Android360: Mobile Device Management – 5 Anbieter im Vergleich ▸93

S&S

JAVA Mag

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 4.2013

Java™ magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Play 2.0/2.1

Jetzt wird's ernst im Tutorial ▸40

OAuth 2.0

Implementierung mit Spring Security ▸56

JAX 2013 hier im Heft! ▸67

Node.js & vert.x

4 194586 709801 04

AngularJS und Java EE 6

Das dynamische Web ▸66

Continuous Integration

In großen Projekten ▸72

Sprach-erweiterung

Mixins mittels AOP oder Quellcodemodifikation ▸13

Alles Scripting oder was?

Da ist man nun seit so vielen Jahren Java-Entwickler, d. h. Spezialist für die beste Programmiersprache der Welt, und plötzlich mehren sich die Rufe nach polyglotter Programmierung. Haben wir nicht über die Jahre bewiesen, dass wir mit Java praktisch alles machen können, was die IT-Fantasie erdenken kann?

Dank zahlreicher Webframeworks haben wir eine ungeheure Auswahl an Architekturen, wenn es um Java-basierende Webanwendungen geht. Für die komplizierteren Fälle können wir zwischen Java EE und Spring wählen, und dank Android können wir auch Smartphone-Apps mit unserer geliebten Alleskönnersprache schreiben.

Und überhaupt – hatte es nicht immer geheißen: „Scripting is for kids“ und „Real men compile“?

Ola Bini, vermutlich einer der wichtigsten Vordenker moderner Programmiersprachen derzeit, hat einmal ein interessantes Modell entworfen. Jedes Softwaresystem solle sich demnach in drei Ebenen gliedern lassen. Die unterste Ebene bilde dabei der „Stable Layer“, typischerweise programmiert in Java (der Sprache) oder in einer anderen kompilierten Sprache wie etwa Scala. Hier werden die Fundamente einer Anwendung programmiert, die wichtigsten APIs, sozusagen der Kernel des Systems.

Darüber befindet sich der „Dynamic Layer“, typischerweise in einer (oder mehreren) dynamischen Sprache(n) geschrieben, wie etwa Groovy, JRuby, Clojure oder JavaScript. Hier wird, gemäß dem Modell, der überwiegende Teil der Geschäftslogik eines Systems entwickelt. Der Vorteil dynamischer Sprachen auf dieser Ebene: Dadurch, dass der Code nicht kompiliert wird, lassen sich Modifikationen „on the fly“ realisieren, was in einer Welt dramatisch beschleunigter Geschäftsprozesse und immer schnellerer Anpassungszyklen von großem Vorteil sein kann. Darüber hinaus erlauben Skriptsprachen typischerweise einen erheblich kompakteren Programmcode, was uns eine höhere Produktivität sowie eine geringere Fehleranfälligkeit verspricht.

On top finden wir den so genannten „Domain Layer“, und hier geht es ans geschäftliche Eingemachte: Hier werden Preise berechnet oder Daten ausgewertet, Zinssätze entwickelt oder Geschäftsregeln definiert und vieles andere mehr. Auf diesem Layer sollten wir den Fachabteilungen, denen unsere Systeme ja letztendlich dienen, größtmögliche Gestaltungsmöglichkeiten geben. Domänenspezifische Sprachen sind daher, laut Ola Bini, das Mittel der Wahl für diesen Bereich.

Gewiss, wir sollten diese Gedanken keineswegs als Dogma betrachten, so wie wir eigentlich nichts in der Welt der Softwarearchitektur dogmatisch verstehen sollten. Aber interessant sind Olas (gar nicht mal so neue) Ideen für eine Strukturierung polyglotter Systeme auf jeden Fall: Unterschiedliche Sphären erfordern unterschiedliche Antworten auf Probleme, und diese lassen sich am besten in zielgerichteten Sprachen formulieren. Aus diesem Grunde haben wir uns in der vorliegenden Ausgabe einigen Scripting-Technologien für die Java-Plattform zugewandt.

