunkt „invokedynamic – Die Zukunft der JVM ist polyglott" lesen. Zu finden im Java Magazin 11.2012: https://jaxenter.de/magazines/JavaMagazin112012.

Video

Auf unserem YouTube-Channel finden Sie ein Kurzinterview zwischen unserer Redakteurin Claudia Fröhling und Marcus Lagergren von Oracle zum Thema Nashorn und invokedynamic: http://bit.ly/15J7cFu.

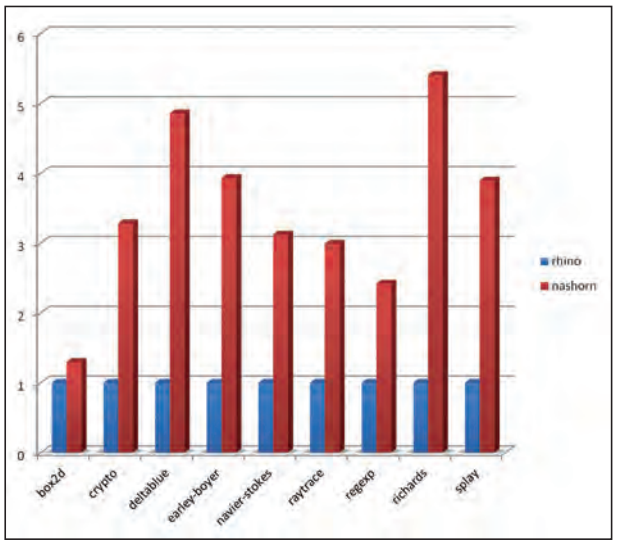


Abb. 2: Rhino vs. Nashorn im Vergleich

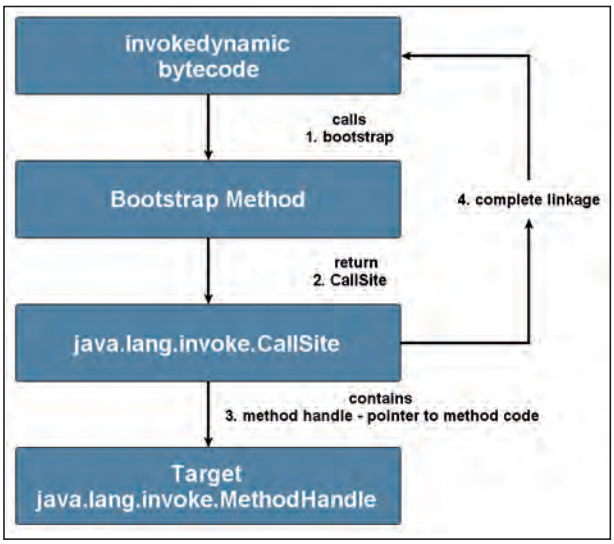


Abb. 3: invokedynamic-Bytecode und „MethodHandle“

JVM in Richtung einer polyglotten Laufzeitumgebung bewegt. Die Herausforderung für Nashorn liegt in der Einbettung von Scripting in die Java-Plattform (JSR 223) mit einer akzeptablen Geschwindigkeit.

JSR 292 invokedynamic als Basis für Nashorn

Der Trend, dass einige neue dynamisch typisierte Sprachen (Non-Java-Programmiersprachen) die JVM als Laufzeitumgebung verwenden, wurde in erheblichem Maß durch invokedynamic gefördert und führte dazu, dass sich eine höhere Performance vom Bytecode erzielen lässt, als zuvor möglich war. Bisher war der Bytecode vorrangig für das Java-Programmiermodell mit strenger Typisierung und anderen nativen Java-Konzepten ausgelegt, was sich jetzt durch invokedynamic ändert und den Vorteil einer tauglichen Implementierung von dynamischen Sprachen auf der JVM ermöglicht.

Erstmalig in der Geschichte der JVM-Spezifikation wurde ein neuer Bytecode eingeführt, der eine abgeschwächte Typisierung erlaubt, aber bezogen auf Primitive- und Objekttypen ist der Bytecode nach wie vor streng typisiert. Mit der Bytecode-Instruktion invokedynamic steht ein Methodenaufruf zur Verfügung, der eine dynamische Verknüpfung vom Aufrufer (Call Site) zum Empfänger (Call Receiver) erlaubt. Damit kann eine Klasse, die gerade einen Methodenaufruf ausführt, mit der Klasse und Methode verknüpft werden, die diesen Aufruf zur Laufzeit erhält. Die anderen JVM-Bytecode-Instruktionen zum Methodenaufruf, wie invokevirtual, bekommen die Target-Type-Information direkt in die kompilierte Klassendatei hineinkodiert. Ähnlich wie beim Funktions-Pointer erhält man mit invokedynamic die Möglichkeit, Aufrufe abzusetzen, ohne die üblichen Java-Sprachprüfungen durchführen zu müssen, komplett benutzerdefinierte Verknüpfungen herzustellen und den direkten Austausch von Methoden-Call-Targets. Die Verknüpfung von Call Site und Call Receiver erfolgt nicht durch *javac*, sondern wird dem Entwick-

Anzeige

ler abverlangt, wenn er den Bytecode für dynamische Sprachen erzeugt. In **Abbildung 3** wird das Konzept von Call Site *java.lang.invoke.CallSite* dargestellt. Über invokedynamic wird die Bootstrap-Methode aufgerufen (Schritt 1) und gibt die *CallSite* zurück *return new CallSite();* (Schritt 2), mit genau einem invokedynamic-Aufruf pro Call Site. Die *CallSite* enthält den *MethodHandle* mit dem Aufruf der tatsächlichen Target-Methodenimplementierung (Schritt 3), und die Call Site ist komplett mit dem invokedynamic-Bytecode verknüpft (Schritt 4). Der *MethodHandle* ist das Target, das auch über *getTarget/setTarget* verändert werden kann (Listing 1).

Das Konzept vom *MethodHandle* (*java.lang.invoke.MethodHandle*) ist in Listing 2 dargestellt und beschreibt einen eigenen „Funktions-Pointer". Die Programmierlogik könnte mit Guards *c = if (guard) a(); else b();* abgesichert sein sowie Parametertransformationen und -Bindings beinhalten (Listing 3). Ein *SwitchPoint* (*java.lang.invoke.SwitchPoint*) enthält eine Funktion zweier *MethodHandles* a und b sowie eine Invalidierung durch Überschreiben der *CallSite* a nach b. Ein *SwitchPoint* ist ein Objekt, um Zustandsübergänge anderen Threads bekannt zu machen. Betrachtet man die Performance von invokedynamic in der JVM, so kennt diese das *CallSite* Target und kann Inlining mit üblichen adaptiven Laufzeitannahmen verwenden, wie beispielsweise „Guard benutzen", und kommt damit ohne komplizierte Mechanismen aus. Theoretisch sollte sich daraus eine passable Performance herleiten lassen; doch wechseln die *CallSite* Targets im Code zu oft, wird der Programmierer durch die JVM-Code-Deoptimierung wieder abgestraft.

Listing 1: java.lang.invoke.CallSite

```
20: invokedynamic #97,0
// InvokeDynamic #0:"func":(Ljava/lang/Object;
    Ljava/lang/Object;)V

public static CallSite bootstrap(
    final MethodHandles.Lookup lookup,
    final String name,
    final MethodType type,
    Object... callsiteSpecificArgs) {

    // look up the target MethodHandle in some way
    MethodHandle target = f(
        name,
        callSiteSpecificArgs);
        // do something
    // instantiate the callsite
    CallSite cs = new MutableCallSite(target);
    // do something

    return cs;
}
```

Listing 2: „MethodHandle" erzeugen

```
MethodType mt = MethodType.methodType(String.class, char.class, char.class);
MethodHandle mh = lookup.findVirtual(String.class, "replace", mt);

String s = (String)mh.invokeExact("daddy", 'd', 'n');
assert "nanny".equals(s) : s;
```

Listing 3: „MethodHandle" mit „Guard" und „SwitchPoint"

```
MethodHandle add =
    MethodHandles.guardWithTest(
        isInteger,
        addInt
        addDouble);

SwitchPoint sp = new SwitchPoint();
MethodHandle add = sp.guardWithTest(
    addInt,
    addDouble);

// do something

if (notInts()) {
    sp.invalidate();
}
```

Implementierung dynamischer Sprachen auf der JVM

Eine Vielzahl von dynamischen Sprachen, wie beispielsweise Jython, JRuby, Groovy, Scala, Kotlin und JavaScript, können die JVM als Ablaufumgebung benutzen, weil sie die Sprachentwickler für die JVM implementiert haben. Die JVM kennt nur Bytecode, keine Sprachsyntax, und der Bytecode ist komplett plattformunabhängig. Damit müssen nur die Call Sites umgeschrieben und die bestehenden Annahmen geändert werden, aber ungeachtet dessen, liegt die große Schwierigkeit bei den JVM-Typen. Das Problem mit sich ändernden Annahmen während der Laufzeit kann von größerem Umfang sein als die üblichen Problemstellungen in einem Java-Programm, bei dem z. B. ein Feld gelöscht wurde. Dann müssen alle betroffenen Stellen geändert werden, bei denen die Annahme zutrifft, dass das Objektlayout zu viele Felder besitzt. Im Gegensatz dazu würde man die Berechnungsergebnisse auf Grundlage mathematischer Funktionalität, wie *Math.cos()* und *Math.sin(),* auf keinen Fall ändern und immer einen konstanten Wert zurückliefern, obwohl sie in JavaScript änderbar sind. Auch wenn dies niemand ändert, muss es doch abgeprüft werden. Setzt man die strenge Typisierung von Java in Bezug mit schwacher Typisierung anderer Sprachen und betrachtet das Problem von schwacher Typisierung genauer und nimmt man unten stehende Java-Methode und schaut sich über *javap –c* den Bytecode an, stellt man fest, dass in Java int-Typen zur Laufzeit bekannt sind (siehe Opcodes und Maschinen-Stack). Soll ein Type Double zur Addition kompiliert werden, ist dies nicht zulässig (Kasten: „Problem von schwacher Typisierung").

Problem von schwacher Typisierung

```
int sum(int a, int b) {
  return a + b;
}
```

```
Method int sum(int,int)
0  iload_1    // Push value of local variable 1 (a)
1  iload_2    // Push value of local variable 2 (b)
2  iadd       // Add; leave int result on operand stack
3  ireturn    // Return int result
```

```
iload_1    // stack: a
iload_2    // stack: a, b
iadd       // stack: (a+b)
ireturn    // stack:
```

Betrachtet man stattdessen eine JavaScript-Funktion, so ist nicht bekannt, ob a und b tatsächlich addiert werden können, da eine Typdefinition fehlt und der JavaScript-Operator + auch zur Verknüpfung von Zeichenketten verwendet werden kann. Bei String-Verarbeitung wird dementsprechend keine Berechnung durchgeführt. Die Definition vom Additonsoperator (+) ist in ECMA 262 so beschrieben, dass + zur Konkatenation von Strings oder zur numerischen Addition verwendet wird.

```
function sum(a, b) {
  return a + b;
}
```

```
???
???
???
???
```

In JavaScript können a und b als Integer behandelt werden, die in 32 Bit passen, aber die Addition kann zum Überlauf führen und das Ergebnis in ein Long oder als Double gespeichert werden. Deshalb ist für die JVM eine JavaScript-„Number“ nur schwer greifbar. Damit ist bei schwacher Typisierung die Typherleitung zur Laufzeit auch zu schwach.

Betrachtet man das Axiom vom adaptiven Laufzeitverhalten, so gleicht es oftmals einem Spiel, bei dem der schlechtere und langsamere Fall wahrscheinlich nicht eintritt. Falls er doch eintreten sollte, wird die Strafe zwar in Kauf genommen, aber erst später und nicht zum aktuellen Zeitpunkt. Mithilfe von Pseudocode wird der Gedankengang veranschaulicht, mit dem Schwerpunkt auf der Typspezialisierung, ohne dabei die Mechanismen von Java 7 und Java 8 zu verwenden:

```
function sum(a, b) {
    try {
        int sum = (Integer)a + (Integer)b;
        checkIntOverflow(a, b, sum);
        return sum;
    } catch (OverFlowException | ClassCastException e) {
        return sumDoubles(a, b);
    }
}
```

Allgemeiner ausgedrückt sieht es so aus:

```
final MethodHandle sumHandle = MethodHandles.guardWithTest(
    intsAndNotOverflow,
    sumInts,
    sumDoubles);

function sum(a, b) {
    return sumHandle(a, b);
}
```

Es können auch andere Mechanismen als Guards verwendet werden, beispielsweise durch Umschreiben vom Target *MethodHandle* im Falle einer *ClassCastException* oder durch *SwitchPoints*. Der Ansatz kann auch auf Strings und andere Objekte erweitert werden. Jedoch sollten die Compile-Time-Typen verwendet werden, falls

Anzeige

sie verfügbar sind. In der Betrachtung werden vorerst keine Integer-Overflows berücksichtigt, die Konvertierung von Primitive Number zu Object wird als übliches Szenario angesehen und Runtime-Analyse und Invalidierung mit den statischen Typen vom JavaScript-Compiler kombiniert. Durch Spezialisierung der *sum function* für die Call Site werden Doubles schneller ausgeführt als semantisch äquivalente Objekte, und dies ist mit nur vier Bytecodes kürzer, ohne die Runtime-Aufrufe:

```
// specialized double sum
sum(DD)D:
    dload_1
    dload_2
    dadd
    dreturn
```

Nachdem in dynamischen Sprachen einiges passieren kann, stellt sich die Frage: Was geschieht, wenn die Funktion *sum* zwischen den Call-Site-Ausführungen wie folgt überschrieben wird?

```
sum = function(a, b) {
  return a + 'string' + b;
}
```

Expliziten Bytecode braucht man dabei nicht zu verwenden, sondern man nimmt einen Switch Point und generiert einen Revert Stub, um den Rückweg abzusichern. Die Call Site zeigt dann auf den Revert Stub und nicht auf die Spezialisierung mit Double. Keiner der Revert Stubs muss als tatsächlicher, expliziter Bytecode generiert werden, sondern die MethodHandle Combinators reichen dafür aus.

```
sum(DD)D:
dload_1
dload_2
dadd
dreturn
```

```
sum_revert(DD)D: // hope this doesn't happen
dload_1
invokestatic JSRuntime.toObject(D)
dload_2
invokestatic JSRuntime.toObject(D)
invokedynamic sum(OO)O
invokestatic JSRuntime.toNumber(O)
dreturn
```

Das Ergebnis sieht dann wie folgt aus:

```
ldc 4711.17
dstore 1
ldc 17.4711
dstore 2
dload 1
invoke JSRuntime.toObject(O)
dload 2
invoke JSRuntime.toObject(O
invokedynamic sum(OO)O
invoke JSRuntime.toDouble(O)
dload 3
dmul
dstore 3
```

```
ldc 4711.17
dstore 1
ldc 17.4711
dstore 2
dload 1

dload 2
//likely inlined:
invokedynamic sum(DD)D

dload 3
dmul
dstore 3
```

Einsatzgebiet von Nashorn

Nashorn ist ein vortrefflicher Anwendungsfall für invokedynamic. Damit existiert eine rein in Java geschriebene und invokedynamic-basierte Implementierung für dynamische Sprachen auf der JVM, die schneller sein soll als andere Implementierungen ohne invokedynamic. Dieser Ansatz ist mit der JVM-Implementierung von JRuby direkt vergleichbar. Für Nashorn wurde JavaScript ausgewählt, einerseits wegen seiner großen Verbreitung und dem hohen Potenzial, neue Anwendungen mit JavaScript zu schreiben. Andererseits, um die langsame Rhino-Engine durch Nashorn im JDK 8 komplett zu ersetzen. JSR 223 mit *javax.script* ist das zentrale API direkt auf der

Listing 4: HTTP Server mit Nashorn in 84 Zeilen

```
#!/usr/bin/jjs -scripting
#

var Thread              = java.lang.Thread;
var ServerSocket        = java.net.ServerSocket;
var PrintWriter         = java.io.PrintWriter;
var InputStreamReader   = java.io.InputStreamReader;
var BufferedReader      = java.io.BufferedReader;
var FileInputStream     = java.io.FileInputStream;
var ByteArray           = Java.type("byte[]");

var PORT = 8080;
var CRLF = "\r\n";
var FOUROHFOUR = <<<EOD;
<HTML>
   <HEAD>
      <TITLE>404 Not Found</TITLE>
   </HEAD>
   <BODY>
      <P>404 Not Found</P>
   </BODY>
</HTML>
EOD

var serverSocket = new ServerSocket(PORT);

while (true) {
   var socket = serverSocket.accept();

   try {
      var thread = new Thread(function() { httpRequestHandler(socket); });
      thread.start();
      Thread.sleep(100);
   } catch (e) {
      print(e);
   }
}

function httpRequestHandler(socket) {
   var out       = socket.getOutputStream();
   var output    = new PrintWriter(out);
   var inReader  = new InputStreamReader(socket.getInputStream(), 'utf-8');
   var bufReader = new BufferedReader(inReader);
```

JVM, über das Java/JavaScript und umgekehrt JavaScript/Java automatisch integriert wird. Beispielhaft für die direkte Verwendung von JSR 223 mit Nashorn ist der von Jim Laskey in 84 Zeilen geschriebene Nashorn-HTTP-Server (Listing 4). Weitere Beispiele sind im Nashorn-Blog aufgeführt (https://blogs.oracle.com/nashorn/), u. a. die Verwendung von Lambda-Ausdrücken mit Nashorn.

Das Einsatzgebiet von Nashorn erstreckt sich über Online-Interpreter in JavaScript mit REPL (Read Eval Print Loop), vgl. *repl.it* für native Mozilla Firefox JavaScript, Shell-Skripte, Build-Skripte mit dem ant-Eintrag *<script language="javascript">*, bis hin zu serverseitigen JavaScript-Frameworks mit Node.jar. Mit dem eigenständigen Projekt Node.jar existiert eine Nashorn-basierte Implementierung des Node.js-Event-API, die mit der Node-Testsuite auf vollständige Kompatibilität getestet wurde, aber ohne ein bisher veröffentlichtes Releasedatum (Stand: Juli 2013, *Anm. d. Red.*). Dabei können das Node.js-Event-Modell, das Modulsystem und die Java-Plattform-APIs verwendet werden. Node.jar verbindet über ein Mapping das Node-API mit korrespondierender Java-Funktionalität (Tabelle 1) und enthält die asynchrone I/O-Implementierung vom Grizzly-Multiprotokoll-Framework.

Die Vorteile beim Betrieb von Node-Anwendungen auf der JVM liegen in der Wiederverwendbarkeit existierender Ja-

```
    var lines = readLines(bufReader);

    if (lines.length > 0) {
       var header = lines[0].split(/\b\s+/);

       if (header[0] == "GET") {
          var URI = header[1].split(/\?/);
          var path = String("./serverpages" + URI[0]);

          try {
             if (path.endsWith(".jjsp")) {
                var body = load(path);
                if (!body) throw "JJSP failed";
                respond(output, "HTTP/1.0 200 OK", "text/html", body);
             } else {
                sendFile(output, out, path);
             }
          } catch (e) {
             respond(output, "HTTP/1.0 404 Not Found", "text/html",
                                                          FOUROHFOUR);
          }
       }
    }

    output.flush();
    bufReader.close();
    socket.close();
}

function respond(output, status, type, body) {
    sendBytes(output, status + CRLF);
    sendBytes(output, "Server: Simple Nashorn HTTP Server" + CRLF);
    sendBytes(output, "Content-type: ${type}" + CRLF);
    sendBytes(output, "Content-Length: ${body.length}" + CRLF);
    sendBytes(output, CRLF);
    sendBytes(output, body);
}

function contentType(path) {
    if (path.endsWith(".htm") ||
        path.endsWith(".html")) {
      return "text/html";
    } else if (path.endsWith(".txt")) {
      return "text/text";
    } else if (path.endsWith(".jpg") ||
               path.endsWith(".jpeg")) {
      return "image/jpeg";
    } else if (path.endsWith(".gif")) {
      return "image/gif";
    } else {
      return "application/octet-stream";
    }
}

function readLines(bufReader) {
    var lines = [];

    try {
       var line;
       while (line = bufReader.readLine()) {
          lines.push(line);
       }
    } catch (e) {
    }

    return lines;
}

function sendBytes(output, line) {
    output.write(String(line));
}

function sendFile(output, out, path) {
    var file = new FileInputStream(path);

    var type = contentType(path);
    sendBytes(output, "HTTP/1.0 200 OK" + CRLF);
    sendBytes(output, "Server: Simple Nashorn HTTP Server" + CRLF);
    sendBytes(output, "Content-type: ${contentType(path)}" + CRLF);
    sendBytes(output, "Content-Length: ${file.available()}" + CRLF);
    sendBytes(output, CRLF);
    output.flush();

    var buffer = new ByteArray(1024);
    var bytes = 0;

    while ((bytes = file.read(buffer)) != -1) {
       out.write(buffer, 0, bytes);
    }
}
```

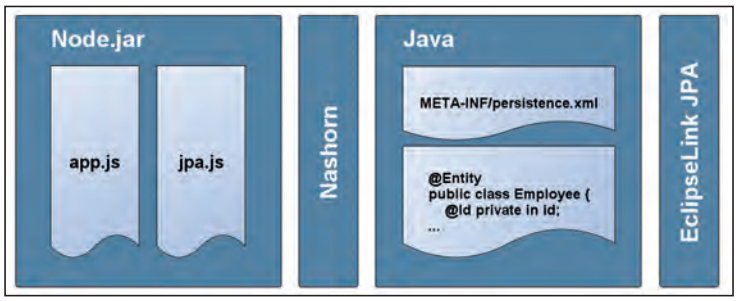


Abb. 4: Annotierte Java-Klassen mit XML-Konfigurator

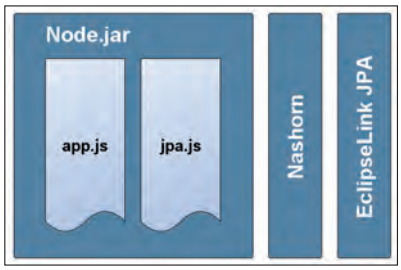


Abb. 5: JavaScript-/JSON-Konfiguration

va-Bibliotheken und Laufzeitumgebungen, der Nutzung mehrerer Java-Threads- und Multi-Core-Prozessoren, im Gegensatz zu einem Node-Port, der immer Single-Threaded und ereignisgesteuert abläuft. Zusätzlich bekommt man bei Java das Java-Security-Modell dazu, und mit Nashorn erhält man alle vom JDK 8 unterstützten Plattformen, auch die für Java SE Embedded.

Bei der Verwendung von Node.jar und Java-Persistierung gibt es mehrere Möglichkeiten: Erstens über einen JPA-Wrapper (**Abb. 4**), durch XML und annotierte Java-Klassen erfolgt der Java-EE-Zugriff von serverseitigem JavaScript. Zweitens die Variante Dynamic-JPA-Wrapper, mit XML-Konfiguration und dem Java-EE-Zugriff von serverseitigem JavaScript, ohne Java-Klassen zu benutzen. Die dritte Variante JavaScript-JPA wird über JavaScript/JSON konfiguriert, einer Java-SE-JPA-orientierten JavaScript-Persistenz mit JSON-Konfiguration, und die vierte Variante JavaScript-Datenbank-JPA benutzt JavaScript-JPA über ein Datenbankschema. Die Java-SE-JavaScript-Persistenz verwendet JPA mit den Mappings vom Datenbankschema. Für die Varianten drei und vier gilt **Abbildung 5**.

Tabelle 1: Mapping der Node.js-Module nach Java

Node.js	Java-Funktionalität
buffer	Grizzly HeapBuffer (java.nio.ByteBuffer)
child_process	java.lang.ProcessBuilder
dns	java.net.InetAddress, java.net.IDN
fs	java.nio.File
http / https	Grizzly
net	java.net.Socket, java.net.ServerSocket, java.nio.channels.Selector
os	java.lang.System, java.lang.Runtime
stream	Grizzly nio Stream

Tabelle 2: Da Vinci Machine Patches

MLVM-Patches	Beschreibung
meth	method handles implementation
indy	invokedynamic
coro	light weight coroutines
inti	interface injection
tailc	hard tail call optimization
tuple	integrating tuple types
hotswap	online general class schema updates
anonk	anonymous classes; light weight bytecode loading

Make Nashorn

Im JDK-8-Early-Access-Release ist Nashorn bereits enthalten, und der Download vom Nashorn Build im Bundle mit JDK 8 kann über java.net durchgeführt werden. Alternativ kann der Nashorn-Forest-Build mit der Umgebungsvariablen *JAVA_HOME* auf dem entsprechenden JDK-8-Installationsverzeichnis auch selbst gemacht werden. Unter dem Verzeichnis *make* kann die ca. 1,5 MB große Datei *nashorn.jar* erstellt werden:

```
hg clone http://hg.openjdk.java.net/nashorn/jdk8/nashorn nashorn

cd nashorn~jdk8/nashorn/make
ant clean jar
```

Um Nashorn verwenden zu können, braucht man das Kommando *jjs* mit dem Interpreter vom JDK 8 und man erhält die REPL:

```
cd nashorn~jdk8/nashorn
sh bin/jjs <your .js file>
```

Für Nashorn gibt es auch eine JavaFX-Integration, die in der Kommandozeile mit dem Aufruf *jjs -fx fxscript.js* verwendet wird. Der Einstellparameter *-fx* für *jjs* macht das Bootstrapping für Skripte, die *javafx.application.Application* verwenden.

Nashorn unterstützt das *javax.script*-API. Damit ist es auch möglich, die Datei *nashorn.jar* in den Classpath aufzunehmen und die Aufrufe an die Nashorn-Script-Engine über den *javax.script.ScriptEngineManager* abzusetzen. Die Dokumentation ist unter *Nashorn JavaDoc*. Sie kann wie folgt erzeugt werden und steht dann unter *dist/javadoc/index.html* zur Verfügung:

```
cd nashorn~jdk8/nashorn/make
ant javadoc
```

Ausblick

Das neue Projekt Nashorn in JDK 8 ist ein großer Schritt, die JVM attraktiver zu machen, gepaart mit dem langfristigen Ziel, die JVM in eine polyglotte Laufzeitumgebung zu überführen, wie im OpenJDK-Projekt Multi-Language VM (MLVM oder auch Da Vinci Machine Project) beschrieben. MLVM erweitert die JVM mit der notwendigen Architektur und Leistungsfähigkeit für andere Sprachen als Java und speziell als effiziente Ablaufumgebung für dynamisch typisierte Sprachen. Der Schwerpunkt liegt auf der Fertigstellung einer existierenden Bytecode- und Ablaufarchitektur zur allgemeinen Erweiterung, anstatt eine neue Funktionalität nur für eine einzige Sprache zu erstellen oder gar ein neues Ablaufmodell

zu erzeugen. Neue Sprachen auf der JVM sollen wie Java von der leistungsstarken und ausgereiften Ablaufumgebung profitieren. Stolpersteine, die die Entwickler neuer Sprachen stören, sollen entfernt werden. Allerdings soll übermäßiger Arbeitseifer an nicht erwiesener Funktionalität oder an Nischensprachen nicht im Vordergrund stehen. Unterprojekte mit erhöhter Aktivität sind beispielsweise: Dynamic Invocation, Continuations und Stack Introspection, Tail Calls und Tail Recursion sowie Interface Injection. Dazu kommt eine Vielzahl von Unterprojekten mit niedriger Priorität. Die im MLVM Incubator befindlichen Unterprojekte Tail-Call-Optimierung und Interface Injection sind hilfreich für Scala. Besonders wünschenswert wäre eine stärkere Beteiligung dieser Entwicklergemeinschaft im Projekt MLVM. Die Gestaltung einer künftigen VM beinhaltet einen Open-Source-Inkubator für aufkommende Merkmale der JVM und sie enthält Codefragmente und Patches (Tabelle 2). Die Migration ins OpenJDK ist erforderlich sowie eine Standardisierung und ein Feature-Release-Plan. Die aus dem Projekt MLVM gewonnenen Forschungsergebnisse sollen dazu dienen, existierenden Bytecode zu verwenden, um die Performancevorteile von Nashorn im JDK 9 zu vergrößern und den HotSpot-Codegenerator für eine verbesserte Unterstützung dynamischer Sprachen anzupassen.

Fazit

Nashorn ist mehr als nur ein Proof of Concept für invokedynamic, es ist eine leistungsstarke und standardkonforme JavaScript-Engine im JDK 8, die auch weitere Versionen von JavaScript und ECMAScript 6 unterstützen wird. Künftige Arbeitspakete umfassen eine Leistungssteigerung innerhalb von Nashorn und der JVM, sodass die schnelle Verarbeitung von JavaScript mit Nashorn auf der JVM nicht mehr in Frage gestellt wird. Mit dem in Nashorn neu implementierten Meta-Objekt-Protokoll werden die JavaScript-Aufrufe von Java-APIs vereinfacht. Dies ermöglicht die nahtlose Interaktion von JavaScript und Java. Denkbar ist auch eine Alternative am Browser, bei der die Entwickler wählen können, ob sie die Google V8 JavaScript Engine mit WebKit oder auch dort mit Nashorn verwenden möchten. Es bleibt offen, wie stark die Verbreitung von Anwendungen mit JavaScript am Browser und auf dem Server tatsächlich zunehmen wird, aber Nashorn ermöglicht es den Infrastrukturanbietern, die Verarbeitung von JavaScript und Java auf eine zentrale JVM zu konsolidieren. Die JVM wird dadurch attraktiver und bietet neben Memory-Management, Codeoptimierung, Scripting für die Java-Plattform (JSR 223) auch viele nützliche JVM-Managementwerkzeuge mit Debugging- und Profiling-Fähigkeiten an. Dies ist ein grundlegender Meilenstein auf dem Weg zur polyglotten VM, eine zuverlässige und skalierbare Infrastruktur für neue Sprachen anzubieten.

Anzeige

Marcus Lagergren hat einen M. Sc. in Computer Science vom Royal Institute of Technology in Stockholm. Er ist ein Gründungsmitglied von Appeal Virtual Machines, der Firma hinter der JRockit JVM. Lagergren war Team Lead und Architekt der JRockit Code Generators und über die Jahre in fast alle Aspekte der JVM involviert. Anschließend wechselte er zu Oracle für die Arbeit an Virtualisierungstechnologien, und seit 2011 ist er Mitglied des Java-Language-Teams bei Oracle und eruiert dynamische Sprachen auf der JVM.

Wolfgang Weigend arbeitet als Sen. Leitender Systemberater bei der Oracle Deutschland B.V. & Co. KG. Er beschäftigt sich mit Java-Technologie und Architektur für unternehmensweite Anwendungsentwicklung.

Links & Literatur

[1] http://openjdk.java.net/jeps/174

[2] http://hg.openjdk.java.net/nashorn/jdk8/nashorn

[3] http://jcp.org/en/jsr/detail?id=223

[4] https://blogs.oracle.com/nashorn

[5] https://blogs.oracle.com/nashorn/entry/jjs_fx

[6] http://octane-benchmark.googlecode.com/svn/latest/

[7] http://openjdk.java.net/projects/mlvm/

Anzeige

Anzeige

Zeitgemäßes Testen von Java-EE-Anwendungen mit Arquillian

Im Container und doch frei

Eine häufige Beobachtung in der Praxis ist, dass Testen immer dann unter den Tisch fällt, wenn es zu kompliziert wird. Das trifft am ehesten für automatisierte Tests zu, die in der Verantwortung der Entwickler entstehen sollen. Wenn man sich dabei noch mit allerlei technischen Aspekten beschäftigen muss, trägt das nicht unbedingt zum Erfolg bei. Warum Arquillian zu einem unverzichtbaren Werkzeug in diesem Bereich geworden ist, soll hier betrachtet werden.

von Christian Heinemann

Das automatisierte Testen von Java-EE-Anwendungen gestaltet sich mitunter recht schwierig. Seit Einführung des POJO-Programmiermodells mit Java EE 5 kann man wenigstens in Unit Tests große Teile der Geschäftslogik automatisiert abprüfen. Darüber hinaus kommen aber noch unzählige Annotationen, Deployment-Deskriptoren, Dependency Injection und Convention over Configuration zum Einsatz. Deren Interpretation obliegt einem Container. Die Nachbildung einer solchen Ausführungsumgebung mag nur ansatzweise gelingen und ist nicht praktikabel.

Die Standardspezifikationen halten sich bei dem Thema vornehm zurück. Mit Java EE 6 wurde zumindest ein API zum testfreundlichen Ausführen eines EJB-Containers eingeführt. Aber was ist mit den anderen Containern bezüglich Servlets, CDI oder OSGi, ganz zu schweigen von deren Zusammenspiel in einem Integrationstest? Diese offensichtliche Lücke möchte Arquillian füllen [1]. Es handelt sich dabei um ein Testframework zur Ausführung von Java-EE-Code im Container. Für den Entwickler bedeutet der Einsatz eine deutliche Entlastung. Das Schreiben der eigentlichen Tests soll im Vordergrund stehen, nicht das Verwalten der Testinfrastruktur.

Funktionen

Arquillian deckt die folgenden Bereiche ab, die im weiteren Verlauf noch genauer beleuchtet werden:

- Verwaltung des Lebenszyklus eines Containers
- Anbindung an verschiedene Container
- Zusammenstellung von Klassen und Ressourcen zu einem Deployment-Artefakt
- Durchführung des Deployments
- Erweiterung der Testklassen u. a. um Dependency Injection per *@EJB*, *@Inject* und *@Resource*

Der Testcode unterscheidet sich auf den ersten Blick nicht sonderlich von klassischen Unit Tests, da er auf JUnit oder alternativ TestNG basiert. Die Abhängigkeiten zu Arquillian innerhalb des Testcodes sind minimal gehalten. Auch bei der Ausführung der Tests muss der Entwickler nicht von seinen Gewohnheiten abrücken. Das ist wie bekannt direkt aus der IDE oder dem Build-Werkzeug möglich. All diese Punkte sorgen dafür, dass sich die Einführung in Projekten relativ unspektakulär gestalten sollte.

Erweiterungen

Arquillian ist in vielerlei Hinsicht erweiterbar [2] und auch unter Berücksichtigung dieses Aspekts entworfen worden. Die nachfolgende Liste stellt nur eine Auswahl der so genannten Extensions dar. Eine nähere Betrachtung würde aber den Umfang hier sprengen.

- *Persistence Extension* unterstützt die Tests von Datenbankanwendungen, indem vorgefertigte Datensätze in Ressourcen (XML, JSON, YML) definiert, eingespielt und validiert werden können.
- *Transaction Extension* erlaubt die deklarative Steuerung von Transaktionen bei der Testausführung durch Annotationen. Sie kommt meist in Verbindung mit der Persistence Extension zum Einsatz.
- *Drone Extension* erleichtert das funktionale Testen von Anwendungen mit Weboberfläche durch die Anbindung von Selenium [3].
- *Jacoco Extension* ermöglicht die Bestimmung der Testabdeckung von Code, der im Container ausgeführt wird.

Deployment

Für die Ausführung von Tests mit Arquillian ist ein Deployment im Container erforderlich. Das Deployment besteht aus einem oder mehreren Archiven (JAR, WAR, EAR). Das Testarchiv wird erst im Rahmen der Testausführung erstellt und gestattet die individuelle Zusammenstellung von Klassen und Ressourcen. Das so genannte *Micro Deployment* unterstützt die Isolation von nicht benötigten Teilen im Classpath und beschleunigt außerdem den Deployment-Prozess. Die Unabhängigkeit der Archiverstellung des Build-Prozesses vereinfacht die Testausführung in der IDE, weil geänderte Klassen sofort angezogen werden, ohne dass zusätzliche Schritte notwendig sind. Damit ist die Ausführung so einfach wie bei Unit Tests ohne Arquillian.

Die Definition der Deployments erfolgt per ShrinkWrap [4]. Dabei handelt es sich um ein Java-API zur Zusammenstellung von Archiven. In der einfachsten Form werden Klassen und Java-Ressourcen aus dem Classpath in die Archive übernommen. Es können aber auch Ressourcen neu erzeugt werden. Davon wird häufig bei Deployment-Deskriptoren Gebrauch gemacht (z. B. *persistence.xml*), die für den Test angepasst werden müssen. Weil die Manipulationen mitunter recht komplex ausfallen können, gibt es mit ShrinkWrap Descriptors einen Aufsatz, der ein Fluent-Java-API zum Umgang mit gängigen XML-Schemata aus dem Java-EE-Umfeld anbietet.

Häufig müssen auch externe Bibliotheken oder weitere Anwendungskomponenten als JARs in das Deployment-Archiv (WAR oder EAR) aufgenommen werden. Mit der Erweiterung ShrinkWrap Resolvers [5] wird dies deutlich vereinfacht. Durch die Anbindung an das Maven-Ökosystem werden damit die entsprechenden Artefakte bereitgestellt, gegebenenfalls auch aus entfernten Repositories geladen, und transitive Abhängigkeiten aufgelöst.

Container

Arquillian unterstützt eine Reihe von Containern [2]. Darunter befinden sich u. a. vollwertige Applikationsserver wie GlassFish oder JBoss AS, Servlet-Container wie Tomcat und Jetty oder auch eigenständige Bean-Container wie Weld und OpenEJB. Die Anbindung erfolgt über Adapter. Damit sind Arquillian im Kern und weitestgehend die Tests unabhängig vom konkreten Container.

Die Anbindung der Container kann in den drei Formen *remote*, *managed* und *embedded* auftreten. Bestimmte Container realisieren nicht alle Formen. Das ist meist technisch bedingt.

- Ein *Remote-Container* läuft im Gegensatz zur Testausführung in einer separaten JVM. Diese kann sich auch auf einer entfernten Maschine befinden. Arquillian verbindet sich mit dem Container, führt das Deployment des Testarchivs durch und startet die einzelnen Tests.

Der Testcode unterscheidet sich auf den ersten Blick nicht sonderlich von klassischen Unit Tests.

- Ein *Managed-Container* verhält sich hier ganz ähnlich. Nur übernimmt Arquillian auch das Hoch- und Herunterfahren. Dazu muss er aber lokal installiert sein.
- Ein *Embedded-Container* schließlich läuft in der gleichen JVM wie die Testausführung. Auch hierbei liegt die Steuerung des Containers in der Hand von Arquillian.

Remote- und Managed-Container setzen eine Installation im Dateisystem voraus. Für Managed-Container baut man in der Regel diese Installation im Rahmen eines automatisierten Build-Prozesses mit auf. Remote-Container werden unabhängig von den Tests verwaltet. Embedded-Container dagegen lassen sich aus dem Classpath heraus ohne Installation betreiben. Im Hinblick auf Continuous Integration und die Reproduzierbarkeit von Builds stellen sie die einfachste Form dar. Sie eignen sich auch besonders für Unit Tests. Weiterführende Testformen wie Integrationstests sollten dagegen mit Managed- oder Remote-Containern durchgeführt werden. In erster Linie sprechen eine isolierte Ausführungsumgebung und bessere Konfigurierbarkeit dafür.

Ausführungsmodus

Bei der Testausführung wird unterschieden, wo der Testcode tatsächlich ausgeführt wird. Es gibt die Möglichkeit der Ausführung im Container. Dabei werden dem Deployment-Archiv die Testklassen und alle ihre Abhängigkeiten hinzugefügt. Das Anstoßen der Testausführung erfolgt wie üblich weiterhin aus der IDE oder einem Build-Werkzeug wie Maven. Jedoch wird die konkrete Ausführung der Testmethoden an die Variante im Container delegiert. Der dahinter liegende Mechanismus unterscheidet sich je nach Containerform.

Anzeige

Für Embedded-Container erfolgt der Aufruf direkt lokal, da sie in der gleichen JVM laufen. Managed- und Remote-Container verwenden ein spezielles Kommunikationsprotokoll, das bei sehr vielen Containern über ein Servlet abgebildet ist. Arquillian übernimmt in diesen Fällen die Anreicherung des Deployments um die dazu notwendigen Artefakte.

In-Container-Tests vereinfachen die Tests dadurch, dass meist keine zusätzlichen Schnittstellen geschaffen werden müssen. Sowohl Anwendungs- als auch Container-Ressourcen lassen sich direkt nutzen. Beispielsweise ist es möglich, EJBs aufzurufen, obwohl sie über keine Remote-Schnittstelle verfügen. Auch das Ansprechen von *EntityManager* und *UserTransaction* gestaltet sich beim Testen von Persistenz sehr praktisch.

Listing 1

```
@Entity
@NamedQuery(name = "Product.byCode",
    query = "SELECT p FROM Product p WHERE p.code=:code")
public class Product implements Serializable {

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

  @NotNull @Column(unique = true)
  private String code;

  @NotNull
  private String name;

  Product() {}
  public Product(String code, String name) {
  this.code = code; this.name = name; }

  public String getCode() { return code; }
  public String getName() { return name; }
}
```

Listing 2

```
@Stateless
public class ProductFinderService {

  @PersistenceContext
  private EntityManager em;

  public Product findProductByCode(String code) {
    TypedQuery<Product> query = em.createNamedQuery(
    "Product.byCode", Product.class);
    query.setParameter("code", code);

    return query.getSingleResult();
  }
}
```

Als weitere Variante gibt es die Testausführung im Clientmodus. Dabei befinden sich weder Testklassen noch zusätzliche Protokollartefakte im Deployment, sondern nur der Anwendungscode. Der Testcode muss nun die Kommunikation mit der Anwendung selbst durchführen und tritt dabei als Remote-Client in Erscheinung. Dieser Modus wird daher häufig für das Testen von Web-Service-, REST- oder Web-UI-Schnittstellen eingesetzt.

Beispielprojekt – erster Testfall

Nach der eher theoretischen Einführung in die Funktionsweise von Arquillian soll nun die Umsetzung anhand eines einfachen und fiktiven Beispielprojekts aus der Handelsdomäne das Ganze veranschaulichen. Der Code ist hier nur auszugsweise dargestellt. Das vollständige Projekt ist online verfügbar [6]. Gegenstand ist die Ausführung einer JPA-Named-Query in einer EJB-Methode. Als Container kommt GlassFish 4.0 Embedded zum Einsatz. Maven bildet den Projektrahmen.

Listing 1 definiert eine JPA-Entität *Product*. Die EJB in Listing 2 enthält eine Methode für die Suche nach *Product* anhand des fachlichen Schlüssels *code*. Das soll die Testgrundlage sein.

Der erste Testfall ist in Listing 3 realisiert und zeigt schon recht deutlich den typischen Aufbau eines Arquillian-Tests in JUnit. Damit die Ausführung überhaupt von Arquillian gesteuert werden kann, ist auf Klassenebene per *@RunWith* die Beziehung herzustellen.

Das Deployment für den GlassFish-Container ist in einer statischen Methode definiert, die mit *@Deployment* annotiert sein muss. In ihr wird ein WebArchive (WAR) per ShrinkWrap zusammengestellt. Dabei wird demonstriert, wie eine einzelne Klasse, ein ganzes Package oder eine Ressource wie *persistence.xml* aufgenommen werden kann. Falls es große Überschneidungen zwischen den Deployments mehrerer Testklassen gibt, empfiehlt sich die Auslagerung in eigene wiederverwendbare Klassen.

Weil die EJB über keine externen Schnittstellen verfügt, benötigt man einen In-Container-Test, um sie ansprechen zu können. Glücklicherweise sind solche Tests Standard in Arquillian, und es muss nichts Besonderes getan werden. Die Testklasse selbst muss auch nicht explizit in das Deployment aufgenommen werden. Das erledigt Arquillian bereits. Sollte es aber zusätzliche Abhängigkeiten geben, die noch nicht enthalten sind oder bereits vom Container bereitgestellt werden, dann müsste das Deployment erweitert werden. Bei einem Embedded-Container kann man durchaus etwas nachlässiger sein, weil der Container in der bestehenden JVM läuft und über einen gemeinsamen Classloader auch Abhängigkeiten außerhalb des Deployments nachladen kann. Allerdings darf es sich dabei nicht um vom Container verwaltete Objekte wie Managed Beans handeln.

Die Testklasse enthält zwei Felder. Das erste ist eine Referenz der EJB, die getestet werden soll und ist mit *@EJB* annotiert. Daneben gibt es noch ein weiteres Feld mit dem *EntityManager* und der Annotation *@PersistenceContext*. Eine ganz praktische Funktion von Ar-

quillian ist die Unterstützung von Dependency Injection in Testklassen. Da es sich um einen In-Container-Test handelt, können sogar interne Ressourcen wie der *EntityManager* verwendet werden. Es sei noch erwähnt, dass auch CDI Beans per *@Inject* oder JNDI-Ressourcen mit *@Resource* so eingebunden werden können. Was aber genau möglich ist, hängt vom konkreten Container bzw. dessen Adapter ab.