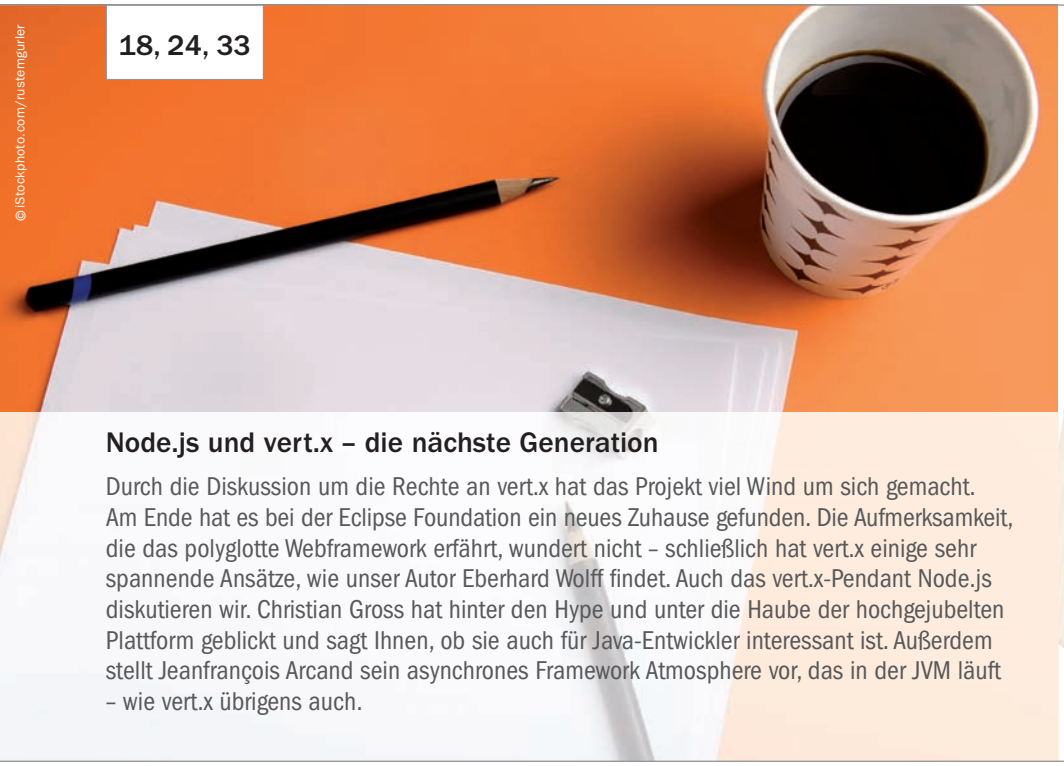
vert.x ist ein polyglottes Framework für die Java-Plattform und wird von Eberhard Wolff (Seite 18) vorgestellt. Node.js wurde von einer aktiven Community lange Zeit stark gehypt und stellt ein interessantes Konzept für JavaScript auf dem Server dar; ob es ein neues Tor zur Webarchitektur der Zukunft öffnet, prüft Christian Gross in seinem Artikel auf Seite 24. Und mit Atmosphere betrachten wir ein Framework, das mit verschiedenen Java-Technologien kompatibel für eine reibungslose Unterstützung von Protokollen wie WebSockets sorgt (Seite 33).

Halten wir‘s mit dem Fazit von Eberhard Wolff: Die Beschäftigung mit Scripting-Frameworks weitet auf jeden Fall den Horizont – zumal vert.x und Atmosphere dezidiert für die Java-Plattform gemacht sind. Und denken Sie bitte daran: Als Java-Enterprise-Entwickler haben Sie zu den Helden der 2000er-Jahre gezählt; aber die Welt verändert sich rapide, und die Softwareentwicklung wird ohne Zweifel mehrsprachiger. Die Plattform hat mittlerweile bewiesen, dass sie zu nachhaltiger Innovation fähig ist. Jetzt ist es an der Java-Community zu beweisen, dass sie trotz (oder gar wegen) ihrer Erfolge bis heute zu einem Umdenken bereit ist.

In diesem Sinne: Viel Spaß beim Lesen!

Ihr Sebastian Meyen, Chefredakteur

@smeyen



Node.js und vert.x – die nächste Generation

Durch die Diskussion um die Rechte an vert.x hat das Projekt viel Wind um sich gemacht. Am Ende hat es bei der Eclipse Foundation ein neues Zuhause gefunden. Die Aufmerksamkeit, die das polyglotte Webframework erfährt, wundert nicht – schließlich hat vert.x einige sehr spannende Ansätze, wie unser Autor Eberhard Wolff findet. Auch das vert.x-Pendant Node.js diskutieren wir. Christian Gross hat hinter den Hype und unter die Haube der hochgejubelten Plattform geblickt und sagt Ihnen, ob sie auch für Java-Entwickler interessant ist. Außerdem stellt Jeanfrançois Arcand sein asynchrones Framework Atmosphere vor, das in der JVM läuft – wie vert.x übrigens auch.

Play – jetzt wird's ernst!

Play, 2. Akt: Nachdem wir uns im ersten Teil unseres Tutorials mit den Grundlagen einer Play-Applikation beschäftigt haben, geht es nun weiter mit Fixtures, Bootstrapping, Forms und deren Validierung und Security. Auch die frisch erschienene Version 2.1 findet Erwähnung.

Magazin

Java Core

Titelthema

Tutorial

Web

Enterprise

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

AngularJS und Java EE 6

Vom „Skript-Kid“ bis zum eingefleischten Java-Entwickler: Wer moderne Webanwendungen schreibt, kommt ohne JavaScript nicht aus. Ist es nicht besser, im Frontend ganz auf Java zu verzichten. Eine kleine CRUD-Anwendung zeigt, dass das geht: Im Backend setzen wir dazu komplett auf Java EE 6, während wir im Frontend Twitter Bootstrap und AngularJS verwenden.

Teamplayer Git

Softwareanwendungen können heutzutage nicht ohne Versionsverwaltung leben. Beim Großteil der Anwendungen werden dafür Systeme wie SVN oder CVS verwendet. Aber auch Git ist wegen seiner Flexibilität und vieler anderer Vorteile inzwischen sehr populär geworden. Besonders interessant wird der Einsatz von Git in der Verknüpfung mit den Features von SVN oder CVS in einem Projekt.

Mobile Device Management

Immer mehr Firmen entscheiden sich dafür, mobile Geräte in den Arbeitsalltag einzubinden. Mit einem Mobile-Device-Management-System lassen sich diese Geräte sicher und zuverlässig verwalten. Auf dem Markt bieten zahlreiche Hersteller Lösungen an, darunter AirWatch, Symantec und MobileIron aus den USA. Wir vergleichen sie mit den europäischen Produkten Bunya und datomo.

Retrospektive

Agile

Android360

Standards

Frisch von JAXenter

News, Kommentare und Interviews von www.jaxenter.de

Jelastic-Statistik zeigt die beliebtesten Datenbanken, Anwendungsserver und Java-Versionen

Welche Technologien stehen in der Cloud-Landschaft derzeit hoch im Kurs, welche Datenbanken und welche Anwendungsserver werden am häufigsten benutzt? Zumindest für die Cloud-Plattform Jelastic PaaS lässt sich diese Frage beantworten, denn jeden Monat veröffentlicht Jelastic Statistiken zur Nutzung und Beliebtheit der verschiedenen Technologien. Im Januar ergab sich dabei folgendes Bild:

Bei den Datenbanken hat die Beliebtheit von MySQL im Vergleich zum Dezember noch einmal zugenommen, sodass die Datenbank nun mit 61 Prozent unangefochten an der Spitze liegt. Den zweiten Platz belegt MariaDB mit 14 Prozent, PostgreSQL und MongoDB teilen sich mit jeweils 11 Prozent den dritten Platz und das Schlusslicht ist CouchDB mit 3 Prozent. In der Nutzung der Datenbanken sind jedoch große regionale Unterschiede zu erkennen. So ist PostgreSQL in Brasilien sehr beliebt, die Finnen nutzen überdurchschnittlich oft MariaDB und CouchDB. In Deutschland führt ebenfalls MySQL mit 54 Prozent, allerdings gefolgt von MongoDB mit 20 Prozent. MariaDB schafft es hier mit 18 Prozent nur auf den dritten Platz, gefolgt von PostgreSQL mit 7 Prozent und CouchDB mit einem Prozent.

Werfen wir nun einen Blick auf die beliebtesten Anwendungsserver. Mit 71 Prozent Anteil und einem 14-prozentigen Zuwachs im Vergleich zum Dezember hebt sich Tomcat 7 weiter von der Konkurrenz ab. Der Zuwachs erfolgte auf Kosten der anderen Angebote, denn außer dem Marktführer haben alle anderen Anteile verloren. Tomcat 6 belegt 15 Prozent, GlassFish 9 Prozent und Jetty 6 Prozent des Markts. In Finnland ist der Anteil von Tomcat 6 besonders hoch, und auch in Deutschland wird er mit 16 Prozent Marktanteil noch etwas häufiger genutzt als im weltweiten Durchschnitt. Der Anteil von Tomcat 7 ist mit 73 Prozent allerdings auch höher, GlassFish belegt 7 und Jetty 4 Prozent.