Vor der Ausführung der EJB-Methode sind ein paar Testdaten notwendig, damit diese dann auch durch die *Named*-Query gefunden werden können. Die Testklasse löst dies einfach durch Verwendung von *EntityManager* in der Methode *insertTestData*. Allerdings setzt dieser Ansatz voraus, dass eine Transaktion gestartet wurde. Dies erreicht man sehr bequem über die Annotation *@Transactional* und die dazugehörige Transaction Extension. Damit man sich nicht selbst um das Abräumen der Testdaten kümmern muss oder Nebeneffekte in späteren Testfällen riskiert, erzwingt man einfach das Zurückrollen der Transaktion für jeden Testfall. Eine Alternative zur Extension wäre die Dependency Injection einer *UserTransaction* und die manuelle Transaktionssteuerung.

Um den Test zu übersetzen und auszuführen, muss die Maven POM noch um ein paar Abhängigkeiten ergänzt werden:

- *junit:junit* (Tests basieren auf JUnit)
- *org.jboss.arquillian.junit:arquillian-junit-container* (Anbindung von Arquillian an JUnit)
- *org.jboss.arquillian.container:arquillian-glassfish-embedded-3.1* (Container-Adapter)
- *org.glassfish.main.extras:glassfish-embedded-all* (konkreter Embedded-Container)
- *org.jboss.arquillian.extension:arquillian-transaction-jta* (Transaction Extension)

Alle Angaben sind in der Form *groupId:artefactId* und im Scope *test*. Auf konkrete Versionen soll hier verzichtet werden. Die vollständige POM ist in der Onlinequelle vorhanden.

Einbinden von Bibliotheken

In einem weiteren Testfall soll nun demonstriert werden, wie externe Bibliotheken in das Test-Deployment aufgenommen werden können. Dabei kommt ShrinkWrap Resolvers, was zuvor schon erwähnt wurde, zum Einsatz. Listing 4 zeigt den Testfall. Es soll der Fehlerfall getestet werden, in dem eine *NoResultException* ausgelöst wird, falls die *Named*-Query in der Servicemethode keinen Datensatz findet. Die gewünschte Exception wird allerdings durch den Aufruf der EJB-Methode in einer *EJBException* gekapselt, womit der übliche Ansatz, die erwartete Exception-Klasse in *@Test* zu deklarieren, nicht funktioniert.

An der Stelle macht man sich das Leben einfach und verwendet die Methode *Throwables.getRootCause* aus Google Guava. Die Bibliothek muss nun an zwei Stellen bekannt gemacht werden. Sie wird auf der einen Seite in der POM als Abhängigkeit benötigt. Schließlich muss der Testcode dagegen kompiliert werden. Auf der anderen Seite muss sie in das Test-Deployment aufgenommen werden. Nochmals zur Erinnerung: Es handelt sich hier um einen In-Container-Test. Es ist also notwendig,

Listing 3

```
@RunWith(Arquillian.class)
@Transactional(TransactionMode.ROLLBACK)
public class FindProductTest {

  @EJB
  private ProductFinderService serviceUnderTest;

  @PersistenceContext
  private EntityManager em;

  @Deployment
  public static WebArchive createArchive() {
    return ShrinkWrap.create(WebArchive.class)
      .addClass(ProductFinderService.class)
      .addPackage(Product.class.getPackage())
      .addAsResource("META-INF/persistence.xml");
  }

  @Before
  public void insertTestData() {
    Product product = new Product("123456", "Testartikel");
    em.persist(product);

    em.flush(); // Änderungen auf DB schreiben
    em.clear(); // und Neuladen erzwingen
  }

  @Test
  public void testFindByCodeSuccess() {
    Product product = serviceUnderTest.findProductByCode("123456");

    assertNotNull(product);
    assertEquals("Testartikel", product.getName());
  }
}
```

Listing 4

```
@Test
public void testFindByCodeNoResult() {
  try {
    serviceUnderTest.findProductByCode("000000");
    fail("EJBException erwartet!");
  } catch (EJBException e) {
    assertEquals(NoResultException.class, Throwables.getRootCause(e).
                                                    getClass());
  }
}
```

dass die Bibliothek während der Testausführung im Container verfügbar ist.

Listing 5 zeigt die angepasste Deployment-Methode der Testklasse. Im ersten Teil wird nun per ShrinkWrap Resolvers die Bibliothek in einem File-Array zur Verfügung gestellt, das nachfolgend dem *WebArchiv* einfach hinzugefügt wird. Jeder Eintrag in dem Array repräsentiert ein lokales Artefakt. In den meisten Fällen sind es JAR-Dateien aus dem lokalen Maven-Repository. Falls es erforderlich sein sollte, würde analog zu einem Maven-Build auch die Kommunikation mit einem entfernten Repository stattfinden und Artefakte heruntergeladen werden. Die Methode *resolve* erwartet die Angabe in der Form *groupId:artefactId*. Die Angabe der konkreten Version ist nicht notwendig, falls wie in diesem Fall die Abhängigkeit bereits in der POM hinterlegt ist. Weitere transitive Abhängigkeiten würden durch Verwendung der Methode *withTransitivity* in das Array aufgenommen werden. Bei Google Guava trifft dies jedoch nicht zu, soll aber die Verwendung demonstrieren.

Suite Extension

Wenn man das vorherige Beispiel tatsächlich einmal ausführt, wird man feststellen, dass der Test doch einige Sekunden benötigt. Der größte Anteil entfällt auf das Hochfahren des Containers. Dazu kommt dann noch das Test-Deployment, das auch ein paar Sekunden dauern kann, obwohl es ziemlich klein ist. Die eigentliche Testausführung geht dann recht flott. Arquillian besitzt die Eigenschaft, für jede Testklasse ein eigenes Deployment durchzuführen. Dabei spielt es auch keine Rolle, ob ein gemeinsam genutztes Deployment in einer Testbasisklasse definiert ist. Der Container bleibt aber wenigstens die ganze Zeit oben und muss nicht immer wieder gestartet werden.

Bei einem größeren Projekt mit Dutzenden Testklassen ergibt sich aus diesem Verhalten eine größere Ausführungszeit. Die Ausführung der Tests mit jedem Build erscheint in gewisser Weise nicht mehr praktikabel. Die Wiederverwendbarkeit von Deployments ist eines der Features, die derzeit am meisten vermisst werden [7].

Listing 5

```
@Deployment
public static WebArchive createArchive() {
  final File[] guavaFiles = Maven.resolver()
        .loadPomFromFile("pom.xml")
        .resolve("com.google.guava:guava")
        .withTransitivity().asFile();

  return ShrinkWrap.create(WebArchive.class)
        .addClass(ProductFinderService.class)
        .addPackage(Product.class.getPackage())
        .addAsLibraries(guavaFiles)
        .addAsResource("META-INF/persistence.xml");
}
```

Mit der Suite Extension [8] von Aslak Knutsen – dem Projektleiter von Arquillian – existiert wenigstens eine inoffizielle Erweiterung, die eine Lösung für das Problem anbietet. Sie lässt sich unter zwei Randbedingungen einsetzen. Die erste ist, dass alle Tests genau ein gemeinsames Deployment verwenden, weil dieses vor dem ersten Testfall durchgeführt und von allen weiteren Testfällen aus dem Lauf verwendet wird. Die zweite Bedingung betrifft den Einsatz von weiteren Extensions. Es kann nämlich durch den Eingriff in den Standardprozess von Arquillian zu Problemen bei deren Einsatz kommen. So ist mindestens die Transaction Extension betroffen und damit auch die auf ihr aufbauende Persistence Extension.

Falls man mit diesen Einschränkungen leben kann, dann sorgt der Einsatz der Suite Extension für eine drastische Verbesserung der Testausführungszeiten. Der Autor setzt diese bereits bei zwei Projekten erfolgreich ein.

Fazit

Arquillian ist aufgrund der Flexibilität, die sich aus In-Container-Tests und Micro-Deployments ergibt, für eine Vielzahl von automatisierten Testformen geeignet. Das reicht von Unit Tests über Integrationstests bis hin zu Akzeptanz- und Performanztests. Beim Schreiben der Tests kann man bekannte Strukturen wie die von JUnit weiter verwenden. Auch der Zugriff auf Ressourcen gestaltet sich mit Dependency Injection denkbar einfach. Die Unabhängigkeit vom Build-Prozess erlaubt die unkomplizierte Ausführung aus der IDE und erleichtert auch die Einführung in bestehende Projekte. Aus Sicht des Autors hat Arquillian aber auch ein ganz wesentliches Ziel erreicht – Testen soll nämlich Spaß machen.

Für eine weitere Beschäftigung mit dem Thema sei auf die Onlinetutorials [9] verwiesen. Weiterhin ist im April 2013 das Buch „Arquillian Testing Guide“ von John D. Ament bei Packt Publishing erschienen [10].

Christian Heinemann ist Diplominformatiker und als Consultant bei der Saxonia Systems AG tätig. Er beschäftigt sich seit über zehn Jahren mit der Entwicklung von Geschäftsanwendungen auf der Java-Plattform, überwiegend im Backend-Bereich.

Links & Literatur

[1] http://www.arquillian.org

[2] http://arquillian.org/modules/

[3] https://docs.jboss.org/author/display/ARQ/Drone

[4] http://www.jboss.org/shrinkwrap

[5] https://github.com/shrinkwrap/resolver

[6] https://github.com/cheinema/arq-javamag

[7] https://issues.jboss.org/browse/ARQ-197

[8] https://gist.github.com/aslakknutsen/3975179

[9] http://arquillian.org/guides/

[10] http://www.packtpub.com/arquillian-testing-guide/book

Arquillian Drone für UI- und Integrationstests

Drohnen, bald mit WARP-Antrieb

Selenium ist ein allgemein einsetzbares Werkzeug, um Browserinteraktionen zu automatisieren und wird häufig für Web-GUI-Tests verwendet. Mit Arquillian Drone existiert eine Arquillian-Erweiterung, die basierend auf Selenium das einfache Testen von webbasiertem GUI im Zusammenspiel mit Java-EE-Komponenten erlaubt.

von Bernd Müller

w-jax Speaker

Mit Arquillian [1] hat JBoss einen neuen Standard im Bereich Testwerkzeuge gesetzt. Die bis dato existierenden Probleme beim Testen von Application-Server-Anwendungen, die vor allem aus dem Mocken und/oder Simulieren der Containerdienste bestanden, wurden obsolet, da Arquillian seine Tests in einem realen Container ausführt und das Mocken/Simulieren somit überflüssig macht.

Selenium [2] ist ein Werkzeug, dessen Entwickler mit „Selenium automates browsers“ eine sehr einfache, aber treffende Beschreibung für ihr Werkzeug liefern. Häufig wird Selenium verwendet, um webbasierte GUIs zu testen. Da Arquillian mit einem Service-Provider-Interface (SPI) versehen ist, ist die Erweiterung von Selenium mit verhältnismäßig geringem Aufwand möglich und wird bereits in etlichen Teilprojekten praktiziert. Eines davon ist Arquillian Drone [3], das Selenium integriert. Damit ist es möglich, Seleniums Java-APIs in Arquillian-Tests zu verwenden.

Wir führen in diesem Artikel Arquillian Drone ein, geben aber auch ein paar Tipps, die die Verwendung von Arquillian bei Nutzung des JBoss-Application-Servers (jetzt WildFly) vereinfachen. Wir setzen grundlegende Arquillian-Kenntnisse voraus und empfehlen dem noch unerfahrenen Leser den Arquillian-Artikel von Christian Heinemann in diesem Java Magazin ab Seite 42.

Arquillian im JBoss-AS

Arquillian ist ein von JBoss entwickeltes Testwerkzeug. Es ist daher nachvollziehbar, dass das Werkzeug insgesamt, aber auch neue Features zunächst für den JBoss-AS und erst danach für andere Application-Server realisiert wurden und weiterhin werden. Auch die Unterstützung verschiedener proprietärer JBoss-AS-Features ist weiter gediehen als entsprechende Features bei anderen Containern. Wir werden hier zunächst auf derartige JBoss-Features eingehen und verwenden dazu einen JBoss-AS 7.1.1 in der Remote-Konfiguration. Auf die Maven-Konfiguration gehen wir nicht ein und verweisen den Leser auf das herunterladbare Projekt [4], das alle Beispiele und Konfigurationen dieses Artikels enthält.

Listing 1

```
@Entity
@Table(uniqueConstraints = @UniqueConstraint(columnNames = "email"))
@NamedQueries({
   @NamedQuery(name = "Customer.getAll",
               query = "Select c from Customer c order by c.id"),
   @NamedQuery(name = "Customer.findByEmailAndPassword",
               query = "Select c from Customer c where c.email = :email and
                                                       c.password = :password")
})
public class Customer implements Serializable {

   @Id @GeneratedValue
   private Long id;
   private String lastname;
   private String firstname;
   @Temporal(TemporalType.DATE)
   private Date dob;
   private String email;
   private String password;
   // Constructors, Getter and Setter
```

JBoss pflegt eigene Maven-Archetypes für Java-EE-6 und den JBoss-AS. Es sind dies u. a. die Archetypes *jboss-javaee6-webapp-blank-archetype* und *jboss-javaee6-webapp-archetype*. Der erste Archetype erzeugt die üblichen Maven-Strukturen, der zweite fügt zusätzlich eine kleine Beispielanwendung hinzu, die JSF, EJB, JPA, CDI und REST verwendet. Besonders interessant im Rahmen dieses Artikels ist ein ebenfalls vorhandener Arquillian-Test in diesem Archetype, sodass sofort ohne weitere Konfiguration mit Arquillian begonnen werden kann.

Für unser eigenes Beispiel verwenden wir das Entity *Customer*, das in Listing 1 dargestellt ist. Da wir uns auf das Testen mit Arquillian konzentrieren wollen, verzichten wir auf die Darstellung von BV-Constraints, Konstruktoren und Getter/Setter-Paaren.

Die zu testende Komponente ist die EJB *CustomerService*, deren Quellcode in Listing 2 abgebildet ist.

Die (bisher einzige) Methode *findByEmailAndPassword()* der EJB soll nun mit Arquillian getestet werden. Man benötigt hierzu u. a. eine Data-Source-Definition und (mindestens) einen Datensatz. Hier kommen nun die erwähnten JBoss-spezifischen Erweiterungen ins Spiel. Im JBoss-AS 7.1 und höher können Data Sources über eine XML-Datei definiert werden, auf deren Inhalt wir nicht näher eingehen, die aber dem Namensschema **-ds.xml* folgt. Hibernate, der JPA-Provider im JBoss-AS, sucht beim Initialisieren nach einer Datei mit dem Namen *import.sql* und führt sie, falls vorhanden, als SQL-Anweisungen auf der in der Data Source definierten Datenbank aus. Die Datei kann SQL-Insert-Anweisungen enthalten, die für eine initiale Befüllung der Datenbank verwendet

Listing 2

```
@Stateless
public class CustomerService {

   @PersistenceContext
   EntityManager em;

   public List<Customer> findByEmailAndPassword(String email, String password) {
     TypedQuery<Customer> query = em.createNamedQuery("Customer.
                                                findByEmailAndPassword", Customer.class);
     query.setParameter("email", email);
     query.setParameter("password", password);
     return query.getResultList();
   }
}
```

Listing 3

```
@RunWith(Arquillian.class)
public class CustomerServiceTest {

   @Deployment
   public static Archive<?> createTestArchive() {
      return ShrinkWrap.create(WebArchive.class, "test.war")
            .addClasses(Customer.class, CustomerService.class, Resources.class)
            .addAsResource("META-INF/test-persistence.xml", "META-INF/persistence.xml")
            .addAsResource("test-import.sql", "import.sql")
            .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml")
            .addAsWebInfResource("test-ds.xml");
   }

   @Inject
   CustomerService customerService;

   @Test
   public void testFindByEmailAndPassword() {
     List<Customer> results = customerService.findByEmailAndPassword("email", "password");
     Assert.assertEquals("Falsche Kunden-Anzahl", 1, results.size());

   }
}
```

Listing 4

```
<h:form id="login">
...
   <h:inputText id="email" value="#{login.email}" />
...
   <h:inputSecret id="password" value="#{login.password}" />
...
   <h:commandButton id="login" action="#{login.login}"
 value="Login" />
...
</h:form>
```

Listing 5

```
@Named
@SessionScoped
public class Login implements Serializable {

   private String email;
   private String password;

   @Inject
   private CustomerService customerService;

   @Inject
   private FacesContext facesContext;

   public String login() {
      List<Customer> results =
            customerService.findByEmailAndPassword(email, password);
      if (results.size() == 1) {
         return "login-successful.jsf";
      } else {
         FacesMessage m = new FacesMessage(FacesMessage.
                        SEVERITY_ERROR, "Login Unsuccessful", null);
         facesContext.addMessage(null, m);
         return null;
      }
   }
...
}
```

werden. Die in Listing 3 dargestellte Arquillian-Klasse *CustomerServiceTest* nutzt die genannten Dateien.

Man erkennt das bei Arquillian übliche Erzeugen eines deploybaren Archivs, hier ein Webarchiv. In der ersten *addAsResource()*-Methode wird der JPA-Deployment-Deskriptor *persistence.xml* in das Archiv gepackt. Man sieht, dass die Datei *test-persistence.xml* verwendet wird. Diese ist im *test*-Verzeichnis des Maven-Projekts enthalten und zeigt somit, dass für Produktion und Test verschiedene Persistenzeinheiten verwendet werden können. Die zweite *addAsResource()*-Methode fügt nach demselben Schema Hibernates *import.sql*-Datei hinzu. Auch hier kann das Maven-Projekt zwei derartige Dateien enthalten. In der ersten *addAsWebInfResource()*-Methode wird eine leere *beans.xml*-Datei in das Archiv gepackt, um CDI zu aktivieren. Die zweite Methode fügt die bereits erwähnte Data-Source-Definition dem Archiv hinzu.

Wird nun das Archiv deployt, so wird zunächst die Data Source angelegt. Durch die Existenz der *persistence.xml* wird daraufhin JPA aktiviert. Hibernate liest diese Datei und erzeugt die Tabellen für die Entities, falls das Property *hibernate.hbm2ddl.auto* auf *create-drop* gesetzt ist. Danach werden die *Insert*-Anweisungen der Datei *import.sql* ausgeführt. Enthält diese ein Insert mit den entsprechenden Daten für *email* und *password*, wird der Test erfolgreich beendet.

Bemerkung: Wir sind uns im Klaren darüber, dass der Unit Test nicht den allgemein gültigen Regeln der Unabhängigkeit eines Unit Tests entspricht [5], da er von der Existenz der entsprechenden Insert-Anweisung in der Datei *import.sql* abhängt. Dies ist hier dem Ziel eines möglichst einfachen Beispiels geschuldet.

Arquillian Drone

Als Nächstes soll nun die Anwendung um eine JSF-Anmeldeseite erweitert werden, die die gerade getestete Methode *findByEmailAndPassword()* zur Authentifizierung verwendet. Drone erweitert Arquillian um verschiedene Selenium-APIs. Es ist damit möglich, Daten in eine Webseite einzugeben, die Seite an den Server zu schicken und die Antwort zu überprüfen. Dies leistet das folgende Beispiel. Wir beginnen mit den wichtigen Teilen der JSF-Seite *login.xhtml*, die in Listing 4 abgebildet sind.

JSF benötigt intern eine eindeutige Identifizierung jeder Komponente des Komponentenbaums. Falls der Entwickler/Designer keine expliziten IDs vergibt, versieht JSF die Komponenten mit synthetischen IDs, was das Testen jedoch praktisch unmöglich macht. In Listing 4 haben daher alle Komponenten (Form, Inputs, Schaltfläche) explizite IDs erhalten.

Die EL-Variable *login* ist eine CDI Managed Bean, abgebildet in Listing 5. Die interessante Methode ist *login()*, die die EJB-Methode verwendet, um die Credentials des Anmeldevorgangs zu prüfen. Bei positivem Ausgang wird zur Seite *login-successful.jsf* navigiert, bei negativem Ausgang wird auf der Login-Seite eine Fehlermeldung angezeigt.

Der Drone-Test ist in Listing 6 abgebildet. Interessant sind hier die beiden Annotationen *@ArquillianResource* und *@Drone*, die den Deployment-URL und das Selenium-Interface injizieren. Beide Injektionsziele werden in der Testmethode verwendet. Wir werden in einem späteren Beispiel ein alternatives Selenium-Interface verwenden. Ebenfalls erwähnenswert ist die explizite Definition der Webarchivvariablen, um diese auf der Konsole ausgeben zu können. Durch den Parameter der *toString()*-Methode wird der Inhalt des WARs auf der Konsole ausgegeben. Alternativ kann in der Datei *arquillian.xml* Arquillian angewiesen werden, das erzeugte WAR ins Dateisystem zu schreiben (Projekt [4]). Neben den bereits bekannten Dateien werden auch die beiden JSF-Seiten sowie der JSF-Deployment-Deskriptor in das WAR gepackt.

Die Testmethode selbst ist intuitiv verständlich. Die beiden Zusicherungen am Ende können alternativ ver-

Anzeige

wendet werden. Die erste geht von einem einfachen Text im Body der Seite aus, die zweite erwartet einen entsprechenden Title in der Seite.

Doch wo und vor allem wie wird der Test aufgeführt? Arquillian geht im Standardfall davon aus, dass die Tests im Container ausgeführt werden. In Listing 6 wird jedoch in der *@Deployment*-Annotation der Parameter *testable* auf *false* gesetzt. Dies bedeutet, dass die Tests nicht im Container ausgeführt werden. Bei der Ausführung des Tests wird zunächst das WAR erstellt und deployt. Dann

Listing 6

```
@RunWith(Arquillian.class)
public class LoginTest {

   private static final String WEBAPP_SRC = "src/main/webapp";

   @ArquillianResource
   URL deploymentURL;

   @Drone
   DefaultSelenium browser;

   @Deployment(testable = false)
   public static Archive<?> createTestArchive() {
      WebArchive war = ShrinkWrap.create(WebArchive.class, "test.war")
            .addPackages(true, "de.pdbm.jm.view")
            .addClasses(Customer.class, CustomerService.class, Resources.class)
            .addAsResource("META-INF/test-persistence.xml", "META-INF/persistence.
                                                                        xml")
            .addAsResource("test-import.sql", "import.sql")
            .addAsWebInfResource("test-ds.xml")
            .addAsWebResource(new File(WEBAPP_SRC, "login.xhtml"))
            .addAsWebResource(new File(WEBAPP_SRC, "login-successful.xhtml"))
            .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml")
            .addAsWebInfResource(new StringAsset("<faces-config
                                          version=\"2.0\"/>"), "faces-config.xml")
            ;
      System.out.println(war.toString(Formatters.VERBOSE));
      return war;
   }

   @Test
   public void testLoginSuccessful() {
      browser.open(deploymentURL + "login.jsf");
      browser.type("id=login:email", "email");
      browser.type("id=login:password", "password");
      browser.click("id=login:login");
      browser.waitForPageToLoad("5000");
      assertTrue(browser.isElementPresent("xpath=//body[contains(text(), 'Login
                                                                  Successful')]"));
      assertTrue(browser.getTitle().equals("Login Successful"));
   }
...
}
```

Listing 7

```
@RunWith(Arquillian.class)
public class CreateCustomerTest {

   private static final String WEBAPP_SRC = "src/main/webapp";

   @Drone
   DefaultSelenium browser;

   @Inject
   CustomerRepository customerRepository;

   @Deployment
   public static Archive<?> createTestArchive() {
      MavenDependencyResolver resolver = DependencyResolvers.
            use(MavenDependencyResolver.class).loadMetadataFromPom("pom.xml");
      return ShrinkWrap.create(WebArchive.class, "test.war")
            .addPackages(true, "de.pdbm.jm.view")
            .addClasses(Customer.class, CustomerService.class,
                                    CustomerRepository.class, Resources.class)
            .addAsResource("META-INF/test-persistence.xml",
                                                "META-INF/persistence.xml")

            .addAsWebInfResource("test-ds.xml")
            .addAsWebResource(new File(WEBAPP_SRC, "create.xhtml"))
            .addAsWebResource(new File(WEBAPP_SRC, "customer-created.xhtml"))
            .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml")
            .addAsWebInfResource(new StringAsset("<faces-config
                                    version=\"2.0\"/>"), "faces-config.xml")
            .addAsLibraries(resolver.artifact("org.seleniumhq.selenium:selenium-
                                                        java").resolveAsFiles())
            ;
   }

   @Test
   @RunAsClient
   @InSequence(1)
   public void createCustomer() {
      browser.open("http://localhost:8080/test/" + "create.jsf");
      browser.type("id=create:firstname", "firstname");
      browser.type("id=create:lastname", "lastname");
      browser.type("id=create:dob", "1.1.2000");
      browser.type("id=create:email", "email");
      browser.type("id=create:password", "password");
      browser.click("id=create:create");
      browser.waitForPageToLoad("5000");
      assertTrue(browser.isElementPresent("xpath=//body[contains(text(), 'Kunde
                                                              angelegt')]"));
   }


   @Test // im Server
   @InSequence(2)
   public void countCustomers() {
     assertEquals(1, customerRepository.getCustomers().size());
   }

}
```

wird eine Firefox-Instanz erzeugt und der URL der Methode *browser.open()* aufgerufen. Eine zweite Firefox-Instanz wird erzeugt, die über JavaScript die Daten des Tests in die erste Instanz eingibt (*browser.type()*) und das Formular abschickt (*browser.click()*). Danach wird die Antwort entsprechend analysiert (*browser.isElementPresent()*, *browser.getTitle()*). Den Test für die nicht erfolgreiche Anmeldung überlassen wir dem Leser als Übung. Er ist im Projekt [4] enthalten. Das Projekt enthält ebenfalls eine Konfigurationsmöglichkeit, um Chrome als Testbrowser zu verwenden. Der interessierte Leser findet diese Option in der Datei *arquillian.xml*. Weitere von Arquillian unterstützte Browser sind u. a. IE, Opera und mobile Endgeräte (Android und iPhone).

Drone im gemischten Modus

Der letzte Test hat das Ein-/Ausgabe-Verhalten der Anwendung auf Ebene des Browsers geprüft. Drone erlaubt jedoch auch die Ausführung von gemischten Tests, die sowohl auf dem Client als auch im Server ausgeführt werden. Listing 7 zeigt einen entsprechenden Drone-Test.

Das Deployment ist nun wieder testbar, jedoch kann die Annotation *@ArquillianResource* nicht verwendet werden, da diese Ressource nur auf dem Client existiert, der zweite Test jedoch ein Servertest ist. Ein weiterer Unterschied zum letzten Test ist die Verwendung des Maven-Dependency-Resolvers, der bereits im Arquillian-Artikel von Christian Heinemann eingeführt wurde und der dafür sorgt, dass die entsprechende Selenium-Bibliothek deployt wird und somit auf dem Server zur Verfügung steht.

Da das Deployment testbar ist, geht Arquillian davon aus, dass die Tests im Server ausgeführt werden. Der erste Test ist daher mit *@RunAsClient* annotiert und wird auf dem Client exekutiert. Die Arquillian-Annotation *@InSequence* sorgt dafür, dass die Tests in der angegebenen Reihenfolge aufgerufen werden. Um den Rahmen nicht zu sprengen, verzichten wir auf die Angabe der EJB-Erweiterung zum Persistieren von Kunden und auf die Darstellung der Klasse *CustomerRepository*, deren Methode *getCustomers()* die Liste aller Kunden zurückgibt.

WARP als Ausblick

Das Pattern des letzten Tests, bei dem eine Aktivität clientseitig initiiert wird und dann serverseitig die Inspektion des Zustands einer oder mehrerer Beans erfolgt, ist sicher ein häufig verwendetes Test-Pattern. Häufig genug, um das Arquillian-Team dazu zu bewegen, dies in einer neuen Arquillian-Erweiterung zu veröffentlichten: Arquillian WARP. Das Pattern macht die in Listing 7

Anzeige

verwendete Reihenfolgebeziehung der beiden Tests überflüssig und folgt damit der Regel, dass Unit Tests reihenfolgeunabhängig sein sollen [5]. WARP [6] befindet sich noch im Alphastatus, und das API wird ständig weiterentwickelt. Trotzdem lohnt es sich an dieser Stelle, einen ersten Blick auf WARP zu werfen, wie wir meinen.

WARP unterstützt Servlet-Anwendungen mit den Annotationen *@BeforeServlet* und *@AfterServlet* sowie JSF-Anwendungen mit den Annotationen *@BeforePhase(Phase)* *@AfterPhase(Phase)* für die sechs JSF-Phasen. Diese Annotationen werden im Inspektionsteil eines WARP-Tests verwendet und definieren die Inspektionsmethode. Die Fluent-artige Syntax für einen WARP-Test ist

```
Warp.initiate(Activity).inspect(Inspection);
```

Listing 8 zeigt die für WARP überarbeitete Version unseres letzten Tests. Die Testklasse ist mit *@WarpTest* und *@RunAsClient* annotiert, um die besondere Behandlung durch WARP zu realisieren. WARP unterstützt im Augenblick nur das *WebDriver*-API von Selenium, sodass die Annotation *@Drone* dieses API injiziert. Die weiteren Details entsprechen dem ursprünglichen Test, sind aber deutlich lesbarer.

Der WARP-Test ist ausführbar, führt aber leider nicht zum gewünschten Ergebnis, da der Aufruf der Zählmethode *vor* dem Persistieren des Kunden erfolgt. Wir sind uns nicht sicher, ob dies ein Bug in WARP oder dem Autor anzulasten ist, und arbeiten an einer Klärung.

Fazit

Arquillian ist ein sehr leistungsfähiges Testwerkzeug, das über ein SPI einfach zu erweitern ist. Mit Drone existiert eine Erweiterung, die intern Selenium verwendet, um browserbasierte Tests zu ermöglichen. Die Tests können in einem gemischten Modus sowohl client- als auch serverseitig ausgeführt werden. Mit WARP steht eine weitere Arquillian-Erweiterung in den Startlöchern, die das Testmuster „erst Aktivität, dann Inspektion“ unterstützt. Da JSFUnit [7] augenscheinlich nicht mehr weiterentwickelt wird – die letzte Version (2.0 Beta) ist fast zwei Jahre alt – scheint WARP die von JBoss favorisierte Alternative für JSF-Tests zu sein und mit der Integration in Arquillian auch besser zum JBoss-Test-Stack zu passen.

Bernd Müller ist Informatiker und als Hochschulprofessor und GmbH-Geschäftsführer tätig. Er ist Autor mehrerer Java-EE-Bücher, Sprecher auf verschiedenen Konferenzen und war Mitglied mehrerer JCP-Expert-Groups.

Links & Literatur

[1] http://www.arquillian.org

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

[3] http://www.arquillian.org/modules/drone-extension/

[4] http://www.pdbm.de/skripte/jm-drone.tgz

[5] Koskela, Lasse: Effective Unit Testing, Manning Publications, 2013

[6] https://docs.jboss.org/author/display/ARQ/Warp

[7] http://www.jboss.org/jsfunit

Listing 8

```
@RunWith(Arquillian.class)
@WarpTest
@RunAsClient
public class CreateCustomerWarpedTest {

  @ArquillianResource
  URL contextPath;

    @Drone
    WebDriver browser;

    @Deployment
    public static Archive<?> createTestArchive() {
      return ShrinkWrap.create( ...
) ;
    }

    @Test
    public void createAndCountCustomers() {
       Warp
        .initiate(new Activity() {
          public void perform() {
            browser.navigate().to(contextPath + "create.jsf");
            browser.findElement(By.id("create:firstname")).sendKeys("firstname");
            browser.findElement(By.id("create:lastname")).sendKeys("firstname");
            browser.findElement(By.id("create:dob")).sendKeys("1.1.2000");
            browser.findElement(By.id("create:email")).sendKeys("email");
            browser.findElement(By.id("create:password")).sendKeys("password");
            browser.findElement(By.id("create:create")).click();
          }})
        .inspect(new Inspection() {
          @Inject
          CustomerRepository customerRepository;

          @AfterPhase(RENDER_RESPONSE)
          public void checkCount() {
            assertEquals(1, customerRepository.getCustomers().size());
          }
        }
      );
      }
}
```

Kolumne: EnterpriseTales

von Lars Röwekamp und Arne Limburg

Auf Jobsuche: Java-Batch-API

Wollte man bisher Batch-Job-Verarbeitung im Umfeld von Enterprise-Java betreiben, war man auf proprietäre Lösungen angewiesen. Diese erfüll(t)en ihren Job in der Regel zwar sehr gut, bringen aber eine häufig nicht gewünschte Herstellerabhängigkeit mit sich. Mit Java EE 7 und der darin enthaltenen Spezifikation „Batch Application for the Java Platform 1.0“ [1] soll sich dies nun ändern.

Batch-Jobs sind Tasks, die ohne Benutzerinteraktion ausgeführt werden. Die Spezifikation „Batch Application for the Java Platform“ ist ein Framework, das eine entsprechende Unterstützung für die Deklaration, Erstellung und Steuerung von Batch-Jobs zur Verfügung stellt. Warum ein Framework? Um die oben genannten Aufgaben zu erfüllen, wird deutlich mehr geboten als nur ein API und dessen Implementierung: eine Batch Runtime, eine XML-basierte Sprache zur Batch-Job-Definition, ein API zur Interaktion mit der Batch Runtime und ein API zur Implementierung von Batch-Artefakten.

Das kleine Batch-Einmaleins

Für erfahrene „Batch-ler“ bringt das neue Batch-Framework bzw. das zugrunde liegende Konzept kaum Überraschungen mit sich. Wer sich schon einmal mit Spring Batch [2] oder WebSphere Compute Grid [3] beschäftigt hat, wird etliche Parallelen entdecken. Ein Batch-Job besteht aus mehreren Schritten, wobei jeder Schritt einzelne Items bzw. Gruppen von ihnen nach vorgegebenen Regeln lesen, verarbeiten und schreiben kann. Unterschiedlichste Callback-Methoden erlauben es dem Entwickler, an nahezu jedem Punkt der Verarbeitung gezielt einzugreifen bzw. das Framework nach eigenen Wünschen zu erweitern. **Abbildung 1** zeigt eine vereinfachte Darstellung der wichtigsten Komponenten.

Das Konzept des Batch-Frameworks sieht vor, dass in der Regel je Schritt eine Menge von Items einzeln eingelesen (*ItemReader*) und verarbeitet (*ItemProcessor*) werden, um dann das Ergebnis der Bearbeitung in Gruppen herauszuschreiben (*ItemWriter*). Da die Summe der bearbeiteten Items als „chunk“ geschrieben wird, spricht man bei dieser Art der Verarbeitung auch von Chunk-basierten Batch-Steps, wobei der in seiner Größe frei definierbare Chunk die Transaktionsgrenze bildet.

Um bei komplexen Batch-Jobs ein gezieltes Unterbrechen und Neustarten zu ermöglichen, können zusätzlich Checkpoints deklariert werden. Diese erlauben es bei der Ausführung eines Steps, eine Art Bookmark zu setzen (inklusive Status des Steps), um so bei einer – gewollten oder ungewollten – Unterbrechung gezielt an dieser Stelle wieder fortfahren zu können. Listing 1 zeigt die Interfaces für *ItemReader*, *ItemProcessor* und *ItemWriter* inkl. Checkpoint, mit deren Hilfe Einfluss auf die Bearbeitung eines Batch-Jobs genommen werden kann.

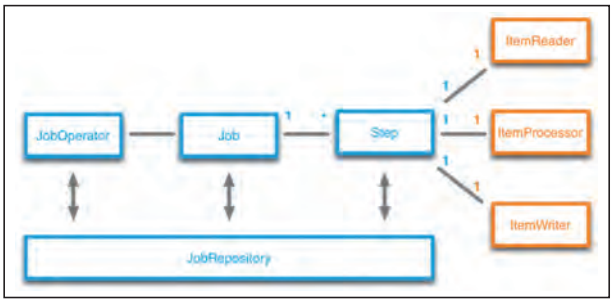


Abb. 1: Batch-Framework aus 10000 m

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

Daneben bietet die Java-Batch-Spezifikation so genannte *Batchlets* an. Ein Batchlet ist im Gegensatz zu einem Chunk ein völlig frei zu definierender Schritt innerhalb der Batch-Verarbeitung, mit dessen Hilfe z. B. ein Dateiupload oder -download angestoßen werden kann. Entsprechend einfach ist auch das Interface eines Batchlets aufgebaut. Neben einer Methode namens *process*() zum Starten des Schritts existiert lediglich eine weitere Methode *stop()*, die das explizite Stoppen von außen ermöglichen soll:

```
public interface Batchlet {
    public String process();
    public void stop();
}
```

Listing 1: ItemReader, ItemProcessor und ItemWriter Interface

```
public interface ItemReader {
  void open(Serializable checkpoint) throws Exception;
  void close() throws Exception;
  Object readItem() throws Exception();
  Serializable ceckpointInfo() throws Exception;
}

public interface ItemProcessor {
  Object processItem() throws Exception();
}

public interface ItemWriter {
  void open(Serializable checkpoint) throws Exception;
  void close() throws Exception;
  Object writeItems(List<Object> items) throws ...;
  Serializable ceckpointInfo() throws Exception;
}
```

Listing 2: Job-Deklaration

```
<job id="loganalysis" restartable="true">
  <properties>
    <property name="input_file" value="input1.txt"/>
    <property name="output_file" value="output2.txt"/>
  </properties>

  <step id="logprocessor" next="cleanup">
    <chunk checkpoint-policy="item" item-count="10" skip-limit="5" retry-limit="5">
      <reader ref="com.xyz.pkg.LogItemReader"></reader>
      <processor ref="com.xyz.pkg.LogItemProcessor"></processor>
      <writer ref="com.xyz.pkg.LogItemWriter"></writer>

      <skippable-exception-classes>
        <include class="pkg.MyItemException"/>
        <exclude class="pkg.MyItemSeriousSubException"/>
      </skippable-exception-classes>
      <retryable-exception-classes>
        <include class="pkg.MyResourceTempUnavailable"/>
      </retryable-exception-classes>

    </chunk>
  </step>

  <step id="cleanup">
    <batchlet ref="com.xyz.pkg.CleanUp"></batchlet>
    <end on="COMPLETED"/>
  </step>
</job>
```

Special Features

Um die laufendende Verarbeitung eines Batch-Jobs zu fast jedem Zeitpunkt via Callback-Mechanismus manipulieren zu können, stehen dem Entwickler mehr als zehn verschiedene Listener-Interfaces zur Verfügung, die den Vertrag zwischen Java-Batch-API und Runtime definieren und in der Regel zusätzlich als abstrakte Implementierungen vorliegen. Mittels *JobListener* und *StepListener* kann, unter Verwendung der Methoden *beforeXYZ()* bzw. *afterXYZ()*, direkt in den Lifecycle vor bzw. nach einem Job/Step eingegriffen werden. Gleiches gilt für den *ChunkListener*, der darüber hinaus noch eine *onError()*-Methode zur gezielten Reaktion auf während der Verarbeitung aufgetretene Fehler bietet, und zwar bevor die Transaktion zurückgerollt wurde. Zum Monitoring stehen die drei folgenden Listener zur Verfügung: *ItemReaderListener*, *ItemProcessorListener* und *ItemWriterListener*.

Batch Runtime

Wir haben nun die eine oder andere Möglichkeit zur Manipulation einer laufenden Batch-Verarbeitung kennengelernt. Aber wie stößt man eine solche Verarbeitung überhaupt an?

Angenommen, es existiert ein Job mit der ID *myJob*, der bei seinem Aufruf verschiedene Parameter, wie zum Beispiel einen Filenamen für eine Datei mit Item-Informationen, erwartet. Um diesen Job zu starten, muss zunächst mittels *BatchRuntime* ein Zugriff auf den *JobOperator* erfolgen, um dann mit dessen Hilfe den Job zu starten:

```
JobOperator jobOperator = BatchRuntime.getJobOperator();
Properties props = new Properties();
props.setProperty("fileName", "oneMillionItems.txt");
...
long execID = jobOperator.start("myJob", props);
```

Der Rückgabewert Execution-ID dient zur eindeutigen Identifikation der Jobinstanz.

Batch-Job-Deklaration

Mithilfe der *Job Specification Language* (kurz: JSL) können Batch-Jobs via XML deklariert werden. Für jedes der oben genannten Konstrukte gibt es ein entsprechendes Pendant innerhalb der JSL. Listing 2 zeigt exemplarisch eine für diesen Artikel leicht modifizierte Job-Deklaration aus dem Java-EE-7-Tutorial [5].

Der Job besteht aus zwei aufeinander basierenden Schritten, wobei der erste eine klassische Chunk-Verarbeitung, inklusive *ItemReader*, *ItemProcessor* und *ItemWriter*, darstellt und der zweite Schritt ein Batchlet ist, das „Aufräumarbeiten" vornimmt. Mit der Zeile

```
<chunk checkpoint-policy="item" item-count="10" skip-limit="5"
                                                retry-limit="5">
```

wird festgelegt, dass unabhängig von der tatsächlich zu verarbeitenden Item-Menge alle zehn Items ein Checkpoint gesetzt wird und so bei Bedarf beim jeweils letzten Checkpoint wieder aufgesetzt werden kann. Mit dem Attribut *skip-limit* wird definiert, wie häufig ein Überspringen der Verarbeitung einzelner Items aufgrund einer unter *skip-exception-classes* aufgeführten Exception maximal erlaubt ist. Mit der Kombination dieser beiden Werte (*skip-limit* und *skip-exception-classes*) lässt sich somit ein Threshold für die Item-Verarbeitung definieren. *retry-limit* und *retry-exception-classes* sind äquivalent dazu, nur dass in diesem Fall die Verarbeitung des Items nicht übersprungen, sondern wiederholt wird.

Die angegebenen Properties *input_file* und *output_file* lassen sich übrigens innerhalb eines Java-Programms direkt injizieren. Die Runtime muss dabei sicherstellen, dass dies auch dann funktioniert, wenn keine Implementierung des JSR 299 oder JSR 330 zur Verfügung steht:

```
@Inject @BatchProperty(name="input_file") String inputFileName;
@Inject @BatchProperty(name="output_file") String outputFileName;
```

Umgekehrt kann innerhalb der XML-Deklaration des Batch-Jobs auf CDI-Ressourcen zugegriffen werden, die mit einer *@Named*-Annotation versehen sind.

Parallelverarbeitung

Mithilfe des *partition*-Elements kann innerhalb einer *step*-Deklaration angegeben werden, dass ein Step, in mehrere Partitionen aufgeteilt, parallel verarbeitet werden soll. Die *BatchRuntime* erzeugt beim Starten des partitionierten Steps für jede Partition einen eigenen *ItemReader*, *ItemProcessor* und *ItemWriter* und stößt so die parallele Verarbeitung des Steps an. Weiß man bereits zum Zeitpunkt der Konfiguration, wie viele Partitionen es am Ende geben wird, können diese inklusive ihrer dynamischen Properties innerhalb des *partition*-Tags durch das Tag *plan* angegeben werden. Alternativ dazu kann das Tag *mapper* verwendet und so, statt der statischen Deklaration, eine konkrete Implementierung zur dynamischen Berechnung eines Partition-Plans angegeben werden.

Was ist nun aber, wenn eine der Partitionen auf (Zwischen-)Ergebnisse einer anderen Partition zugreifen möchte? Oder was passiert, wenn eine Partition fehlerhaft aussteigt, die anderen Partitionen aber mit ihrer Abarbeitung noch nicht am Ende sind? Für diese Anwendungsfälle wurden drei weitere Tags eingeführt: *reducer*, *collector* und *analyzer*. Mithilfe des *PartitionReducer* erhält man zu Beginn und zum Ende eines jeden Partition-Steps bzw. bei einem Rollback via Callback-Methoden die Kontrolle und kann so gezielt Ergebnisse einzelner Partitionen bewerten, verdichten oder ggf. die gesamte Verarbeitung beenden. Der *PartitionCollector* wird ebenfalls regelmäßig während der parallelen Verarbeitung aufgerufen und stellt eine Art gemeinsamen Sammelpunkt innerhalb des Parent-Threads für alle Partitionen bzw. deren Berechnungen dar. Mit seiner Hilfe können die jeweiligen Zwischenergebnisse in Form des serialisierbaren Rückgabewerts der Callback-Methode *collectPartitionData()* an einen *PartitionAnalyser* weitergeleitet werden.

Der *PartitionAnalyser* erfüllt gleich zwei Aufgaben: Zum einen analysiert er nach jedem Aufruf des *PartitionCollector* dessen aggregierte Daten; zum anderen prüft er am Ende eines jeden Partition-Steps den aktuellen Status, um so ggf. den weiteren Verlauf der Batch-Job-Verarbeitung beeinflussen zu können.