Abschließend hat sich Jelastic noch die in der eigenen Cloud verwendeten Java-Versionen angesehen. Groß war die Auswahl nicht, denn nur Java 7 sowie Java 6 finden Anwendung in Jelastic PaaS. 82 Prozent bevorzugen die neuere Version, in Deutschland sind es sogar ganze 84 Prozent.

Jelastic hat die gesamte Auswertung auf dem Unternehmensblog öffentlich gemacht. Sicher finden Sie dort noch die ein oder andere interessante Zahl.

▶ http://bit.ly/XLKaWE

Onlinepetition: Java-Installation bitte ohne „Crapware“

In der Java-Community wurde in den letzten Wochen des Öfteren kritisiert, dass der Windows Installer für die Java Runtime einen Dialog zur Installation einer Ask Toolbar enthält. Per Default ist der Haken in der Checkbox gesetzt: „Install the Ask Toolbar and make Ask my Default search provider.“ Cay Horstmann, Universitätsprofessor und geschätzter Java-Aktivist, bloggt nun über diese Unsitte, „Crapware“ von Oracle-Partnern mit Java-Updates in Verbindung zu bringen.

Schlechte Zeiten für Desktop-Java seien es derzeit, lauten seine Worte. Nicht nur wird Java von Hackern unter Dauerbeschuss genommen. Nicht nur schalten Betriebssysteme wie Browser Java per Default ab. Jetzt wird auch noch jeder Versuch Oracles, durch zeitnahe Updates Glaubwürdigkeit wiederherzustellen, dadurch unterminiert, dass lästige Zusatzsoftware zur Installation empfohlen wird.

„You don’t earn people’s trust by ‚recommending‘ to install stuff that they are likely to hate“, so Cay Horstmann, der auch deshalb allergisch gegen diesen „Ask“-Dialog ist, da seine Studenten bei der Installation von Java erfahrungsgemäß nicht jeden Wizard hinterfragen. So wird auch per Default der JavaFX-Installationswizard angeworfen. Entweder wird die Ask-Toolbar-Checkbox also übersehen und die Leiste ungewollt mitinstalliert. Oder JavaFX wird auf dieselbe Crapware-Stufe wie die Ask-Bar gestellt, und Studenten fragen: Ask und JavaFX kann man also weglassen, oder?

Horstmann hat jedenfalls keine guten Worte für diese Praxis übrig und fragt in die Runde, wer seiner Meinung ist: Wer auch die Ask Toolbar wieder verschwinden lassen will, kann sich an einer eigens initiierten Online-Petition beteiligen. Horstmann beendet seinen Blogeintrag mit der eindringlichen Bitte an Oracle: „Come on, Oracle. Tear down this toolbar!“

▶ http://chn.ge/YrkNuo

Ouya in den Startlöchern: Die Spielkonsole der Zukunft spricht Android

Es war einmal ein kleines Projekt, das im Juli 2012 versuchte, über Kickstarter genügend Kapital zur eigenen Finanzierung zu bekommen. Eine Spielkonsole sollte es werden, auf Android basierend und mit dem schönen Namen Ouya. Was dann jedoch geschah, überraschte die Entwickler selbst wohl am meisten: Das Projekt ging nahezu durch die Decke, das anvisierte Finanzierungsziel wurde innerhalb von nur acht Stunden erreicht, und auch daraufhin wollte das Geld einfach nicht aufhören zu fließen, sodass mittlerweile über 8,5 Millionen US-Dollar Starthilfe eingegangen sind.

Alle Fans, die sich mit kleinen oder großen Beträgen am Projekt Ouya beteiligt haben, warten seitdem sehnlich darauf, die Spielkonsole endlich in ihren Händen halten zu dürfen. Dieses Warten hat nun schon bald ein Ende, denn im kommenden März wird die Konsole an die Kickstarter-Förderer ausgeliefert, im Juni kommt sie dann auch in den Handel. Für den Marktstart haben sich die Entwickler namhafte Partner sichern können: Sowohl Amazon und Gamestop als auch Target und Best Buy werden die Konsole im Sortiment führen. Bei allen Händlern soll sie inklusive einem Controller 99,99 US-Dollar kosten, einen zusätzlichen Controller mit Touch Input gibt es für 49,99 US-Dollar. Gemäß der plattformübergreifenden Idee hinter der Ouya-Konsole kann dieser übrigens auch für andere Geräte, wie beispielsweise Apple TV, verwendet werden.