Möchte man, anders als eben beschrieben, nicht einzelne Schritte parallelisieren, sondern pro Thread unterschiedliche Teile des gesamten Batch-Jobs verarbeiten lassen, kann dies via *split*- und *flow*-Tag erreicht werden.

Fazit

Auch wenn bei ersten Tests mit dem GlassFish 4 nicht alles so lief wie in der Spezifikation beschrieben, macht bereits die erste Version der „Batch Application for the Java Platform 1.0" einen ausgereiften Eindruck. Was auf den ersten Blick vielleicht den einen oder anderen Early Adopter überraschen dürfte („never, ever use a version 1.0 in production!"), erklärt sich aus der Tatsache, dass etliche namhafte und etablierte Vertreter des Genres „Batch" (u. a. [2], [3] und [4]) Pate für die neue Spezifikation gestanden haben und somit bereits in der ersten Version auf eine langjährige Erfahrungshistorie zurückgegriffen werden konnte.

Die CDI-Integration scheint derzeit noch ein wenig rudimentär zu sein, bietet aber immerhin schon die Möglichkeit, eine CDI Bean innerhalb der XML-Deklaration zu referenzieren bzw. eine Property aus der XML-Deklaration direkt innerhalb des Java-Codes zu injizieren.

Man darf gespannt darauf sein, welches Bild sich ergibt, wenn die ersten großen Projekte auf Basis des Frameworks umgesetzt und/oder migriert wurden. In diesem Sinne: Stay tuned.

Links & Literatur

[1] Batch Application for the Java Platform 1.0: http://jcp.org/en/jsr/detail?id=352

[2] Spring Batch: http://static.springsource.org/spring-batch/

[3] WebSphere Compute Grid: http://www.ibm.com/developerworks/downloads/ws/wscg/

[4] z/OS Batch: http://www.redbooks.ibm.com/abstracts/sg247779.html

[5] Java EE 7 Tutorial: http://docs.oracle.com/javaee/7/tutorial/doc/home.htm

Anzeige

Anzeige

Technische Evolution und ein wichtiges Signal! Die Community atmet auf

GWT 2.5 ist da

Das aktuelle Release des Google Web Toolkits liefert neben der Einführung einiger teilweise noch als experimentell eingestufter Features vor allem Verbesserungen am Herzstück der Plattform, dem Java nach JavaScript Crosscompiler. Doch allein die Veröffentlichung eines neuen Release lässt die Community aufhorchen …

von Ferdinand Schneider

Das Google Web Toolkit ermöglicht auf einzigartige Weise die ingenieursmäßige Entwicklung komplexer Webanwendungen unter Verwendung der gängigen Java-Toolchain mit IDEs, Bibliotheken, Debugging, Build-Tools und Testing sowie dem Einsatz bekannter Entwicklungsprozesse. Möglich macht dies der einzigartige GWT-Compiler, der Java-Code direkt in crossbrowserfähige Ajax-Anwendungen übersetzt. GWT wird daher von den Unternehmen in einer Vielzahl von produktiven webbasierten Enterprise-Anwendungen eingesetzt, und hat bei den Nutzungszahlen mit den etablierten Industriestandards wie JSF und Struts gleichgezogen.

GWT wurde im Mai 2006 von Google veröffentlicht und konnte in den Anfängen nur einen geringen Teil des Java-Sprachumfangs abbilden, die Einschränkungen bei der Entwicklung waren gegenüber anderen Java-UI-Frameworks noch sehr groß. Doch Google investierte viele Ressourcen in die neue Plattform und machte in den folgenden fünf Jahren aus GWT ein sehr mächtiges und produktives Framework zur Entwicklung von „large scale web applications“. Dies spiegelt sich auch in den Nutzungszahlen wieder. Offizielle Statistiken von Google sprechen von über 100 000 aktiven Entwicklern und 1 Million SDK-Downloads.

Doch nach dem Release 2.4 im September 2011 wurde es ruhiger. Von Google gab es keine Auskünfte zu einer Roadmap und damit der Zukunft der Plattform, gleichzeitig verunsicherte der Internetkonzern die GWT-Community mit Meldungen über die massive Verlagerung von Entwicklerkapazitäten auf die neue Webprogrammiersprache Dart. Spekulationen über die Aufgabe von GWT zugunsten von Dart machten in der Blogosphäre die Runde. Viele Unternehmen, die inzwischen große Investitionen in ihre GWT-Anwendungslandschaft getätigt hatten, bekamen deutlich zu spüren, dass GWT kein vollständig offenes Projekt, sondern von den Planungen des US-Konzerns abhängig ist, der selbst viele seiner Anwendungen auf Basis von GWT entwickelt hat.

Auf der Hausmesse Google I/O im Juni 2012 kam dann der Befreiungsschlag. Ray Cromwell, Technical Lead bei Google, verkündete neben der Vorstellung des neuen GWT-Release 2.5 die Gründung eines Steering Committee, dem die Verantwortung für die zukünftige Entwicklung der Plattform übertragen wird [1]. Gleichzeitig soll eine stärkere Einbindung der Community erfolgen. Damit haben Unternehmen zum ersten Mal die Möglichkeit, die weitere Entwicklung von GWT maßgeblich zu beeinflussen, da Google in Zukunft nur noch ein gleichberechtigter Partner in den Entscheidungsprozessen ist.

Die neue Offenheit zeigt sich erstmals, als das Steering Committee eine Umfrage unter Entwicklern startete und unter anderem eine „Wishlist“ für die Plattform zusammenstellte [2], [3]. Ganz oben auf der Liste der meistersehnten Features waren unter anderem Performanceverbesserungen am Compiler und dem generierten Ajax-Sourcecode sowie eine effizientere Entwicklungsumgebung. Mit den neuen Features der Version 2.5 wird vielen dieser Wünsche Rechnung getragen und gleichzeitig Googles Botschaft untermauert: GWT ist am Leben und hat noch eine lange Zukunft vor sich.

Performanceverbesserungen

Bekanntermaßen ist die Start-up-Performance der generierten Ajax-Anwendungen stets eine Herausforderung in der GWT-Entwicklung. Die Problematik ist das Konzept von GWT, alle Änderungen an der Benutzeroberfläche, wie bspw. den Wechsel in einen anderen Dialog, nicht wie bei klassischen Webframeworks durch erneute HTTP-Requests, sondern durch DOM-Manipulationen mit JavaScript auszuführen. Daraus folgt, dass bereits im initialen Page Load der Webapplikation die gesamte Anwendungslogik inklusive der Benutzeroberfläche geladen wird, was häufig zu sehr großen Downloads und damit einer langen Ladezeit beim Aufruf der Seite führt. Neben weiteren Optimierungsmöglichkeiten ist die Reduzierung des vom Compiler generierten Ajax-Codes also einer der wesentliche Stellhebel, wenn es um die Start-up-Performance von GWT-Anwendungen geht.

Genau hier wurden in GWT 2.5 große Fortschritte gemacht. Durch Optimierungen ist es gelungen, die Größe des generierten Ajax-Codes signifikant zu senken. Um die Verbesserungen zu quantifizieren, hat der Autor zwei Anwendungen jeweils mit GWT 2.4 und 2.5 kompiliert und die Größe des generierten JavaScripts (Permutationen plus Fragmente) gegenübergestellt. Zur besseren Nachvollziehbarkeit durch die Leser wurden die öffentlich verfügbaren GWT-Sample-Projekte DynaTableRF und der GWT-Showcase für den Vergleich herangezogen. Die Analyse ergibt, dass allein der Wechsel des Compilers den generierten JavaScript-Output um ca. fünf Prozent verringert. Tatsächlich dürfte diese Zahl noch höher ausfallen, da beispielsweise der Showcase von GWT 2.4 weniger umfangreich als sein aktuelles Pendant ist und somit relativ einen noch größeren Compileroutput hat. Die Betrachtung weiterer Real-Life-Projekte zeigt, dass mit steigender Komplexität der Anwendungen die Reduzierung noch größer ausfällt und häufig bis zu 20 Prozent beträgt.

Doch das genügt den GWT-Entwicklern nicht. Für GWT 2.5 bedienen sie sich aus dem hauseigenen Sortiment und haben den aus der konventionellen JavaScript-Entwicklung bereits bekannten JavaScript-Compiler aus dem Closure-Toolkit von Google in GWT integriert [4]. Der Closure-Compiler optimiert bestehenden JavaScript-Code durch eine Reihe von Maßnahmen wie bspw. die Reduktion von Funktionen und das Entfernen toten Codes. Da lediglich der von GWT erzeugte Code nochmals in effizienteres JavaScript übersetzt wird, hat der Einsatz des Closure-Compilers keine Auswirkungen auf die Entwicklung des Java-Quellcodes und kann somit bedenkenlos durch das Compiler-Flag *-XenableClosureCompiler* aktiviert werden. In der bereits bekannten Teststellung wird durch den Closure-Compiler eine weitere Reduzierung des generierten JavaScripts um sechs (Showcase) bis acht (DynaTableRf) Prozent erreicht. **Abbildung 1** veranschaulicht die Verbesserungen bei der Größe des Compiler-Outputs in einem Diagramm.

Trotzdem sollte man den Einsatz des Closure-Compilers mit Vorsicht genießen. Nicht ohne Grund wurden die Closure-Optimierungen nicht direkt in den GWT-Compiler integriert, sondern müssen expliziert durch ein Flag aktiviert werden. Motivation hierfür ist offensichtlich die Dauer des Compilevorgangs, ein weiteres Dauerthema unter GWT-Entwicklern. Diese schnellt nämlich beim Einsatz der neuen Compileroption eklatant in die Höhe. Und mit eklatant ist eine tatsächliche Erhöhung der Kompilierungszeit in der Teststellung um bis zu 150 Prozent gemeint! Wer also auf häufige Builds seiner Anwendung angewiesen ist (bspw. wegen der Entwicklung für mobile Browser, für die kein Dev-Mode-Plug-in existiert) sollte sich zumindest überlegen, den Closure-Compiler erst zum Ende der Entwicklungsphase zu aktivieren. Da keine funktionellen Änderungen hierdurch zu befürchten sind, kann dies zum Glück bedenkenlos getan werden.

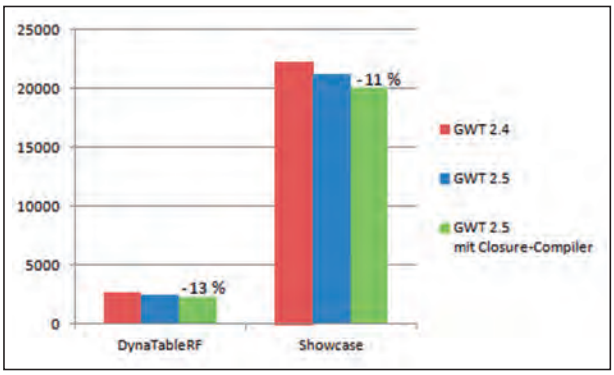


Abb. 1: Größe des vom GWT-Compiler generierten JavaScripts in Kilobyte

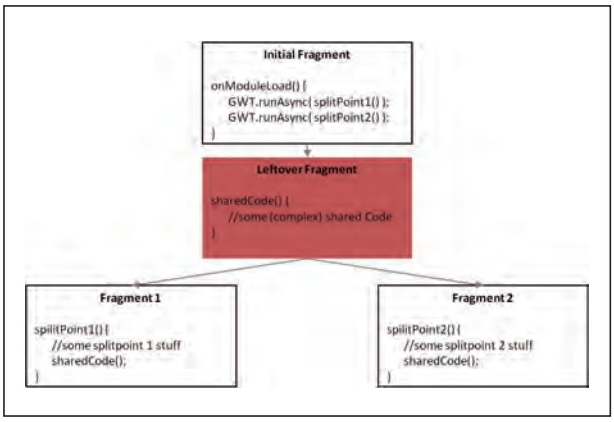


Abb. 2: Bisheriger Umgang mit gemeinsam genutztem Code in Fragmenten

Aktiviert man neben dem Einsatz des aktuellen GWT-Compilers also zusätzlich noch den Closure-Compiler und diverse weitere Optimierungen, kann die Start-up-Time einer GWT-Anwendung sehr stark verringert werden. Doch nicht nur die Größe der Applikationen, auch die Laufzeiteffizienz wurde mit der neuen Version verbessert. Fragment Merging optimiert das Nachladen von Funktionalitäten in einer Anwendung zur Compile-Zeit. Durch das Einfügen von Split Points mittels der Funktion *runAsync* wird der Crosscompiler angewiesen, den Inhalt von *runAsync* in eine eigene JavaScript-Datei zu verpacken, die dann beim Aufruf der Funktion vom Browser nachgeladen werden kann. Der initiale Page Load einer Anwendung wird damit reduziert. Beim GWT-Compiler entsteht im Idealfall für jeden Split Point eine eigene JavaScript-Datei, die dann am Client asynchron geladen wird. Problematisch hierbei ist von mehreren Fragmenten gemeinsam genutzter Code. Bislang wurde dieser komplett in ein Leftover Fragment gepackt (**Abb. 2**).

Sobald der erste Split Point in einer Anwendung erreicht ist, muss das komplette Leftover Fragment mit geladen werden. In großen Anwendungen kann dieses sehr umfangreich werden und damit der erste asynchrone Funktionsaufruf extrem lange dauern.

GWT 2.5 optimiert diesen Prozess durch das Fragment Merging. Der GWT-Compiler erkennt, welcher Code gemeinsam von welchen Fragmenten genutzt

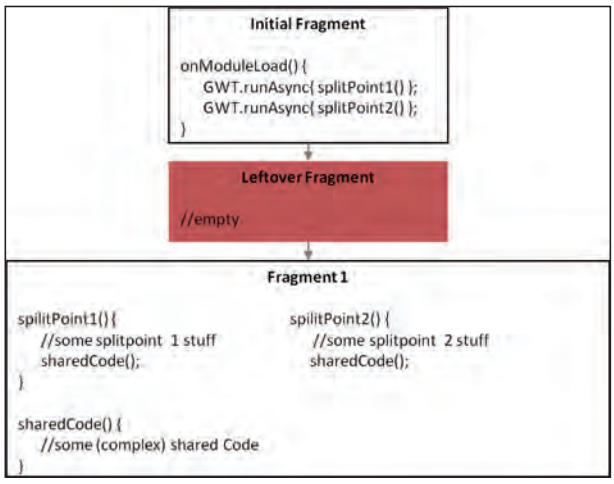


Abb. 3: Leeres Leftover Fragment durch Fragment Merging

wird und fasst diese inklusive des gemeinsam genutzten Codes zu einem neuen Fragment zusammen (**Abb. 3**).

Das Leftover Fragment wird bei dieser Vorgehensweise auf ein Minimum reduziert, wodurch die Ladezeit für gemeinsam genutzten Code besser auf alle asynchronen Funktionsaufrufe verteilt wird. Fragment Merging wird durch das Compiler-Flag *-XfragmentCount* aktiviert, wobei als Parameter noch die gewünschte Maximalzahl an exklusiven Fragmenten mitgegeben werden muss. Hier ist einiges an Experimentieren gefragt, um eine für die Anwendung ausgewogene Einstellung zu finden. Praktischerweise wird einem auch hier vom GWT-Compiler unter die Arme gegriffen, indem dieser sich nicht strikt an die Vorgabe hält, sondern die Anzahl selbstständig erhöht, wenn eine derart geringe Zahl an einzelnen Fragmenten nicht sinnvoll ist. Übrigens: Im Gegensatz zum Closure-Compiler hat der Einsatz von Fragment Merging kaum Auswirkungen auf die Compile-Zeit.

Als letzte Performanceoptimierung wurde für GWT 2.5 die Geschwindigkeit des Development Modes verbessert und auch die für Entwickler sehr wichtige Refresh Time verkürzt.

Super Dev Mode

Der bislang für die GWT-Entwicklung zum Einsatz kommende Development Mode hat einige strukturelle Schwächen. Das Konzept basiert auf Browser-Plug-ins, die das Ausführen einer GWT-Anwendung als Java-Bytecode in einer JVM ermöglichen. Nach jedem Refresh der Anwendung im Browser wird die komplette Applikation neu gebaut und durch das Plug-in im Browser angezeigt. Bei der Entwicklung großer Anwendungen kann dieser Refresh sehr lange dauern, was den Entwicklungs-Roundtrip zur Geduldsprobe macht. Eine weitere Herausforderung ist die Aktualisierung der für den Dev Mode nötigen Plug-ins. Die Releasezyklen der Browserhersteller sind extrem kurz, und somit ist es sehr aufwändig, stets für die neuesten Browserversionen funktionierende Plug-ins bereitzustellen. Die Entwicklung auf mobilen Geräten bleibt mangels passender Plug-ins derzeit in der GWT-Entwicklung völlig außen vor. Hier bleibt einem derzeit nur der Test mit einem echten Deployment der Anwendung.

Für den Entwicklungsmodus wäre eine auf Standardtechnologien basierende Lösung, die viele der wesentlichen Probleme des Dev Modes aus der Welt schaffen würde, wesentlich eleganter. Mit Source Maps existiert seit Kurzem eine Spezifikation, die auf lange Sicht den klassischen Dev Mode von GWT überflüssig machen und viele neue Möglichkeiten eröffnen könnte. Source

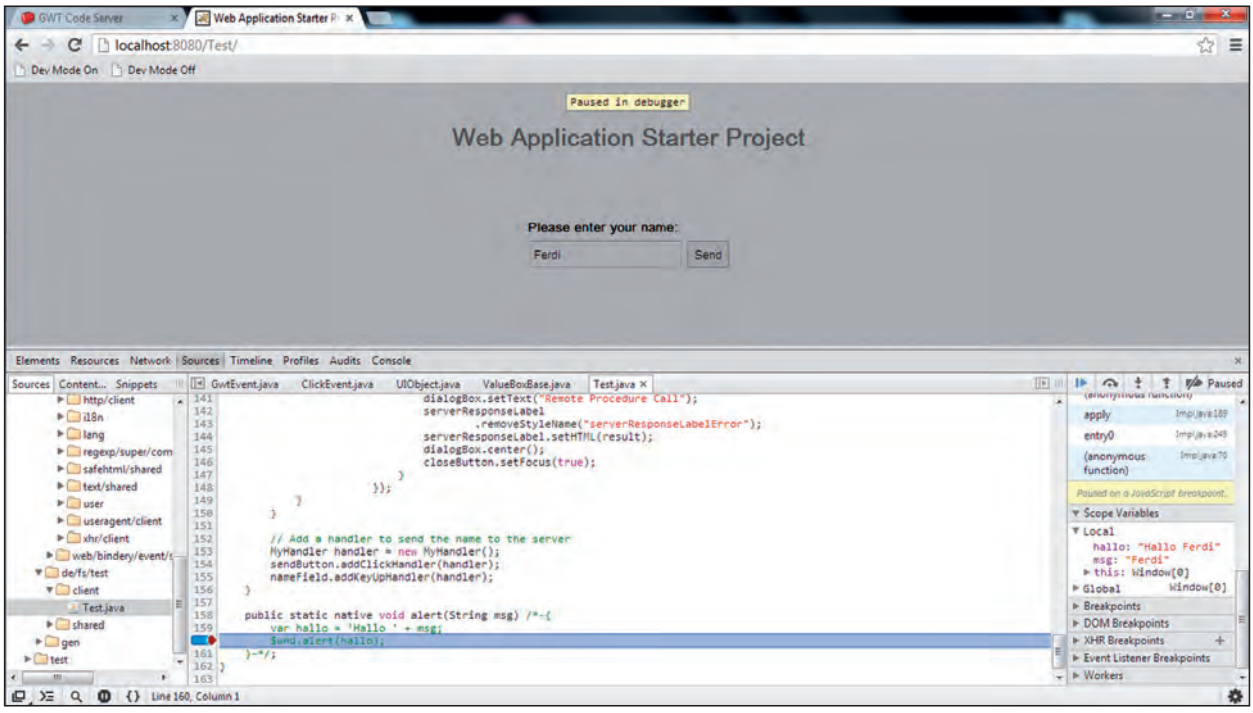


Abb. 4: Der neue Super Dev Mode ermöglicht das Debuggen der Java-Sources direkt im Browser

Maps ermöglichen dem Browser das Mapping von ausgeführtem JavaScript auf andere Source-Dateien als die, die gerade tatsächlich geladen sind. Ein mögliches Anwendungsbeispiel ist das Debugging von mit dem Closure-Compiler optimierten Ajax-Anwendungen. Statt dem nicht menschenlesbaren Code werden im Debugging-Fenster des Browsers die original Sourcen angezeigt und können auch debuggt werden.

Basierend auf dieser Technologie haben die GWT-Entwickler in 2.5 einen neuen Entwicklungsmodus integriert, den Super Dev Mode. Dieser folgt einem völlig anderen Konzept als der bekannte Dev Mode. Statt im Hintergrund direkt den Java-Quellcode in einer JVM auszuführen, wird die Anwendung tatsächlich in JavaScript übersetzt. Dies macht ein Browser-Plugin überflüssig, da nun schon während der Entwicklung eine vollwertige Ajax-Anwendung vorliegt. Um den Code dennoch debuggen zu können, werden die generierten JavaScript-Sources via Source Maps auf die entsprechenden Java Sources gemappt. Somit ist es möglich, mit den Entwicklungstools des jeweiligen Browsers den Java-Code zu debuggen. **Abbildung 4** zeigt dies anhand eines einfachen Beispiels. Bei genauem Hinsehen veranschaulicht das dargestellte Snippet einen weiteren großen Vorteil des Super Dev Modes, nämliche das nahtlose Debugging von Java Sources und JSNI-Methoden. Dies ist vor allem wichtig, wenn komplexe, bestehende JavaScript-Bibliotheken in die GWT-Anwendung eingebunden werden müssen. Bislang ist dies häufig mit einer langen und aufwendigen try-and-error-Session verbunden.

Wer jetzt schon von der GWT-Entwicklung auf seinem Smartphone träumt, bei der via Remote-Debugging der Java-Code direkt im Browser des mobilen Devices debuggt werden kann (auch der Autor träumt seit Langem davon), der muss sich leider noch etwas gedulden. Der Super Dev Mode ist hoch experimentell und derzeit noch weit von einem produktiv einsetzbaren Stadium entfernt. Viele, vor allem größere, Anwendungen lassen sich häufig überhaupt nicht starten, müssen für die Entwicklung mit dem neuen Werkzeug speziell vorbereitet werden und unterliegen einigen Restriktionen. Ein weiteres Hemmnis ist die derzeit geringe Verbreitung von Source Maps in den Browsern. Lediglich Google Chrome sowie seit der Beta von Version 23 Firefox unterstützen dieses für die Entwicklung mit dem Super Dev Mode unerlässliche Feature, und auch hier ist das Zusammenspiel häufig alles andere als reibungslos.

Wer das neue Tool trotzdem ausprobieren möchte, sollte sich auf viel Ausprobieren gefasst machen. Bereits das korrekte Einrichten und Starten des Tools erfordert

Anzeige

einiges an Einstellungen, und die sehr dürftige Dokumentation im GWT Development Guide [5] ist hier nur selten eine Hilfe. Wohl nicht umsonst wünschen die Entwickler dem interessierten Leser zum Abschluss der Super-Dev-Mode-Dokumentation „Happy hacking!".

Neue Features

Eine Eigenschaft vieler Frameworks ist, dass sie die darunterliegende Technologie abstrahieren und damit verbergen. Weiterentwicklungen an der Basistechnologie müssen erst von den Frameworkentwicklern nach-

Super Dev Mode in Action

Sie wollen den Super Dev Mode einmal selbst ausprobieren? Mit der folgenden Anleitung wird Ihnen alles Wichtige Schritt für Schritt erklärt. Grundkenntnisse im Umgang mit GWT werden vorausgesetzt, Kenntnisse mit der Entwicklungsumgebung Eclipse und dem GWT-Plug-in sind von Vorteil.

Vorbereitungen

Der Super Dev Mode wird derzeit noch sehr stark weiterentwickelt. Es wird daher dringend empfohlen, stets die neueste GWT-Version einzusetzen, um in den Genuss der letzten Hotfixes zu kommen. Es kann also passieren, dass das hier gezeigte Tutorial nicht mehr funktioniert, sobald neue Hotfixes nach Redaktionsschluss erscheinen.
Laden Sie sich also die derzeit aktuellste GWT-Version 2.5.1 auf der Projektseite herunter. Vermeiden Sie den Einsatz von GWT 2.5.0 für den Super Dev Mode, sie werden damit auf Grund von Inkompatibilitäten erheblich mehr Probleme haben.

Einrichtung der GWT-Projekte

Richten Sie das heruntergeladene SDK als GWT-Bibliothek für die Projekte ein, mit denen Sie den Super Dev Mode erproben wollen. In Eclipse gehen sie dazu wie folgt vor:

1. Window | Preferences | Google | Web Toolkit | Add...
2. Wählen Sie im Pop-up das Installationsverzeichnis des GWT SDK aus
3. Rechtsklick auf das Projekt | Properties | Google | Web Toolkit
4. Aktivieren Sie *Use Google Web Toolkit* und wählen Sie das 2.5.1 SDK aus

Anpassungen der GWT-XML-Dateien

Zur Verwendung mit dem Super Dev Mode müssen die Anwendungen derzeit noch mit speziellen Konfigurationen in der GWT-XML ausgestattet werden. Fügen Sie folgende Deklarationen ein:

- <add-linker name="xsiframe" />
- <set-configuration-property name="devModeRedirectEnabled" value="true" />
- <set-property name="compiler.useSourceMaps" value="true" />

Die erste Deklaration weist den GWT-Compiler an, den für den Super Dev Mode derzeit noch unverzichtbaren iFrame Linker zu verwenden. Danach folgen Anweisungen, um Redirects für den Dev Mode zu aktivieren und für die Generierung der Source Maps.

Starten des Super Dev Modes

Ihre Anwendung muss mit folgender Konfiguration gestartet werden:

- Im Classpath liegt neben den für das Projekt benötigten Sources die *gwt-codeserver.jar* aus dem GWT SDK
- Main Class ist die Klasse *com.google.gwt.dev.codeserver.CodeServer*
- Folgende Aufrufparameter sind zwingend zu setzen: *[-src dir]* und *[module]*, wobei *dir* das Source-Verzeichnis im Projekt und *module* der vollqualifizierte Name des zu startenden GWT-Moduls ist

Mit Eclipse erstellen Sie einfach eine Run-Konfiguration auf das entsprechende Projekt. Dort legen Sie wie oben beschrieben die Main Class fest, setzen unter *Arguments* die Aufrufparameter und ergänzen im Reiter Classpath die *gwt-codeserver.jar*. Alles Weitere erledigt Eclipse.

Starten der Anwendung im Browser

1. Stellen Sie sicher, dass Ihre Anwendung auf einem Webserver deployt ist.
2. Ist der Super Dev Mode gestartet, öffnen Sie den in der Ausgabe angegebenen URL (per Default http://localhost:9876) in Ihrem Browser.
3. Ziehen Sie nun die beiden Buttons *Dev Mode On* und *Dev Mode Off* in die Bookmark-Leiste des Browsers.
4. Öffnen Sie Ihre GWT-Anwendung.
5. Klicken Sie auf *Dev Mode On* und wählen das GWT-Modul der Applikation im geöffneten Pop-up aus.

Ihre Anwendung ist nun im Super Dev Mode gestartet, herzlichen Glückwunsch!

Debugging mit Source Maps

Um die Anwendung debuggen zu können, müssen Sie im jeweiligen Browser die Source Maps aktivieren (Chrome, Firefox). Vergewissern Sie sich, dass Sie eine kompatible Browserversion installiert haben.
Unter Chrome gehen Sie in die Entwicklungstools und aktivieren dort in den Einstellungen die Konfiguration Enable source maps. In Firefox wechseln sie unter Extras | Web-Entwickler in den Debugger. Dort erreichen Sie rechts über das Einstellungensymbol ein Menü, in dem Sie Ursprüngliche Quelle anzeigen auswählen.
Rufen Sie nun die Anwendung erneut wie oben beschrieben im Super Dev Mode auf. Im Debugger des jeweiligen Browsers finden Sie nun unter Domäne/Port des Super Dev Modes (bspw. http://localhost:9876) neben den JavaScript-Sources der Anwendung auch die entsprechenden Java-Dateien. Hier können Sie wie gewohnt Breakpoints setzen und Variablen auswerten. Viel Spaß damit!

gezogen werden, ehe die Anwender in den Genuss neuer Features kommen. Auch bei GWT hat man durch das „Wrappen“ von JavaScript mit einer anderen Programmiersprache eine solche Innovationsbremse installiert. Neue Features und Anpassungen, die derzeit durch das intensive Vorantreiben von HTML5 in der Webentwicklung sehr häufig sind, können aller frühestens in der nächsten GWT-Version, häufig jedoch erst viele Releases in der Zukunft, zur Verfügung gestellt werden. Will oder kann ein Unternehmen nicht so lange warten, muss bislang auf sehr umständliche Weise nativer JavaScript-Code in der Anwendung verbaut werden, was ein enormer Komplexitätstreiber ist.

Das zeitliche Lag, mit dem neue Features aus der Webwelt in GWT zur Verfügung stehen, kann zukünftig durch das neue, derzeit noch experimentelle Elemental-API drastisch verkürzt werden. Hierbei handelt es sich um ein komplett neues API mit eigener Typhierarchie, das direkt aus dem JavaScript-Standard, der Web IDL des W3C [6], generiert ist. Die Entwicklung mit GWT fühlt sich damit fast an wie „echte“ JavaScript-Programmierung.

Listing 1 zeigt, wie mit Elemental ein einfaches Hello World gebaut wird. Wer schon einmal JavaScript entwickelt hat, dem wird sofort auffallen, dass man den Code mit nur wenigen Änderungen auch als JavaScript-Anwendung laufen lassen könnte. Leider merkt man bereits bei einem solch einfachen Beispiel das experimentelle Stadium des Frameworks. Ein Starten im Dev Mode ist leider nicht möglich, dieser streikt reproduzierbar bei der Registrierung von Event Handlern. Erst

Listing 1

```
public class Elemental implements EntryPoint {

  public void onModuleLoad() {
    final ButtonElement button = Browser.getDocument()
        .createButtonElement();

    button.setTextContent("Say Hello!");

    Browser.getDocument().getBody().appendChild(button);

    button.addEventListener(Event.CLICK, new EventListener() {
      @Override
      public void handleEvent(Event evt) {
        Browser.getWindow().alert("Hello World!");
      }
    }, false);
  }
}
```

Anzeige

ein Build und Deploy im Webserver lassen das in **Abbildung 5** dargestellte Ergebnis erscheinen.

Die Java-Typen, die im Elemental-API zum Einsatz kommen, sind größtenteils so genannte Overlay Types, die ein Minimum an Overhead beim Crosscompilen nach JavaScript erzeugen. Dies führt dazu, dass mit Elemental entwickelte GWT-Anwendungen extrem schlank und schnell sind.

Um den Nutzen zu veranschaulichen, soll in einem weiteren Beispiel mittels Elemental das neue HTML5-File-API, das einen besseren Zugriff auf das Dateisystem des Clients ermöglicht, verwendet werden. Bislang wäre dies in GWT nur über den Einsatz nativer JavaScript-Methoden möglich gewesen, die in der Regel von Open-Source-Projekten in Form von Bibliotheken bereitgestellt wurden. Allein für das File-API existieren auf Google Code mehrere GWT-Wrapper. Wie wir gleich sehen werden, sind sie ab sofort überflüssig!

In Listing 2 wird die für unseren Versuchsaufbau nötige Oberfläche nach bekanntem Muster erstellt. Zuerst werden mit *createXXXElement* die benötigten DOM-Elemente erzeugt und via *set*-Methoden die entsprechenden Attribute gesetzt. Zum Schluss werden die Elemente noch dem Body hinzugefügt. Die Beispielanwendung soll die Auswahl beliebig vieler Dateien auf dem Dateisystem des Clients durch den Benutzer ermöglichen und deren Eigenschaften wie Name, Dateityp und -größe in einer Liste ausgeben [7].

Nun fehlt noch die Dynamik in unserer Anwendung. Hierfür registrieren wir einen *CHANGE* Event Handler am Input-Element, der sich zunächst die Liste der vom Benutzer ausgewählten Dateien holt und daraufhin für jede Datei einen Eintrag in die Outputliste schreibt. Der entsprechende Code findet sich in Listing 3. Hier wird noch einmal das Konzept hinter Elemental deutlich. Vom Event bis zum DOM-Element kommen durchgängig eigene Typen zum Einsatz, auch ein eigenes Collections-API gibt es. Die Manipulation des DOMs erfolgt wie in JavaScript durch Metaprogrammierung von HTML direkt in der Anwendungslogik. Ruft man die so entwickelte Applikation im Browser auf und wählt einige Dateien im File-Dialog aus, wird die in **Abbildung 6** dargestellte Ausgabe erzeugt.

Listing 2

```
public void onModuleLoad() {
  // Creates the following HTML snippet:
  // <input type="file" name="files[]" multiple />
  // <output id="list"></output>
  final InputElement filesElement = getDocument().createInputElement();
  final OutputElement outputElement = getDocument().createOutputElement();

  filesElement.setType("file");
  filesElement.setMultiple(true);
  filesElement.setName("files[]");

  Element body = getDocument().getBody();
  body.appendChild(filesElement);
  body.appendChild(outputElement);
}
```

Listing 3

```
filesElement.addEventListener(Event.CHANGE, new EventListener() {
  @Override
  public void handleEvent(Event evt) {
    // get the file list
    FileList files = ((InputElement) evt.getTarget()).getFiles();

    // list some properties
    ArrayOf<String> output = Collections.arrayOf();
    for (int i = 0; i < files.length(); i++) {
      File f = files.item(i);

      String type = f.getType().isEmpty() ? "n/a" : f.getType();
      String ouputString = "<li><strong>" + f.getName()
          + "</strong> (" + type + ") - " + f.getSize()
          + " bytes</li>";

      output.push(ouputString);
    }

    // insert into output element
    outputElement.setInnerHTML("<ul>" + output.join("") + "</ul>");
  }
}, false);
```

Listing 4

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'>
  <ui:with field='name' type='java.lang.String' />

  <div ui:field="divElement">
    Hallo <ui:text from='{name}' />.
  </div>
</ui:UiBinder>
```

Listing 5

```
public class HalloWeltCell extends AbstractCell<String> {
  interface HalloWeltUiRenderer extends UiRenderer {
    void render(SafeHtmlBuilder sb, String name);
    DivElement getDivElement(Element parent);
  }

  private static HalloWeltUiRenderer renderer = GWT
      .create(HalloWeltUiRenderer.class);

  @Override
  public void render(Context c, String value, SafeHtmlBuilder sb) {
    renderer.render(sb, value);
  }
}
```

Wie man sieht, können mit Elemental schnell und unkompliziert alle Browserfunktionalitäten, inklusive aller „Hot"-Features bspw. aus der neuesten HTML5-Entwicklung, verwendet werden. Von einer Produktionsreife ist das API leider noch weit entfernt. An zu vielen Stellen merkt man dem Framework den experimentellen Status an, und es gibt bislang keine Möglichkeit zur Integration von Elemental mit dem GWT-Typensystem, was die Einbindung in bestehende Projekte schwierig macht. In Zukunft kann das neue API jedoch die GWT-Entwicklung auf eine neue Ebene heben, da man vermehrt auf den Einsatz von nativem JavaScript verzichten und hochperformante GWT-Anwendungen entwickeln kann.

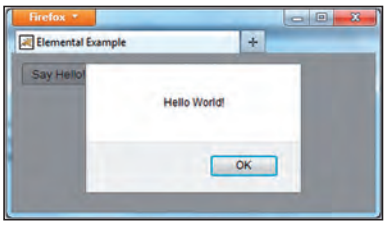


Abb. 5: Hello World mit Elemental-API

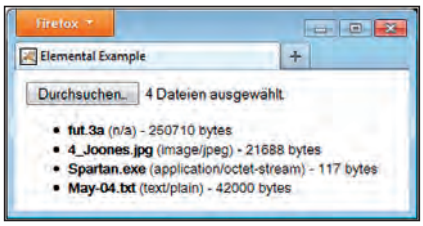


Abb. 6: Elemental-Beispiel mit HTML5-File-API

Eine sehr wichtige Control-Kategorie in GWT sind die Cell-Widgets wie bspw. DataGrid oder CellTree. Sie ermöglichen die Entwicklung komplexer Widgets basierend auf vorgefertigten Cells wie z. B. DatePicker oder Text Input, aber auch mit selbst entwickelten Zellentypen. Bislang werden eigene Cells auf Basis von HTML-Schnipseln, die dann von GWT an der entsprechenden Stelle im Cell-Widget eingefügt werden, implementiert. Die Entwicklung eigener Controls auf Basis einer Konkatenation von HTML-Code ist sehr umständlich und fehleranfällig. GWT 2.5 bietet hier eine neue Möglichkeit auf Basis des UiBinders. Die Cells werden dabei, wie von anderen Widgets vor GWT 2.5 bereits bekannt, in einer XML-Datei via HTML-Templates beschrieben und können dann in Java über einen *UiRenderer* angesprochen werden. Listing 4 und 5 zeigen ein sehr einfaches Beispiel auf Basis des neuen API.

Die Cell wird in einer XML-Datei entworfen, Parametrisierung erreicht man durch das *<ui:with… />*-Tag. Anhand des Namens werden die Elemente mit dem Ui-Renderer *HalloWeltUiRenderer* an Java-Methoden gebunden, was man am Beispiel des UiFields *divElement* sehen kann. Auch das Event Handling sowie CSS-Styling kann auf bekanntem Weg durch Annotationen wie *@UiHandler* und DeferredBinding im Java-Code implementiert werden, für die umfassende Erläuterung all dieser Möglichkeiten sei jedoch auf die Dokumentation verwiesen. In jedem Fall vereinfacht dieser Mechanismus die Entwicklung eigener Cell-Typen enorm und schafft endlich mehr Durchgängigkeit bei der Entwicklung von Oberflächen mit dem UiBinder.

GWT 2.5 bietet noch einige weitere Neuerungen. Zur Entwicklung barrierefreier Anwendungen wurden bislang die statischen Methoden der Klasse *Accessibility* verwendet. Mit ihnen konnte man auf Ebene von HTML-Elementen entsprechende ARIA-Rollen und Status für Widgets setzen. Dieses API wurde durch die neue Klasse *Roles* ersetzt, die aber dem gleichen Prinzip folgt. Generell sollte man für die Implementierung von Barrierefreiheit weiterhin der Empfehlung im GWT Developer Guide folgen und nach Möglichkeit Standard-HTML-Controls einsetzen, die von jedem Screenreader etc. verstanden werden.

Mit den experimentellen Typen *IsRenderable* und *RenderablePanel* wurde der Grundstein für eine neue Art von hochperformanten Controls gelegt, die die Rendering Time signifikant verringern sollen und somit die Performance komplexer Widgets verbessern. Weiterhin ist das Validation-API auf Basis von JSR 303 nicht mehr als experimentell eingestuft und damit für den produktiven Einsatz gewappnet. Inhaltlich hat sich gegenüber GWT 2.4 an der Validierung kaum etwas geändert.

… and the Future …

Mit GWT 2.5 wurden viele Wünsche der GWT-Community angegangen und eine Preview auf kommende Key-Features wie den Super Dev Mode und das Elemental-API gegeben. Durch die Initiierung solcher umfangreicher Innovationen beweist das GWT-Entwicklungsteam, dass das Projekt alles andere als tot ist und in nächster Zeit wieder volle Fahrt aufnehmen wird. Auf der Google I/O 2013 wurde erstmals eine offizielle Roadmap vorgestellt, die die inhaltliche Fokussierung bei der Weiterentwicklung von GWT in den nächsten Jahren festlegt [8]. Mit den kommenden Releases kann sich die Community demnach unter anderem über mehr Offenheit des Frameworks, Performanceverbesserungen sowie Mobile als Schlüsselthema freuen. Sollte dieser Kurs der Offenheit und Anwenderorientierung beibehalten werden, wird GWT in Zukunft wohl noch mehr an Bedeutung gewinnen und den Entwicklern auch weiterhin viel Freude bereiten.

Ferdinand Schneider ist Software Engineer bei der Schwäbisch Hall Kreditservice AG in Baden-Württemberg. Er ist im Java-Ökosystem zu Hause und beschäftigt sich seit einiger Zeit intensiv mit dem Google Web Toolkit, durch das er wieder Spaß an der Webentwicklung gefunden hat.

Links & Literatur

[1] https://developers.google.com/events/io/2012/sessions/gooio2012/218/

[2] https://vaadin.com/gwt/report-2012/wishlist

[3] https://vaadin.com/gwt/report-2012

[4] https://developers.google.com/closure/compiler/

[5] http://www.gwtproject.org/doc/latest/ReleaseNotes.html

[6] http://www.w3.org/TR/WebIDL/

[7] http://www.html5rocks.com/de/tutorials/file/dndfiles/

[8] https://developers.google.com/events/io/sessions/327833110

Anzeige

Anzeige

Anzeige

Red5-Flash-Server: weiterführende Konzepte

Streaming für Anspruchsvolle

Red5 ist ein kostenloser Java-basierter Open-Source-Medienserver, der die Entwicklung von Streaming-Anwendungen durch die Verwendung bekannter Technologien aus dem Java-EE-Umfeld ermöglicht. Red5 eignet sich vor allem zur Realisierung von verteilten Multimedia-Streaming-Anwendungen mit vielen Clients. Der zweite Teil dieser Serie beleuchtet weiterführende Aspekte des Audio- und Videostreamings.

von Sebastian Weber und Cornelius Moucha

Der Red5-Flash-Server [1] stellt eine kostenlose Alternative zu dem kommerziell verfügbaren Flash Media Server (FMS) von Adobe dar. Streaminganwendungen werden hierbei serverseitig in Java geschrieben, und als Clients kommen Flash-basierte Lösungen (zumeist Apache Flex oder Adobe AIR) zum Einsatz. Im ersten Teil dieser Artikelserie wurden Grundlagen der Client-Server-Kommunikation mit Red5 am Beispiel einer einfachen Meeting-Place-Anwendung „Red5Chat" erläutert. Der Fokus lag insbesondere auf dem Datenaustausch zwischen Red5-Server und Flash-Client. Bei der Betrachtung des Clients wurden Red5-spezifische Aspekte und die Netzkommunikation via *NetConnections* hervorgehoben, über die Remote Procedure Calls (RPC) und Streaming ermöglicht werden.

Aufbauend auf den im ersten Teil vorgestellten Grundfunktionen geht der zweite Teil abschließend auf weiterführende Aspekte des Audio-/Videostreamings ein, wie z. B. Audiokommunikation zwischen allen Teilnehmern eines Chatraums oder bidirektionales Videostreaming zwischen zwei Teilnehmern in einem separaten Raum. Im Folgenden sind die Features dieser Beispielanwendungen aufgelistet:

- In Chaträumen können Benutzer mit einem oder mehreren anderen Benutzern Textnachrichten austauschen. Benutzer haben die Möglichkeit, neue separate Chaträume zusätzlich zum Standardraum anzulegen bzw. in andere Räume zu wechseln.
- Weiterhin besteht die Möglichkeit der Audiokommunikation zwischen allen Benutzern eines Chatraums.
- Zusätzliche Chaträume können als „offen" oder „geschlossen" deklariert werden. Letzteres verlangt das Versenden von Einladungen, um dem Raum beitreten zu dürfen.
- Abschließend können Videochaträume explizit für zwei Benutzer erstellt werden, in denen jeweils beide Benutzer per Videostreaming miteinander kommunizieren können.

Erweiterung der Red5-Serveranwendung

Für die Implementierung der serverseitigen Streamingfunktionalität nimmt das Red5-API dem Entwickler vieles ab, da notwendige Basisfunktionen bereits von der Red5-Klasse *MultiThreadedApplicationAdapter* zur Verfügung gestellt werden. Lediglich anwendungsspezifisches Verhalten muss je nach Anforderungen ergänzt werden. Dazu bietet das API mehrere Methoden an, um auf spezifische Streamingevents reagieren zu können.

Für die Umsetzung der Audiokommunikation zwischen allen Teilnehmern eines Chatraums ist es erforderlich, alle verbundenen Teilnehmer über die Verfügbarkeit eines neuen Audiostreams zu informieren. Daraufhin können sich diese Teilnehmer für den neuen Audiostream anmelden („Publish-Subscribe-Prinzip") und ihn abspielen. Dazu ist es notwendig, die Methode *streamBroadcastStart* zu überschreiben und die Zusatzfunktionalität zu implementieren. Darüber hinaus ist

Artikelserie

Teil 1: Einführung in die Entwicklung mit Red5
Teil 2: Weiterführende Red5-Konzepte und Sicherheit

für die angestrebte Audiostreamingfunktionalität auf Serverseite nichts weiter vorzubereiten, da Red5 die eigentliche Arbeit im Hintergrund übernimmt. Die notwendigen Umsetzungsschritte für den Client werden im späteren Verlauf dieses Artikels erläutert.

Als weiteres Beispiel lässt sich das benötigte Datentransfervolumen eines Benutzers nach Beendigung des Streamings protokollieren. Hierfür bietet das Red5-API ebenfalls Unterstützung an (*streamBroadcastClose* in Listing 1).

Für die Streamingkommunikation zwischen Red5-Server und Client existieren zwei verschiedene Modi: *live* und *record*. Im *live*-Modus empfängt der Server die Streamingdaten und verwirft sie automatisch, wenn sich kein Client für diesen Stream registriert hat und die Daten abruft. Erst mit angemeldeten Clients werden die empfangenen Daten intern im Red5-Server gepuffert. Im *record*-Modus dagegen werden empfangene Streamdaten unter dem angegebenen Streamnamen im Verzeichnis der Red5-Anwendung (z. B. *RED5-Root/webapps/Red5Chat/streams/…*) abgespeichert.

In der Standardkonfiguration des Red5-Servers ist es für jeden Benutzer möglich, sowohl einen Stream auf dem Server zu veröffentlichen als auch existierende Streams abzurufen, sofern die entsprechenden Informationen, d. h. Anwendungs-URL und Streamname, zur Verfügung stehen. Um die Sicherheit einer Anwendung in Bezug auf die angebotene Streamingfunktionalität zu erhöhen, bietet das Red5-API die beiden Interfaces *IStreamPlaybackSecurity* und *IStreamPublishSecurity* zur Definition von Handlern an, die die Berechtigungen zur Wiedergabe bzw. Veröffentlichung von Streams kontrollieren. Listing 2 zeigt eine einfache beispielhafte Umsetzung eines Handlers, in der überprüft wird, ob der zu veröffentlichende bzw. abzuspielende Stream über das Namenspräfix „Red5Chat“ verfügt. An dieser Stelle lassen sich natürlich realistischere Authentifizierungsinformationen verarbeiten, z. B. mittels JAAS, um die Sicherheit zu erhöhen. Der implementierte Sicherheits-Handler wird in der Startmethode der Red5-Anwendung registriert. So werden die späteren Streamingfunktionen abgesichert.

Erweiterung der Clientanwendung von „Red5Chat“

Im ersten Teil der Artikelserie wurde die Basisfunktionalität des Red5-Clients vorgestellt, wobei der Fokus auf dem Verbindungsaufbau und -abbau sowie dem Aufruf von entfernten Red5-Methoden und dem Datenaustausch zwischen Client und Server lag. Im Folgenden wird die entwickelte ActionScript-Klasse *Red5Manager* erweitert. Dabei zeigt Listing 3 die notwendigen clientseitigen Veränderungen zur Verarbeitung von Audio- und Videostreaming.

Listing 1

```
public class Red5Chat extends MultiThreadedApplicationAdapter {
 ...
 @Override
 public void streamBroadcastStart(IBroadcastStream stream) {
  super.streamBroadcastStart(stream);

  IScope  curScope=Red5.getConnectionLocal().getScope();
  IClient client=Red5.getConnectionLocal().getClient();
  String  clientID=(String)client.getAttribute("clientID");

  for(Set<IConnection> connections : curScope.getConnections()) {
   for(IConnection conn: connections) {
    if(conn instanceof IServiceCapableConnection) {
     IServiceCapableConnection curConn=(IServiceCapableConnection)
                                                              conn;
     String connUserID=(String)curConn.getClient().
                                                 getAttribute("clientID");
     if(!clientID.equals(connUserID)) {
      conn.invoke("clientPublishedStreams", new Object[]{clientID,
              stream.getPublishedName()});
 }}}}}
 ...
 @Override
 public void streamBroadcastClose(IBroadcastStream stream) {
  super.streamBroadcastClose(stream);
  if(stream instanceof ClientBroadcastStream) {
    ClientBroadcastStream cbs=(ClientBroadcastStream)stream;
    log.debug("bytesReceived: " + cbs.getBytesReceived());
  }
 }
```

Listing 2

```
//// Red5ChatStreamSecurity.java
public class Red5ChatStreamSecurity
            implements IStreamPlaybackSecurity, IStreamPublishSecurity {
  @Override
  public boolean isPublishAllowed(IScope scope, String name, String
                                                              mode) {
    if(name.startsWith("Red5Chat")) return true;
    return false;
  }

  @Override
  public boolean isPlaybackAllowed(IScope scope, String name, int start,
                                        int length, boolean flushPlayList) {
    if(name.startsWith("Red5Chat")) return true;
    return false;
 }}

//// Red5Chat.java
public class Red5Chat extends MultiThreadedApplicationAdapter {
 ...
  public synchronized boolean start(IScope scope) {
    streamingSecurity=new Red5ChatStreamSecurity();
    registerStreamPlaybackSecurity(streamingSecurity);
    registerStreamPublishSecurity(streamingSecurity);

    return super.start(scope);
  }
 ...
```

Allgemein sind clientseitig mehr Schritte notwendig als im entsprechenden Servercode. Dies liegt an der Vorbereitung der Peripheriegeräte wie Mikrofon und Kamera und der Verarbeitung der Datenströme für das Streaming. Nach erfolgreichem Herstellen einer Verbindung mit dem Red5-Server durch das *NetConnection*-Objekt wird diese Verbindung mit einem *NetStream* überlagert. Über diesen erfolgt anschließend der Datenaustausch von Audio- und Videodaten mit dem Server. Zusätzlich muss das Mikrofon initialisiert und mit dem *NetStream* verknüpft werden. Analog zu *NetConnection*-Events werden auch für das Streaming zahlreiche *NetStatus*-Events empfangen. Daher werden hierfür Event Handler benötigt. Abschließend wird mittels *publish* der Audio-/Videostream an den Server veröffentlicht. Listing 3 zeigt den schematischen Ablauf für das Streaming von Audiodaten.

Bei der serverseitigen Erweiterung wurde bereits erläutert, wie die Verarbeitung des Streamings abläuft. Die Verfügbarkeit eines neuen Audiostreams wird hierbei an verbundene Clients mitgeteilt (Aufruf der Client-Methode *clientPublishedStreams*). Diese können sich anschließend für den Stream registrieren und ihn abspielen. Analog zur Veröffentlichung eines Streams wird hierfür ebenfalls die *NetConnection* mit einem *NetStream* überlagert. Eine eventuelle Überlagerung mit mehreren *NetStreams*, z. B. einen zur Veröffentlichung und ggf. mehrere zum Abspielen, ist dabei kein Problem. Listing 4 zeigt die notwendigen Schritte zur Verarbeitung eines eintreffenden Streams. Die Zwischenspeicherung des lokalen *NetStream*-Objekts in der *userStreams*-Liste ist lediglich in diesem Beispiel notwendig, damit der Garbage Collector nach Verlassen der Methode das Objekt nicht „aufräumt“ und das Playback somit beendet.

Bisher wurde bei der Beschreibung der Fokus auf das Streaming von Audiodaten gelegt. Abschließend wird der Client noch insofern erweitert, als zusätzlich Videodaten für die bidirektionale Kommunikation in einem separaten Chatraum veröffentlicht bzw. empfangen werden können. Hierfür wird der in *toggleAudioStreaming* vorgestellte Code erweitert und innerhalb der Methode *toggleVideoStreaming* ein Kameraobjekt akquiriert und initialisiert (Listing 5). Die Qualitätseinstellungen werden zunächst auf Standardwerte gesetzt. *AVAILABLE_BANDWIDTH* ist hier ein Platzhalter für Bandbreite, die das Videostreaming maximal nutzen soll. Nachdem der Server den Veröffentlichungsstart durch das Event *NetStream.Publish.Start* bestätigt hat, wird das eigene Kamerabild in dem lokalen *VideoDisplay*-Objekt *ownVideo* dargestellt. Hierfür existiert in Flex die UI-Komponente *mx:videoDisplay*, die als *Child* das anzuzeigende Videoobjekt erhält.

Die Verarbeitung des entfernten Videostreams erfolgt analog zum Empfang von Audiodaten. Nach Erhalt der Information vom Red5-Server, dass ein anderer Datenstrom veröffentlicht wurde, registriert sich der Client für diesen und stellt das enthaltene Video in einem zweiten *VideoDisplay*-Objekt dar. Das entsprechende Vorgehen ist in der Methode *receiveIncomingStream* dargestellt.

Streaming Codecs und Enhanced Audio API

Das vorgestellte *Microphone*-Objekt bietet die Möglichkeit, über die Eigenschaft *codec* zwischen vier Audio-Codecs zu wählen, die für die Kompression von

Listing 3

```
public function red5Connect(userName:String, password:String=null):void {
  nc=new NetConnection();
  nc.client=this;
  nc.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
  ...
  nc.connect(Red5Manager.URL_RED5 + model.currentChatRoom.roomName, userName);
}
private function netStatusHandler(event:NetStatusEvent):void {
  if(event.info.code == "NetConnection.Connect.Success") {
    connected=true;
    toggleAudioStreaming();
  }

private function startMicrophone():void {
  mic=Microphone.getMicrophone();
  if (mic != null) {
    mic.encodeQuality = 10; mic.setSilenceLevel(0);
    mic.setUseEchoSuppression(true);
} }
public function toggleAudioStreaming():void {
  if(!netStream) {
    netStream=new NetStream(nc);
    netStream.client=this;
    startMicrophone();
    netStream.attachAudio(mic);
    netStream.addEventListener(NetStatusEvent.NET_STATUS, handleStreamEvents);
    netStream.publish("Red5ChatAudioStream"+model.currentUserName, "live");
  } else stopStreaming();
}
private function stopStreaming():void {
  if(netStream) {
    netStream.attachAudio(null);
    netStream.close();
}}
```

Listing 4

```
private var userStreams:ArrayList=new ArrayList();
public function clientPublishedStreams(userID:String, streamName:String):void {
  if (nc != null) {
    var stream:NetStream=new NetStream(nc);
    stream.play(publishedStreamName);
    stream.addEventListener(NetStatusEvent.NET_STATUS, incomingStreamsEvents);
    userStreams.addItem([curUserID,stream]);
  }
}
```

Listing 5

```
private function startCamera():void {
  camera=Camera.getCamera();
  if (camera != null) {
    camera.setQuality(AVAILABLE_BANDWIDTH, 0);
    camera.setMode(640, 480, 25);
} }

public function toggleVideoStreaming():void {
    // Initialisierung NetStream analog zu toggleAudioStreaming()
    // ...
    startMicrophone();
    startCamera();
    netStream.attachAudio(mic);
    netStream.attachCamera(camera);
    // ...
    netStream.publish("Red5ChatVideoStream"+model.currentUserName,
                                                          "live");
}

public var ownVideo : Video;
public function handleStreamEvents(event:NetStatusEvent):void {
  if (event.info.code == "NetStream.Publish.Start") {
    ownVideo=new Video(320, 240);
    ownVideo.attachCamera(camera);
    videoDisplayOwnStream.addChild(ownVideo);
} }
// Verarbeitung des entfernten Video-Stroms
public function receiveIncomingStream(otherUserID:String):void {
  // NetStream Initialisierung und Handler-Registrierung
  netStreamInc.play("Red5ChatVideostream" + otherUserID);

  remoteVideo=new Video(640, 480);
  if(remoteVideo != null) {
    remoteVideo.attachNetStream(netStreamIncoming);
    videoDisplayRemote.addChild(remoteVideo);
} }
```

Listing 6

```
private function startMicrophone():void {
  mic=Microphone.getEnhancedMicrophone();
  if (mic != null) {
    var options:MicrophoneEnhancedOptions=new
                                        MicrophoneEnhancedOptions();
    options.mode=MicrophoneEnhancedMode.FULL_DUPLEX;
    options.autoGain=false;
    options.echoPath=128;
    options.nonLinearProcessing=true;
    mic['enhancedOptions']=options;
  }
  else { /* kein EnhancedMicrophone verfügbar, benutze Standard
                                                        API */ }
}
```

Listing 7

```
public function toggleVideoStreaming():void {
  if(!netStream) {
    // Initialisierung siehe Listing 13
    startCamera();
    camera.setKeyFrameInterval(15);
    netStream.attachCamera(camera);
    // Aktivierung H.264-Codec
    h264Settings.setProfileLevel(H264Profile.BASELINE,
                                              H264Level.LEVEL_3_1);
    netStream.videoStreamSettings = h264Settings;
    netStream.publish("Red5ChatVideoStream"+model.currentUserName,
                                                        "live");

    var metaData : Object = new Object();
    metaData.codec = netStream.videoStreamSettings.codec;
    metaData.profile = h264Settings.profile;
    metaData.level = h264Settings.level;
    metaData.fps = camera.fps;
    metaData.bandwidth = camera.bandwidth;
    metaData.height = camera.height;
    metaData.width = camera.width;
    metaData.keyFrameInterval = camera.keyFrameInterval;
    netStream.send("@setDataFrame", "onMetaData", metaData);
  }
}
```

Listing 8

```
public class Red5Chat extends MultiThreadedApplicationAdapter {
...
@Override
 public boolean roomStart(IScope room) {
  ...
  createSharedObject(room, "roomClients", false);
  ISharedObject roomClientSO = getSharedObject(room, "roomClients");
  HashSet<String> roomClients = new HashSet<String>();
  roomClientSO.setAttribute("clients", roomClients);
  ...
 }
...
 @Override
 public boolean roomConnect(IConnection conn, Object[] params) {
  ...
  ISharedObject roomClientSO = getSharedObject(conn.getScope(),
                                                    "roomClients");
  if(roomClientSO!=null) {
   roomClientSO.lock();
   HashSet<String> roomClients = roomClientSO.getAttribute("clients");
   roomClients.add( ((String)conn.getClient().getAttribute("clientID") ));
   roomClientSO.setAttribute("clients", roomClients);
   roomClientSO.unlock();
  }
  ...
 }
```

Audiodaten verwendet werden können. Standardmäßig verwendet Flash den Speex-Codec für Audiodaten. Über die Verwendung der Konstanten *SoundCodec.NELLYMOSER*, *SoundCodec.PCMA*, *SoundCodec.PCMU* und *SoundCodec.SPEEX* kann der Audio-Codec manuell geändert werden. Weitere Informationen sind in der ActionScript-Referenz der *Microphone*-Klasse zu finden [5].

Seit Flash Player 10.3 steht zusätzlich das „Enhanced Audio API" zur Verfügung [6]. Neben der Echokompensation stellt die Rauschunterdrückung ein weiteres Feature dar. Weiterführende Audioeinstellungen sind über die *enhancedOptions*-Eigenschaft des *Microphone*-Objekts möglich. Bedingung ist die Anbindung eines entsprechenden Mikrofons über den Aufruf *getEnhancedMicrophone*. Listing 6 zeigt eine überarbeitete Methode *startMicrophone* zur Nutzung des Enhanced Audio API.

Für das Videostreaming stehen ebenfalls verschiedene Codecs zur Verfügung ([7], [8], [10]). Standardmäßig wird der Sorenson Spark Codec verwendet. Zur Verbesserung der Videoqualität empfiehlt es sich, den H.264-Codec für das Streaming von Videodaten zum Red5-Server zu verwenden. Dies kann je nach Qualitätseinstellungen zu einem höheren Datentransfervolumen führen. Durch die Angabe von Zusatzoptionen in der Methode *toggleVideoStreaming* kann der H.264-Codec aktiviert werden (Listing 7).

Shared Objects mit Red5

Zusätzlich zu den bisher vorgestellten Funktionalitäten bietet der Red5-Server so genannte Shared Objects (SO) an. Sie bieten Entwicklern die Möglichkeit, gemeinsam genutzte Daten innerhalb einer Anwendung auszutauschen. Im Gegensatz zu anwendungsweit definierten Variablen sind SO dabei lediglich innerhalb eines spezifischen Scopes gültig. Damit lassen sich beispielsweise gemeinsame Informationen zentral verwalten, ohne auf die notwendige Scope-Zuordnung im Falle einer globalen Variable achten zu müssen.

Für die Beispielanwendung wird in Listing 8 ein SO pro Chatraum definiert, das eine Liste der verbundenen Chatclients enthält. Alternativ könnte man beispielsweise eine Klasse für das Verwalten der Clientinformationen (Vorname, Name, IP, E-Mail-Adresse etc.) definieren, die über den Benutzernamen eindeutig identifiziert wird und diese im SO speichern. Um einen synchronisierten Zugriff auf die SOs zu gewährleisten, werden vom Red5-API die Methoden *lock* bzw. *unlock* zur Verfügung gestellt.

Der Red5-Server bietet zusätzlich die Möglichkeit, Listener auf SOs zu registrieren, um sie auf verschiedene Ereignisse wie Registrierung eines neuen Clients oder Veränderung zu überwachen. Listing 9 zeigt exemplarisch die Definition und Registrierung eines einfachen Listeners, der bei Aktualisierung des Chatraum-SO andere Benutzer dieses Chatraums benachrichtigt, dass sich ein neuer Benutzer verbunden bzw. den Raum verlassen hat. Abschließend muss der erstellte Listener nach der Erstellung des SO mit diesem verknüpft werden.

Verbindungssicherheit

Das Real-Time-Messaging-Protokoll (RTMP) sendet sämtliche Kommunikationsdaten unverschlüsselt. Damit kann potenziell jeder Angreifer die Kommunikation

Unterscheidung RTMP, RTMPE und RTMPS

Der Unterschied zwischen dem unverschlüsselten RTMP und den verschlüsselten Alternativen RTMPE und RTMPS ist vergleichbar mit dem Unterschied zwischen den Standard-Webprotokollen HTTP und der verschlüsselten Variante HTTPS. Tatsächlich wird bei RTMPS intern der Datenverkehr über eine HTTPS-Verbindung geleitet und ermöglicht damit eine verschlüsselte Kommunikation zwischen Server und Client unter Verwendung etablierter Techniken von HTTPS. Die initiale Aushandlung der Verschlüsselung (der so genannte Handshake) zu Beginn einer Verbindung ist ein rechenintensiver Vorgang. Für performante Server mit hohem Traffic-Aufkommen wurde daher von Adobe das RTMPE-Protokoll entwickelt, das die RTMP-Session in einem einfachen Verschlüsselungs-Layer einbettet. Allerdings werden hierbei unsichere kryptographische Verfahren eingesetzt, wodurch RTMPE nur einen geringen Grad an Sicherheit bietet. Insbesondere Man-In-The-Middle-Angriffe sind weiterhin möglich, da keine Überprüfung der Identität von Client oder Server durchgeführt wird.

Listing 9

```
//// Red5ChatSharedObjectListener.java
public class Red5ChatSharedObjectListener extends ISharedObjectListener {
 public void onSharedObjectUpdate(ISharedObjectBase soBase, String key, Object value) {
   // Benachrichtigung anderer Chatraum-Benutzer
}
 ...
}

//// Red5Chat.java
public class Red5Chat extends MultiThreadedApplicationAdapter {
 ...
 @Override
 public boolean roomStart(IScope room) {
  ...
  this.createSharedObject(room, "roomClients", false);
  ISharedObject roomClientSO = this.getSharedObject(room, "roomClients");
  roomClientSO.addSharedObjectListener(new Red5ChatSharedObjectListener());
  ...
 }
...
}
```

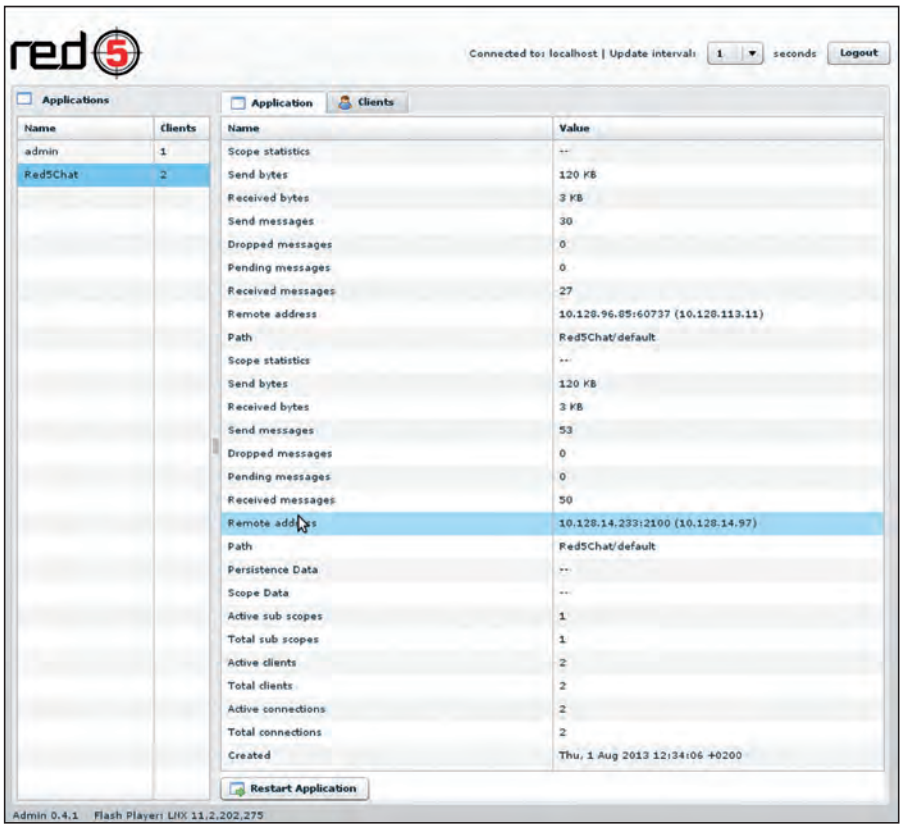


Abb. 1: Red5-Administrationsanwendung

abfangen oder verändern. Zur Verbesserung der Sicherheit der Anwendung bietet Red5 Möglichkeiten an, die Übertragung der Daten via RTMPE und RTMPS zu verschlüsseln (Kasten: „Unterscheidung RTMP, RTMPE und RTMPS“).

In der Standardkonfiguration des Red5-Servers ist die Verwendung von RTMPS bereits weitgehend vorbereitet. So muss lediglich der entsprechende Abschnitt in den Konfigurationsdateien *red5-core.xml* auskommentiert werden bzw. in *red5.properties* um benötigte Angaben wie das Keystore-Passwort ergänzt werden. Anschließend muss noch der in der Bean-Property *keystoreFile* der RTMPS-Konfiguration angegebene Keystore *conf/keystore* erstellt werden, damit eingehende Verbindungen verschlüsselt werden können.

Für die Beispielanwendung wird zunächst ein selbstsigniertes Zertifikat für die Verschlüsselung der RTMP-Verbindung erstellt. Dazu wird das in der Java-JRE enthaltene Programm *keytool* verwendet. Nach Eingabe des in Listing 5 definierten Passworts für den Keystore werden einige für das Zertifikat benötigte Informationen abgefragt (Listing 10). Wichtig ist das Common-Name-(CN-)Feld des Zertifikats, das die Domain darstellt, über die der Red5-Server erreichbar ist. Dieses wird als „first and last name“ abgefragt. Die restlichen Informationen dienen der Identifikation des Serverbetreibers für verbindende Benutzer.

Mittels des keytool-Programmarguments *-list* lassen sich installierte Zertifikate in einem Keystore/Truststore anzeigen. Zur Einbindung eines von einer regulären CA signierten Zertifikats kann es je nach vorliegendem Format des Zertifikats und Schlüssels notwendig sein, dieses zunächst in einen PKCS12-Container zu konvertieren und anschließend diesen Container mittels *keytool -importkeystore -srcstoretype PKCS12 …* in den eigenen Keystore zu importieren.

Listing 10

```
$ keytool -genkey -keyalg RSA -alias red5 –keystore conf/keystore.jks -validity 365
                                                                    -keysize 2048
[CN] red5.example.domain
[OU] Meeting-Place
[O]  Red5Chat
[L]  Kaiserslautern
[ST] Rheinland-Pfalz
[C]  DE
```

Listing 11

```
public function red5Connect(userName:String, password:String=null):void {
  nc=new NetConnection();
  nc.client=this;
  nc.proxyType="best";
  ...
  nc.connect("rtmps://red5.example.domain/" + curChatRoom.roomName, userName);
```

Clientseitige Shared Objects und verschlüsselte Verbindungen

Zur Nutzung einer verschlüsselten Verbindung zwischen Client und Server wurden bei den serverseitigen Erweiterungen des Beispiels bereits notwendige Änderungen an der Red5-Anwendung vorgestellt. Für eine entsprechende Anpassung verbindender Flash-Clients sind weniger Vorbereitungsschritte notwendig. Es muss hierzu lediglich das Verbindungsprotokoll von RTMP auf RTMPS abgeändert und der *proxyType* gesetzt werden, damit eine verschlüsselte Verbindung etabliert werden kann. Listing 11 zeigt den geänderten Code für den Verbindungsaufbau.

Analog zu den vorgestellten lokalen SOs in der Red5-Anwendung besteht die Möglichkeit, diese auch im

Flash-Client zu nutzen. Dazu akquiriert der Client ein entferntes SO, das in der Red5-Anwendung initialisiert wurde. Durch Registrierung eines entsprechenden Event Handlers für Synchronisierungs-Events besteht die Möglichkeit, direkt auf serverseitige Änderungen an diesem Objekt zu reagieren. In Listing 12 registriert sich der Client nach erfolgreichem Verbindungsaufbau für das entfernte SO, das die Liste aller verbundenen Clients in diesem Raum enthält. Da diese Objekte relativ zum aktuellen Raum (Scope) instanziiert werden, erhält der anfragende Client jeweils genau das Objekt, das für den Raum gültig ist, zu dem er sich soeben verbunden hat.

Monitoring von Red5-Anwendungen

Für den erfolgreichen Betrieb eines Servers ist ein Monitoring des Servers und der darauf ausgeführten Anwendungen hilfreich. Der Red5-Server bietet hierfür in den Demoanwendungen eine entsprechende Starthilfe. In der Standardkonfiguration lassen sich über den URL http://RED5-URL:5080/installer verschiedene Beispiel-Apps auf dem Red5-Server installieren. Dazu zählt auch die Administrationsanwendung *admin* zur Überwachung installierter Anwendungen und verbundener Clients. Nach der Installation kann diese über den URL http://RED5-URL:5080/admin aufgerufen werden (**Abb. 1**). In der Standardeinstellung lauten die Anmeldedaten: Benutzer *admin* und Passwort *admin*. Für den Produktivbetrieb sollten diese Daten in der Datei *admin.h2.db* natürlich geändert werden.

Fazit

Red5 stellt eine kostenfreie Alternative zu kommerziellen Mediaserverlösungen wie z. B. Adobe Flash Media Server (FMS) dar. Neben dem einfachen Datenaustausch, der im ersten Teil der Artikelserie beschrieben wurde, ermöglicht das Red5-API das Streaming von Audio- und Videodaten. Livestreaming von hochauflösenden Videos via Webcams unter Verwendung des H.264-Codecs erlaubt es, komplexe Multimediaanwendungen, wie z. B. Video-Conferencing, zu realisieren. Durch die Verwendung von Remote Shared Objects können Daten in Echtzeit zwischen Clients synchronisiert werden oder auch Methoden auf mehreren Clients gleichzeitig aufgerufen werden. Vor allem Entwicklerteams, die sich auf Java spezialisiert haben, bietet Red5 eine sehr gute Möglichkeit, einen einfachen Einstieg in die Entwicklung von Multi-Client-Streaming-Anwendungen zu finden. Das Red5-API versteckt vieles der Komplexität, die für die Kommunikation zwischen Server und Client nötig ist, und ermöglicht es Entwicklern, sich hauptsächlich auf die Anwendungslogik zu konzentrieren.

Listing 12

```
private var remoteSO : SharedObject = null;
private function netStatusHandler(event:NetStatusEvent):void {
  if (event.info.code == "NetConnection.Connect.Success") {
    ...
    try {
      remoteSO = SharedObject.getRemote("roomClients", nc.uri, false);
      remoteSO.connect(nc);
      remoteSO.addEventListener(SyncEvent.SYNC, handleSOsync);
    }
    catch(e: Error) { /* Fehler beim Akquirieren des SO */ }
} } }