Doch das Wichtigste an einer Spielkonsole sind natürlich weder der Preis noch die zugehörige Hardware, sondern die darauf laufenden Spiele. Diese basieren auf Android, und jedes soll zumindest teilweise kostenlos sein. Inzwischen gibt es mehrere Community-Sites mit Listen von bis zu zweihundert Titeln, die allesamt auf die Ouya kommen sollen, viele davon sogar exklusiv. Um sie verfügbar zu machen, müssen Entwickler ihre Spiele in den mittlerweile eröffneten Ouya Store hochladen. Wer jetzt aber denkt, dass das Spieleangebot für die Ouya-Konsole ausschließlich aus von Hobby-Entwicklern erstellten Spielereien bestehe, liegt weit daneben. So haben sich eine Reihe bekannter Spieleentwickler dazu bereit erklärt, ihre Produkte auf die freie Konsole zu bringen. Final Fantasy 3 von Square Enix gehört dazu, und dank dem Engagement von Namco Bandai möglicherweise auch bald Games wie Tekken oder Soul Calibur.

Ob die offene Android-Konsole tatsächlich Erfolg haben wird, lässt sich natürlich nur schwer sagen. Immerhin haben es Spielkonsolen derzeit allgemein schwer, da sie nach und nach von den handlicheren Smartphones und Tablets abgelöst werden. Aber auch wer weiterhin auf ausgefeiltere, grafisch und Gameplay-technisch ausgereifte Spiele setzt, muss nicht unbedingt zur Ouya greifen. Schließlich stehen ebenfalls neue Versionen der Playstation und der Xbox in den Startlöchern.

▶ http://on.wsj.com/WWxGx8

Alles besser mit der 2.0: Apache Log4j 2.0-beta4 erschienen

Version 2 des Logging-Frameworks Apache Log4j soll alles besser machen. Gegenüber ihrem Vorgänger soll sie deutliche Vorteile bieten, beispielsweise, was die Logback-Architektur angeht. Doch noch ist es nicht soweit: Mit der vierten Betaversion ist nun bereits das sechste Pre-Release von Log4j 2.0 erschienen.

Auch die vierte Beta tut wieder ihr Bestes, um das Framework auf die Final-Version vorzubereiten. So wurden über zwanzig Bugs gefixt und einige neue Features eingeführt. Beispielsweise gibt es jetzt einen Log4j-2.0-zu-SLF4J-Adapter. Man hat Flume-Appender-Beispiele, *MessageFormatMessage* und *FormattedMessage* hinzugefügt. Im Console Appender gibt es ab sofort ein *Follow*-Attribut, und wenn die Socket-Verbindung fehlschlägt, können die geworfenen Ausnahmen fortan durch einen Unit Test bestätigt werden.

Wie immer ist Feedback aus der Community sehr willkommen. Um die aktuelle Version zu testen, wird allerdings mindestens Java 5 benötigt. Das log4j-1.2-API garantiert die Kompatibilität zu Log4j 1.x.

▶ http://bit.ly/WkfwnZ

Play 2.1 erschienen: Scala 2.10 an Bord

Die Version 2.1 des Webframeworks Play wurde freigegeben. Typesafe-Ingenieur James Roper verkündet die frohe Botschaft in der Google-Group und verweist auf die Highlights des Releases. Hervorzuheben ist zunächst einmal die Anbindung an das neue Scala 2.10. Das Runtime API für Play ermöglicht die Nutzung aller Scala-2.10-Features. Gleichzeitig wurde die Abhängigkeit zwischen einer zur Runtime benutzten Scala-Version und der vom Buildsystem (sbt) benutzten Scala-Version aufgebrochen, sodass es nun möglich ist, Play auch mit experimentellen Branches der Scala-Sprache zu benutzen.

Play selbst wurde modularisiert, d. h. in verschiedene Unterprojekte aufgeteilt, sodass jeder seine eigene Konfiguration der benötigten Komponenten zusammenstellen kann. Zur Verfügung stehen jdbc, anorm, javaCore, javaJdbc, javaEbean, javaJpa und filters. Der Play Core selbst konnte so auf ein minimales Set von Abhängigkeiten reduziert werden und eignet sich als ultraschlanker asynchroner HTTP-Server.