public function handleSOsync(event : SyncEvent) : void {
  var array : Array = event.changeList;
  for each(var o : Object in array) {
    // Verarbeitung geänderter Properties
  }
  // Protokollierung verbundender Clients (serverseitig definiert)
  trace(remoteSO.data['clients']);
}
```

Dipl.-Inf. Sebastian Weber arbeitet seit 2007 am Fraunhofer IESE im Bereich UX und Mobile. Er entwickelt seit einigen Jahren Applikationen auf Apache-Flex- und Adobe-AIR-Basis.

M. Sc. Cornelius Moucha arbeitet seit 2011 am Fraunhofer IESE im Bereich Security und Software Engineering.

Links & Literatur

[1] https://code.google.com/p/red5

[2] http://web.archive.org/web/20030629213736/nellymoser.com/products/audio_compression_asao_fst.htm

[3] http://tools.ietf.org/html/rfc3551#page-28

[4] http://speex.org/docs/manual/speex-manual/

[5] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/media/SoundCodec.htm

[6] http://www.adobe.com/devnet/flashplayer/articles/acoustic-echo-cancellation.html

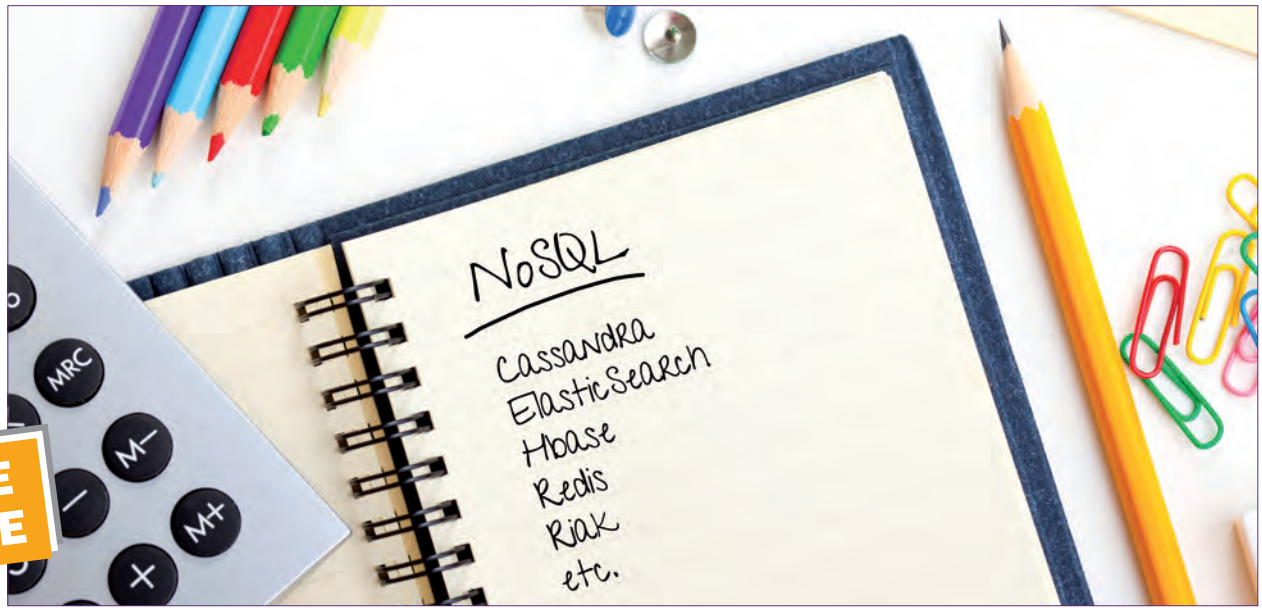
[7] http://www.digitalpreservation.gov/formats/fdd/fdd000066.shtml

[8] http://web.archive.org/web/20090228040914/http://on2.com/file.php?127

[9] http://ip.hhi.de/imagecom_G1/assets/pdfs/csvt_overview_0305.pdf

[10] http://www.adobe.com/devnet/adobe-media-server/articles/h264_encoding.html

[11] http://www.adobe.com/devnet/adobe-media-server/articles/encoding-live-video-h264.html



NEUE SERIE

©iStockphoto.com/fotostorm

Wie können Datenbanken verglichen werden?

Die NoSQL-Übersicht

NoSQL führt eine Vielzahl neuer Technologien in die Datenbankszene ein. Das Java Magazin wird in dieser Serie einen genauen Blick darauf werfen, was es alles gibt und wie sich die Datenbanken unterscheiden.

von Eberhard Wolff

w-jax Speaker

Ein technischer Überblick über die Konzepte und Modelle von NoSQL-Datenbanken findet sich schon unter [1]. Wir fassen die Kernpunkte des Artikels zusammen:

- Als Gründe für das Entstehen der NoSQL-Bewegung werden oft das Wachstum der Datenmengen und die höheren Anforderungen an Performance vor allem aus dem Webbereich angeführt. Das ist sicher richtig, aber nur einer der Treiber. Bei der ersten Generation von Datenbanken haben sich nämlich die Internetriesen genau dem Problem der großen Datenmengen angenommen: Google mit BigTable, Amazon mit Dynamo und Facebook mit Cassandra.
- Ein weiterer Treiber ist die Struktur der Daten: User-generated Content aus Social Networks ist nur wenig strukturiert und kann daher nur schwer in einer relationalen Datenbank mit festem Schema gespeichert werden. Aber auch Datenmodelle aus dem klassischen IT-Bereich beispielsweise für Finanzprodukte, Versicherungsverträge usw. sind sehr komplex – und können mit bestimmten NoSQL-Datenbanken einfacher bearbeitet werden. Einige vernetzte Strukturen, z. B. die Beziehungen in sozialen Netzwerken wie Facebook oder auch Verkehrs- oder Telekommunikationsnetze, sind mit relationalen Ansätzen nur mit sehr schlechter Performance abbildbar. Auch hier können NoSQL-Datenbanken helfen, und zwar vor allem die Graphendatenbanken.

Wie der gesamte NoSQL-Markt wird die Artikelserie eine Vielzahl von Technologien und Ansätzen für die Lösung ganz unterschiedlicher Probleme umfassen.

- Ein weiteres Problem ist die Flexibilität: Schemaänderungen bei relationalen Datenbanken sind insbesondere dann schwierig, wenn bereits große Datenmengen vorhanden sind. Durch Continuous Delivery werden viel öfter neue Softwareversionen mit ihren dazugehörigen Schemaänderungen deployt. So kann die Datenbank zu einem Flaschenhals für Änderungen an der Software werden. Um mit diesem Problem umzugehen, gibt es zahlreiche Workarounds – oft sind das interessante, manchmal aber auch einfach nur lustige Lösungen. Jeder hat wohl schon Schüssel-Wert-Tabellen oder Tabellen mit in XML serialisierten Java-Objekten gesehen. Außerdem gibt es Ansätze zum strukturierten Refactoring von Datenbanken [2].

NoSQL-Spielarten

Weil es so viele unterschiedliche Gründe für neue Persistenztechnologien gibt, ist NoSQL auch ein Sammelbegriff. Wesentliche Spielarten von NoSQL sind:

- Key-Value Stores speichern unter einem Schlüssel einen Wert (beides binäre Blobs). Sie sind sehr einfach aufgebaut, aber das Datenmodell kann eine Herausforderung darstellen – Zugriff lediglich über einen Schlüssel ist eben für einige Anwendungen doch zu wenig. Einige Lösungen aus diesem Bereich skalieren sehr gut. Beispiele sind Redis als schnelle In-Memory-Datenbank mit eher begrenzter Skalierung bezüglich der Datenmenge und Riak als sehr skalierbare Lösung.
- Wide Column Stores verwalten die Werte in Tabellen, die sowohl in der Anzahl Spalten als auch in der Anzahl Zeilen sehr groß werden können. Im Gegensatz zu relationalen Datenbanken dürfen die Tabellen aber keine Beziehungen zueinander haben. Beispiele sind HBase aus dem Apache-Hadoop-Projekt und Cassandra, ebenfalls ein Apache-Projekt.
- Dokumentenorientierte Datenbanken speichern hierarchisch strukturierte Dokumente ab, meistens als JSON. Dadurch können auch sehr komplexe Datenstrukturen abgebildet werden. Typische Vertreter sind MongoDB und CouchDB.
- Graphendatenbanken sind vor allem auf die effiziente Speicherung von komplexen, vernetzten Domänen ausgerichtet. In den Graphen sind die Entitäten als Knoten durch getypte Kanten miteinander verbunden. Beide können beliebige Attribute enthalten. Dieses Datenmodell unterscheidet sich fundamental von den anderen drei NoSQL-Varianten, die Beziehungen zwischen Datensätzen weitestgehend vermeiden. Wenn Datensätze stark miteinander in Beziehung stehen, ist es schwer, die Daten auf mehrere Server zu verteilen. Um eine bessere Skalierung über die Nutzung von mehr Servern zu erreichen, gehen daher die anderen NoSQL-Datenbanken bewusst den Tradeoff ein, Beziehungen zwischen Daten weniger gut zu unterstützen, um so Skalierbarkeit zu ermöglichen. Als Vertreter dieser Datenbankkategorie ist vor allem Neo4j zu nennen. Mit dieser Datenbank beschäftigt sich auch der erste Artikel der Serie in der aktuellen Ausgabe (siehe Michael Hunger, „Die Welt ist ein Graph“ auf Seite 80).

Kriterien für einen Vergleich

Wie der gesamte NoSQL-Markt wird die Artikelserie eine Vielzahl von Technologien und Ansätzen für die Lösung ganz unterschiedlicher Probleme umfassen. Um dennoch eine Vergleichbarkeit herzustellen, gibt es für jede Datenbank eine tabellenartige Übersicht. Im Folgenden werden die wesentlichen Begriffe aus der Tabelle erläutert – sie sind an der *kursiven* Schrift erkennbar. Das erste Beispiel für diese Tabelle finden Sie in Michael Hungers Artikel zu Neo4j.

Als Erstes ist natürlich das *Datenmodell* relevant. So kann die Datenbank einer Kategorie wie Key-Value dokumentenorientiert, Graphen oder Wide Column zugeordnet werden. Ein weiterer Punkt sind die *Suchmöglichkeiten*. Viele Datenbanken bringen ihre eigenen Anfragesprachen mit, andere unterstützen SQL-ähnliche Sprachen. Besonders spannend ist die Integration mit einer Suchtechnologie wie Lucene, Solr oder Elasticsearch. Dadurch bekommen die Datenbanken sofort sehr mächtige Anfragemöglichkeiten – nicht nur für Text, sondern auch für andere strukturierte Daten. Ein Key-Value Store, der sonst nur den Zugriff über bekannte Schlüssel ermöglicht, bekommt so ein vollwertiges und ausgesprochen mächtiges Anfragesystem. Für das Durchsuchen und Auswerten gerade großer Datenmengen hat sich MapReduce als Algorithmus etabliert, das ebenfalls eine spezialisierte Suchmöglichkeit darstellt.

Interessant ist natürlich auch die *Integration in Business-Intelligence-Werkzeuge* wie QlikView oder Pentaho. Damit kann die Analyse gerade großer Datenbestände oft direkt ohne Implementierung eigener Werkzeuge erfolgen. Business-Intelligence-Werkzeuge gibt es schon lange, aber erst in letzter Zeit fängt die

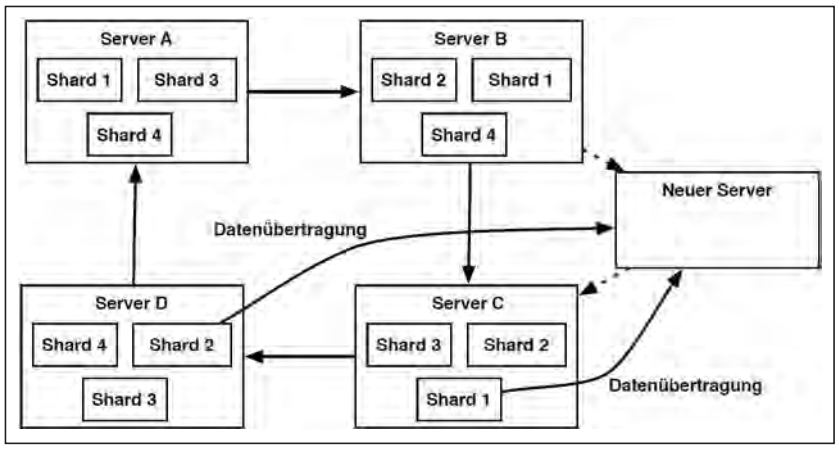


Abb. 1: Hinzufügen eines Servers im Dynamo-Modell; die Server sind als Ring organisiert; jeder Shard ist auf einem und den beiden nachfolgenden Servern; der neue wird zwischen B und C in den Ring integriert und übernimmt daher Shard 1 von Server C und Shard 2 von Server D; in der Realität gibt es meist wesentlich mehr Shards

Unterstützung von NoSQL-Datenbanken in diesem Bereich an. Da letztere oft kein SQL unterstützen, ist die Integration auch eine größere Herausforderung, als dies bei relationalen Datenbanken der Fall wäre.

Aus diesen Eigenschaften ergibt sich dann oft das *typische Einsatzszenario* – Datenbanken können für ganz unterschiedliche Bereiche wie Datenanalyse und Reporting, als Onlinedatenspeicher oder eben für die Modellierung beispielsweise von Graphen genutzt werden.

Klassische Datenbanken skalieren meistens vertikal, also durch die Nutzung größerer Server. Für die Bewältigung großer Datenmengen ist die *horizontale Skalierbarkeit* aber oft die bessere Alternative. Das bedeutet, dass für größere Datenmengen oder höhere Performanceanforderungen einfach mehr Server genutzt werden. Es gibt grundsätzlich unterschiedliche Möglichkeiten, horizontale Skalierbarkeit zu ermöglichen. Beim Sharding wird die Gesamtmenge der Daten in verschiedene Shards aufgeteilt. Jeder Server ist für einen Shard zuständig, also beispielsweise Server A für Shard 1 und Server B für Shard 2. Die Zuordnung kann anhand des Werts eines bestimmten Attributs erfolgen.

Anders beim von Amazon eingeführten Dynamo-Modell [3]. Dort wird die Gesamtdatenmenge aufgeteilt und zwar durch einen Consistent-Hashing-Algorithmus. Jeder Knoten ist für bestimmte Schlüssel aus diesem Consistent-Hashing-Wertebereich zuständig. Dabei werden die Daten gleich auf mehreren Knoten gespeichert. Also kann Server A Daten für die Schlüsselbereiche 1, 3 und 4 enthalten, während Server B die Schlüsselbereiche 2, 1 und 4 enthält.

Interessant ist das Verhalten beim Hinzufügen eines Servers: Wird beim Dynamo-Modell ein neuer Knoten in den Cluster aufgenommen, so bekommt er Daten von mehreren anderen Knoten geliefert und steht auch gleich für Schreiboperationen zur Verfügung. Beim obigen Beispiel könnte ein neuer Knoten die Schlüsselbereiche 3 und 4 übernehmen und sich dabei die Daten für 3 von A holen und die für 4 von B. Dieses Vorgehen wird auch durch die Eigenschaften des Consistent-Hashing-Algorithmus unterstützt (**Abb. 1**).

Beim Sharding hingegen wird der Shard aufgeteilt und die Daten des neuen Shards auf den neuen Server übertragen. Im Beispiel würde also der Shard 1 auf Server A aufgeteilt und die Daten dann nur von Server A auf den neuen Server übertragen. Erst danach kann der neue Server auch Schreiboperationen entgegennehmen (**Abb. 2**).

Das Dynamo-Modell ist durch die redundante Speicherung der Daten auch eine Lösung für *Hochverfügbarkeit und Replikation*. Bei Sharding werden die Shards für Hochverfügbarkeit auf andere Server repliziert, beispielsweise mit Master/Slave-Replikation. So kann ein hochverfügbares System mit Servern umgesetzt werden, die selbst nicht hochverfügbar sind. Im Zusammenhang mit Replikation sind auch das CAP-Theorem und BASE sehr wichtig, diese Aspekte wurden aber schon in [1] näher beleuchtet.

Ebenfalls interessant ist, in welcher *Programmiersprache* die Datenbank *implementiert* ist. Das wirkt auf den ersten Blick wie ein technisches Detail, aber in Wirklichkeit hängt dahinter eine Vielzahl von Implikationen. Ist die Datenbank in Java implementiert, ist für Java-Entwickler oder mit Java erfahrene Administratoren noch alles ganz wie gewohnt. C++- oder C-Anwendungen sind ebenfalls gang und gäbe. Anders bei Erlang: Es bringt seine eigene virtuelle Maschine (BEAM) mit. Die Ergebnisse sind mannigfaltig: Von einem einfachen „Was tut dieser BEAM-Prozess auf dem Server?“ bis hin zu Fragen rund um die Optimierung und den Betrieb einer solchen VM.

Hier zeigt sich schon eine wesentliche Herausforderung im Bereich NoSQL: Die Entscheidung für eine Datenbank betrifft nicht nur die Entwicklung, sondern auch den Betrieb. Beide Bereiche wachsen durch DevOps zunehmend zusammen, aber auf jeden Fall ist eine NoSQL-Datenbank ein zusätzlicher Aufwand, wenn es um den Betrieb der Anwendung geht und nicht nur für die Entwicklung. Dieser Aspekt ist bei der Entscheidung für eine Technologie nicht zu vernachlässigen und muss daher auch Architekten und technischen Projektleitern präsent sein. Ebenso relevant für den Betrieb sind auch *die unterstützten Betriebssysteme*. Sollen die Datenbanken dann in Produktion gehen, muss eine Lösung für das *Monitoring* und *Backup* vorhanden sein – oder das Team mag extrem stressige Feuerwehreinsätze. Beim Monitoring ist oft eine Integration in eine schon vorhandene Lösung der geeignete Weg. Beim Backup ist dieser Weg auch gangbar. Da NoSQL-Datenbanken aber oft in Clustern laufen, gibt es zusätzliche Herausforderungen. Ein Backup von vielen Knoten muss erfolgen, und

die Daten auf den Knoten müssen so konsistent sein, dass später auch ein konsistenter Zustand der Datenbank wiederhergestellt werden kann – und das idealerweise, während der produktive Betrieb weitergeht.

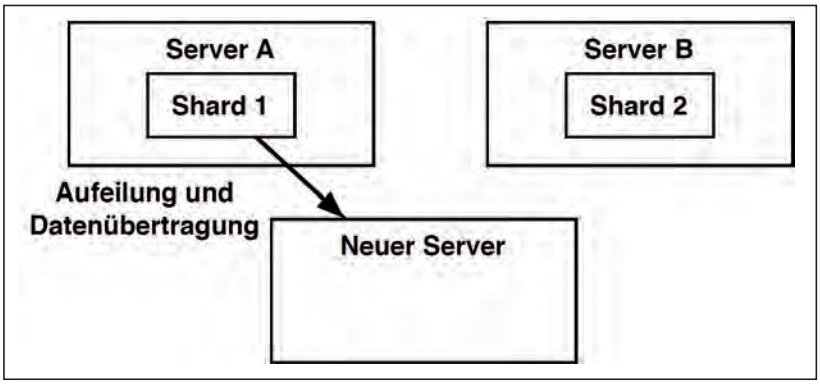


Abb. 2: Hinzufügen eines Servers bei Sharding; der Shard des Servers 1 muss aufgeteilt und übertragen werden

Lizenzen und Support

Lizenzen sind interessant, weil NoSQL-Datenbanken meistens Open-Source-Projekte sind. Die Apache-Lizenzen erlauben es Nutzern, praktisch beliebige Dinge mit dem Code zu tun. Anders bei GPL: Zusammen mit einem System oder eines abgewandelten Systems muss dann auch immer der Sourcecode zugänglich gemacht werden. Bei LGPL entfällt der Passus zum abgewandelten System: Das ist beispielsweise bei Libraries nützlich. Eine GPL-Library würde jedes System, das diese Library nutzt, sozusagen infizieren und zu einer GPL-Lizenz zwingen – bei LGPL ist das nicht der Fall. AGPL ist sogar aggressiver: Der Code muss allen Nutzern zugänglich gemacht werden. Der Unterschied zu GPL ergibt sich, wenn das System nur zur Nutzung über ein Netzwerk (wie das Internet) verfügbar gemacht wird. In dem Fall muss bei GPL der Sourcecode nicht zugänglich gemacht werden, weil kein System ausgeliefert wird. Bei AGPL ist das notwendig, da jeder Nutzer des Systems Zugriff auf den Code haben soll.

Im NoSQL-Bereich ist die Open-Source-Lizenz oft mit einer kommerziellen Version kombiniert. Dank Open Source kann der Nutzer die Datenbank ohne zusätzliche Kosten verwenden – auch in Produktion. Wenn *Support* als zusätzliche Sicherheit unverzichtbar ist, kann er zugekauft werden. Ebenso gibt es oft kommerzielle Versionen der Software mit *zusätzlichen Features*, die zumeist aber nur die Nutzung vereinfachen, aber nicht beispielsweise die Skalierbarkeit beschränken.

Und Java?

Bei der Nutzung von der JVM aus gibt es oft einfache *Java-APIs*, mit denen direkt in die Datenbank geschrieben werden kann. Eine Alternative sind Ansätze, bei denen die Java-Objekte automatisch in Datenbankstrukturen übertragen werden. Ein wesentliches Projekt in diesem Zusammenhang ist Spring Data [4, 5]. Das Ziel dieses Projekts ist es, möglichst viele unterschiedliche Persistenzalternativen zu unterstützen und dabei die Details der APIs soweit es geht anzugleichen. Es liegt in der Natur der Sache, dass ein Key-Value Store anders angesprochen wird als eine Graphendatenbank – und diese Unterschiede versteckt Spring Data auch nicht. Es hat außerdem auch Ansätze, um Java-Objekte ohne zusätzlichen Code direkt in der Datenbank zu speichern. Lösungen wie JPA oder Hibernate bieten das zwar auch, sind aber auf relationale Datenbanken ausgelegt. Eine Unterstützung für NoSQL ist meistens nicht sonderlich sinnvoll, da die Besonderheiten dieser Technologien mit einem O/R Mapper, der auf relationale Technologien ausgerichtet ist, kaum ausgenutzt werden können.

Neben Java gibt es natürlich auch andere Sprachen, die auf der JVM laufen. Einige – wie beispielsweise Clojure – sind so anders als Java, dass ein eigener Treiber auf jeden Fall angemessen ist.

Fazit

Im Rahmen der Serie werden unterschiedliche Datenbanken wie Cassandra oder Neo4j anhand dieser Kriterien näher betrachtet. Entwickler und Architekten werden dadurch in die Lage versetzt, die Chancen dieser neuen Technologien besser zu bewerten – eine wesentliche Voraussetzung für den erfolgreichen Einsatz! In Zukunft reicht es nämlich nicht mehr aus, einfach eine relationale Datenbank zu nutzen, sondern man muss sich gegebenenfalls für eine NoSQL-Spielart entscheiden – und dann für ein konkretes Produkt.

Eberhard Wolff arbeitet als Architecture and Technology Manager für die adesso AG in Berlin. Er ist Java Champion, Autor einiger Fachbücher und regelmäßiger Sprecher auf verschiedenen Konferenzen. Sein Fokus liegt auf Java, Spring und Cloud-Technologien.

@ewolff

Links & Literatur

[1] http://bit.ly/15LzteY

[2] http://www.agiledata.org/essays/databaseRefactoringCatalog.html

[3] http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[4] http://www.springsource.org/spring-data

[5] Pollack, Mark; Gierke, Oliver; Risberg, Thomas; Brisbin, Jon; Hunger, Michael: „Spring Data“, O'Reilly, 2012

Wir sind umgeben von einem Netz aus Informationen. Neo4j ist bereit dafür

Die Welt ist ein Graph

Von all den Informationen, die tagtäglich verarbeitet werden, ist ein beträchtlicher Anteil nicht wegen seines Umfangs interessant, sondern wegen der inhärenten Verknüpfungen, die darin gespeichert sind. Denn diese machen den eigentlichen Wert solcher Daten aus.

von Michael Hunger

Verknüpfungen reichen von historischen Ereignissen, die zu Orten, Personen und anderen Ereignissen in Beziehung stehen (und selbst in der heutigen Politik ihre Auswirkungen zeigen), bis hin zu Genstrukturen, die unter konkreten Umwelteinflüssen auf Proteinnetzwerken abgebildet werden. In der IT-Branche sind es Netzwerke, Computer, Anwendungen und Nutzer, die weitreichende Netze bilden, in denen Informationen ausgetauscht und verarbeitet werden. Und nicht zuletzt stellen soziale Netzwerke (ja, neben den virtuellen gibt es auch noch reale) aus Familien, Kollegen, Freunden, Nachbarn bis hin zu ganzen Kommunen einen wichtigen Aspekt unseres Lebens dar.

Jeder Aspekt unseres Lebens wird von zahlreichen Verbindungen zwischen Informationen, Dingen, Personen, Ereignissen oder Orten bestimmt. Große Internetfirmen versuchen natürlich, sich diese Informationen zunutze zu machen. Beispiele für großangelegte Projekte in dem Zusammenhang sind der Google Knowledge Graph [1] oder Facebook Graph Search [2].

Vernetzte Informationen und Datenbanken

Wenn diese vernetzten Informationen in Datenbanken abgespeichert werden sollen, müssen wir uns Gedanken darüber machen, wie wir mit den Verbindungen umgehen. Normalerweise werden sie ignoriert, denormalisiert oder zusammengefasst, um in das Datenmodell der Datenbank zu passen und auch Abfragen schnell genug zu machen. Was dabei jedoch verlorengeht, ist die Informationsfülle, die in anderen Datenbanken und Datenmodellen erhalten geblieben wäre. Genau in dieser Situation spielen Graphdatenbanken und das Graphdatenmodell ihre Stärken aus. Stark vernetzte Daten fallen in einer relationalen Datenbank sofort durch die schiere Menge an JOIN-Tabellen und JOIN-Klauseln in Abfragen auf (und durch die daraus resultierende schlechtere Abfragegeschwindigkeit).

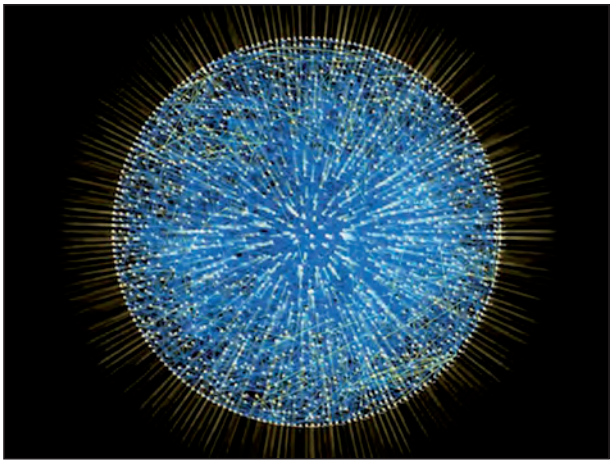


Abb. 1: Visualisierte Informationen

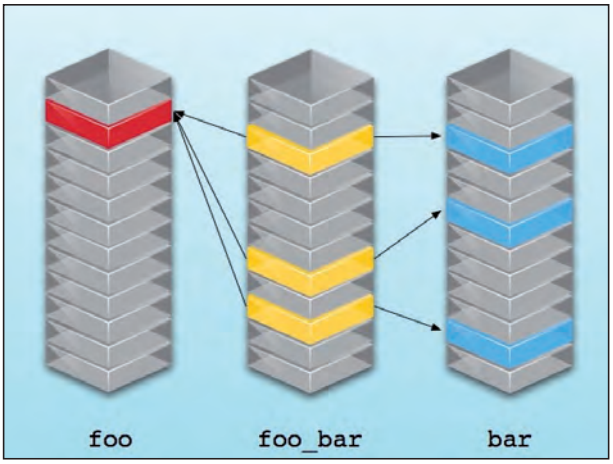


Abb. 2: Abfragen

Graphdatenmodell

Die mathematische Theorie zu Graphen ist viel älter als man denkt. Leonard Euler begründete sie, als er einen Weg über die sieben Brücken des damaligen Königsbergs finden wollte, ohne eine doppelt überqueren zu müssen [3]. Die Mathematik hat sich seitdem sehr ausführlich mit Graphtheorie und Graphalgorithmen befasst. Das soll aber nicht der Gegenstand dieses Artikels sein.

Graphdatenbanken

Die Kombination aus Management von Graphstrukturen (und damit von vernetzten Daten) und Datenbankeneigenschaften wie Transaktionalität und ACID ist aber eine neuere Erscheinung. Graphdatenbanken, die dies leisten, sind Teil der NoSQL-Bewegung, die zumeist nicht relationale Datenbanken umfasst. Diese Datenbanken sind größtenteils Open Source, entwick-

Technische Eigenschaften	
Datenmodell	Property-Graph-Modell: Entitäten sind Knoten, verbunden durch getypte, gerichtete Kanten, beide mit beliebigen Attribut-Wert-Eigenschaften, Knoten können beliebig viele Labels erhalten
Suchmöglichkeiten	Deklarative Cypher Query Language mit Mustersuche im Graph und SQL-ähnlicher Filterung, Projektion, Sortierung und Paginierung Lucene-Index-Suche für Startknoten (exakt nach Schlüsseln, Volltext) Java-API mit Traversal-DSL, Cypher-DSL REST-/HTTP-API mit explorativen URLs Gremlin/Groovy DSL Spring Data Repositories
Integration in BI-Tools	JDBC-Treiber, Konnektoren für Talend, Jaspersoft, QlikView, Pentaho Kettle
Typisches Einsatzszenario	Alle Szenarien mit einem anspruchsvollerem Datenmodell, z. B. Routing, soziale Netzwerke/Spiele, Konfigurationsmanagment, Netzwerk-/Rechenzentrumsmanagement, Organisiationsmanagement, Jobsuche, Partnerbörsen (Matchmaking), Empfehlungsermittlung, Betrugsermittlung, Zugriffsrechteermittlung, Vertragsmanagement ...
Horizontale Skalierbarkeit	Master-Slave Cluster, Read Scaling, Cache Sharding, Global Cluster
Hochverfügbarkeit	Master-Slave Replication, High Availability, Global Cluster
Implementiert in	Java, Scala (Abfragesprache)
Unterstützte Betriebssysteme	Windows, Mac OS, Unix (Linux, BSD, Solaris ...)
Monitoring-Werkzeuge	JMX
Back-up-Lösung	Online-Back-up (inkrementell und voll) von einzelnen Instanzen und Clustern

Lizenz und Support	
Aktuelles stabiles Release	1.9.3
Open Source?	Ja, Quellcode aller Editionen auf github.com/neo4j
Lizenz	Community Edition GPL Enterprise Edition (HA, Monitoring, Onlinebackup) mit AGPL, kommerziell
Kosten der kommerziellen Version	Listenpreis 6 000 bis 24 000 Euro VHB und separate Angebote für OEMs, Start-ups, Existenzgründer
Features der kommerziellen Version	High Availability Erweitertes Monitoring Online-Back-up Turbocache
Zusätzlicher professioneller Support	Von Neo Technology 24/7 sowie von globalem Partnernetzwerk

Nutzung mit der JVM	http://neo4j.org/drivers
Java-APIs	Embedded Mode: Java-API auch nutzbar von anderen JVM-Sprachen Server-Mode: JDBC-Treiber, Java-REST-Binding
APIs für andere JVM-Sprachen	Clojure, Scala, Groovy, JRuby, Jython, JavaScript
APIs für andere Non-JVM-Sprachen	Native Python-Bindings HTTP-/REST-Bindings für viele Sprachen, z. B: .NET, Python, PHP, JavaScript/Node.js, Ruby, Erlang, Haskell
Object Mapper	Spring Data Neo4j Neo4j Django Neo4j.rb

Tabelle 1: Spezifikation der NoSQL-Serie anhand Neo4j

Abb. 3: Property-Graph

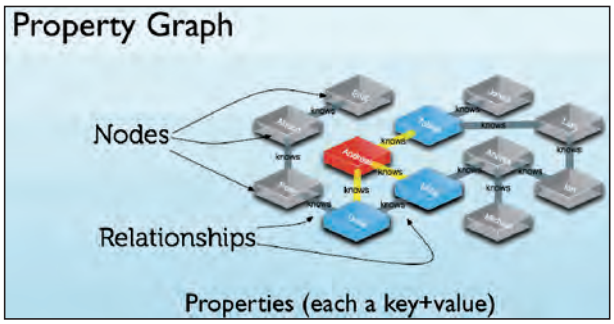


Abb. 4: Visualisierung von Mustern

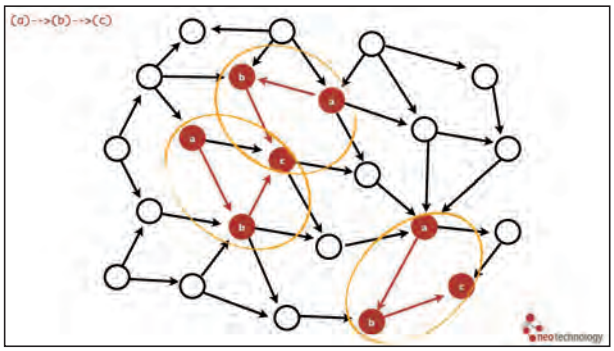
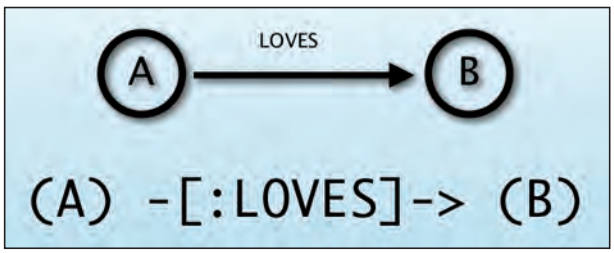


Abb. 5: Abfragesprache nach ASCII-Art



lerorientiert und mit einem Datenmodell versehen, das bestimmte Anwendungsfälle besonders gut unterstützt.

Graphdatenbanken sind dafür prädestiniert, relevante Informationsnetzwerke transaktional zu speichern und besonders schnell und effizient abzufragen. Das Datenmodell besteht aus Knoten, die mittels gerichteter, getypter Verbindungen miteinander verknüpft sind. Beide können beliebige Mengen von Attribut-Wert-Paaren (Properties) enthalten. Daher wird dieses Datenmodell auch als „Property-Graph" bezeichnet (**Abb. 3**).

Jeder hat definitiv schon einmal mit Graphen gearbeitet. Sei es bei der Modellierung für eine relationale Datenbank (ER-Diagramm), beim Skizzieren von Domänenaspekten auf einem Whiteboard/Tafel (Symbole und Linien) oder einfach während der kreativen Sammlung von Informationen (Mindmaps). Graphen sind aufgrund der Einfachheit des Datenmodells und einer besonders leichten Visualisierung gut verständlich und leicht zu handhaben.

Aber was ist nun so besonders an Graphdatenbanken? Dieser Artikel geht näher auf dieses Thema anhand des Beispiels Neo4j, einer Open-Source-Graphdatenbank, ein. Sie ist nativ und in Java implementiert. Nativ bedeutet, dass Knoten und Beziehungen direkt in den internen Datenbankstrukturen als Records in den Datenbankdateien repräsentiert sind. Neo4j nutzt keine andere Datenbank als Persistenzmechanismus, sondern baut auf einer eigenen Infrastruktur auf, die speziell dafür entwickelt wurde, vernetzte Daten effizient zu speichern.

Bezug nehmend auf die neue Java-Magazin-Serie zu NoSQL fasst Tabelle 1 alle wichtigen Eigenschaften zusammen.

Wie schafft es eine Graphdatenbank, die hochperformante Navigation im Graphen zu realisieren? Das ist ganz einfach: Mit einem Trick. Statt bei jeder Abfrage rechen- und speicherintensiv Entitäten immer wieder zu korrelieren, werden die Verbindungen beim Einfügen in die Datenbank als persistente Strukturen abgelegt. So wird zwar beim Speichern ein Zusatzaufwand in Kauf genommen, aber beim viel häufigeren Abfragen der Informationen können die direkt gespeicherten Verknüpfungsinformationen zur schnellen Navigation in konstanter Zeit genutzt werden.

Neo4j repräsentiert Knoten und Beziehungen in seinem Java-API als Java-Objekte (Node, Relationship) und im HTTP-API als JSON-Objekte. In der eigens für Graphen entwickelten Abfragesprache Cypher hingegen wird „ASCII-Art" verwendet.

Neo4js Abfragesprache Cypher

Was? ASCII-Art? Wie soll das denn funktionieren? Man denke einfach an Kreise und Pfeile auf einer Tafel oder einem Whiteboard, die man zum Diskutieren von Modellen schnell aufzeichnen kann. Das klappt, solange die Datenmengen, die es zu visualisieren gilt, klein genug oder nur konzeptionell sind, richtig gut. Bei größeren Graphen kann es schnell passieren, dass man den Wald vor Bäumen (oder Subgraphen) nicht mehr sehen kann. Aber wir wissen eigentlich, wonach wir suchen. Wir sind an ganz bestimmten Mustern im Graphen interessiert und ausgehend von diesen Strukturen wollen wir Daten aggregieren und projizieren, sodass unsere Fragen beantwortet und Use Cases abgebildet werden können. In einer Visualisierung können wir diese Muster z. B. mit anderen Farben hervorheben (**Abb. 4**).

Aber wie würden wir diese Muster in einer textuellen Abfragesprache beschreiben? Dort kommt die ASCII-Art ins Spiel. Wir „zeichnen" einfach Knoten als geklammerte Bezeichner und Beziehungen als Pfeile aus Bindestrichen (ggf. mit Zusatzinformationen wie Richtung oder Typ) (**Abb. 5**). Attribute werden in einer JSON-ähnlichen Syntax in geschweiften Klammern dargestellt.

Viel klarer wird das mit einem Beispiel, hier aus der Domäne der Filmdatenbanken (wie z. B. moviepilot):

```
(m:Movie {title: "The Matrix"})
<-[:ACTS_IN {role:"Neo"}]-
(a:Actor {name:"Keanu Reeves"})
```

Es ist ganz leicht, die ersten Schritte mit Cypher zu machen. Auf neo4j.org gibt es einen Cypher-Track [4], ein hilfreiches Cheat Sheet [5] und eine umfangreiche Referenzdokumentation [6]. Der Neo4j-Server [7] ist in wenigen Minuten heruntergeladen, ausgepackt, gestar-

tet und bereit zum Ausprobieren. Alternativ kann man Cypher in der Onlinedemokonsole [8] ausprobieren.

Wir betrachten Cypher als eine „menschenfreundliche“ Abfragesprache, die auf Lesbarkeit und Verständlichkeit optimiert ist. Man stellt dar, an welchen Mustern/Strukturen man im Graphen interessiert ist und welche Operationen, Filter, Aggregationen, Sortierungen usw. man anwenden möchte.

In Cypher wird, ähnlich wie in SQL, deklarativ die Frage dargestellt, die man beantworten möchte, und keine imperative, programmatische Anweisungsabfolge vorgegeben. Trotzdem ist Cypher viel mächtiger als SQL, wenn es um die Darstellung komplexer Beziehungen, Pfade oder Graphalgorithmen geht. Weitere Highlights sind die angenehme Arbeit mit Listen (*filter, extract, reduce*, Quantoren), das Verketten von mehreren Teilabfragen und die Weiterleitung von (Teil-)Ergebnissen bzw. Projektionen an nachfolgende Abfragen.

Cypher kann nicht nur komplexe Anfragen einfach darstellen und schnell ausführen, sondern auch Daten und Strukturen im Graphen erzeugen, modifizieren und korrigieren (Listing 1).

Cypher ist der schnellste Weg, um mit Neo4j produktiv zu werden und funktioniert in beidem, dem HTTP-API des Servers und dem eingebetteten Java-API. Um mit Neo4j zu interagieren, kann man sich einfach einen Treiber in der Lieblingsprogrammiersprache [9] aussuchen und benutzen.

Ganz neu: Neo4j 2.0

Gerade wird an der nächsten Version von Neo4j gearbeitet – 2.0. Welche neuen Features machen den Versionssprung möglich?

Zum einen wurde zum ersten Mal in zehn Jahren das Datenmodell erweitert. Neben Beziehungen können jetzt auch Knoten optionale Bezeichner (Labels) erhalten. Das macht es zum einen viel leichter, Typen im Graphen abzulegen (sogar mehrere pro Knoten). Diese Zusatzinformationen erlauben auch Optimierungen in der Cypher-Engine und an anderen Stellen. Aufbauend auf den Knotenbezeichnern kann man automatische Indizes, die pro Bezeichner und Attribut definiert werden, anlegen. Zudem wird es möglich, zusätzliche Restriktionen für das Datenmodell einzuführen (Eindeutigkeit, Wert- und Typbeschränkungen). Wie sieht so etwas aus? Listing 2 zeigt es.

Neo4j 2.0 führt auch ein neues Cypher-Schlüsselwort ein, das beim Aktualisieren von Graphen eine „Get or Create“-Semantik hat. Mit *MERGE* [10] kann man wie bei *MATCH* Muster angeben, die im Graphen gefunden werden sollen. Wenn dies erfolgt ist, werden die Knoten und Beziehungen direkt genutzt, ansonsten werden die fehlenden Bestandteile des Musters angelegt und können mit dedizierten Klauseln (*ON CREATE*, *ON MATCH*) aktualisiert werden.

```
MERGE (keanu:Person {name:'Keanu Reeves'})
ON CREATE keanu SET keanu.created = timestamp()
ON MATCH keanu SET keanu.lastSeen = timestamp()
RETURN keanu;
```

Ein weiteres, wichtiges neues Feature ist der transaktionale Cypher-HTTP-Endpunkt [11]. Bisher unterstützte das Server-API nur eine Transaktion pro HTTP-Request. Jetzt kann eine Transaktion mehrere HTTP-Anfragen umfassen und auch mit einem Mal mehrere Cypher-Abfragen beinhalten. Bis zum Timeout oder dem expliziten Abschluss der Transaktion mittels *commit* (POST to /transaction/id/commit URL) oder *rollback* (DELETE /transaction/id) wird die Transaktion offen gehalten und kann weiterverwendet werden. So werden z. B. eigene Änderungen innerhalb der Transaktion sichtbar, sind aber für andere Konsumenten nicht vorhanden (Isolation aus ACID). Die Anfrage- und Antwortdaten werden zum und vom Server gestreamt. Ein weiterer großer Vorteil ist das deutlich kompaktere Format der Ergebnisse, das nur noch die reinen Ergebnisdaten, keine Metadaten mehr enthält.

Dieses neue API erlaubt eine ganz neue Generation von Treibern (wie z. B. den JDBC-Treiber [12]), die in beiden Anwendungsszenarien (Server und Embedded) ein transaktionales Cypher-API anbieten. Damit wird die Lücke zur bekannten Interaktion mit SQL-Datenbanken weiter geschlossen und die Integration mit existierenden Werkzeugen und Tools erleichtert.

Erste Schritte

Was wäre ein guter Weg, um mit dem Graphdatenmodell und Graphdatenbanken durchzustarten? Zuerst sollte man einen Schritt zurücktreten und das größere

Listing 1

```
// Erzeugt einen Film mit der gesamten Besetzung in einem Zug
// Benutzt Parameter wie in Prepared-Statements
CREATE (m:Movie {title:{movie_title}})
FOREACH (a in {actors} :
  CREATE (a:Actor {name:a[0]})-[:ACTS_IN {role:a[1]}]->(m))

// Findet die Top-10-Schauspielerkollegen von Keanu Reeves
MATCH (keanu:Actor)-[:ACTS_IN]->()<-[:ACTS_IN]-(co:Actor)
WHERE keanu.name="Keanu Reeves"
RETURN co.name, count(*) as times
ORDER BY times DESC
LIMIT 10
```

Listing 2

```
CREATE INDEX ON :Movie(title);

// diese Abfrage benutzt den Index, statt alle "Movie"-Knoten zu durchsuchen
MATCH (m:Movie) WHERE m.title = "The Matrix";

// Beispiel für eine Eindeutigkeitsrestriktion
CREATE CONSTRAINT ON (actor:Actor) ASSERT actor.name IS UNIQUE
```

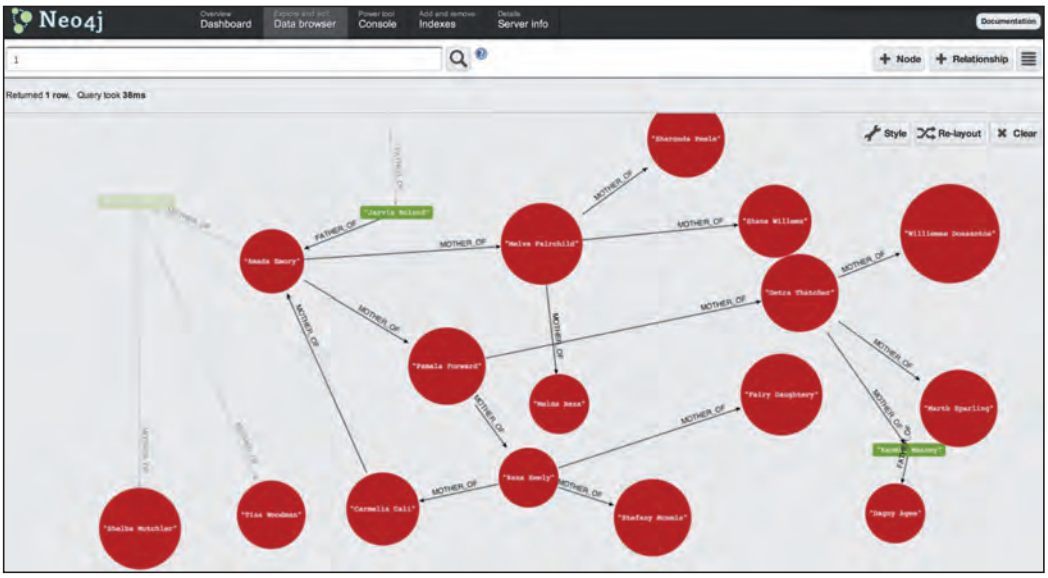


Abb. 6: Integrierter Neo4j-Browser

Ganze betrachten. Einige der Lösungen, die man sich mit relationalen Datenbanken erarbeitet hat, sollten überdacht und kritisch hinterfragt werden, wenn man die Technologie und das Datenmodell wechselt. Um ein gutes Graphmodell der eigenen Domäne zu erarbeiten, braucht man nicht viel. Ein Kollege oder Domänenexperte und ein Whiteboard sind genug, um ein Modell aufzuzeigen, das alle notwendigen Informationen und Beziehungen enthält, um Antworten für die wichtigsten Fragen und Anwendungsfälle zu liefern. Dies entspricht dem üblichen Vorgehen bei der Projektentwicklung. Dieses konzeptionelle Modell wird von Graphdatenbanken so gut unterstützt, dass man sich nicht auf eine technologiegetriebene Abbildung beschränken muss.

Mit diesem Modell im Hinterkopf kann man sich an den Import der Daten [13] in die Graphdatenbank machen. Dazu reichen das Herunterladen und der Start des Neo4j-2.0-Servers. Im Browser kann man mit dem Neo4j-Web-UI den Graphen visualisieren, hat aber auch eine Unix-ähnliche, interaktive Shell zur Verfügung. Sie erlaubt es, Cypher-Abfragen auszuführen, genauso, wie man es von SQL-Tools auch kennt. Diese Statements können sowohl Informationen liefern als auch den Graphen aktualisieren und anreichern (wie schon gesehen). Es ist hilfreich, dazu das Cypher Cheat Sheet zur Hand zu haben, um Syntaxfragen zu beantworten.

Jetzt geht es daran, Daten aus existierenden Datenbanken (oder einem Datengenerator) in ein Format zu transformieren, das man einfach in Neo4j importieren kann.

Ein Ansatz nutzt separate CSV-Dateien für Knoten und Beziehungen. Um diese tabellarischen Daten in einen Graphen zu konvertieren, benötigt man nur einige Cypher-Statements, die ähnlich wie SQL INSERTs ausgeführt werden. Entweder schreibt man sich ein kleines Skript, das die notwendigen Statements erzeugt, oder benutzt allgegenwärtige Tabellenkalkulationen [14] (ein nützlicher Trick, der mir während meiner Consulting-Engagements untergekommen ist.) Die Cypher-Statements sollten dann in einen transaktionalen Block gekapselt werden:

```
BEGIN
CREATE ...;
CREATE ...;
COMMIT
```

um die atomare Erzeugung des (Sub-)Graphen zu ermöglichen und die Einfügegeschwindigkeit zu erhöhen. Dann können diese Statements in die Web-UI-Konsole eingefügt oder noch besser mittels der Neo4j-Shell-Kommandozeilenanwendung aus einer Datei gelesen werden: *bin/neo4j-shell -file import.cql*. Es gibt auch eine Reihe anderer Tools [15], die den Datenimport mit der Neo4j-Shell noch viel einfacher gestalten.

Und das war es schon. Jetzt kann der Graph einfach visualisiert und abgefragt werden. Programmatischer Zugriff von eigenen Anwendungen ist wie bereits angesprochen mit der Vielzahl von Treibern für viele Programmiersprachen problemlos möglich. Dankenswerterweise hat sich unsere aktive Neo4j-Community stark gemacht und diese Treiber entwickelt und zur Verfügung gestellt.

Für die JVM gibt es neben dem originalen Java-API, mit dem Neo4j als eingebettete Datenbank (ähnlich Derby/HSQL) benutzt werden kann, auch Treiber für andere Programmiersprachen wie Clojure, Scala, JRuby, Groovy. Des Weiteren kann von Java sowohl über einen JDBC-Treiber als auch über das Java-REST-Binding oder direkt über eine HTTP-/REST-Bibliothek Zugriff auf den Neo4j-Server erlangt werden, um z. B. Cypher-Abfragen auszuführen.

Für alle Programmiersprachen außerhalb der JVM wie beispielsweise JavaScript, Ruby, Python und .NET kann man mittels der Treiber auf das HTTP-/REST-API des Neo4j-Servers zugreifen. Natürlich kann man auch einen der Treiber nutzen, um mittels Cypher oder des Low-Level-API Knoten und Beziehungen im Graphen anzulegen.

Für die Visualisierung von Graphen kann zum einen der integrierte Neo4j-Browser genutzt werden. Dazu wird der Neo4j-Server mit einem integrierten Web-UI ausgeliefert, das neben Monitoring und Statusinformationen sowie einer Onlinekonsole auch einen Visualisierungsmodus bietet. Dieser kann interessante Strukturen des Graphen darstellen und erlaubt die kontinuierliche Erforschung der vernetzten Informationen. Die Visualisierung kann zusätzlich mit Farben, Formen und Icons gestaltet werden (**Abb. 6**).

Aber auch eine eigene Visualisierung ist sehr leicht zu implementieren. Die verbreitete JavaScript-Bibliothek D3.js [16] bietet eine Vielzahl von Graphvisualisierungen, die mit einem Minimum an Aufwand realisiert werden können. Dazu muss man nur z. B. mit Cypher eine Knoten- und Kantenliste für den relevanten Ausschnitt des Graphen erzeugen und diese als JSON-Struktur für D3.js bereitstellen (**Abb. 7**).

Zum Schluss möchte ich noch anhand einiger Beispiele die breite Palette an Anwendungsmöglichkeiten des Graphmodells und von Graphdatenbanken demonstrieren. Jedes Datenmodell, das einigermaßen anspruchsvoll ist, beinhaltet eine Menge wichtiger Beziehungen und kann einfach als Graph repräsentiert werden. Das wird noch offensichtlicher, wenn man sich das Objektmodell der Anwendung anschaut und Objekte durch Knoten und Objektreferenzen durch Beziehungen ersetzt (auch wenn das noch nicht das optimale Graphmodell darstellt, da dort Beziehungen noch anämisch sind). Hier ein paar interessante Anwendungen:

- Facebook Graph Search [17] von Max De Marzi importiert Informationen aus Facebook und transformiert Anfragen in natürlicher (englischer) Sprache in Cypher-Statements.
- Typology ist ein ähnliches Projekt, das im Rahmen von „Jugend forscht“ einen ersten Preis gewonnen hat und Wortvorhersage in deutschen Sätzen auf der Basis des Google-N-Gram-Datensets ermöglicht.
- Rik Van Bruggens Biergraph [18] zeigt, dass auch Nutzer, die keine Entwickler sind, aus ihren Daten Graphen erzeugen, visualisieren und abfragen können.
- Open Tree Of Life [19] arbeitet daran, einen Graphen der kompletten biologischen Systematik (alle Pflanzen, Tiere) zu erstellen.
- moviepilot nutzt Neo4j, um Filmempfehlungen für seine Nutzer bereitzustellen und auch den Filmstudios hochqualitatives Feedback zu ihren Neuerscheinungen zu geben.
- Dshini ist ein soziales Netzwerk, das unter anderem Pinterest-Funktionalität mit Neo4j nachempfunden hat.
- Shutl [20] findet den besten Kurier und die optimale Route innerhalb einer Stadt für Sofortlieferungen (innerhalb von Minuten).
- Telenor [21] löst komplexe ACL-Autorisierungen in Sekundenbruchteilen auf.

Doch dieser Artikel zeigt zunächst einen relativ kleinen Ausschnitt aus der faszinierenden Welt der Graphen und Graphdatenbanken. Ist Ihr Interesse geweckt? Dann gibt es zahlreiche weiterführende Informationen. Im kostenlosen O’Reilly-E-Book „Graph Databases“ [22] werden viele Details und Modellierungsansätze für konkrete Domänen ausführlich von Experten dargestellt. Es gibt auch die Möglichkeit, ein Neo4j-Training [23] zu besuchen oder an einer der vielen Meetup-Events [24] teilzunehmen. Die neo4j.org-Seite ist ein guter Anlaufpunkt für alle Informationen rund um Neo4j. Für die Herbstausgaben des Java Magazins planen wir eine ausführlichere Artikelserie über Neo4j mit detailliertem Tutorial und Praxisbeispielen.

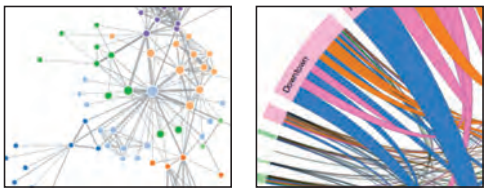


Abb. 7: Graphvisualisierung mit D3.js

Zu **Michael Hunger**s großen Leidenschaften gehört es, Software zu entwickeln. Der Umgang mit den beteiligten Menschen ist ein besonders wichtiger Aspekt. Zu seinen Interessen gehören außerdem Software Craftsmanship, Programmiersprachen, Domain Specific Languages und Clean Code. Seit Mitte 2010 arbeitet er eng mit Neo Technology zusammen, um deren Graphdatenbank Neo4j noch leichter für Entwickler zugänglich zu machen. Hauptfokus sind dort Integration in Spring (Spring Data Graph Project) und Hosting-Lösungen. Zurzeit hilft er der Neo4j-Community dabei, mit der Graphdatenbank ihre Wünsche wahr werden zu lassen. Michael arbeitet(e) an mehreren Open-Source-Projekten mit, ist Autor, Editor, Buch-Reviewer und Sprecher bei Konferenzen. Neben seiner Familie betreibt er noch ein Buch- und Kulturcafé (die-buchbar.de) in Dresden, ist Vereinsvorstand des letzten großen deutschen MUDs (mg.mud.de) und hat viel Freude an kreativen Projekten aller Art.

Links & Literatur

[1] http://www.google.com/insidesearch/features/search/knowledge.html

[2] https://www.facebook.com/about/graphsearch

[3] http://de.wikipedia.org/wiki/K%C3%B6nigsberger_Br%C3%BCckenproblem

[4] http://www.neo4j.org/tracks/cypher_track_start

[5] http://docs.neo4j.org/refcard/1.9/

[6] http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html

[7] http://www.neo4j.org/download

[8] http://console.neo4j.org/

[9] http://www.neo4j.org/develop/drivers

[10] http://docs.neo4j.org/chunked/milestone/query-merge.html

[11] http://docs.neo4j.org/chunked/milestone/rest-api-transactional.html

[12] http://www.neo4j.org/develop/tools/jdbc

[13] http://www.neo4j.org/develop/import

[14] http://blog.neo4j.org/2013/03/importing-data-into-neo4j-spreadsheet.html

[15] https://github.com/jexp/neo4j-shell-tools

[16] http://d3js.org/

[17] http://maxdemarzi.com/2013/01/28/facebook-graph-search-with-cypher-and-neo4j/

[18] http://blog.neo4j.org/2013/01/fun-with-beer-and-graphs.html

[19] http://blog.opentreeoflife.org/

[20] http://www.neotechnology.com/watch-how-shutl-delivers-even-faster-with-nosql/

[21] http://de.slideshare.net/verheughe/how-nosql-paid-off-for-telenor

[22] http://graphdatabases.com/

[23] http://www.neotechnology.com/tutorials/

[24] http://neo4j.meetup.com/

NEUE SERIE

Ehcache Advanced: In-Memory Data Management mit Java

Cache me if you can

Längst weiß man, dass Big Data nicht nur ein Hype ist. Trotzdem stellen sich viele die Frage: Gab es Big Data nicht schon vor zehn, zwanzig Jahren? Groß waren die Datenmengen damals auch. Also, was hat sich geändert? Die Antwort: vieles. Diese Artikelserie zeigt nicht nur, wie man mit In-Memory Data Management Caching- oder Garbage-Collection-Herausforderungen meistern kann, sondern auch, wie gut die In-Memory-Technologie und Big Data zusammenpassen.

von János Vona

In Sachen Big Data hat sich in den letzten Jahren und Jahrzehnten vieles verändert: Pro Minute werden 100 000 Tweets abgesendet, 277 000 neue Facebook-Anmeldungen vorgenommen und über zwei Millionen Google-Suchen getätigt [1]. Es gibt mehr als sieben Milliarden mobile Endgeräte, die bis 2016 mehr als elf Exabytes mobilen Datenverkehr erzeugen [2] – pro Monat!

Im Big-Data-Kontext sprechen wir immer über die *drei Vs*: *Volume, Velocity, Variety*. Es ist offensichtlich, dass die Datenvolumen in den letzten Jahren exponentiell gewachsen sind. Dazu kommt, dass die Daten immer schneller erzeugt, gesendet und empfangen werden. In jedem Smartphone steckt ein Mini-PC, wobei sich das „Mini" nur auf die physische Größe und nicht auf die Leistung bezieht. Wir haben immer schnellere (mobile) Internetleitungen (Gigabit, bald 400 Gigabit [3] oder Terabit) und Netzwerke, außerdem Sensoren in Ampeln, Autos und bald in den meisten elektronischen Geräten („Internet of Things"). Das führt zu der dritten Herausforderung, nämlich der Anzahl der verschiedenen Daten, Sender und Empfänger. Diese exponentielle Menge an Daten, ihre Komplexität sowie die Geschwindigkeit der Erzeugung und Verarbeitung gab es vor zwanzig Jahren noch nicht. Nun stehen viele von uns vor diesen neuen Herausforderungen. Zum Glück sind diese Daten nicht von heute auf morgen entstanden, und parallel zu dieser „Evolution" der Daten sind auch neue Lösungen entstanden, wie etwa Hadoop, MapReduce, NoSQL, DataGrid und In-Memory. Im Zentrum dieser Artikelserie steht das In-Memory Data Management von Terracotta [4], das als Open-Source-, Free- oder Enterprise-Lösung zur Verfügung steht.

Lesetipp

Die Ehcache-Grundlagen können Sie in der Ausgabe 3.2013 in János Vonas Artikel „Die Macht der Daten" nachlesen: https://jaxenter.de/magazines/JavaMagazin3.13.

Artikelserie

Teil 1: Ehcache Advanced
Teil 2: BigMemory und Distributed In-Memory Data Management

Ehcache Advanced

Die Grundfunktionen von Ehcache [5] habe ich in der Ausgabe 3.2013 bereits beschrieben. Zur Wiederholung fasse ich die ersten Schritte kurz zusammen.

Ehcache ist eines der beliebtesten Caching-Frameworks mit mehr als zwei Millionen Benutzern. Die Entwickler von Ehcache sind am Java Specification Request 107 (JCache) aktiv beteiligt [6], und es gibt eine JCache-Referenzimplementierung. Ehcache bietet in fast allen Bereichen der modernen Java-Entwicklung Integrationsmöglichkeiten wie z. B. für Hibernate, Spring, JRuby, Google App Engine usw.

Wenn man Ehcache über das API benutzen möchte, muss man zuerst die aktuelle Version herunterladen [5]. Nachdem wir die Datei heruntergeladen und ausgepackt haben, werden wir zuerst die JAR-Dateien (*/lib-Verzeichnis*) brauchen. Außer den JARs brauchen wir

noch eine Konfiguration. Ein Beispiel davon finden wir unter */src/ehcache-failsafe.xml*.

Wir fügen die JARs zu unserem Java-Projekt hinzu und kopieren die Konfigurationsdatei mit dem Namen *ehcache.xml* in unseren *classpath*. Nachdem wir den *CacheManager* initialisiert haben (siehe den o. g. Artikel in der Ausgabe 3.2013), können wir auf den in *ehcache.xml* definierten Cache zugreifen. Den dort bereits definierten *defaultCache* kann man nicht direkt benutzen. Er dient den Grundeinstellungen aller definierten Caches. So muss man sie nicht bei jeder Cache-Definition wiederholen. Außerdem kann man auf der *CacheManager*-Ebene auch einige Default-Einstellungen angeben. Wenn man mehrere *CacheManager* innerhalb der JVM definieren möchte, muss man unterschiedliche Namen für die *CacheManagers* in der jeweiligen Konfigurationsdatei angeben:

```
Ehcache1.xml: <ehcache name="distributedCacheManager">
Ehcache2.xml: <ehcache name="SimpleCacheManager"
maxBytesLocalHeap="4G">
CacheManager distributedCacheManager=CacheManager.create("ehcache1.
                                                          xml");
CacheManager singleCacheManager=CacheManager.create("ehcache2.xml");
```

Bevor wir die Konfiguration näher ansehen, werfen wir einen Blick auf den Aufbau von Ehcache. Ehcache ist ein Open-Source-Projekt und kann die gecachten Objekte entweder On-Heap (In-Memory) speichern und/oder auf die Festplatte schreiben. Es gibt jedoch noch eine dritte Möglichkeit, die Objekte Off-Heap (In-Memory) zu schreiben. Diese Lösung heißt *BigMemory* [7]. BigMemory und Distributed In-Memory Data Management sind Themen des zweiten Teils des Artikels. Bleiben wir bei Ehcache. Zuerst muss man die On-Heap-Cachegröße pro Cache definieren. Das kann man entweder in der Konfigurationsdatei mit der Angabe *maxEntriesLocalHeap* oder *maxBytesLocalHeap* machen oder über die gleichnamige *setter*-Methode zur Laufzeit einstellen. Bei Laufzeitkonfiguration muss jedoch der *dynamicConfig*-Parameter auf *true* gesetzt werden. Bei *maxEntriesLocalHeap* gibt man die Anzahl der gecachten Objekte an und bei *maxBytesLocalHeap* die Größe des Caches entweder in Byte oder in Prozent:

```
<ehcache name="myCacheManager" maxBytesLocalHeap="1G"
                                                dynamicConfig="true">
<!-- Cache1 hat 500MB On-Heap Cache -->
<cache name="Cache1" ... maxBytesLocalHeap="50%"></cache>
```

Wenn der On-Heap-Cache nicht genug ist, gibt es zwei Möglichkeiten, ihn zu erweitern: Entweder nutzt man BigMemory mit Off-Heap Caching [7] oder man lässt den Cache auf die Festplatte übergreifen. Unabhängig davon, für welche Möglichkeit man sich entscheidet, müssen die gecachten Objekte ab jetzt serialisierbar sein. Wenn diese Voraussetzung erfüllt ist, kann man die so genannte *persistent strategy* aktivieren (Listing 1). Sie kann *localTempSwap* sein, was es ermöglicht, unseren On-Heap-Cache mit der Festplatte zu erweitern, oder *distributed*, was Thema des zweiten Teils des Artikels sein wird. Die letzte Möglichkeit ist die *lokalRestartable*, was eine Ausfallsicherheit für unseren Cache ermöglicht. In dem Fall werden die gecachten Objekte synchron oder asynchron (optionale Parameter *synchronousWrites*) auf die Festplatte geschrieben (persistiert). Falls der Server, die Anwendung oder der Cache ausfallen, werden beim Neustart die persistierten Daten in den Cache automatisch geladen.

Bei *LocalTempSwap* wird zwar der Cache mit der Festplatte (*DiskStore*) erweitert, diese Daten überleben aber einen Neustart oder Stromausfall nicht. Der *DiskStore* muss beim *localTempSwap* oder *lokalRestartable* auf die Ehcache-Ebene konfiguriert werden (Listing 1).

Wenn wir unseren Cache mit einem *DiskStore* erweitern wollen, müssen wir den Geschwindigkeitsunterschied berücksichtigen. Wenn man keine SSD-Festplatten hat, kann man zum Beispiel in den Cache Byte-Arrays statt Strings speichern, weil Byte-Arrays wesentlich schneller serialisiert werden. Ehcache führt eine eigene Serialisierung durch, damit die Größe bzw. die Geschwindigkeit der Serialisierung kleiner bzw. schneller wird.

Wenn wir einen Cache planen, ist es sehr wichtig zu wissen, wie lange die Objekte im Cache bleiben werden. Wenn sie bis zum Löschen im Cache bleiben (*eternal*="*true*"), muss man genügend Speicher zur

Listing 1

```
ehcache.xml
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi=... >
<diskStore path="/auto/default/path"/>
   <defaultCache
        maxEntriesLocalHeap="10000"
        eternal="false"
        timeToIdleSeconds="50"
        timeToLiveSeconds="1000">
   </defaultCache>
<cache name="myCache"
  timeToIdleSeconds="60"
  timeToLiveSeconds="3600">
<persistence strategy="localTempSwap"/>
</cache>
</ehcache>
```

Listing 2

```
<cache name="myCache"
  maxEntriesLocalHeap ="10000" eternal="false" timeToIdleSeconds="3600"
  timeToLiveSeconds="0" memoryStoreEvictionPolicy="FIFO">
</cache>
```

Verfügung haben oder den Cache mit einem *DiskStore* erweitern. Man kann aber auch festlegen, wie lang ein Objekt im Cache bleiben soll bzw. welche Objekte aus dem Cache gelöscht werden, wenn dieser voll ist.

Mit dem Cacheparameter *timeToLiveSeconds* (TTL) können wir festlegen, wie lange ein Objekt im Cache bleiben kann. Der Parameter *timeToIdleSeconds* (TTI) gibt an, wie lange das Objekt „überleben" darf, ohne dass jemand auf das Objekt zugreift (*get* oder *update*). Listing 2 zeigt eine Konfiguration, in der maximal 10 000 Objekte im Cache gespeichert werden können. Jedes Objekt kann im Cache bleiben (*timeToLiveSeconds=0*), solange auf das Objekt mindestens einmal pro Stunde zugegriffen wird (*timeToIdleSeconds=3600*). Wenn die Anzahl der Objekte das Maximum erreicht hat, wird beim nächsten Hinzufügen eines neuen Objekts zuerst ein altes aus dem Cache gelöscht, damit das neue Objekt im Cache Platz hat. Wenn hier nicht die Anzahl, sondern die Größe in Bytes definiert ist, werden so viele Objekte aus dem Cache gelöscht, dass das neue Objekt von der Größe her hineinpasst. Als Regel (*memoryStoreEvictionPolicy*) kann man *First In, First Out* (FIFO), *Least Recently Used* (LRU) und *Least Frequently Used* (LFU) definieren. Der Erfahrung nach nutzt man meistens *Least Frequently Used*, wobei es die Möglichkeit gibt, eine eigene Regel zu definieren. Dazu muss man dem Cache die neue Regel einfach übergeben. Letztere muss nur das *Policy*-Interface implementieren, das im Wesentlichen eine *compare*- und eine *selectedBasedOnPolicy*-Methode hat. Die zuletzt genannte gibt das ausgewählte *Element* zurück. Da jedes Cacheobjekt sein TTL/TTI und sonstige Statistiken zur Verfügung hat, kann man seine eigene Methodik einfach nachimplementieren:

```
getCache().getCacheConfig().setMemoryStoreEvictionPolicy(Policy policy)
```

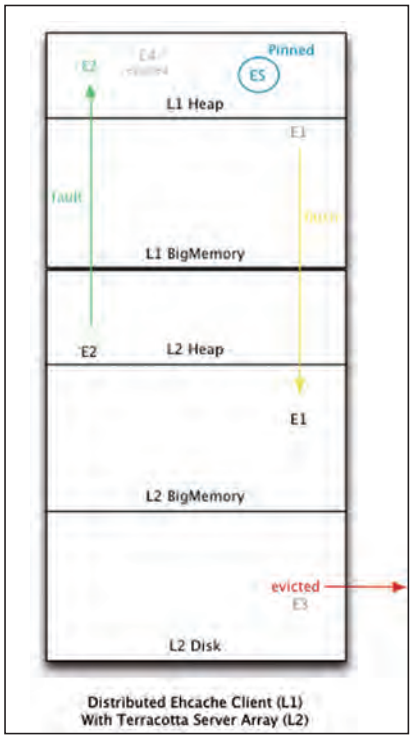


Abb. 1: Automatic Resource Control (ARC)

Es gibt jedoch einen wesentlichen Unterschied, ob ein Objekt nach dieser Regel aus dem Cache gelöscht wird oder nach TTL/TTI „abgelaufen" ist: Wenn die *EvictionPolicy* zum Einsatz kommt, bedeutet das immer, dass der Cache voll ist und für das nächste Objekt wieder Platz gebraucht wird. Das ausgewählte Objekt wird also sofort aus dem Cache entfernt. Das bedeutet aber nicht unbedingt gelöscht. Falls *LocalTemSwap* konfiguriert ist, wird das ausgewählte Objekt zuerst in den *DiskStore* verschoben. Falls es dort auch keinen Platz mehr gibt, wird es aus dem Cache endgültig gelöscht.

Wenn die „Lebenszeit" (TTL oder TTI) des einen oder anderen Objekts abgelaufen ist, bedeutet das nicht unbedingt, dass der Cache voll ist. Aus Performanzgründen werden also diese Objekte nicht sofort, sondern bei nächster Gelegenheit aus dem Cache entfernt (wie bei *EvictionPolicy*). **Abbildung 1** zeigt, wie das Automatic Resource Control (ARC) in einem komplexen System mit mehreren Cache-Layers die gecachten Objekte (E1–E5) je nach Konfiguration, Status, TTI/TTL und Regel managt. Die mehrschichtige In-Memory-Data-Management-Architektur werden wir im zweiten Teil des Artikels näher betrachten.

Die Suche

Eine der wichtigsten Funktionen eines Caches ist die Suche. Hier macht es einen Unterschied, ob man nur Ehcache ohne BigMemory (also nur On-Heap-Cache), oder mit BigMemory (Off-Heap-Cache) verwendet. Mit BigMemory werden Indices generiert, die entweder auf die Festplatte geschrieben oder In-Memory gehalten werden. Hier wird intern eine Variante von Apache Lucene [8] verwendet. Ohne BigMemory nutzt Ehcache eine schnelle Iteration durch die Elemente, um das gesuchte Objekt zu finden. Wenn man also nur Ehcache (On-Heap) benutzt und eine performante Suche haben möchte, sollte man nicht mehr als eine Million Objekte im Cache halten. Laut Dokumentation dauert eine sehr einfache Suche über eine Million Objekte nur 427 ms. Bei größeren Datenmengen sollte man eher BigMemory bzw. ein Distributed System verwenden.

Das Search-API von Ehcache kann man mit einer JPA-Query vergleichen – mit dem Unterschied, dass hier kein Join möglich ist. Außerdem müssen die gesuchten Informationen in Form von Attributen definiert werden. Wenn man zum Beispiel Mitarbeiterdaten cachen will und nach Alter, Standort usw. suchen möchte, müssen diese wie in Listing 3 definiert werden.

Listing 3

```
<cache name="cache" maxEntriesLocalHeap="10000">
  <persistence strategy="localRestartable"/>
  <searchable>
    <searchAttribute name="age" expression="value.getAge()"/>
    <searchAttribute name="city" expression="value.adress.getCity()"/>
    <searchAttribute name="netto" class="com.test.TestAttributeExtractor"/>
  </searchable>
</cache>
```

Im Beispiel sehen wir, wie man die *getter*-Methode des *value*-Objekts aufruft. Das *value* repräsentiert das gecachte Objekt, dessen *getAge()*-Methode für das *searchAttribute age* das gewünschte Ergebnis liefert. Man kann das *searchAttribute* über mehrere Objekte hinweg definieren, wenn die notwendige *getter*-Methode zur Laufzeit zur Verfügung steht (*value.adress.getCity()*). Um komplexere *searchAttribute*s zu bilden, kann man eigene *AttributeExtractor* schreiben. Das Interface sieht so aus:

```
Object attributeFor(Element element, String attributeName) throws
                                                AttributeExtractorException;
```

Diese Klasse bekommt den definierten Attributnamen und das aktuelle *Element* (Schlüssel plus gecachtes Objekt) als Parameter. Daraus muss man das gewünschte Suchattribut extrahieren. Falls es nicht genügt, kann man in der Konfigurationsdatei nach der Klassendefinition eigene Properties hinzufügen. In dem Fall braucht die Klasse einen Konstruktor mit *java.util.Properties*-Parameter. Achten wir darauf, dass ein *AttributeExtractor* immer nur einfache Java-Typen zurückgeben kann (String, Integer usw.). Wenn wir komplexe Typen oder Listen gecacht haben, nutzen wir immer einen eigenen AttributeExtractor, um die *SearchAttribute* zu extrahieren.

Wenn wir noch nicht wissen, wonach wir suchen werden, schalten wir das *allowDynamicalIndexing* an. Diese Einstellung ermöglicht, nach der Cacheinitialisierung weitere Attribute hinzuzufügen. In Listing 4 kann man ein paar Beispiele für das Search-API sehen. Eines von *includeKeys*, *includeValues*, *includeAggregator* oder *includeAttribute* muss obligatorisch angegeben werden. *IncludeKeys* bedeutet, dass das *Results*-Objekt die Schlüssel der gesuchten Cacheobjekte enthält. Bei *includeValues* werden die gecachten Objekte selbst im *Results* sein. Es klingt zunächst logisch, dass wir die Objekte sofort im *Results* haben. Wenn man aber nachdenkt, ist es nicht unbedingt notwendig bzw. in bestimmten Fällen zu vermeiden. Ehcache deserialisiert die Objekte (im Fall von Off-Heap oder *DiskStore*) nach dem Lazy-Loading-Prinzip. Also nur dann, wenn die entsprechende *getter*-Methode aufgerufen wurde. Wenn wir nur die *keys* im Ergebnis benötigen, werden nur diese bereitgestellt. Mit den *keys* können wir nach Bedarf mit *cache.get(key)* das gesuchte Objekt aus dem Cache holen.

Listing 4

```
final Query intQuery = cache.createQuery();
intQuery.includeKeys();
intQuery.addCriteria(age.eq(35));
intQuery.end(); // Execute Time: 62ms
final Query stringQuery = cache.createQuery();
stringQuery.includeKeys();
stringQuery.addCriteria(state.eq("DE"));
stringQuery.end(); // Execute Time: 125ms
final Query iLikeQuery = cache.createQuery();
iLikeQuery.includeKeys();
iLikeQuery.addCriteria(name.ilike("JAVA*"));
iLikeQuery.end(); // Execute Time: 180ms
```

Sehr oft stellen wir in unserer Applikation nur einen Teil der Information als Suchergebnis zur Verfügung. Die Information, die wir in der Ergebnisliste zeigen wollen, kann man mit *includeAttribute* explicit hinzufügen:

```
Query q = cache.createQuery();
Attribute<String> city = cache.getSearchAttribute("city");
q.includeAttribute(city);
```

Außerdem hat man auch die Möglichkeit, zu *Query*

- Sortierung – *addOrderBy*,
- Gruppierung – *addGroupBy*,
- Aggregation – *includeAggregator*

hinzuzufügen. Die Sortierungsmethode erwartet zum einen das *Attribute*-Objekt, das sortiert werden soll, und zum anderen die Richtung der Sortierung (*Direction*). Die Gruppierung funktioniert ähnlich wie bei SQL. Man übergibt die *Attribute*-Objekte, die man gruppieren möchte. Es gibt ein paar Regeln, die man bei der Gruppierung beachten muss, die aber nicht wesentlich anders sind als bei SQL. Die Funktion *includeAggregator* erwartet *Aggregator*-Objekte, die die Art der Aggregation und die zu aggregierenden Daten beschreibt. Man kann diese Objekte entweder über das *Attribute*-Objekt oder über die statische *Aggregators*-Klasse erzeugen:

```
q.includeAggregator(money.min()) oder
q.includeAggregator(Aggregators.min(money))
```

Wenn man nur einen Teil der Ergebnisse zurückbekommen möchte, kann man das *Resultset* mit der Methode *query.maxResults()* begrenzen.

Wie man sieht, kann man mit dem Search-API von Ehcache ohne großen Implementierungsaufwand sehr weit kommen. Wenn man auf seine SQLs nicht verzichten kann, sollte man auf die kommende Version warten. Hier wurden nämlich einige Erweiterungen angekündigt.

Besondere Funktionen

Es gibt sehr viele spezielle Funktionen von Ehcache, die man nur in Verbindung mit BigMemory (Off-Heap Caching) oder mit einem Distributed System verwenden kann. Es gibt jedoch einige, die man ohne BigMemory oder Distributed System sinnvoll nutzen kann. Dazu gehören die *Cache Decorators*. Sie basieren auf den gleichnamigen Design Patterns [9] und erlauben es, die Cachefunktionen zu beeinflussen. Einen solchen Cache kann man mit der eigenen *DacoratorFactory*-

Es gibt unter anderem Ehcache-Integrationen für Hibernate, OpenJPA, Spring, JRuby, Rails.

Klasse erzeugen, was von der *net.sf.ehcache.constructs.CacheDecoratorFactory* abgeleitet werden soll. Die *Factory*-Klasse überschreibt nur zwei Methoden: Die erste ist für die Erzeugung beliebiger Caches und die zweite für den Default-Cache zuständig. Mit Hibernate ist es häufig notwendig, den Default-Cache zu erweitern, da Hibernate nur den Default-Cache als L2-Cache benutzen kann. Wenn wir unsere *DecoratorFactory*-Klasse nicht in der Konfiguration definiert haben, muss man beim *CacheManager* den Cache manuell austauschen:

```
EhCache myCache = new MyCache(cache);
cacheManager.replaceCacheWithDecoratedCache(cache,myCache);
```

In dem Fall muss man darauf achten, dass der Cache vom Typ *Ehcache* statt *Cache* ist. Diesen Unterschied merkt man spätestens dann, wenn man vom *CacheManager* den Cache holen will. Im Fall von *Decorators* liefert *cacheManager.getCache()* *null* zurück. Stattdessen muss man *cacheManager.getEhcache()* verwenden.

Mit den *Decorators* kann man zum Beispiel die Objekte verschlüsseln, bevor diese in den Cache gespeichert werden, zusätzliche Indizes, Statistiken, Logs generieren oder Objekte verändern, filtern usw. Außer diesen individuellen Möglichkeiten gibt es zwei eingebaute *Decorators*, die man verwenden kann. Der erste ist der *BlockingCache*, der das gleichzeitige Lesen des Caches erlaubt. Wenn ein Objekt im Cache noch nicht vorhanden ist, werden die Lese-Threads dieses Cacheobjekts solange blockiert, bis das Objekt in den Cache geladen ist (via *put* oder *replace*). Vorteil hier ist, dass nicht der ganze Cache blockiert wird, wie beim *Hashtable* oder *SynchronizedMap*, sondern immer nur das fehlende Objekt. Dadurch kann man einen sehr hohen Lesedurchsatz erreichen und das Lesen auf mehrere Threads problemlos verteilen.

Der andere eingebaute Cache mit *Decorators* ist der *SelfPopulatingCache*. Er ist vom *BlockingCache* abgeleitet und ist in der Lage, Cacheobjekte selbst zu erzeugen, falls sie noch nicht vorhanden sind. Es ist sehr nützlich für experimentelle Aufgaben, wo man nicht immer die nötigen Daten vorhanden hat. Zur Realisierung braucht man nur eine eigene Klasse, die das *CacheEntryFactory*-Interface implementiert. Hier gibt es nur eine Methode, die das Objekt für den angegebenen Schlüssel manuell erzeugt:

```
Object createEntry(Object key) throws Exception;
```

Es gibt noch ein paar sehr interessante Funktionen wie *Event Listeners*, *Exception Handlers* und *CacheWriter*-Funktionen wie *write-through*, *write-behind*, *Transaction-Handling* oder *Explicit Locking*, die ich aus Platzgründen nicht mehr vorstellen kann. Mehr Informationen darüber findet man in der API-Dokumentation [10].

Fazit

Nicht ohne Grund ist Ehcache eines der beliebtesten Caching-Frameworks unter Java-Entwicklern. Mit Out-of-the-Box-Integrationen für Hibernate, OpenJPA, Spring, JRuby, Rails usw. ist der Einsatz von Ehcache überall möglich. Durch das einfache API und dank der flexiblen Erweiterungen kann man ihn problemlos in jeder Java-Anwendung verwenden. Ehcache hat gezeigt, dass In-Memory Data Management deutlich mehr bedeutet, als nur Informationen im Anwendungsspeicher (On-Heap) zu halten. Die Erweiterung des Speichers mit *DiskStore* (Festplatte), *Off-Heap* (BigMemory) und *Distributed* (Terracotta Server Array) eröffnet einerseits zahlreiche neue Möglichkeiten, andererseits muss man die eigene Architektur oft überdenken.

Wie wäre es, wenn Informationen immer und sehr schnell zur Verfügung stehen würden und die Datenbank nur zum Backup oder für sehr komplexe und nicht zeitkritische Abfragen benutzt würde? Diese Fragen werden wir im nächsten Teil beantworten.

János Vona ist Senior Manager ARIS Customized Solutions der IDS Scheer Consulting GmbH, einer Tochtergesellschaft der Software AG. Er leitet Individualprojekte im Kundenauftrag und beschäftigt sich seit mehr als zehn Jahren mit Java.

Janos.Vona@softwareag.com

Links & Literatur

[1] http://scoop.intel.com/what-happens-in-an-internet-minute/

[2] http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html

[3] http://www.ieee802.org/3/400GSG/

[4] http://terracotta.org

[5] http://ehcache.org

[6] http://jcp.org/en/jsr/detail?id=107

[7] http://terracotta.org/products/bigmemorygo

[8] http://lucene.apache.org/core/

[9] http://oreilly.com/catalog/hfdesignpat/chapter/ch03.pdf

[10] http://ehcache.org/documentation/apis/index

Anzeige

Anzeige

Professionelle Datenvisualisierung mit Java

Visualize me

In den vorherigen Artikeln der Serie haben wir die Programmierumgebung Processing vorgestellt und gezeigt, wie 2-D- und 3-D-Objekte im leeren Raum erstellt werden können, sowie die Realisierung von Augmented Reality mithilfe von Marker Codes dargelegt. Da Processing auf Java basiert, lässt es sich mit der ganzen Vielfalt der Java-Bibliotheken kombinieren und somit als Java-Entwickler einfacher in der jeweiligen IDE effektiver nutzen. In diesem vierten Artikel unserer Serie liegt der Fokus auf der Generierung von Grafiken.

Quellcode zum Download

von Dimitar Robev

Die Datenmengen, mit denen wir es heute zu tun haben, werden immer größer. Infolgedessen wachsen auch die Informationen, die darin verborgen sind, ständig. Aufgrund der Masse an Daten und den darin enthaltenen Informationen, kann die Verarbeitung zu einer Herausforderung werden. Da wir Menschen Grafiken besser und leichter verstehen können als reine Datenansammlungen, ist Datenvisualisierung oft der einzige Weg, sich und anderen einen Überblick zu verschaffen. Infografiken ermöglichen es uns, auch komplexe Themen in kurzer Zeit umfassend zu verstehen. Dadurch, dass die Daten vereinfacht und in visueller Form dargestellt werden, wird das menschliche Gehirn unterstützt, da es grafische Zusammenhänge besser verarbeiten kann.

Selbstverständlich wäre es zu unübersichtlich und auch nicht unbedingt sinnvoll, sämtliche gesammelten Daten zu visualisieren. Stattdessen geht es darum, die Daten grafisch darzustellen.

Ben Fry, ein Genie der Datenvisualisierung und Mitentwickler von Processing, stellt in seinem Buch „Visualizing Data" einen siebenstufigen Prozess vor, der den Weg der Antwortfindung sehr gut beschreibt:

1. acquire (erwerben, beschaffen): Beschaffen der Daten aus Intranet, Internet, aus dem lokalen Laufwerk etc.
2. parse (parsen): Strukturieren und kategorisieren der Daten.
3. filter (filtern): Entfernen aller unnötigen Daten.
4. mine (Muster erkennen): Anwenden von statistischen Techniken und solchen aus dem Gebiet des Data Minings, um Muster zu erkennen oder die Daten in einem mathematischen Kontext zu setzen.
5. represent (repräsentieren): Wählen einer Basisdarstellung für die Repräsentation, wie beispielsweise Bar Graph, Liste, Baum etc.
6. refine (verfeinern): Verfeinern der Basisdarstellung, um ein besseres Verständnis zu erzielen.
7. interact (interagieren): Hinzufügen von Funktionen, um die Daten zu manipulieren, oder um zu kontrollieren, welche Features angezeigt werden sollen [1].

Nach diesem siebenstufigen Prozess zu arbeiten, kann beeindruckende Ergebnisse hervorbringen. Processing bietet dabei Unterstützung für jeden einzelnen Schritt. **Abbildung 1** veranschaulicht den Prozess noch einmal.

An dieser Stelle möchte der Autor die Gelegenheit nutzen, nicht mit der PDE, sondern mit Eclipse zu entwickeln. Wenn man sich einmal an die Autovervollständigung gewöhnt hat, ist es schwierig, ohne sie zu arbeiten.

Zunächst gilt es, ein einfaches Java-Projekt zu erstellen. Hierzu ist im Project Explorer mit der rechten Maustaste New | Project zu wählen und anschließend im Wizard Java Project. Über Wizard ist im nächsten Schritt der *data*-Ordner als Source-Ordner hinzuzufügen. Bei Bedarf kann dies auch erst im Nachhinein ohne Wizard getan werden. Sollte man zusätzliche Ressourcen, wie Bilder, Textfonts etc. benötigen, ist der *data*-Ordner gut geeignet, um sie dort abzulegen, da man sich dann um die Lokalisierung und das Laden der benötigten Daten nicht kümmern muss. Hier gilt das Prinzip „Convention over Configuration" (CoC).

Da in der Beschreibung in diesem Artikel eine externe Bibliothek verwendet wird, ist der *lib*-Ordner anzule-

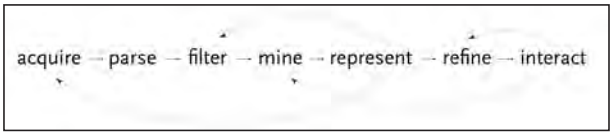


Abb. 1: Der siebenstufige Prozess nach Ben Fry

Artikelserie

Teil 1: Einführung ins Processing, Nutzen der 2-D-Rendering-Engine
Teil 2: Nutzen der 3-D-Rendering-Engine mit Kamerafahrten
Teil 3: Computervision und Augmented Reality mit Processing
Teil 4: Professionelle Datenvisualisierung mit Java

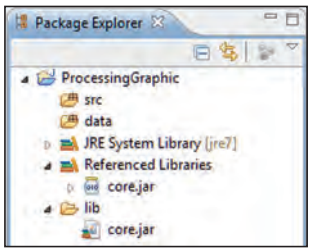


Abb. 2: Die Projektstruktur

gen. Der Name ist dabei nicht zwingend, da hier keine CoC stattfindet. Die benötigten Libraries werden in den Klassenpfad aufgenommen und können in einem beliebigen Ordner liegen – sowohl in der fertigen JAR als auch außerhalb.

Die Processing-Kernfunktionalität ist in der *core.jar* verpackt. Sie liegt in der Windows-Installation unter *processing-2.0.1\core\library* und muss in den Klassenpfad aufgenommen werden. Dafür ist sie in das *\lib*-Verzeichnis zu kopieren und im Klassenpfad aufzunehmen.

Die Hauptklasse muss von *PApplet* erben. So bekommt man Zugriff auf alle schon bekannten Processing-Methoden und Variablen, wie *fill()*, *noLoop()*, *keyPressed()*, *mousePressed()*, *width*, *height* usw.

Möchte man keine Multi-Applet-Anwendung bauen und doch seine eigenen Klassen zur grafischen Ausgabe schreiben, müssen diese mit einer Instanzvariablen vom Typ *PApplet* versehen und über den Konstruktor initialisiert werden. Eine detaillierte Ausführung folgt an späterer Stelle dieses Artikels. Eine Schritt-für-Schritt-Anleitung ist daneben auch auf der Processing-Seite [2] zu finden.

Über die Export-Funktion von Eclipse lässt sich das Projekt mit wenigen Klicks in eine lauffähige JAR exportieren. Hierzu ist für die Hauptklasse eine *Main*-Methode notwendig, wobei die *Main*-Methode von *PApplet*, je nach Wunsch, einen oder mehrere Parameter aufruft. Zuerst kommt der vollqualifizierende Klassenname, in dem die *Main*-Methode liegt, danach folgen die Aufrufargumente, verpackt in einem String-Array:

```
public static void main(String args[]) {
  PApplet.main("processing.graphic.tutorial.ForbesEu" , new String[]
                                                    {"--full-screen"} );
}
```

Laut Anleitung sollte man für diese Zwecke den *present*-Mode von Processing einschalten. Da dieser mittlerweile jedoch *deprecated* ist, empfiehlt es sich stattdessen, *full-screen* zu verwenden. Welche Argumente hierbei zur Verfügung stehen, kann in der Javadoc [3] der *PApplet*-Klasse nachgelesen werden (**Abb. 2**).

Alternativ kann auch das Proclipsing-Plug-in [4] verwendet werden, auf das an dieser Stelle jedoch nicht näher eingegangen werden soll.

Projektstruktur

Nach der Vorbereitung der IDE für Processing kann mit der Anwendung begonnen werden. In diesem Beispiel soll die Anzahl der Unternehmen pro EU-Staat in der Forbes-2000-Liste ausgewertet und dargestellt werden. Die vollständige Liste ist auf der Forbes-Website unter forbes.com einsehbar. In Listing 1 ist ein kurzer Ausschnitt zu sehen.

In Listing 2 ist die bekannte *setup()*-Methode zu sehen, in der die Fenstergröße gesetzt wird. Danach werden die Fonts initialisiert sowie die Daten geholt und geparst. Die Fonts können mittels *createFont()* erstellt oder alternativ mit *loadFont()* vom *data*-Ordner geladen werden. Es ist möglich, in der PDE über *Create Font…* eigene Fonts zu erzeugen und sie im *.vlw*-Format zu speichern. Mit *PFont.list()* kann man sich darüber hinaus alle im System verfügbaren Fonts ausgeben lassen. Wichtig dabei ist, dass Operationen, die von der Festplatte oder aus dem Internet Daten laden, in der *setup()*-Methode platziert sind, damit die Daten nicht bei jedem Aufruf von *draw()* wieder geholt/geladen werden.

Listing 1

```
Company;Industry;Country;Market Value;Sales;Profits;Assets;Rank;Forbes Webpage;
ICBC;Major Banks;China;237,3;134,8;37,8;2813,5;1;http://www.forbes.com;
China Construction Bank;Regional Banks;China;202;113,1;30,6;2241;2;http://www.forbes.com;
JPMorgan Chase;Major Banks;United States;191,4;108,2;21,3;2359,1;3;http://www.forbes.com;
```

File Handling	
BufferedReader	Zeilenweise aus einer Datei lesen
createInput()	Initialisiert und öffnet einen InputStream
createReader()	Erzeugt einen BufferedReader
loadBytes()	Lädt den Dateiinhalt in einem *byte[]*
loadJSONArray()	Lädt Array von JSON-Objekten aus der Platte oder aus einem URL
loadJSONObject()	Lädt ein einziges JSON-Objekt von der Platte oder aus einem URL
loadStrings()	Lädt den Dateiinhalt in einem *String[]*, Zeile pro String
loadTable()	Lädt eine .csv-Datei in einem Table-Objekt
loadXML()	Lädt eine XML-Datei in einem XML-Objekt
parseXML()	Lädt einen String, der in XML formatiert ist, und liefert ein XML-Objekt zurück
selectFolder()	Öffnet einen plattformspezifischen FileChooser, um interaktiv eine Datei auszuwählen
selectInput()	

Tabelle 1: File Handling

Die Schritte

SCHRITT 1: acquire

Die Basisdaten können überall gespeichert sein – sowohl auf der Festplatte als auch im Intra- oder Internet. Alle Dateien, die über Processing geladen und/oder gespeichert werden, sind im UTF-8-Zeichensatz. Um an sie heranzukommen, können Processing-Boardmittel verwendet werden. Hierzu steht eine breite Palette an Möglichkeiten zur Verfügung. Mit *loadTable()* ist es möglich, CSV-Dateien in einem Table-Objekt zu laden. *loadStrings()* lädt den Dateiinhalt in einem String-Array, wobei in

jedem String eine Zeile gespeichert wird. *loadBytes()* liefert die Datei in einem Byte-Array zurück. *loadJSONArray()* kann eine in *[]* eingeschlossene Liste von JSON-Objekten einlesen. *loadXML()* regelt das Einlesen von XML-Dateien. Tabelle 1 liefert einen Überblick über die Möglichkeiten des File Handlings.

Damit die Applikation auch offline ihre Dienste tut, wurde im nächsten Schritt eine CSV-Datei im *data*-Ordner ausgewählt:

```
acuire() {
  data = loadStrings(SOURCE_FILE);
}
```

SCHRITT 2: parse

Im Parse-Schritt gilt es, die Rohdaten in eine Struktur zu überführen, die geeignet für die weitere Verarbeitung ist. Dabei ist es notwendig, die gelieferten Daten zu inspizieren und zu verstehen, wie sie aufgebaut sind.

Hierzu kann man selbstverständlich eine externe Library zum Einlesen von CSV verwenden. An dieser Stelle soll jedoch gezeigt werden, dass dies auch mit Processing einfach zu bewerkstelligen ist:

```
void parse() {
  ...
  int lineCount = data.length;
  createTableHeader(data[0]);
  for(int i = 1; i < lineCount; i++) {
    forbesGlobal.addRow(replaceBtwColumns(data[i], 3, 8, ',', '.'));
  }
  ...
}
```

Die gewählte interne Struktur ist die einer *processing.data.Table*. Da Table nur kommaseparierte und Tab-separierte CSV-Formate unterstützt und die Daten im Beispiel zwischen Spalte drei und acht Double-Werte enthalten, die bei den Dezimalzeichen Kommata beinhalten, müssen diese angepasst werden. Eine Aufgabe, die gut zum Parse-Schritt passt.

Es ist üblich, das Bereitstellen und Parsen der Daten nicht in Processing vorzunehmen. Stattdessen kann dies im Backend unter Verwendung mächtiger Bibliotheken geschehen.

Zwar wirkt es, als gehöre das Parsen nicht unbedingt zum Prozess, jedoch gibt es gute Gründe, weshalb es dennoch ein fester Prozessbestandteil ist. Da die Daten in den meisten Fällen außerhalb der eigenen Kontrolle (man denke an sich ständig wechselnde Wetterdaten) und in nicht eigens definierten Formaten vorliegen, wird viel Zeit in die Interpretation und das Verstehen der Daten investiert.

SCHRITT 3: filter und SCHRITT 4: mine

In Schritt 3 gilt es, alle Daten zu entfernen, die für die Visualisierung nicht benötigt werden. In Schritt 4 werden die Daten anschließend in einen Kontext gesetzt, in dem sie besser zu handhaben sind (Listing 3).

Listing 2

```
public void setup() {
  size(750, 750);
  initFonts();
  acquire();
  parse();
  filterAndMine();
  createBarChart(euCountries);
  noLoop(); // einmal zeichnen reicht
}
```

Im Beispiel soll die Anzahl der Unternehmen pro EU-Land angezeigt werden. Dafür ist es notwendig, über alle Einträge zu iterieren und zu zählen, wie oft ein Unternehmen aus einem EU-Land gelistet ist.

Für die Darstellung dieses Falls wurde ein Bar Chart gewählt. In diesem Zusammenhang bietet sich die *FloatDict*-Klasse von Processing an, um die gefilterten Daten aufzubewahren. Diese speichert ihre Werte, ähnlich einer Map, als (Key-Value-)Paare und verfügt über Methoden, sie einfach zu manipulieren. So ist es möglich, über *set(key, amount)* einen Wert zu setzen, über *add(key, amount)* einen Wert zu addieren usw. Dabei werden alle mathematischen Operationen unterstützt. Man kann sich alle Keys und Werte ausgeben lassen, sie sortieren und das Vorhandensein abfragen. Hierbei stehen ähnliche Strukturen, die Integers und Strings als Werte speichern, zur Verfügung. Die *xyzList*-Klassen (*xyz -> [Float, Integer, String]*) bieten eine ähnliche Funktionalität. Dabei wird jedoch nicht über einen Key auf den Wert zugegriffen, sondern über einen Index. Tabelle 2 bietet eine Übersicht der vorhandenen Strukturen.

SCHRITT 5: represent

Wie bereits erwähnt, wurde in diesem Beispiel ein Bar Chart ausgewählt. Selbstverständlich ist es möglich, alles selbst zu zeichnen und alle Abstände, Richtungen, x- und y-Achse, Überschriften und Werte zu berechnen oder alternativ eine externe Bibliothek zu nutzen. In diesem Fall fiel die Wahl auf *giCenterUtil* [5]. Diese muss aus dem Internet heruntergeladen, in den */lib*-Ordner kopiert und in den Klassenpfad aufgenommen werden (Listing 4).

Damit der Bar Chart über die Funktionalität der *PApplet* verfügt, wird er damit erzeugt. Er bietet die Möglichkeit,

Listing 3

```
filterAndMine() {
  ...
  FloatDict euCountries = new FloatDict();
  while (rows.hasNext()) {
    TableRow row = rows.next();
    String country = row.getString(COUNTRY_INDEX);
    if (allEuCountries.contains(country)) {
      euCountries.add(country, 1f);
    }
  }
  euCountries.sortValues();
  ...
}
```

Strukturen
Array
ArrayList
FloatDict
FloatList
HashMap
IntDict
IntList
JSONArray
JSONObject
Object
String
StringDict
StringList
Table
TableRow
XML

Tabelle 2: Strukturen (Namen sind selbsterklärend)

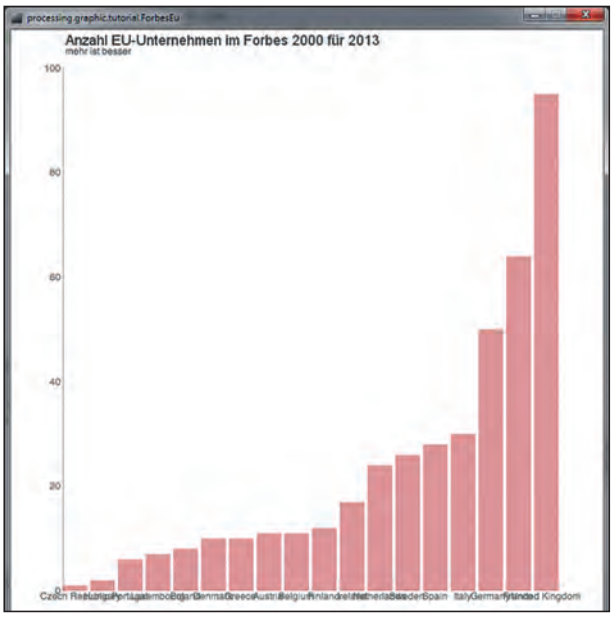


Abb. 3: Basisdarstellung – Namen überlappen sich

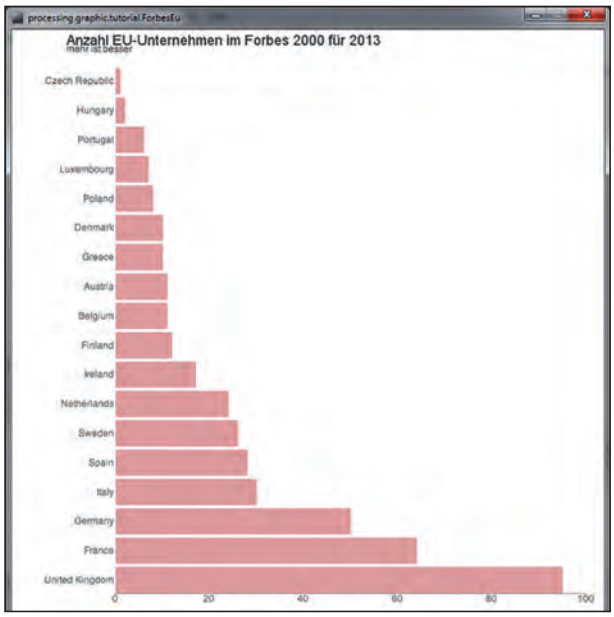


Abb. 4: Die Grafik ist jetzt übersichtlicher

die Werte und die Beschriftungen zu setzen. Hier zeigt die *FloatDict*-Klasse ihre Stärken, indem sie bereits die passenden Methoden bietet. Die Ländernamen werden auf der x-Achse und die Anzahl der Unternehmen auf der y-Achse dargestellt.

SCHRITT 6: refine

In diesem Schritt wird die Basisdarstellung angepasst, damit die Anzeige übersichtlicher und ansprechender wird. Betrachtet man die Darstellung, fällt schnell auf, dass die Namen sich überlappen, sodass an dieser Stelle ein Optimierungsbedarf besteht. Zwar ist zwischen den einzelnen Balken der Abstand vergrößerbar (*setBarGap(int)*), jedoch trägt dies zur Problemlösung nicht gänzlich bei (**Abb. 3**). Damit auch längere Namen, wie „Czech Republic" oder „United Kingdom" ausreichend Platz haben, ist der Graph sehr weit auseinanderzuziehen. Ebenso besteht die Möglichkeit, die Balken anstatt nach oben, zur Seite wachsen zu lassen. In diesem Fall würden die Ländernamen untereinander geschrieben, sodass sie sich nicht in die Quere kommen, was eine übersichtlichere Grafik zum Ergebnis hat. Dies wird mit *transposeAxes(boolean)* erreicht (**Abb. 4**):

```
...
barChart.setBarGap(4);
barChart.transposeAxes(true);
...
```

SCHRITT 7: interact

Interaktivität mit Processing umzusetzen, ist nicht kompliziert, da die Sprache sämtliche von Swing bekannten Aktionen, wie Mausklicks, Mausbewegungen, Tastenschläge, Eingabefelder etc., bietet. Beispielhaft wurden im nächsten Schritt zwei weitere Charts implementiert. Der erste beantwortet die Frage „Wie groß ist der Marktwert aller Unternehmen pro EU-Land?", der zweite „Auf welchem Platz ist das bestplatzierte Unternehmen pro EU-Land?". Zu den entsprechenden Ansichten gelangt man mit der rechten Maustaste bzw. mit der Taste *r* oder *R*. Die Default-Ansicht erhält man über die linke Maustaste.

Listing 4

```
createBarChart() {
  BarChart barChart new BarChart(this);
  barChart.setData(dict.valueArray());
  barChart.setBarLabels(dict.keyArray());
  barChart.setBarColour(color(200, 80, 80, 100));
  barChart.setBarGap(2);
  barChart.setValueFormat("###,###");
  barChart.showValueAxis(true);
  barChart.showCategoryAxis(true);
  barChart.draw(40, 40, width - 50, height - 50);
}
```

Fazit

Processing ermöglicht es, mit relativ wenig Code einen umfangreichen Prozess zur Datenvisualisierung umzusetzen. Da Processing auf Java aufsetzt, kann ein Großteil der Verarbeitungsschritte auch im Backend durchgeführt werden und somit auch anspruchsvolle und professionelle Grafiken erstellt werden. Das gesamte beispielhafte Projekt kann auf GitHub [6] heruntergeladen werden. Enjoy!

Dimitar Robev ist als IT-Berater im Java-Umfeld bei der EXXETA AG in Karlsruhe tätig. Seit ca. zehn Jahren entwickelt er Java-Anwendungen. In letzter Zeit ist er im Bankenumfeld unterwegs, wo er Real-Time-Analysesysteme entwickelt.

Links & Litratur

[1] Fry, Ben: „Visualizing Data: Exploring and Explaining Data with the Processing Environment", 11.01.2008, ISBN-10: 0596514557, ISBN-13: 978-0596514556

[2] http://processing.org/tutorials/eclipse/

[3] http://processing.org/reference/javadoc/core/

[4] http://code.google.com/p/proclipsing/

[5] http://gicentre.org/utils/

[6] https://github.com/drobev/ProcessingGraphic

Codequalität in Alt- und Wartungsprojekten: Tests mit Mockito

Erhalt und Verbesserung

Das Testen von Software bildet einen wichtigen Bestandteil in der Softwareentwicklung. Dabei bieten unterschiedliche Vorgehensmodelle den Entwicklern mehrere Ansätze, Komponenten zu entwickeln. Besonders bei Altprojekten sollte der Fokus auf die Testabdeckung gelegt werden. Mockito [1] kann helfen, Testfälle für alte und neu in die Anwendung zu integrierende Module zu entwickeln, ohne dass das Fachwissen zahlreicher Jahre nötig wäre. Auch der Umgang mit unerwartet auftretenden fachlichen und technischen Fehlern erfordert ein individuelles Vorgehen. Denn nicht jeder Fehler erfordert eine umgehende Behebung. Im Gegensatz dazu sollte die Dokumentation immer den aktuellen Projektstand widerspiegeln und zeitnah aktualisiert werden.

von Daniel Winter

Nachdem durch die Einführung von Checkstyle und PMD in das Altprojekt ein einheitlicher Formatierungsstil erarbeitet und die Grundlage für weitere Verbesserungen an der Software geschaffen wurde, überprüfen wir im nächsten Schritt die Testabdeckung. Die Sicherstellung der fachlichen Korrektheit der Anwendung gilt als Voraussetzung, um die Codequalität weiter verbessern zu können. In neuen Projekten geschieht dies unter anderem mit der Implementierung zahlreicher Unit-, Regressions- und Integrationstests. Zusätzlich besitzt das Projektteam bei der Umsetzung die Möglichkeit, unterschiedliche Vorgehensmodelle (Test First vs. Test Last) einzusetzen. Die Entwickler von Altprojekten sind gezwungen, Testfälle für historischen Code nachträglich zu implementieren. Ausnahmen bilden anstehende Erweiterungen. Hier kann auch der Ansatz von TDD (Test-driven Development) [2] verfolgt werden.

Testen mit Mockito

Ein Hauptproblem bei der Testfallerstellung von Altcode ist das unvollständige Wissen über das Verhalten komplexer Komponenten infolge fehlender oder unzureichender fachlicher Dokumentation. Ebenso erschwert die schiere Größe einiger Interfaces, Klassen und Methoden die vollständige Testabdeckung. Müssen anstehende Erweiterungen, aufbauend auf solch komplexen Datenstrukturen, entwickelt werden, ist eine intensive Einarbeitung für das Verständnis nötig und ein erhöhter Zeitaufwand für die Implementierung einzuplanen. Durch den Einsatz von Mockito kann eine deutliche Verkürzung beider Phasen erreicht werden.

Mockito ist eine Bibliothek, mit der Klassen samt ihrer Funktionalität nachgebildet werden können (Mock-Objekte), ohne ihren kompletten Funktionsumfang selbst implementieren zu müssen. Der Entwickler entscheidet selbst, welche Werte oder Objekte er bei den einzelnen Methoden zurückgeben lassen will. Weiterhin besteht die Möglichkeit eines „partiellen Mockings“ (Spy) von Objekten. Dabei wird ein so genannter „Spion“ erzeugt, der die tatsächlichen Methoden des Objekts aufruft, solange deren Funktionen nicht „gemockt“ wurden. Weiterführende Informationen und konkrete Anwendungsbeispiele können der offiziellen Webseite [3] entnommen werden.

Die Anwendung von Mockito soll am Beispiel einer seit Jahren zuverlässig arbeitenden Finanzsoftware verdeutlich werden. Eine Hauptaufgabe der Anwendung ist die Erstellung von Refinanzierungen und Tilgungsplänen, basierend auf den Daten von Leasingverträgen und Leasinggesellschaften. Um auch die zukünftige Nutzung zu gewährleisten, muss die Software mit einem neuen Modul erweitert werden, das SEPA-Lastschriften [4] auswertet und einen Abgleich mit dem eigenen Bestand durchführt. Voraussetzung des Abgleichs ist die Ermittlung der IBAN, einer Kombination aus Kontonummer, Bankleitzahl und Länderkennung des Kontos der Leasinggesellschaft. In der SEPA-XML-Datei wird IBAN als ID verwendet. Die korrekte Berechnung muss daher mit

Artikelserie

Teil 1: Erste Lösungsansätze mit Checkstyle, PMD und einem Code-Formatter
Teil 2: Höhere Testabdeckung mit Mockito

einer umfassenden Testabdeckung einhergehen. Jedoch wurden in der Finanzsoftware die zentralen Klassen *Leasinggesellschaft, Bankverbindungsdaten* und deren Interfaces über die Zeit immer wieder erweitert und umfassen aktuell einige hundert Methoden. Zusätzlich sind die benötigten Setter und Getter der Bankverbindungsdaten mit einem direkten Datenbankzugriff versehen. Durch diese Umstände erweist sich die Testfallgestaltung als komplex und unübersichtlich. Mockito präsentiert sich in diesem Fall als die eleganteste Möglichkeit, um die Funktionalitäten des SEPA-Abgleichs zu testen.

Dabei werden, wie in Listing 1 dargestellt, einmalig die *Leasinggesellschaft* und die *Bankverbindungsdaten* in der *setup*-Methode „gemockt". In den einzelnen Testfällen können mit wenigen Zeilen Quellcode die Kontonummer und die Bankleitzahl ausgetauscht und mehrere Testszenarien abgedeckt werden. Dabei kann auf für den Testfall nebensächliche Tätigkeiten, wie das umständliche Bereitstellen unterschiedlicher Bankverbindungsdaten oder die Implementierung einer Leasinggesellschaft, verzichtet werden. Ebenso entfällt eine zeitintensive Einarbeitung in die Fachlichkeit der *LG*-Klasse. Mockito erlaubt eine Fokussierung auf das eigentliche Problem und ist dadurch ideal geeignet, Unit-Testfälle in Altprojekten einzuführen.

Einarbeitung mittels Testen

Versucht man während der Fehlerbeseitigung konzentriert allein an der Komponente des Problems zu arbeiten, sind bei der Einarbeitung in ein Modul alle Klassen, die die Fachlichkeit betreffen, von Relevanz. Ein tieferes Verständnis für die Funktionsweise lässt sich durch ein simples Codereview nur schwer erreichen, ist jedoch eine Voraussetzung zur Verbesserung der Codequalität durch die Einführung neuer und komplexerer Unit Tests. Als Beispiel wird die Erstellung von Tilgungsplänen für Refinanzierungen näher betrachtet. Diese Thematik stellt einen zentralen Bestandteil der bereits genannten Finanzsoftware dar. Auf den ersten Blick erscheint die Berechnung der einzelnen Tilgungsplanzeilen für jemanden, der in dem Thema unerfahrenen ist, sehr komplex und unübersichtlich. Um trotzdem die Thematik verstehen und gegebenenfalls notwendige fachliche Änderungen am Quellcode vornehmen zu können, bietet sich die Erstellung eigener Testfälle mithilfe von Mockito an. Dabei ist zu Beginn nicht beabsichtigt, einen korrekt kalkulierten Tilgungsplan zu erzeugen, sondern wichtig ist nur die Tatsache, dass überhaupt ein Tilgungsplan berechnet oder dass die Erstellung mit einer Exception scheitern wird. Entgegen dem Vorgehen bei einem normalen Unit Test, kennt der Entwickler das Ergebnis nur grob. Ja, es ist sogar ungewiss, ob das endgültige Resultat einen brauchbaren Testfall abbildet. Die Gefahr bei dieser Art der Einarbeitung in eine unbekannte Fachlichkeit mithilfe von Unit Tests besteht darin, die eigentlich unbekannten Resultate als fachlich korrekt zu betrachten, daraus falsche Schlussfolgerun-

Listing 1

```
/*
 * Klasse, die die IBAN aus Kontonummer und BLZ errechnet
 */
private IBANCalculator calc;
/*
 * Bankverbindungsdaten der Leasinggesellschaft
 */
private Bankverbindungsdaten bvdaten;
/*
 * Leasinggesellschaft
 */
private LG lg;

@Before
protected void setup() throws Exception {
  this.calc = new IBANCalculator();

  // Mocken der Leasinggesellschaft
  this.lg = Mockito.mock(LG.class);
  // Mocken der Bankverbindungsdaten
  this.bvdaten = Mockito.mock(Bankverbindungsdaten.class);
  Mockito.doReturn(this.bvdaten).when(lg).getBankverbindungsdaten();
}

/*
 * Bestimmung der IBAN aus einer normalen Kontonummer und BLZ
 */
@Test
public void testCalculateIBAN() throws Exception {
  /*
  * Wenn die Kontonummer abgefragt wird, wird der Wert 0648489890 als String
  * zurückgegeben
  */
  Mockito.doReturn("0648489890").when(bvdaten).getKontonummer();
  /*
   * Wenn die BLZ abgefragt wird, wird der Wert 50010517 als String zurückgegeben
   */
  Mockito.doReturn("50010517").when(bvdaten).getBLZ ();
  // Bei der Länderkennung wird "DE" zurückgegeben
  Mockito.doReturn("DE").when(lg).getLaenderkennung();

  // calculateIBAN ruft die oben gemockten Methoden auf, um die IBAN zu ermitteln
  assertEquals("DE12500105170648489890", calc.calulateIBAN(this.lg));
}

/**
 * Bestimmung der IBAN aus einer Kontonummer und BLZ mit Leerzeichen
 */
@Test
public void testCalculateIBANWithValidWhitespaces() throws Exception {
  Mockito.doReturn("140 015 18 ").when(bvdaten).getKontonummer();
  Mockito.doReturn("130 000 00 ").when(bvdaten).getBLZ ();
  Mockito.doReturn("DE").when(lg).getLaenderkennung();

  assertEquals("DE26130000000014001518", calc.calculateIBAN(this.lg));
}

// ... Implementierung weiterer Testfälle
```

gen zu ziehen, andere Testfälle darauf aufzubauen und den Eindruck zu erwecken, dass Kernkomponenten des Altprojekts korrekt getestet wären. Der Entwickler ist sich dieser Problematik durchaus bewusst, denn er hatte auch nicht vorgehabt, die Testabdeckung zu verbessern, sondern nur die Einarbeitung zu erleichtern. Gefährlich wird es allerdings, wenn spätere Mitarbeiter das Vorhandensein von Testfällen zum Anlass nehmen, Refactorings von Methoden und Klassen vorzunehmen. Deswegen sollten die erstellten Testklassen immer mit *@Ignore* gekennzeichnet und zusätzlich mit hinweisenden Kommentaren versehen werden.

In Listing 2 wird beispielhaft ein erster Testfall für die Tilgungsplanberechnung erstellt. Der Fokus liegt auf der Grundlage jedes Tilgungsplans, der *KontoParameter*-Klasse. Das Ziel ist der Aufbau des Wissens über die notwendigen Werte für eine fehlerfreie Erstellung, nicht das Eruieren des Einflusses einzelner Kontoparameter auf die Zeilen des Tilgungsplans. Durch das Mocken der Methoden (*getZinssatz()*, *getZinsmethode*() u. a.) und das Verändern der Rückgabewerte erhält der Entwickler einen ersten Eindruck über das Verhalten der *Tilgungsplan*-Klasse. Das Fehlschlagen des Tests könnte zum Beispiel mit *null* als Rückgabewert provoziert werden.

Aufbauend auf diesen Erkenntnissen erstellen wir im folgenden Schritt einen Tilgungsplan mit monatlichen Zahlungen (Listing 3). Dafür wird dem Testfall der „Spy“ der Zahlungsreihe einer Refinanzierung hinzugefügt. Die Notwendigkeit des Spy besteht in der komplexen, vom Tilgungsplan genutzten Finanzmathematik der Zahlungsreihe, die mit dem aktuellen Wissen des

Listing 2

```
/*
 * Einarbeitungstest, um einen einfachen Tilgungsplan zu erstellen
 * ACHTUNG: Bildet keinen validen Unit Test ab
 */
@Ignore
public void testCalculateSimpleTilgungsplan() throws Exception() {
  KontoParameter kp = Mockito.mock(KontoParameter.class);
  // Mocken der Kontoparameter angefangen bei Zinssatz und Zinsmethode
  Mockito.doReturn(4.0).when(kp).getZinssatz();
  Mockito.doReturn("30_360").when(kp).getZinsmethode();
  // Mocken weiterer wichtiger Parameter wie Zahlungshäufigkeit ...

  Tilgungsplan tp = new Tilgungsplan(kp);
  // Holen der leeren Liste mit den eigentlichen Zeilen des Tilgungsplans
  List<TPZeilen> tpZeilen = tp.getTPZeilen();

  // Tilgungsplan existiert, ist aber leer => Test erfolgreich
  Assert.assertNotNull(tpZeilen);
  Assert.assertEquals(0, tpZeilen.size());
}
```

Anzeige

Entwicklers nicht adäquat gemockt werden könnte. Ausschlaggebend ist die Erkenntnis, dass die Methode *getZahlungen()* einen direkten Einfluss auf die Zeilen des Tilgungsplans besitzt, anders als die Angabe der Anzahl der Vorperioden, die aber ebenfalls benötigt wird.

In weiteren Testfällen kann der Entwickler die Beschaffenheit der Zahlungsreihe analysieren, indem er zum Beispiel Wissen über die angesprochenen Vorperioden aufbaut oder eine vorzeitige Ablösung dem Tilgungsplan hinzufügt. Pro Testfall sollte der Fokus auf ein konkretes Problem gelegt werden. Die Komplexität der Testfälle kann mit dem wachsenden fachlichen Wissen immer weiter ausgebaut werden. Indem der Entwickler Schicht für Schicht die Arbeitsweise einzelner Komponenten versteht, erreicht er nicht nur ein besseres Verständnis für das komplizierte Gesamtsystem, das die Grundlage für umfassende, tatsächliche Unit Tests bildet, sondern schafft gleichzeitig einen hervorragenden Einstiegspunkt in die Anwendung für neue, unerfahrene Entwickler. Diese haben die Möglichkeit, ihre eigenen Testfälle, basierend auf den vorhandenen, zu entwerfen, bestehende Einarbeitungstestfälle zu richtigen Unit Tests auszubauen und dabei die Robustheit des Systems weiter zu verbessern.

Fehler dokumentieren

Die Fehlersuche und Behebung umfasst den größten Teil des Wartungslebenszyklus von Software – seien es Schwierigkeiten mit der Performance durch stark ansteigende Datenströme und stetig wachsende Datenbanktabellen, sporadisch auftretende Probleme mit der Nebenläufigkeit oder Bugs in den neu entwickelten Erweiterungen. Im Idealfall analysiert der Entwickler gedanklich zuerst den Problemfall, versucht den Fehlerursprung auf wenige Klassen einzugrenzen, sondiert mögliche Lösungsvarianten und implementiert die optimale Variante, um den Ausfall des Produktivsystems so kurz wie möglich zu halten. Neue Projekte mit einer klaren Architektur, genau definierten Schnittstellen und einer umfangreichen Dokumentation sind dabei leichter zu warten als historisch gewachsene und bereits mit zahlreichen Bugfixes ausgestattete Wartungsprojekte. Der Idealfall kann durch die bescheidene Codequalität nur selten in Betracht gezogen und damit ausgeschlossen werden. Vielmehr ist es von Vorteil, zuerst zu analysieren, welche Art von Fehler genau vorliegt.

Bei vom Kunden gemeldeten technischen Problemen ist meist davon auszugehen, dass die Ursachen im eigenen Quellcode zu finden sind, besonders, wenn im Vorfeld Patches ausgeliefert wurden. Die Fehlersuche kann sich hierbei auf die geänderten Klassen beschränken, die durch die Versionsverwaltung identifiziert werden können. Nicht so trivial gestaltet sich die Analyse von fachlichen Fehlern, wie die Übertragung falscher Werte in ein angebundenes Fremdsystem. Ein reger Austausch mit der Fachabteilung des Kunden und eine detaillierte Schilderung der eigentlichen Programmlogik können zu einer raschen Aufklärung beitragen und klären, ob ein Bedienungsfehler oder ein jahrelang unentdecktes technisches Problem vorliegt. Notwendige Änderungen am Quellcode sollten dabei immer in Verbindung mit einer ausführlichen Dokumentation realisiert werden. Zwingend erforderlich werden detaillierte Beschreibungen bei Schnittstellen zu Fremdsystemen.

Bei Weitem schwieriger zu beheben, aber auch seltener sind technische Fehler, die bei der Wartung oder Weiterentwicklung der Altsoftware vom Entwicklerteam in den Kernkomponenten gefunden werden. Diesen Bugs liegt meist ein Designfehler oder ein unbekanntes Nebenläufigkeitsproblem zugrunde. Ein besonderes Merkmal dieser Art von Fehler ist das bisherige Nichtauftreten im Betrieb. Somit ist auszuschließen, dass kritische Anwendungsfälle betroffen sind, die einen Ausfall des Systems verursachen könnten. Eine Behebung des Fehlers sollte deshalb mit Bedacht geschehen. Mögliche Auswirkungen auf andere Komponenten des Altsystems müssen zuerst analysiert werden, um eine Abschätzung über den möglichen Aufwand treffen zu können. Eine Überprüfung der Testabdeckung der betroffenen Module gibt Aufschluss, ob nach einem Refactoring die fachliche Korrektheit des Systems garantiert werden kann. In jedem Fall muss ein entsprechender Eintrag in der Fehlerverwaltung und im Wiki verfasst werden, der Auskunft über die Ursache, Auswirkungen und Lösungsansätze des Problems bietet. Dadurch ist sichergestellt, dass eine Behebung zeitnah oder zu einem späteren Zeitpunkt erfolgen kann.

Listing 3

```
@Ignore
public void testCalculateNotSoSimpleTilgungsplan() throws Exception {
  // Mocken der Kontoparameter wie in Listing 1
  KontoParameter kp = Mockito.mock(KontoParameter.class);
  // Mocken von Zinssatz, Zinsmethode

  // Datumsangaben werden für die Zahlungsreihe benötigt
  final FinanzDatum valuta = new FinanzDatum(1, 1, 2013);
  final FinanzDatum ersteZahlung = new FinanzDatum(1, 2, 2013);

  // Erstellen einer Zahlungsreihe und Erzeugen eines Spions
  final Zahlungsreihe zr = new Zahlungsreihe(valuta, ersteZahlung, kp);
  final Zahlungsreihe spyZr = Mockito.spy(zr);

  // Mocken der für den Tilgungsplan wichtigen Methoden
  Mockito.doReturn(1).when(spyZr).anzahlVorperioden(Mockito.any(FinanzDatum.class));
  final double[] zahlungen = new double[] {500, 600, 700, 800, 900, 1000};
  Mockito.doReturn(zahlungen).when(spyZr).getZahlungen();

  Tilgungsplan tp = new Tilgungsplan(kp);
  tp.addRefinanzierungsZahlungsreihe(spyZr);

  // Tilgungsplan existiert und hat sieben Zeilen (sechs Zahlungen plus den Monat
  // ohne Zahlung, hier der 1.1.2013)
  Assert.assertNotNull(tpZeilen);
  Assert.assertEquals(7, tpZeilen.size());
  // weitere Überprüfungen oder auch eine Ausgabe des Tilgungsplans
  // zur Analyse möglich
}
```

Pflege der Dokumentation

Die Dokumentation ist die zentrale Komponente, um neue Mitarbeiter einzuarbeiten, Bugs zu beheben, Sonderfälle im Code zu identifizieren und neue Erweiterungen zeitnah in das Altprojekt integrieren zu können. Leider stellt sie auch den am häufigsten vernachlässigten Bereich dar – angefangen bei einer Anleitung zur Einrichtung des Workspace über eine technische und fachliche Beschreibung der Kernkomponenten der Anwendung, Aufbau oder Erhalt des eigenen Datenwikis bis hin zu ausführlichen Codekommentaren. Mit hoher Sicherheit haben alle Abschnitte Verbesserungspotenzial oder müssen in entscheidenden Punkten aktualisiert werden.

Ein guter Einstieg für den Neuaufbau ist die Erstellung der Anleitungen zur Einrichtung des Workspace und die Installation und Einrichtung zusätzlich benötigter Software. Anhand der eigenen Erfahrung können die nötigen Schritte und möglichen Fehlerquellen rekonstruiert werden und erlauben die Erstellung einer detaillierten Anleitung. Die technische und fachliche Beschreibung der Anwendung sollte dagegen nur von einem erfahrenen Entwickler vorgenommen werden. UML-Tools und deren Eclipse-Plug-ins [5] können durch die Generierung von Klassendiagrammen einen ersten Überblick über die Architektur bieten und helfen, Komponenten voneinander abzugrenzen oder deren Schnittstellen zu identifizieren. Auch eine schematische Darstellung der Datenbanktabellen unterstützt den Verständnisprozess. Abschließend ermöglicht ein Wiki die Bereitstellung und Archivierung aller erstellten Dokumente und Diagramme. Bei einer Neuinstallation des Wikis muss auf eine saubere und komplette Datenmigration geachtet werden. Nichts ist frustrierender als der Umstand, noch mehr Zeit und Aufwand in den Erhalt von überarbeitungswürdigen Dokumenten zu investieren.

public void doSomething()

Verständlicher Quellcode in Verbindung mit aussagekräftigen Kommentaren bildet das Grundgerüst, um eine problemlose Wartung von Altprojekten zu gewährleisten. Durch die Mitarbeit vieler Entwickler zu unterschiedlichen Zeiten und mit eigenen Vorstellungen der Quellcodeformatierung sinkt die Verständlichkeit des Quellcodes. Gleichzeitig werden, bedingt durch den zusätzlichen Zeitdruck, neue Kommentare nicht verfasst und bereits bestehende angepasst. Eine nachträgliche Bearbeitung und Korrektur dieser Problematik ist nur selten möglich. Gründe hierfür sind die pure, nicht zu bewältigende Masse an unkommentiertem Quellcode und die Unkenntnis über den eigentlichen Zweck von Methoden nach zahlreichen Erweiterungen, Bugfixes und kryptischen Variablenbezeichnungen. Besonders heikel erweist sich das Nichtkommentieren von Sonderfällen. Hier ist neben einer aussagekräftigen Beschreibung (wieso und was) auch die entsprechende Bug-ID – falls vorhanden – Pflicht. Spätere Erweiterungen sind auf exakte Schilderungen dieser Sonderfälle angewiesen. Insgesamt bilden Erweiterungen und Patches mitunter die einzige Möglichkeit, mit der Tradition zu brechen und eine vollständige Dokumentation von Code sicherzustellen oder Namenskonventionen für Klassen und Variablen einzuhalten. Die nachträgliche Kommentierung von altem Quellcode sollte nur von erfahrenen Entwicklern vorgenommen werden, die im besten Fall selbst Testfälle für die betreffenden Klassen verfasst haben.

Erhalt der Codequalität

Der letzte und besonders für nachfolgende Entwickler essenziell wichtige Schritt umfasst den Erhalt der Codequalität. Um zu verhindern, dass die durchgeführten Maßnahmen, wie die Einführung von Checkstyle, PMD und einheitlichen Formatierungsregeln, einmalige Aktionen bleiben, müssen Vorkehrungen zum Erhalt getroffen werden. Eine Aktualisierung des Build-Systems mit der Verwendung der Checkstyle- und PMD-Plug-ins stellt die Integrität der festgelegten Regelsätze sicher. Die lokale Installation und Pre-Commit-Verwendung der dazugehörigen Eclipse-Plug-ins verringert zusätzlich das Auftreten von Warnungen aufgrund falsch formatierter Klassen. Doch nicht nur das Altprojekt benötigt umfassende Wartungsarbeiten, um einen stabilen Programmablauf zu garantieren, auch die Einträge und Dokumente des Datenwikis sollten in zyklischen Abständen überprüft und gegebenenfalls umgeschrieben werden. Der Einsatz der Tools Vagrant [6] und Puppet [7] ist ebenfalls eine Überlegung wert, um eine einheitliche Arbeitsumgebung bei allen Teammitgliedern sicherzustellen. Aber bereits mit den in beiden Artikeln beschriebenen Maßnahmen, ausreichender Motivation, bewilligter Zeit und Budget kann durch eine Codequalitätsverbesserung die Produktivität im Altprojekt signifikant und nachhaltig verbessert werden.

Daniel Winter studierte Informatik an der FH Zittau/Görlitz und arbeitet seit 2011 als Consultant für Softwareentwicklung bei der Saxonia Systems AG (daniel.winter@saxsys.de). Sein aktueller Fokus liegt in den Möglichkeiten der Codequalitätsverbesserung bei Projekten jedweder Art. Ausgiebige Erfahrungen in diesem Bereich sammelte er bei der Wartung einer Leasing-Refinanzierungssoftware.

Links & Literatur

[1] Mockito: http://code.google.com/p/mockito/

[2] TDD: https://en.wikipedia.org/wiki/Test-driven_development

[3] Tutorial für Mockito: http://docs.mockito.googlecode.com/hg/latest/org/mockito/Mockito.html

[4] SEPA: http://de.wikipedia.org/wiki/Einheitlicher_Euro-Zahlungsverkehrsraum

[5] Eclipse-UML-Plug-in: http://www.eclipse.org/modeling/mdt/?project=uml2

[6] Vagrant: http://www.vagrantup.com/

[7] Puppet: https://puppetlabs.com/

Smart Bluetooth seit Android 4.3

Bluetooth revisited

GOOD TO KNOW

Eines der großen Features, die das Android-Update auf die Version 4.3 mit sich bringt, ist das so genannte „Smart Bluetooth“ oder auch „Bluetooth Low Energy“. Dass es dabei um eine stromsparende Variante von Bluetooth geht, verrät ja schon der Name. Aber was verbirgt sich genau hinter diesem Begriff? Wie unterscheidet sich diese Technologie von dem bisher in Android genutzten Bluetoothstandard? Auf welche Änderungen des Bluetooth-API muss sich ein Entwickler gefasst machen, wenn er das neue API verwenden will? Diese Kolumne wird den neuen Bluetoothstandard und seine Einbettung in Android näher beleuchten. Dabei liegt der Fokus auf den Neuerungen, die das Android-API in diesem Bereich bringt, und wie diese mit dem bisherigen Bluetooth-API zusammenspielen.

von Lars Röwekamp und Arne Limburg W-JAX Speaker

Bluetooth ist eine lange bekannte Technologie, die es Geräten ermöglicht, sich gegenseitig auf kurze Distanz zu finden, zu verbinden und miteinander zu kommunizieren oder Daten auszutauschen. Aufgrund des geringeren Energieverbrauchs im Vergleich zu WLAN ist diese Technologie vor allem dann geeignet, wenn eines der kommunizierenden Geräte keinen festen Stromanschluss hat, sondern auf Batteriebetrieb basiert. Bekannte Anwendungsfälle sind drahtlose Mäuse und Tastaturen, drahtlose Headsets, drahtloses Drucken, Audiostreaming und mittlerweile auch die Anbindung von medizinischen Messgeräten zum Messen von Blutdruck, Körpertemperatur, Gewicht, EKG, Puls, Blutzucker usw.

Der Fokus auf den niedrigen Energieverbrauch wurde in der Version 4.0 der Bluetoothspezifikation (nicht zu verwechseln mit Android 4.0) noch einmal verstärkt, und zwar mit der Einführung eines komplett neuen Protokolls, dem „Bluetooth Low Energy“-Protokoll. Generell wird Energie bei Bluetooth vor allem in zwei Phasen verbraucht: Beim Scannen nach Devices und bei der Übertragung von Daten. Bluetooth Low Energy legt daher den Fokus auf genau diese beiden Phasen und sorgt dafür, dass sie seltener notwendig sind und kürzer andauern.

Bluetooth in Android

Das Android SDK unterstützt Bluetooth seit der Version 2.0 [1]. Damit die eigene App Bluetooth verwenden kann, muss die Permission *android.permission.BLUETOOTH* im Manifest eingetragen werden. Wenn die Anwendung zusätzlich auch das Scannen von und das Verbinden mit Devices anstoßen können soll, wird auch die Permission *android.permission.BLUETOOTH_ADMIN* benötigt. Die Kommunikation mit der Bluetoothschnittstelle erfolgt dann über die Klasse *android.bluetooth.BluetoothAdapter*. Eine Instanz dieser Klasse erhält man über die statische Methode *BluetoothAdapter.getDefaultAdapter()*. Sollte diese Methode *null* zurückliefern, wird von dem vorliegenden Gerät kein Bluetooth unterstützt.

Im zweiten Schritt muss sichergestellt werden, dass Bluetooth überhaupt aktiviert ist. Erfragen lässt sich dies über die Methode *isEnabled()* des *BluetoothAdapter*s. Sollte es nicht aktiviert sein, kann der Benutzer über *startActivityForResult(new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE), 42)* darum gebeten werden, Bluetooth zu aktivieren. Dies geschieht dann automatisch durch das Android-System über einen Dialog. Der zweite Parameter (in diesem Fall die 42) ist eine beliebige positive Zahl, mit der sich das System nach abgeschlossenem Aktivierungsversuch bei der Activity über die Methode *onActivityResult* zurückmeldet.

Als Alternative zu der expliziten Aktivierung kann die App im Übrigen auch auf das Broadcast-Intent *BluetoothAdapter.ACTION_STATE_CHANGED* lauschen, um bei einer Bluetoothstatusänderung direkt informiert zu werden.

Hat man sichergestellt, dass Bluetooth aktiviert ist, kann man entweder die Liste der verbundenen Geräte über *BluetoothAdapter.getBondedDevices()* nach einem konkreten Gerät durchsuchen oder alternativ über *BluetoothAdapter.startDiscovery()* das Scannen nach Devices in der Umgebung starten.

Hat man ein *BluetoothDevice* gefunden, mit dem man sich verbinden will, kann die Kommunikation über einen *BluetoothSocket* gestartet werden. Daten können dann low-level in Form von Bytearrays ausgetauscht werden.

Verwendung von Profilen ab Android 3.0

Da der Austausch über Bytearrays recht umständlich ist, wurde in Android 3.0 ein High-Level-API zur Kommunikation über Bluetooth eingeführt: die so genannten *BluetoothProfiles*. In Android 3.0 wurden die Profile *BluetoothA2dp* (für Audiostreaming) und *BluetoothHeadset* (für Headsets) mitgeliefert. Seit Android 4 gibt es zusätzlich das Profil *BluetoothHealth* zur Verbindung mit medizinischen Geräten.

Proprietäre Bluetooth-Low-Energy-APIs

Die komplette Funktionalität, die bis zu diesem Punkt hier beschrieben wurde, basiert auf dem „klassischen" Bluetoothstack. Betrachtet man die Tatsache, dass die Bluetoothversion 4.0 bereits am 30.06.2010 verabschiedet wurde, also bereits seit drei Jahren gültig ist, erscheint es etwas seltsam, dass Android diesen Standard nicht schon eher implementiert hat. Zum Vergleich: Im Dezember 2010 erschien die Version 2.3 von Android. Der Grund, dass Android diesen Standard so lange verschmähte, ist vermutlich die geringe Verfügbarkeit von passenden Chipsätzen zu dem damaligen Zeitpunkt.

Allerdings verwundert es da nicht, dass Hersteller, deren Geräte Bluetooth Low Energy unterstützen, auch APIs für die Verwendung des Protokolls zur Verfügung stellen. Und so gibt es bereits seit einiger Zeit von Samsung [2] und HTC [3] herstellerspeziefische APIs, um Bluetooth Low Energy in Android auf den jeweiligen Geräten zu unterstützen. Diese können auch in Android-Versionen < 4.3 verwendet werden.

Bluetooth Low Energy in Android 4.3

Wie bereits oben geschrieben, bietet Android mit der gerade veröffentlichten Version 4.3 anbieterunabhängige Unterstützung von Bluetooth Low Energy [4]. Es basiert auf dem Generic Attribute Profile (GATT), einer Spezifikation zur Kommunikation mit kleinen Datenmengen, durch die sich Bluetooth Low Energy auszeichnet.

Datenstrukturen innerhalb der Kommunikation von GATT werden durch so genannte Characteristics repräsentiert. Sie sind in der Bluetooth-Low-Energy-Spezifikation festgelegt [5]. In Android werden sie durch die Klasse *BluetoothGattCharacteristic* repräsentiert.

Ein Characteristic repräsentiert einen oder mehrere zusammengehörende Datenwerte inklusive Datentypen und Wertebereich. So besteht das Characteristic *org.bluetooth.characteristic.date_time* z. B. aus den einfachen Datentypen

- *org.bluetooth.unit.time.year*
- *org.bluetooth.unit.time.month*
- *org.bluetooth.unit.time.day*
- *org.bluetooth.unit.time.hour*
- *org.bluetooth.unit.time.minute und*
- *org.bluetooth.unit.time.second*

Anzeige

Das Characteristic *org.bluetooth.characteristic.day_date_time* wiederum ist zusammengesetzt aus

- *org.bluetooth.characteristic.date_time und*
- *org.bluetooth.characteristic.day_of_week*

Neben den beiden genannten, recht allgemeingültigen Characteristics gibt es aber auch so konkrete wie „Heart Rate Measurement“ oder „Cycling Power Measurement“.

Characteristics werden dann in so genannten Services gesammelt [6]. Diese werden in Android durch die Klasse *BluetoothGattService* repräsentiert. Auch Services können wiederum weitere Services enthalten.

Um sich mit einem Bluetooth Low Energy Device zu verbinden, muss zunächst Bluetooth aktiviert werden. Schnittstelle ist auch hier der *BluetoothAdapter*. Die Methode zum Scannen nach Devices unterscheidet sich aber vom klassischen Bluetooth-API. Seit Android 4.3 hat die Klasse *BluetoothAdapter* hierzu die Methode *startLeScan*, die als Parameter ein *LeScanCallback* bekommt, das informiert wird, wenn ein Bluetooth Low Energy Device gefunden wurde (Listing 1). Mit dem gefundenen Device kann sich dann über die Methode *connectGatt* verbunden werden. Diese Methode erwartet drei Parameter. Als erster wird das obligatorische Context Object erwartet. Der zweite Parameter ist ein Boolean-Wert, der anzeigt, ob sich automatisch mit dem entsprechenden Device verbunden werden soll, sobald es in Reichweite ist. Der dritte Parameter ist ein Callback, das die gesamte Kommunikation mit dem GATT beinhaltet. Es enthält drei Methoden, über die man informiert wird, wenn sich ein GATT-Device verbindet, wenn sich ein GATT-Service registriert und wenn Characteristics vom Device gelesen werden. In diesem Listener kann die Logik implementiert werden, um die Daten aus dem Bluetooth Low Energy Device auszulesen (Listing 2).

Listing 1

```
bluetoothAdapter.startLeScan(new BluetoothAdapter.LeScanCallback() {
    public void onLeScan(BluetoothDevice device, int rssi, byte[] scanRecord) {
        ...
    }
});
```

Listing 2

```
device.connectGatt(this, true, new BluetoothGattCallback() {
    public void onConnectionStateChange(BluetoothGatt gatt, int s, int newState) {
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            ...
        } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
            ...
        }
    }

    public void onServicesDiscovered(BluetoothGatt gatt, int status) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            ...
        } else {
            ...
        }
    }

    public void onCharacteristicRead(
            BluetoothGatt gatt,
            BluetoothGattCharacteristic characteristic,
            int status) {
        if (status == BluetootGatt.GATT_SUCCESS) {
            ...
        }
    }
});
```

Fazit

Bluetooth Low Energy wurde entwickelt, um es Peripheriegeräten wie z. B. Herzfrequenzmessern, Pulsmessern, Blutzuckermessgeräten oder auch Fahrradcomputern zu ermöglichen, mit einem sehr geringen Energieverbrauch Informationen zur Verfügung zu stellen.

Google hat sich lange Zeit gelassen, um diesen Standard in das Android-API aufzunehmen. Im Gegenzug ist das entstandene Bluetooth-LE-API ausgereift und bildet den kompletten Bluetooth-Low-Energy-Standard ab. Mit ihm ist es möglich, zwischen einem Android-Device und Peripheriegeräten wie den oben genannten mit niedrigem Energieverbrauch zu kommunizieren.

Sollte eines der beiden Geräte diesen Standard nicht unterstützen, kann jederzeit auf das klassische Bluetooth zurückgegriffen werden. Das dann zu verwendende API unterscheidet sich dann allerdings komplett vom Bluetooth-LE-API.

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://developer.android.com/guide/topics/connectivity/bluetooth.html

[2] http://developer.samsung.com/ble

[3] http://www.htcdev.com/devcenter/opensense-sdk/bluetooth-smart/

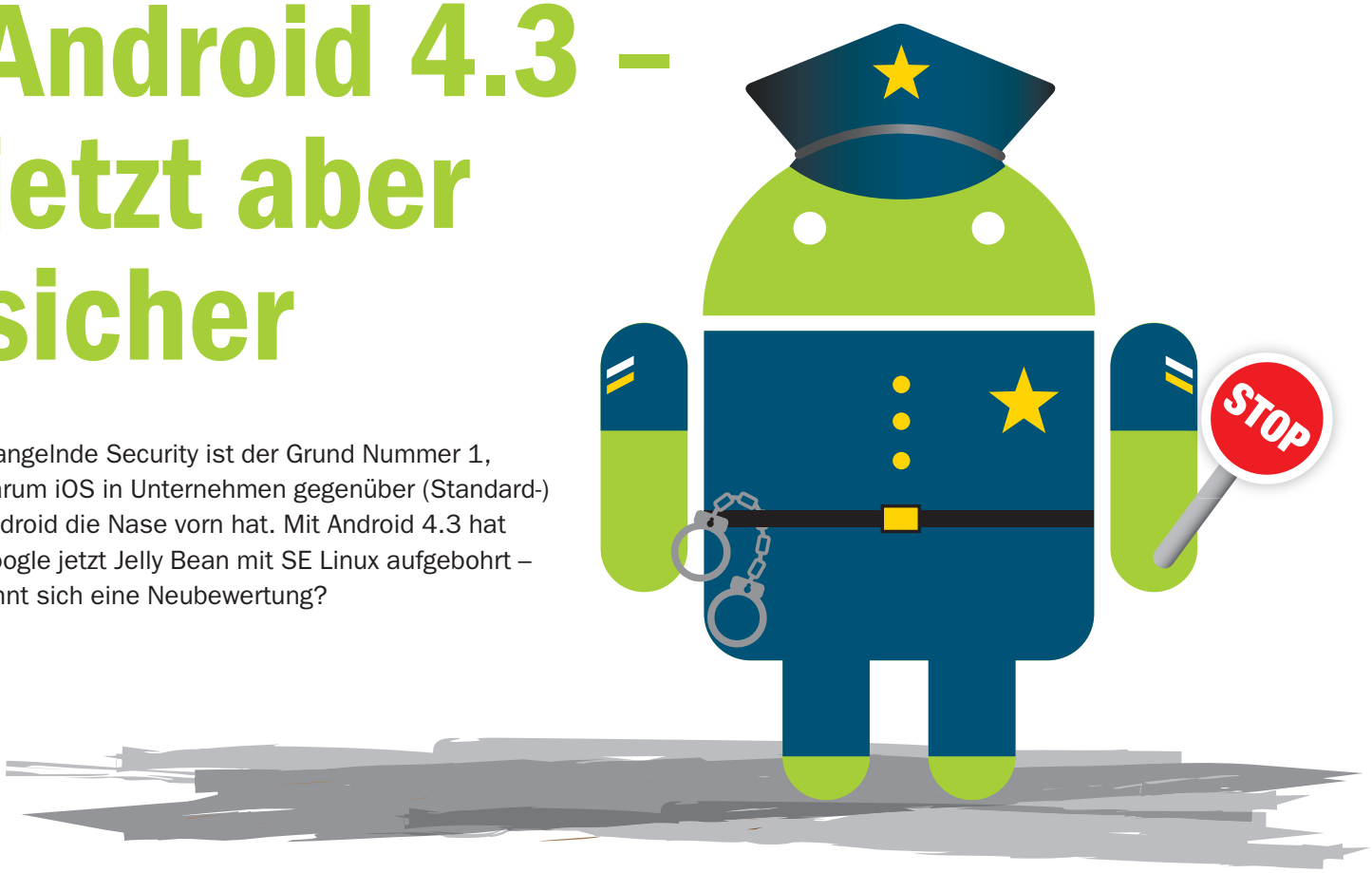
[4] http://developer.android.com/guide/topics/connectivity/bluetooth-le.html

[5] https://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicsHome.aspx

[6] https://www.bluetooth.org/en-us/specification/adopted-specifications

Ein erster Eindruck der neuen Securityfeatures in Android 4.3

Android 4.3 – jetzt aber sicher



Mangelnde Security ist der Grund Nummer 1, warum iOS in Unternehmen gegenüber (Standard-) Android die Nase vorn hat. Mit Android 4.3 hat Google jetzt Jelly Bean mit SE Linux aufgebohrt – lohnt sich eine Neubewertung?

von Jan Peuker

w-jax Speaker

Am 24. Juli wurde die Version 4.3 der Android-Plattform, Jelly Bean MR2, vorgestellt. Schon auf der Google I/O wurde die neue Version erwartet, und erste Leaks machten bereits die Runde. Was neben Verbesserungen bei der Performance auffällt, sind einige erweiterte Securityfeatures. Deshalb lohnt es sich, aus Unternehmenssicht einen Blick auf die Neuerungen zu werfen. Einiges mag negativ klingen – ich möchte aber vorausschicken, dass ich Android 4.3 für das aus Securitysicht wichtigste Release seit 2.3 halte.

Vor einem knappen Jahr unternahm ich auf der MobileTech Con 2012 eine kleine Befragung unter den Konferenzteilnehmern zu „Android's Missing Features“ (siehe Mobile Technology 1.13 „Die Android-5-Wunschliste für Unternehmen“). Anhand dieser Liste möchte ich einen Vergleich ziehen: ein Jahr danach – was hat sich getan? Der größte Schritt ist sicher die Einführung von SE Linux. Auf der MTC 2012 war dieses Feature von Samsung KNOX auf Platz 5 der am meisten für die Allgemeinheit geforderten Features – und hat es jetzt endlich geschafft. Zusammen mit dem in Android 4.2 eingeführten Mehrbenutzerbetrieb bildet SE Linux das Fundament der neuen Sicherheitsarchitektur.

Profile

Im Rückblick auf das vergangene Jahr könnte man diagnostizieren, dass Google den Fokus der Aktivitäten deutlich auf Google Now, also den Nicht-Open-Source-Teil, verlegt hat. Indikatoren waren die I/O, die Ankündigung des Moto X [1] und seinem „OK,

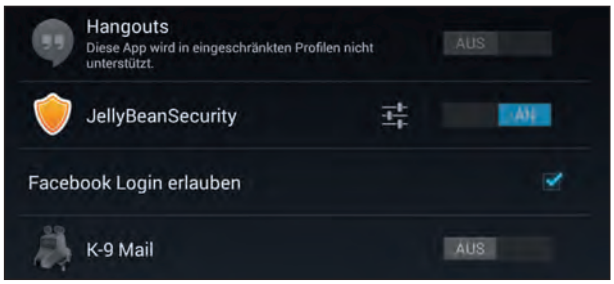


Abb. 1: Anzeige der definierten Einschränkungen

Google Now“ sowie die weiterhin fehlenden Updates für den Android-Standardbrowser. Konsequenterweise hat Google die neue Sicherheitsarchitektur auf Google-Konten ausgerichtet, was mit Android 4.2 als Multi-User eingeführt wurde. Die Geräteeinrichtung beginnt pro User wie nach einem Factory Reset, die Daten sind getrennt ähnlich zu Profilen, wie wir sie von BlackBerry kennen. Eine Data Loss Prevention analog dem von Motorola übernommenem und dann von Google eingestellten 3LM ist damit zwar noch nicht erreicht, der Grundstein aber gelegt – deshalb gibt's ein „Halb“ für Punkt 2 der MTC-Liste.

Heise [2] bezeichnete die Android-Security als „marode“, weil Benutzer keine freie Wahl bei der Bestätigung von Zugriffsbeschränkungen haben. iOS bietet hier in der Tat ein wenig mehr Komfort als Android. Im Datenschutzmenü können Benutzer dort z. B. den Zugriff auf Kontakte beschränken; Apps fragen vor dem Zugriff auf Daten nach, und dieser Zugriff kann separat bestätigt werden. Zudem bietet iOS „Einschränkungen“, die hauptsächlich für Jugendschutz und zur Unternehmenssicherheit gedacht sind, wie etwa die Deaktivierung der Kamera oder „anstößiger Sprache“. Und mit iOS 7 wird „Managed Open“ – eine Vorstufe zur Data Loss Prevention – eingeführt. Doch auch iOS ist insofern „marode“, als dass nur recht spezifische Beschränkungen bearbeitbar sind, nicht etwa eine generelle Unterbindung von Internet- oder Speicherzugriff.

Android 4.3 geht deutlich weiter und führt das *Restricted Profile*-API ein. Mit diesem wird es jeder App ermöglicht, vollkommen frei Beschränkungen zu definieren, bis hin zum Einzelbetrieb von Anwendungen im Kioskmodus, wie ihn iOS auch unterstützt. Das Restricted-Profile-API erlaubt spezifische Einstellungen pro und sogar innerhalb der Anwendung. Dies wurde genutzt, um im Standard bereits ähnlich den Möglichkeiten von iOS den Play Store, In-App Payment, Ortung oder Spiele einzuschränken – das API steht aber jeder Anwendung offen. Sie können auf einen Broadcast *ACTION_GET_RESTRICTION_ENTRIES* reagieren, damit ihre möglichen Beschränkungen publizieren und dann später über den *UserManager* abfragen, die davon gesetzt wurden. In Listing 1 definieren wir als Antwort auf den Broadcast eine Einschränkung unserer Anwendung: ob ein Login über Facebook erlaubt ist oder nicht. In Listing 2 lesen wir die Wahl dieser Einschränkung aus.

Leider gehen Profile und Benutzer aber noch nicht Hand in Hand. Profile sind unterhalb von Benutzern angesiedelt. Eingeschränkte Apps können zwar auf Daten des Benutzerkontos zugreifen, eine 1:1-Kopplung von Benutzer (Google-Konto) und Rolle (Restricted Profile) ist aber noch nicht möglich. Die App muss spezielle Berechtigungen anfragen, um auf den Account Manager zuzugreifen – leider tun dies die Google-Apps wie z. B. E-Mail/GMail noch nicht und sind deshalb nicht mit eingeschränkten Profilen, Punkt 5 der MTC-Liste, nutzbar.

Mobile Device Management

Es ist weiterhin nicht möglich, als „System“ gekennzeichnete Apps (wie z. B. Google Hangouts) völlig zu deaktivieren und jegliche installierten Apps aus der Ferne zu deinstallieren. Denn die meisten Whitelisting-Lösungen im Mobile Device Management wie z. B. Afaria oder AirWatch klinken sich in den Installer ein – der hier leider zu spät kommt. Umgehen könnte man dies, indem

Platz	Nennung	Status Android 4.3
1	Black-/Whitelisting für Apps	Nein – weiterhin nur mit speziellen Versionen von Samsung, Motorola etc.
2	Trennung private/Corporate-Daten (DLP)	Halb – verschiedene Profile können angelegt werden, aber nicht aus der Ferne konfiguriert.
3	Nutzbares NFC-API ohne Wallet	Nein.
4	Policies für Debug- und Share-Menü	Halb – Secure Debugging mit RSA seit 4.2.2, aber keine Policies.
5	Zertifikate und S/MIME im Standard-E-Mail-Client	Nein – SCEP und x.509 S/MIME weiterhin nur mit TouchDown o. Ä., aber wenigstens können Mails lokal verschlüsselt abgelegt werden.
6	Samsung/Afaria AES in den Standard	Halb – nicht die umfangreichen Policies und Installer übernommen, aber SE Linux und Profile.
7	Bluetooth-Druck-API	Halb – mit Android 4.2 neue Bluetooth-Funktionen und BLE in Android 4.3.
8	Standard-SQLite-Implementierung	Nein.
9	EAP-SIM für WiFi	Halb – EAP und WPA2 werden mit Android 4.3 unterstützt.
10	Enterprise App Store	Halb – Private Channel in Play und Profile für Apps, aber nicht zertifikatsbasiert und nachträglich.

Tabelle 1: Die Top 10 „Missing Features“ auf der MTC 2012 und der Status zu Android 4.3

Google ein Berechtigungskonzept für Anwendungen auf Zertifikatsbasis einführt – analog zu internen Android-Schnittstellen oder iOS. Doch genau hier ist Androids Play-Integration zu fundamental und die Deaktivierung zu halbherzig, Punkt 10 in der MTC-Liste. So können beispielsweise auf Samsung-Geräten Play deaktiviert und Anwendungen OTA ohne Benutzerinteraktion installiert werden – für Systemupdates muss das Gerät aber bei Samsung registriert werden, und der Samsung App Store bleibt immer offen. Zudem ist es dann notwendig, „nicht vertrauenswürdige" Anwendungen zur Installation freizugeben. Das ist auch mit Whitelisting gefährlich, da die Installationsdateien trotzdem heruntergeladen werden und so Schwachstellen ausnutzen könnten.

Security-APIs

Bereits mit Android 4.2 wurde Multi-User-Funktionalität eingeführt. Damit ist es für Entwickler wichtiger, Daten an die richtigen Orte zu schreiben, keine Hardwareschlüssel mehr zur Identifizierung zu verwenden (wie es iOS auch schon lange forciert) und zu wissen, welche Einstellungen wirklich benötigt werden, denn einige, wie z. B. Funkverbindungsdetails, Debug, Proxy oder Flugmodus, wurden verschoben. Zudem wurden viele Sicherheitsfunktionen wie z. B. SecureRandom neu an OpenSSL delegiert, was insbesondere die Schlüsselverwaltung sicherer macht. Doch auch in Androids Open-SSL-Verbindung wurde gerade ein Bug mit JCA-Nutzung gefunden.

In Android 4.3 werden jetzt im Rahmen der neuen WPA2-, Media- und DRM-Funktionen neue APIs zur Schlüsselverwaltung bereitgestellt. Die bereits in ICS eingeführte KeyChain unterstützt dafür jetzt auch Hardwaresecuritymodule (Trusted Platform Module) und neue Algorithmen. Ein neuer, in Java-Security eingebetteter Provider „AndroidKeyStore" erlaubt einfacheren Zugriff auf den Keystore. Das macht es für Anwendungen einfacher, sicher Schlüssel zu verwalten, wenn man diese nicht der KeyChain anvertrauen will. Ein fiktives Beispiel dafür (Listing 3) könnte die Implementierung von Clientzertifikaten sein. Mit SE Linux und einigen Änderungen unter der Haube (wie z. B. Einschränkungen beim *setuid*, was allerdings auf dem Samsung S4 bereits ausgehebelt wurde) wären diese im Speicher auch gegen Root-Zugriff gefeit – bleibt nur, darauf zu hoffen, dass bald ein Browser sie zusammen mit SCEP implementiert.

Kommunikation

Der dritte wirklich große Schritt nach Profilen und der KeyChain ist WPA2 Enterprise, also Wi-Fi-Konfiguration für Unternehmen, Punkt 9 auf der MTC-Liste. WPA2 war zwar grundsätzlich bereits ab Android 4 möglich, die Zertifikate dafür muss man aber manuell herunterladen, eine Proxykonfiguration war nicht möglich. Jetzt wurde WPA2 um ein ausführliches Konfigurations-API erweitert, das reichhaltige EAP-Möglichkeiten bietet. Neben Standard-PEAP und Passwortschutz kann jetzt auch EAP-TLS mit X.509-Clientzertifikaten konfiguriert werden. Einzig EAP-SIM fehlt weiterhin – und daher auch nur ein „Halb" auf der Liste.

Bluetooth Low Energy und Bluetooth Smart haben Einzug gehalten und damit die Vormachtstellung von

Listing 1: Android-4.3-Restrictions-API – Empfang des Broadcasts