Migriert wurde auf nebenläufige Scala Futures. Neu ist auch ein Scala-JSON-API und ein Filter-API mit eingebauter CSRF Protection. Insgesamt listet der Issue Tracker 182 Bugfixes und Feature-Erweiterungen auf. Wer mehr über das neue Play-Release wissen möchte, schaue sich am besten die Highlightsseite auf http://www.playframework.com an.

▶ http://www.playframework.com/documentation/2.1.0/Highlights

Hat IcedTea eine Zukunft?

Im Mai 2007 veröffentlichte Sun Microsystems mit dem OpenJDK eine vermeintlich freie Open-Source-Implementierung der Java-Standard-Edition. Doch komplett frei war das Open Java Development Kit zu diesem Zeitpunkt noch nicht. Tatsächlich blieben ganze vier Prozent der Class Library proprietär, und zur Erstellung nutzte man Werkzeuge, ohne auf deren freie Zugänglichkeit zu achten. All dies rief Red Hat auf den Plan.

Bereits im Sommer 2007 entwickelte das Unternehmen aus North Carolina mit IcedTea eine Version des OpenJDK, die nicht nur selbst frei war, sondern auch gänzlich ohne proprietäre Software erstellt wurde. Man bemühte sich, nur in der Distribution selbst vorhandene Pakete sowie Werkzeuge zu nutzen, wenn möglich auch ohne Netzzugang. Hierfür wurden einige Plugs durch Teile des GNU Classpath ersetzt.

Bot IcedTea anfangs eine wirklich freie Alternative zum nur vorgeblich freien OpenJDK, hat sich seitdem einiges verändert. Die proprietären Plugs des OpenJDK gehören nun schon lange der Vergangenheit an, genauso wie die anfänglichen Tarballs, die durch ein *Mercurial*-Verzeichnis ersetzt wurden. Auch im OpenJDK gibt es mittlerweile Arbeitsgruppen und Projekte, die sich Problemen im Development Kit annehmen. Und so stellt sich zum heutigen Datum die Frage: Ist es überhaupt noch sinnvoll, IcedTea weiterzuführen? In seinem Blog hat der Entwickler Andrew John Hughes drei Probleme zusammengestellt, die beim Einstellen des IcedTea-Projekts zum Tragen kommen würden:

1. IcedTea verfügt über eine große Anzahl an lokalen Patches, ständig kommen neue hinzu. All diese in OpenJDK zu integrieren, ist eine Menge Arbeit und geht nur langsam voran.
2. Mittlerweile gibt es einige Sub-Projekte, die von IcedTea abhängig sind, beispielsweise einen eigenen Fork der Jtreg Testsuite, einen PulseAudio Soundtreiber oder ein Plug-in sowie eine Web-Start-Implementierung, die im OpenJDK nicht verfügbar sind.
3. Zur Arbeit am OpenJDK muss man bei Oracle ein so genanntes Contributor Agreement unterzeichnen. Das schreckt viele Entwickler ab, die stattdessen lieber bei IcedTea bleiben und dort ihre Patches bauen.

Nichtsdestotrotz steht die Frage nach der Zukunft des Projekts IcedTea weiter im Raum. Hughes sieht zum einen die Möglichkeit, primär mit OpenJDK zu arbeiten. Zum anderen wäre aber auch ein IcedTea-Neustart direkt vom OpenJDK aus eine Option. Seiner Meinung nach muss IcedTea definitiv weitergeführt werden, allein schon der freien Java Virtual Machines CACAO und JamVM wegen.

▶ http://bit.ly/12yqADl

Links und Downloadempfehlungen zu den Artikeln im Heft

- Liste grafischer Git-Frontends und Tools: https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools
- Umfrage zu Versionierungssystemen im Februar 2010: http://martinfowler.com/bliki/VcsSurvey.html
- Atmosphere-Framework auf GitHub: http://github.com/Atmosphere/atmosphere
- Wicket 6 Bean Validations auf GitHub: https://github.com/dilgerma/wicket6-bean-validations
- Zentrales vert.x-Modul-Repository: http://github.com/vert-x/vertx-mods
- Tonr und Sparklr, OAuth-2.0-Beispielapplikationen von Spring Security: https://github.com/SpringSource/spring-security-oauth/tree/master/samples/oauth
- Beispielanwendung: Web-Apps mit Angular.js und JavaEE: https://github.com/nilspreusker/crm-demo

O/R-Mapping in Android:

- ORMLite: http://ormlite.com
- greenDAO: http://greendao-orm.com/
- stORM: https://code.google.com/p/storm-gen/

Anzeige

Autor des Monats

Martin Dilger ist freiberuflicher Software-Consultant und Wicket-Trainer. Er beschäftigt sich seit Jahren intensiv mit der Entwicklung von Webanwendungen im Enterprise-Java-Umfeld.
Kontakt: martin@effectivetrainings.de

Wie bist du zur Softwareentwicklung gekommen?

Schon während meiner Schulzeit habe ich erste Programme geschrieben. Damals noch mit Pascal, da mich leider niemand darauf hingewiesen hat, dass das schon zu dieser Zeit völlig veraltet war. Danach kam der klassische Weg über das Informatik-Studium und unterschiedliche Jobs in interessanten Projekten. Schlussendlich bin ich dabei geblieben, weil es einfach richtig Spaß macht.

Was ist für dich der schönste Aspekt in der Softwareentwicklung?

Softwareentwicklung fängt erst an, richtig Spaß zu machen, wenn man es ernsthaft betreibt. Eine schöne, einfache Lösung für ein richtig kompliziertes Problem zu finden macht richtig Freude.

Was ist für dich ein weniger schöner Aspekt?

Wenn diese einfache Lösung nicht auffindbar ist.

Wie und wann bist du auf Java gestoßen?

Zum ersten Mal auf Java gestoßen bin ich 2001. Java war außerdem die Lehrsprache während des Studiums. Rückblickend die richtige Entscheidung, da man im Studium normalerweise sehr viel Zeit für Experimente hat und dadurch eine Sprache sehr genau lernt. Seit dem Studium mache ich bis auf einige Seitensprünge in die PHP-Welt fast ausschließlich Java.

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

Ich würde alle verfügbaren Ressourcen auf die Entwicklung und Fertigstellung vom Projekt Jigsaw verwenden.

Was ist zurzeit dein Lieblingsbuch?

Practice Perfect von Doug Lemov, Erica Woolway und Katie Yezzi. Das Buch inspiriert mich, mich ständig weiterzuentwickeln, sowohl im Beruf als auch für Hobbys und Privates.

Was machst du in deinem anderen Leben?

In meinem anderen Leben bin ich verheiratet und habe eine wundervolle Frau. Die Zeit, die dann noch übrig bleibt, verbringe ich beim Joggen im Englischen Garten in München.

JUG-Kalender* Neues aus den User Groups

WER?	WAS?	WO?
JUG **Ostfalen**	04.03.2013 – RESTful Web Services	http://www.jug-ostfalen.de
JUG Schweiz	05.03.2013 – Java oder Scala? Scala und Java!	http://www.jug.ch
JUG **Darmstadt**	07.03.2013 – JavaFX	http://jugda.wordpress.com
JUG **Hannover**	13.03.2013 – Continuous Integration	http://www.java.de/stammtisch-hannover
JUG **Rostock**	20.03.2013 – WebGL und Drools	https://sites.google.com/site/jughro
JUG Stuttgart	21./22.03.2013 – Stuttgarter Test-Tage	http://www.jugs.de
JUG **Hessen**	28.03.2013 – Stand-up Coding	http://www.jugh.de
JUG Hessen	27.03.2013 – In 60 Minuten von der Excel- zur JEE-Anwendung	http://www.jugf.de
JUG Ostfalen	28.03.2013 – Wicket 6 Bootcamp	http://www.jug-ostfalen.de
JUG Augsburg	28.03.2013 – CDI	http://www.jug-augsburg.de
JUG **Hannover**	29.03.2013 – Java-Stammtisch	http://www.java.de/stammtisch-hannover
JUG **Münster**	17.04.2013 – JavaEE 7	http://www.jug-muenster.de