```
if (intent.getAction().equals(Intent.ACTION_GET_RESTRICTION_ENTRIES)) {
// Create PendingResult to offload permission creation to Thread
final PendingResult pendingResult = goAsync();
new Thread() {
 @Override
 public void run() {

  final Bundle extras = new Bundle();

  // Constructor for Boolean permission
  final RestrictionEntry facebookLoginPermissionEntry
= new RestrictionEntry(Constants.PERMISSION_ALLOW_FACEBOOK_LOGIN, true);
  facebookLoginPermissionEntry.setTitle(context.getResources().getString(R.string.
                                                    permission_facebook_login));

  final ArrayList<RestrictionEntry> permissionEntries
= new ArrayList<RestrictionEntry>();
  permissionEntries.add(facebookLoginPermissionEntry);

  extras.putParcelableArrayList(Intent.EXTRA_RESTRICTIONS_LIST, permissionEntries);

pendingResult.setResult(Activity.RESULT_OK, null, extras);
pendingResult.finish();

 }

}.start();

}
```

Listing 2: Android-4.3-Restrictions-API – Abfrage der Permissions

```
    @SuppressLint("NewApi")
  private void createUiElements() {
   if (android.os.Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR2) {
    final UserManager userManager = (UserManager) getSystemService(Context.USER_
                                                                    SERVICE);
    final Bundle restrictions = userManager.getUserRestrictions();
    if (!restrictions.getBoolean(Constants.PERMISSION_ALLOW_FACEBOOK_LOGIN, false))
{
     // Do not show Facebook login option
     final EditText facebookLogin = (EditText) findViewById(R.id.editTextFacebookLogin);
     facebookLogin.setVisibility(View.GONE);
    }
   }
  }
```

Android gegenüber iOS bei allem, was Sensoren und Peripheriegeräte anbelangt, weiter ausgebaut, Punkt 7 der Liste ist damit auch halb erfüllt.

Sicherheit heisst auch Monitoring und Support. Für Unternehmen stehen zudem mittlerweile sehr gute Supportlösungen zur Verfügung. Mit TeamViewer und Konsorten kann man z. B. auf Samsung-Geräten mittlerweile Screen Sharing aktivieren und Prozesse administrieren. Dem gegenüber steht die Unsicherheit der neuen „parallelen" Entwicklungsslinie IntelliJ und Gradle, denn die meisten größeren Unternehmen setzen auf Eclipse und Maven als zentrale Werkzeuge. Auch wenn ich persönlich IntelliJ und Gradle toll finde, so ist es doch schade, wie schlecht z. B. die Support-Library über Maven unterstützt wird. Und das, obwohl die Support-Library gerade eine exzellente Überholung inklusive ActionBar erhalten hat.

Listing 3: Android 4.3-AndroidKeyStore-API

```
        final KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
    keyPairGenerator.initialize(rsaSpec);

// Generate Private/Public Key
final KeyPair keyPair = keyPairGenerator.generateKeyPair();

// Type is android.security.AndroidKeyStore.NAME
final KeyStore androidKeyStore = KeyStore.getInstance("AndroidKeyStore");
androidKeyStore.load(null, null);
androidKeyStore.setEntry("JavaMagazinAlias", new PrivateKeyEntry(keyPair.getPrivate(),
                                                chain), (new KeyStoreParameter.Builder(
    getApplicationContext())).build());

// Example: Use Private key to set Client Certificate
final KeyManagerFactory keyManagerFactory = KeyManagerFactory.
                                        getInstance(keyPairGenerator.getAlgorithm());
keyManagerFactory.init(androidKeyStore, password);
```

Fragmentierung als Chance

Auf dem Markt der SDKs, Responsive Design Cross-Platform Toolkits, MADPs und Mixed-Model-Plattformen findet derzeit bei konstant hoher Ausdehnung eine Konsolidierung statt. Ein Blick auf die Sicherheit bei Android muss daher auch ein Blick auf die Security bei Cross-Plattform-Toolkits sein. Abgesehen von der in den letzten Wochen religiös geführten Performancediskussion [2] stellt die auf den meisten Plattformen im Vergleich zu Chrome stiefmütterlich behandelte WebView das größte Problem für die Cross-Plattform-Lösungen dar. Sie unterstützt zu wenige SSL-Features (z. B. keine Clientzertifikate aus der KeyChain) – und das hat sich auch mit Android 4.3 nicht geändert.

Doch Fragmentierung bietet auch eine Chance: Da Google generell Webtechnologien gut unterstützt und man davon ausgehen kann, bald einige der neuen Security-APIs in PhoneGap und Konsorten wiederzufinden, könnten bald echte Cross-Device-Anwendungen mit Single Sign-On Realität werden. Wenn der Container sicher ist, aber man mehr oder weniger gezwungen ist, die Daten in der (eigenen) Cloud zu synchronisieren, führt das auch zu effizienten, über Gerätegrenzen hinweg nutzbaren Anwendungen. Was heute oft als „Contextual Fragmentation" [4] bezeichnet wird, also die Fragmentierung nicht von Device-Layouts (die hat iOS auch), sondern von Inhalten, kann man vom Fernseher über Tablet und Smartphone bis hin zur Smartwatch ausweiten – damit werden die APIs zu Geschäftsprozessen wieder zentraler und wichtiger.

iOS

Andere Plattformen haben in der Zwischenzeit nachgelegt. Die VPN-Konfiguration ist eine traditionelle Stärke von iOS. Mit App-abhängigen VPN-Kanälen wurde dies in iOS 7 noch verbessert. Sogar ein Enterprise-Single-Sign-On über mehrere Apps wurde angekündigt. Auch wurde die ohnehin schon gute hardwareverstärkte Verschlüsselung erweitert und auf alle Apps als Fast-DLP ausgedehnt – etwas Ähnliches wird es mangels strikter Hardwarestandards unter Android wohl noch lange nicht geben. Und einer Keychain analog Safari würden wohl die wenigsten Google-Nutzer vertrauen.

Was gab es sonst noch? Die Accessibility-Funktionen und Sprachunterstützung wurden deutlich ausgebaut, und mit dem *UI Automation Framework* gibt es jetzt eine offizielle Alternative zu Robotium-UI-Tests. Zu guter Letzt eine kleine Änderung: Eine neue Systrace-Funktion erlaubt die tiefere Analyse von ANRs, und in Android 4.2 wurde ja bereits das Debugging deutlich sicherer gemacht.

Eine sehr solide Sicherheitsarchitektur

Zusammenfassend kann man sagen, dass Google mit Android 4.3 ein großer Wurf in Richtung Security gelungen ist. Viele neue APIs wurden zu einer stimmigen neuen Plattform kombiniert. Hier und da wünscht man sich ein wenig mehr Integration zwischen den neuen Funktionalitäten, sie stimmen aber endlich wieder zuversichtlich, dass Google die Bedenken ernst nimmt. Android 5 wird iOS ebenbürtig – und es vielleicht sogar abhängen.

Jan Peuker leitet bei Accenture die Spezialistengruppe zur Systemintegration mobiler Technologien. In Projekten beschäftigt er sich mit Android, BYOD und Java-API-Management.

@janpeuker

Links & Literatur

[1] http://www.wired.com/gadgetlab/2013/08/inside-story-of-moto-x/

[2] http://www.heise.de/mobil/meldung/Android-4-3-bringt-eingeschraenkte-Profile-und-Bluetooth-4-0-1923324.html

[3] http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/

[4] http://opensignal.com/reports/fragmentation-2013/

Quellcode zum Download

Nudeln, onCreate und das Geheimnis von R

Die 5-Minuten-App

Image licensed by Ingram Image

Kann man eine App in fünf Minuten erstellen? Ein gewagtes Unterfangen, aber solange man nicht auf die Uhr schaut, könnte es klappen. Zeit für eine ganz kurze Zusammenfassung, um sich noch einmal in Erinnerung zu rufen, was wir bisher gemacht haben. Danach gehen wir dem Geheimnis von „R“ auf den Grund.

von Stephan Elter

W-JAX Speaker

Wir haben ein neues Projekt in Eclipse angelegt, genauer gesagt ein Android-Application-Projekt, die Grundlage für eine erste, einfache App. Wir haben ein Icon erstellt und uns ist aufgefallen, dass automatisch mehrere Icons in unterschiedlichen Größen erstellt wurden, eine grundlegende Technik, mit der in Android verschiedene Auflösungen bedient werden können. Wir haben erfahren, was eine Activity ist und dass eine App aus einer oder mehrerer dieser Activities bestehen kann. Wir haben ein wenig am Aussehen unserer App bzw. unserer bis dahin einzigen Activity gebastelt und haben erfahren, woher die Texte in unserer Activity wirklich herkommen, was Views, ViewGroups und LayoutManager sind und haben sogar noch ein Bild eingebunden.

Was bleibt uns denn da noch übrig? Tatsächlich noch eine ganze Menge, denn unsere 5-Minuten-App ist noch nicht fertig. Zeit für etwas Gehaltvolleres, Zeit für die Nudeln: Wir werden uns erstmals den Java-Quellcode unserer Activity ansehen und damit arbeiten, das Rätsel von „R“ lösen, ein (unvermeidliches) Toast zubereiten. Und dann geht es richtig los. Wir werden eine zweite Activity in unserer App anlegen und sie dann von der ersten Activity aus starten. Und natürlich werden wir wieder einigen unerklärlichen Begriffen begegnen, die bei genauerem Hinsehen doch ganz einfach sind.

Nudeln, onCreate und das Geheimnis von „R“

Wir öffnen unser Projekt und sehen uns den Java-Quellcode an, der zu unserer ersten Activity gehört, die Datei *StartSchirm.java* (natürlich immer vorausgesetzt, Sie hatten die Datei genauso benannt wie in den Screenshots unseres ersten Beitrags). Als Erstes fällt positiv auf, dass es sich tatsächlich um echtes Java handelt, ein paar Importe sind dabei, die verdächtig nach Android aussehen. Auffällig ist auch das Fehlen einer *main*-Funktion. Dafür wird von einer Klasse „Activity“ geerbt. Erben ist nie schlecht, also sehen wir uns ganz unbefangen, aber in gebotener Kürze an, was wir hier haben: Zwei Methoden von der ursprünglichen Klasse, die überschrieben werden. Die Namen deuten bereits an, dass wir in Android sehr stark ereignisorientiert arbeiten.

Die Methode *onCreate* wird einmalig beim Start unserer App bzw. beim Start der Acitvity aufgerufen. Besonders interessant ist hier *setContentView()*. Hier wird unser Layout, also unsere Oberfläche in der Form *R.layout.activity_start_schirm* aufgerufen und quasi mit der Activity verknüpft. Das geschieht wie der Aufruf vieler (eigentlich fast aller) Android-Elemente über „R“. R ist dabei nicht das Oberhaupt einer mysteriösen Geheimgesellschaft, sondern ganz einfach eine bzw. **die** zentrale Klasse, über die wir Zugriff auf praktisch alle Elemente in unserem Projekt erhalten. Glücklicherweise brauchen wir uns um die Einbindung der Elemente in R nicht selbst zu kümmern, diese undankbare Aufgabe wird uns freundlicherweise von der Entwicklungsumgebung abgenommen. Das alles geschieht automatisch,

Lesetipp

Lesen Sie auch Stephan Elters E-Book „Android-Entwicklung für Einsteiger – 20 000 Zeilen unter dem Meer“, erschienen bei entwickler.press: http://entwickler.de/press/Android-Entwicklung-fuer-Einsteiger-0

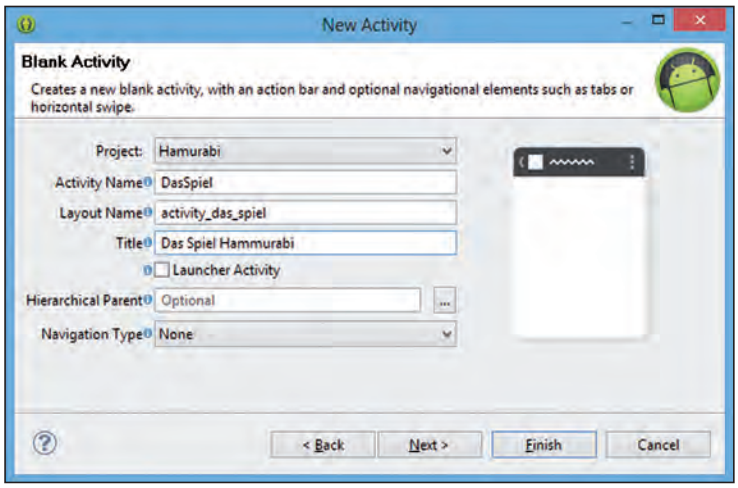


Abb. 1: Natürlich geht nichts kaputt, wenn man andere Bezeichnungen verwendet, man muss sie dann aber natürlich konsequent weiterverwenden

sobald ein neues Element erzeugt oder in das Projekt integriert wird. Woher wir wissen, wie die jeweiligen Elemente heißen und wie wir sie ansprechen, werden wir im Laufe der beiden Artikel an einigen Beispielen sehen.

Als Erstes sei gezeigt, was wir machen müssten, wenn wir ein ganz anderes Layout verwenden wollten, denn auch das geht nämlich problemlos. Man müsste sich nur über STRG + N mit dem passenden Wizard „Android XML Layout File“ ein neues Layout in Form einer XML anlegen, auf das man dann alternativ verweist. Der Aufruf im Quellcode wäre dann bis auf den Namen gleich: *setContentView(R.layout.mein_ganz_anderes_layout);*. Der (selbst vergebene) Name der entsprechenden XML-Datei wäre in diesem Falle *mein_ganz_anderes_layout.xml*. Und sinnvollerweise liegt diese Datei in dem Ordner *res/layout*. Jeder darf jetzt einmal raten, woher jetzt der Name *R.layout.mein_ganz_anderes_layout* herrührt und kann sicher auch noch die Frage beantworten, wie der Name unserer ursprünglichen, ersten XML lautet und in welchem Ordner man sie finden kann.

Tipp

Ein wichtiger Hinweis an dieser Stelle zu „R“. Ein Aufruf funktioniert natürlich nur, wenn die Elemente auch tatsächlich exakt so benannt sind. Haben Sie beispielsweise Ihr Layout anders benannt, vielleicht ganz klassisch *activity_main.xml*, dann wird ein Aufruf natürlich nicht mit einem anderen Namen funktionieren. So banal das klingen mag: Dieses Problem der Benennung tritt gerade bei den ersten Versuchen mit Android sehr gerne auf, wenn Teile von Lösungen aus dem Internet verwendet werden, die aber eben meist andere Bezeichnungen verwenden. Ein Aufruf in der Form *R.id.seekBar3* kann nicht klappen, wenn man im eigenen Projekt noch kein entsprechendes Element hat oder es möglicherweise gar nicht in den eigenen Kontext passt. Ein Element vom Typ Seekbar ist eben kein Bild und ein Text ist eben ein Text und noch kein Button. Übrigens, falls es nach der Übernahme von fremdem Quellcode plötzlich Probleme gibt, ist auch ein vorher nicht aufgefallener Import in der Art von *import.irgendetwas.R* (oder ähnlich) ein verdächtiger Kandidat, um einmal testweise auskommentiert zu werden.

Die zweite Methode unserer Klasse, *onCreateOptionsMenu*, ist an dieser Stelle uninteressant für uns. Wie der Name der Funktion andeutet, bezieht sich die Funktion auf ein Menü, das hier aber noch leer bleibt. Das gehört zu der Gewürzmischung, die wir im dritten und letzten Teil unserer 5-Minuten-App noch hinzufügen werden.

Ein Toast zur 5-Minuten-App

Kein Beispiel für die Android-Entwicklung kann ohne das obligatorische Toast vollständig sein, erst recht nicht unsere 5-Minuten-App. Ein Toast ist eine kurze Meldung auf dem Schirm, die keine weitere Interaktion zulässt und nach einem kurzen Moment wieder verschwindet. Wer würde das bei der Bezeichnung Toast nicht auch genauso vermuten?

Sie müssen dazu nicht einmal ein neues Objekt erschaffen, sondern können Ihr Toast direkt verwenden. Wir übergeben mit *getApplicationContext()* den Context, so weiß das Toast quasi, wo es hingehört. Schließlich soll es zur 5-Minuten-App auch am richtigen Tisch serviert werden. Ein Application Context sollte dem einen oder anderen nicht ganz unbekannt sein, durchaus vergleichbare Konzepte gibt es ja beispielsweise auch in Spring. Wir geben dem Toast noch einen Text mit und bestimmen, dass er etwas länger dargestellt werden soll, *.LENGTH_SHORT* wäre die kürzere Variante. Um ein Toast verwenden zu können, muss natürlich noch ein Import von *android.widget.Toast* vorgenommen werden (Listing 1).

Der verwendete Text sollte bereits einen Hinweis darauf geben, was wir als Nächstes programmieren wollen und was als Nächstes im Spiel zu tun ist: Wir machen das Bild anklickbar und rufen damit eine zweite Activity auf, die das eigentliche Spiel darstellt. Bevor wir das aber hier in unserer Activity umsetzen, sollten wir aber erst noch die aufzurufende, zweite Activity tatsächlich zur Verfügung haben, und werden diese in einer einfachen Form anlegen.

Listing 1

```
public class StartSchirm extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_start_schirm);

    String dieInfo = "Hallo\nBitte auf das Bild tippen";
    Toast.makeText(getApplicationContext(), dieInfo,
                                          Toast.LENGTH_LONG).show();

  }

  // ...aus Platzgründen gekürzt, nur deshalb, wirklich
}
```

Butter bei die Fische, oder eine zweite Activity

Um doch ohne eher lästige Fehlerhinweise in Eclipse weiterarbeiten zu können, benötigen wir eine tatsächlich bereits vorhandene Activity, die wir uns als Nächstes anlegen. Wir sorgen mit STRG + N wieder für etwas Magie und wählen unter Android den Wizard für eine „Android Activity“. Wie gehabt lassen wir uns wieder eine „Blank Activity“ erstellen, die wir dann noch entsprechend benennen, wir nennen sie einfach „DasSpiel“ (**Abb. 1**).

Egal, mit welcher Bezeichnung, wir klicken uns noch weiter bis zum Finish. Im letzten Schritt fällt dem aufmerksamen Beobachter ins Auge, dass Eclipse unter anderem verschiedene Eintragungen in die zentrale Datei *AndroidManifest.xml* ankündigt. Als kleine Wiederholung und Erinnerung aus dem ersten Artikel: In dieser zentralen Datei im Stamm des Projekts werden alle Activities, Informationen über die App wie ihr Name, die Versionsnummer oder auch benötigte Berechtigungen eingetragen, quasi der Ausweis, die Papiere unserer App.

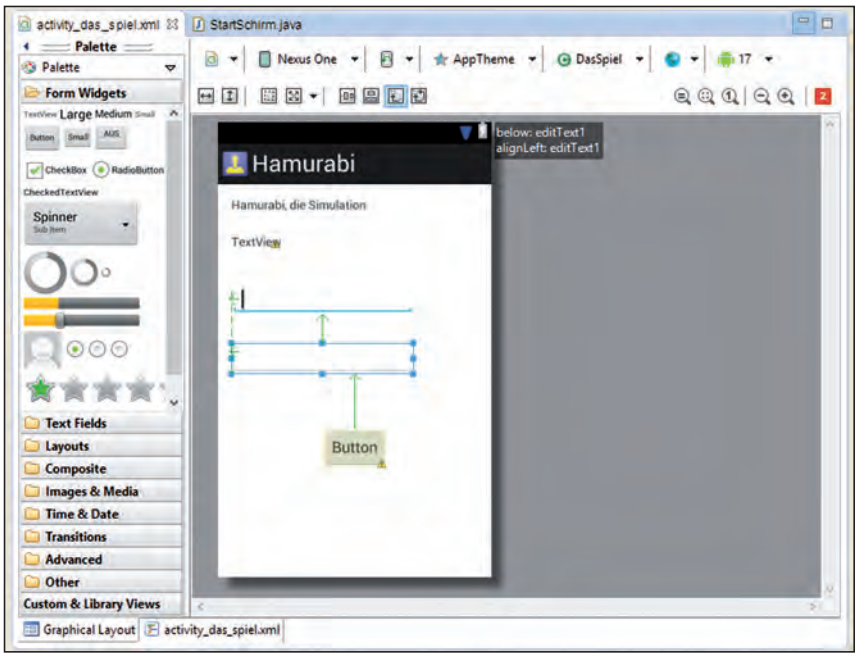


Abb. 2: So ähnlich sollte die neue Oberfläche aussehen, basteln und ausprobieren ist erlaubt, wird aber bitte nicht auf die fünf Minuten unserer 5-Minuten-App angerechnet

Zurück zu unserer neuen Activity, die auch als Blank Activity bereits einen einfachen Text auf der Oberfläche hat. Auch hier verweist diese *TextView* tatsächlich auf einen dahinterliegenden Text *@string/hello_world*. Wir erinnern uns, dass in einer Textdatei *strings.xml* in unserer Projektstruktur unter *res/values* ein entsprechender Eintrag existiert. Ändern wir dort den Text, ändert er sich an allen aufgerufenen Stellen – genau diese Änderung hatten wir ja bereits gemacht, und deshalb haben wir hier auch schon wie von Geisterhand den Text „Hamurabi, die Simulation“ und kein schnödes „Hello world“ mehr.

Da wir schon einmal hier sind, legen wir ein paar Views an, die wir im Folgenden für unser Spiel benötigen werden. Eine weitere TextView, in der wir den jährlichen Bericht des Spiels ausgeben können, zwei Felder, um dort Eingaben für die zu verteilende Nahrung und das Korn, das ausgesät werden soll, machen zu können. Und einen Button, um das Jahr abzuschließen, brauchen wir auch noch. Die Möglichkeit, Land zu kaufen oder zu verkaufen, lassen wir erst einmal unter den Tisch fallen. Das ist eine kleine Fleißaufgabe, die Sie selbst angehen dürfen.

In der grafischen Ansicht unserer Activity (die *activity_das_spiel.xml*, zu finden im Projekt unter *res/layout*) ziehen wir uns links aus der Palette unter Form Widgets eine TextView in die Oberfläche, unter Text Fields zwei Mal Eingabefelder für Zahlen (etwas weiter unten zu finden). Und zuletzt ziehen wir noch einen Button in unsere Oberfläche, auch zu finden unter Form Widgets. Damit ist unsere Oberfläche fertig, zumindest fürs Erste, denn auch an Aussehen und an den Texten kann in einer ruhigen Stunde gerne noch etwas gefeilt werden (**Abb. 2**).

Aber auch mit den neuen Views nützt uns unsere neue Activity alleine noch nicht viel. Nicht nur, dass sie noch keine Funktionen hat, das werden wir gleich noch ändern, nein, bei einem Aufruf der App bekommen wir diese Activity nicht einmal zu Gesicht. Wir müssen also noch dafür sorgen, dass unsere zweite Activity auch von der ersten Activity aus aufgerufen werden kann. Aber kein Problem, das ist einfacher, als man vielleicht vermuten würde.

Mehr als nur gute Absichten: Intents

Intents sind ein weiteres wichtiges Konzept in der Android-Entwicklung, und auch hier ist die Namensfindung weder schön, noch unbedingt selbsterklärend. Ein Intent ist die „Absicht“ (oder vielleicht der Wunsch?) einer Activity, etwas vom System erledigen zu lassen. So wie der Ruf nach dem Kellner, etwas Salz zu bringen. Das kann abstrakt die Absicht sein, dass ein Bild mit einer beliebigen, aber passenden App geöffnet wird, oder aber ganz konkret die Absicht, dass eine ganz bestimmte App bzw. eine ganz bestimmte Activity gestartet wird. Im ersten Fall wird dem jeweiligen System selbst überlassen, mit welcher App eine Aufgabe (Intent) ausgeführt wird (beispielsweise muss es ja gar nicht klar sein, welche App zum Öffnen eines Bilds installiert ist). Im anderen konkreten Fall wird eine App oder eine Activity direkt angesprochen. Wir öffnen also noch einmal den Quellcode unserer ersten Activity und ergänzen die Methode *onCreate* um folgenden Code, die dafür notwendigen Im-

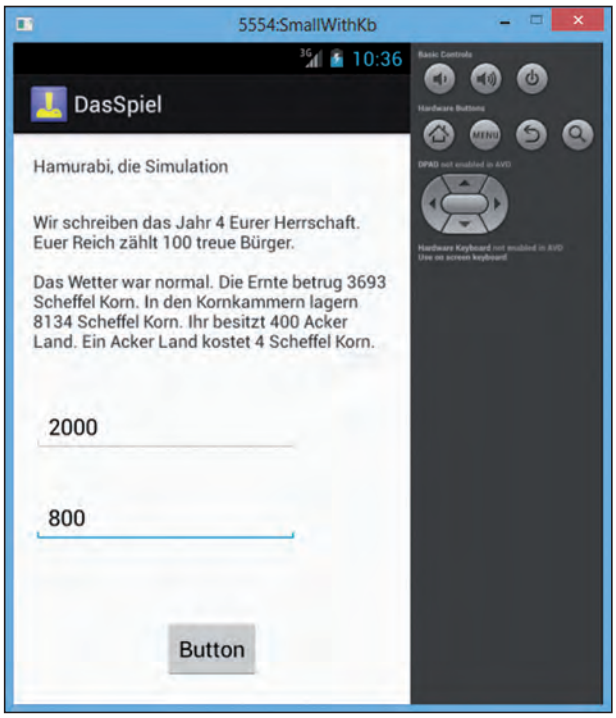


Abb. 3: Das Spiel ist fertig, zumindest in seiner allerersten, sehr einfachen Version – herzlichen Glückwunsch!

porte von *android.content.Intent*, *android.view.View* und *android.widget.ImageView* überlassen wir getrost Eclipse (Listing 2).

Was hier passiert, ist einfacher als es auf den ersten Blick vielleicht aussehen mag. Um mit einer View (einem Element der Oberfläche) arbeiten zu können, brauchen wir (natürlich) ein passendes Objekt, das für diese View steht. Das geschieht hier konkret mit dem Objekt „meinBild", dem wir das tatsächliche Element, unser Bild, über dessen ID zuordnen.

Jetzt müssen wir nur noch dafür sorgen, dass unser neues Objekt über einen Listener klickbar wird. Und

Listing 2

```
// Um mit dem Bild arbeiten zu können benötigen
// wir ein passendes Objekt vom Typ ImageView
ImageView meinBild = (ImageView) findViewById(R.id.imageView1);

// Jetzt machen wir unser neues Objekt klickbar...
meinBild.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {

    // ...und legen hier fest, was bei einem
    // Klick passiert: Zeit für unseren Intent
    Intent meinErsterIntent = new Intent(getApplicationContext(),
                                                    DasSpiel.class);
    startActivity(meinErsterIntent);
    finish();
  }
});
```

dann legen wir noch fest, was bei einem Klick passiert. Hier arbeiten wir zum ersten Mal mit einem Intent, der „Absicht" unsere neue Activity (in Form unserer Klasse *DasSpiel.class*) zu starten. Ach ja: Und weil wir nicht wollen, dass unser Startschirm bei einem „Zurück" des Users dann später noch einmal erscheint, geben wir unserer ersten Activity mit *finish* den Todesstoß. Das mag nicht schön sein für die Activity, aber darauf wollen wir keine Rücksicht nehmen. Der normale Weg wäre es sonst, sich über den Zurück-Button von Android von Acitvity zu Activity zurückzuhangeln, was hier ja sicher nicht gewollt ist. Damit verlassen wir unsere *StartSchirm.java* und unsere erste Activity, um uns endlich auf das eigentliche Spiel zu konzentrieren.

Die Spiele mögen beginnen

Zum ersten Mal kommt unsere Klasse *Hamurabi* zum Zuge, die Sie sich herunterladen und in das eigene Projekt integrieren können (Sie finden den Link auf www.javamagazin.de bei den Infos zu dieser Ausgabe bzw. auch unter www.punktuelles-im-web.net).

Dann geht es auch schon recht schnell mit dem eigentlichen Spiel los. Wir arbeiten jetzt im Quellcode unserer zweiten Activity weiter, in der Datei *DasSpiel.java*. Von der Klasse *Hamurabi* muss dort ein Objekt erzeugt werden. Das Spiel wird damit sofort gestartet und befindet sich im Jahr 1 der hoffentlich glorreichen Herrschaft des Spielers. Mit der Methode *getBericht()* kann der aktuelle Status des eigenen Reichs jederzeit als ein String abgefragt werden. Eine neue Runde bzw. neues Jahr startet die Methode *zug(int Landkauf, int Landverkauf, int Nahrung, int Saat)*. In unserem Beispiel wollen wir aus Platzgründen auf den Kauf und Verkauf von Land verzichten und übergeben hier deshalb nur jeweils den Wert 0 (deshalb haben wir bisher auch keine Eingabemöglichkeit dafür vorgesehen, aber das kriegen Sie sicherlich inzwischen selbst hin). Der dritte Wert stellt das Korn dar, das als Nahrung an das Volk verteilt wird. Jeder Bürger benötigt 20 Scheffel Korn. Der letzte Wert stellt die Menge an Korn dar, die für die Aussaat vorgesehen ist. Auf jedem Stück Land können 2 Scheffel Korn ausgesät werden und ein Bürger kann sagenhafte 10 Acker Land bewirtschaften.

Den jährlichen Bericht, den Sie aus der Methode *meinSpiel.erstelleBericht()* als String erhalten, könnten Sie über ein Toast ausgeben, vorausgesetzt Sie können sehr, sehr schnell lesen. Viel sinnvoller ist es aber, die Ausgabe in eine TextView zu machen. Zufälligerweise hatten wir ja weiter oben genau so eine TextView in unserem Layout angelegt, die wir jetzt dafür verwenden können. Und genauso zufällig hatten wir ja einen Button in unserer Oberfläche angelegt, den wir verwenden können, um bei einem Klick darauf ein neues Jahr zu starten.

Wir erschaffen uns also die notwendigen Objekte, um auf unsere Views (die Elemente der Oberfläche) zugreifen zu können und natürlich auch, um ein Objekt unserer Klasse *Hamurabi* zur Verfügung zu haben. Bei allen notwendigen Importen lassen wir uns wieder von Eclipse helfen:

```
public class DasSpiel extends Activity {

  TextView meinBericht;
  EditText wertA;
  EditText wertB;
  Button berichteButton;
  Hamurabi meinSpiel = new Hamurabi();
```

Der Rest der Geschichte ist schnell erzählt bzw. der Becher unserer 5-Minuten-App ist schnell umgerührt. Wir arbeiten hier ausschließlich in unserer Methode *onCreate()*. Das mag sicher nicht der Normalfall sein, für unser Beispiel ist es aber ausreichend und wir müssen schließlich auch daran denken, dass schon eine ganze Menge unserer fünf Minuten verstrichen sind.

Gehen wir also rasch weiter und fügen noch weitere Nudeln zur Activity bzw. noch etwas mehr Java-Code zu unserer Methode *onCreate* (Listing 3).

Was ist hier passiert? Unsere TextView existiert bisher nur als leeres Objekt ohne Inhalt oder weiteren Bezug. Mit der ersten Zuweisung zu *meinBericht* verknüpfen wir unser leeres Objekt mit der View im Layout und befüllen die TextView im zweiten Schritt mit dem Inhalt, den wir als String von unserem Objekt Hamurabi erhalten – wir schreiben den Spielbericht des aktuellen Jahres in das Textfeld. Ähnliches machen wir jetzt noch mit unserem Button, den wir sogar noch klickbar machen. Das kennen wir inzwischen ja schon in ähnlicher Form von dem Bild unserer ersten Activity. Nur rufen wir jetzt bei einem Klick auf den Button keine andere Activity über einen Intent auf, sondern lesen den Inhalt der beiden Formularfelder aus, konvertieren die erhaltenen Strings in Integer und übergeben dann alles in einem Spielzug unserer Klasse *Hamurabi*. Schließlich aktualisieren wir noch unser Textfeld mit dem jeweils aktuellen Bericht. Bleibt nur noch die Frage, woher wir diese schicken Bezeichnungen wie *R.id.button1* oder *R.id.editText1* bekommen? Ganz einfach, in der Layoutansicht schauen wir rechts unter Properties der Elemente (Views) nach der ID. Ein Klick auf den kleinen grauen Button mit den drei Punkten enthüllt uns den Namen unter Java – und ändern könnten wir die Namen hier auch gleich noch (Listing 4).

Und da der Listener bei jedem neuen Klick auf den Button reagiert, spielen wir uns so von Runde zu Runde bzw. von Jahr zu Jahr: Wir lesen die Werte der Formularfelder aus, konvertieren die Werte zu Integer, machen einen Zug und aktualisieren den Bericht! Unsere Klasse *Hamurabi* ist dabei so freundlich, selbst zu merken, wann das Spiel beendet ist, und gibt dann über den Bericht einen passenden Hinweis aus. Auf weitere Eingaben reagiert die Klasse dann auch nicht mehr.

„Hasta la vista, Suppe“

Die Werte in den Eingabefeldern bleiben in jeder Runde unverändert stehen, zumindest solange der Spieler sie nicht selbst ändert. Wer möchte, kann die Felder natürlich auch immer wieder mit einer Anweisung in der Art *wertA.setText(" ");* leeren oder, schöner ausgedrückt, mit keinem Zeichen füllen, was natürlich auf das Gleiche hinausläuft. Spätestens hier merkt der aufmerksame Entwickler, dass es an der Zeit wäre, zumindest leere Eingaben abzufangen. Wer das richtige Eingabefeld in sein Layout gezogen hat, der verhindert zwar von vornherein, dass andere Zeichen als reine Zahlen eingegeben werden können, ein leeres Feld ist damit aber immer noch möglich und sorgt für einen Abbruch.

Ich möchte mich auch diesmal wieder für Ihre Aufmerksamkeit bedanken und hoffe, dass Sie einiges lernen konnten und wir uns in der nächsten Ausgabe wiedertreffen, wenn die letzte, noch spannendere und noch buntere Ausgabe unserer „5-Minuten-App“ folgt, in der wir die schnöden und langweiligen Eingabefelder durch schicke Schieberegler ersetzen werden, das Menü befüllen und am Spielende dann nicht mehr benötigte Felder ausblenden lassen, sowie noch das eine oder andere Gimmick ausprobieren werden. Es bleibt spannend, versprochen!

Stephan Elter arbeitet als Entwickler für den Bereich Internet bei der NORDSEE-ZEITUNG GmbH in Bremerhaven. Neben PHP und Java sind seit dem Palm III mobile Geräte seine Leidenschaft. Er ist auch Autor beim PHP Magazin. Neben seiner eigenen Webseite schreibt er häufig als Gastautor für andere Blogs.

Listing 3

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_das_spiel);

  meinBericht = (TextView) findViewById(R.id.textView1);
  meinBericht.setText(meinSpiel.getBericht());
```

Listing 4

```
  berichteButton = (Button) this.findViewById(R.id.button1);
  berichteButton.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {

      wertA = (EditText) findViewById(R.id.editText1);
      wertB = (EditText) findViewById(R.id.editText2);

      int nahrung = Integer.parseInt(wertA.getText().toString());
      int saat = Integer.parseInt(wertB.getText().toString());

      meinSpiel.zug(0, 0, nahrung, saat);
      meinBericht.setText(meinSpiel.getBericht());
    }
  });

}
```

Vorschau auf die Ausgabe 11.2013

Continuous Delivery

Continuous Integration hat seinen Ursprung in der Lehre des Extreme Programming (XP) und ist mittlerweile fester Bestandteil unserer Softwareentwicklung. Continuous Delivery geht einen Schritt weiter und will die letzte Meile bis zur Produktion beschleunigen. Doch die Umsetzung dieser Softwarepipeline ist nicht trivial. Nächsten Monat beschäftigen wir uns daher mit wichtigen Architektur-Patterns für Continuous Delivery und dem Weg von Continuous Integration zu Continuous Delivery.

Aus redaktionellen Gründen können sich Themen kurzfristig ändern.

Die nächste Ausgabe erscheint am 9. Oktober 2013

Querschau

eclipse MAGAZIN — Ausgabe 5.2013 | www.eclipse-magazin.de

- **Hello Kepler! Alles Wissenswerte rund um das zehnte Simultaneous Release**
- **Test Drive: Einführung in Java 8 und Java Development Tools**
- **Umstieg heißt Umdenken: Migration auf Eclipse 4**

entwickler magazin — Ausgabe 5.2013 | www.entwickler-magazin.de

- **Go with the Flow: Code Completion in Java**
- **Ember.js: MVC im Client**
- **Internationalisierung: Global mit dem Google Web Toolkit**

PHPmagazin — Ausgabe 5.2013 | www.php-magazin.de

- **Grenzenlose Freiheit: Was Onlinehändler beachten müssen**
- **Follow the White Rabbit: JCR und Magnolia CMS**
- **Werkzeugkasten: Chrome Developer Tools**

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
Leitung Grafik & Produktion: Jens Mainz
Layout, Titel: Tobias Dorn, Flora Feher, Karolina Gaspar, Dominique Kalbassi, Laura Keßler, Nadja Kesser, Maria Rudi, Petra Rüth, Franziska Sponer

Autoren dieser Ausgabe:
Stephan Elter, Dirk Fauth, Christian Heinemann, Michael Hunger, Klaus Kreft, Heiner Kücker, Marcus Lagergren, Angelika Langer, Arne Limburg, Cornelius Moucha, Bernd Müller, Michael Müller, Jan Peuker, Dimitar Robev, Lars Röwekamp, Ferdinand Schneider, János Vona, Sebastian Weber, Wolfgang Weigend, Daniel Winter, Eberhard Wolff

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

Bildnachweis: Der Android Robot steht unter der Creative-Commons-Lizenz.

© Software & Support Media GmbH

Alle Rechte, auch für Übersetzungen, sind vorbehalten. Reproduktionen jeglicher Art (Fotokopie, Nachdruck, Mikrofilm oder Erfassung auf elektronischen Datenträgern) nur mit schriftlicher Genehmigung des Verlages. Jegliche Software auf der Begleit-DVD zum Heft unterliegt den Bestimmungen des jeweiligen Herstellers. Eine Haftung für die Richtigkeit der Veröffentlichungen kann trotz Prüfung durch die Redaktion vom Herausgeber nicht übernommen werden. Honorierte Artikel gehen in das Verfügungsrecht des Verlags über. Mit der Übergabe der Manuskripte und Abbildungen an den Verlag erteilt der Verfasser dem Herausgeber das Exklusivitätsrecht zur Veröffentlichung. Für unverlangt eingeschickte Manuskripte, Fotos und Abbildungen keine Gewähr. Java™ ist ein eingetragenes Warenzeichen von Oracle und/oder ihren Tochtergesellschaften.

Anzeige

Anzeige