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

DevOps for Developers

von Michael Hüttermann

DevOps, ein Portmanteau aus „Developers“ und „Operators“, ist nicht einfach nur ein Kunstwort, sondern auch der Begriff einer Bewegung. Entwickler, so der Autor, streben nach kurzen Entwicklungszyklen, nach der Umsetzung von Anforderungen, hoffentlich auch nach Tests. Die Admins auf der anderen Seite sind dafür verantwortlich, dass die Systeme stabil laufen; jedes Update stellt dabei ein potenzielles Ausfallrisiko dar. Zwei Gruppen, die häufig in verschiedenen Teams arbeiten, oder wie Michael Hüttermann es beschreibt, in getrennten Silos. Und wenn dann mal die Software nicht richtig läuft, seien je nach Standpunkt die Admins schuld, weil sie die Umgebung nicht richtig konfiguriert haben. Oder die Entwickler, die mäßige Qualität abliefern. Der Autor beschreibt das Spiel der gegenseitigen Schuldzuweisung sowie anderer Probleme, kurz: all die negativen Auswirkungen, die DevOps vermeiden möchte. Es gehe darum, die Silos einzureißen und die beiden Gruppen einander näher zu bringen, optimal ein Team zu formen. Hier beschreibt der Autor verschiedene Ansätze. Der Bevorzugte, und das ist bereits aus dem Untertitel ablesbar, ist die Integration der Administration in die Entwicklung. Nicht, dass nun das Entwicklungsteam auch administrieren soll. Nein, vielmehr geht es um die Erweiterung des Entwicklungsteams um den oder die Administratoren – und damit um eine Ausweitung agilen Vorgehens auf das Feld der Administration. Das bedeutet nicht, Ziele wie stabile Systeme aufzugeben. Es bedeutet die direkte Kommunikation von Entwickler zu Admin über die benötigte Umgebung. Und die Nutzung von Tools, die ein agiles Vorgehen unterstützen – nicht nur für die Erstellung, sondern auch für das (automatisierte) Testen und Ausliefern von Releases. So können letztendlich Releases in hoher Qualität und mit kurzen Releasezyklen ausgeliefert werden, nicht als Big Bang, sondern iterativ und inkrementell. Umso höher ist die Wahrscheinlichkeit, dass diese in der Produktivumgebung auf Anhieb laufen. Und sollte dennoch etwas schiefgehen, so ist der Rollback nur ein kleiner, im Idealfall kaum merklicher.

Was hier kurz zusammengefasst ist, beschreibt Michael Hüttermann ausführlich. Dabei trägt er wichtige Informationen und Hintergrundwissen aus unterschiedlichen Quellen zusammen und zeigt so, wo die Probleme liegen und wie eine gute Lösung zu erlangen ist. Dabei liefert er in drei von vier Buchabschnitten überwiegend das Know-how, um dann im letzten Teil auch auf Tools einzugehen, die eine solche enge und agile Anbindung von Operation an die Entwicklung unterstützen. Somit entsteht insgesamt ein runder Eindruck. Interessante Lektüre für Entwickler (inklusive Tester), Admins und IT-Manager.

Michael Müller

Michael Hüttermann

DevOps for Developers

Integrate Development and Operations, the Agile Way

196 Seiten, 29,99 US-Dollar
Apress, 2012
ISBN 978-1-4302-4569-8

Radikal führen

von Reinhard K. Sprenger

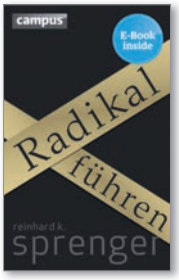
Wenn Reinhard Sprenger ein neues Buch veröffentlicht, gilt es, genauer hinzusehen. Er schafft es regelmäßig in die Bestsellerlisten und ist dem „Spiegel“ zufolge Deutschlands meistgelesener Managementautor. Entsprechend hoch sind die Erwartungen an „Radikal führen“.

Der Titel wirkt erst einmal provokant. Doch Sprenger klärt schnell, was sich dahinter verbirgt: Die ursprüngliche Bedeutung von „radikal“ ist hier gemeint. „Radix“ ist lateinisch für „Wurzel“. Es geht nicht um einen Neuansatz, sondern um die universalen und zeitlosen Grundsätze der Führung. Sprenger reduziert die unzähligen Theorien zum Thema Führung auf das Wesentliche. Ergebnis sind fünf Kernaufgaben von Führung, die eingehend besprochen werden. Zuvor aber stellt Sprenger kritisch Fragen in den Raum: Wofür werden Führungskräfte bezahlt? Welchen Mehrwert muss Führung im Unternehmen leisten? Fragen, die trivial klingen, dann aber doch nicht ganz einfach zu beantworten sind. Die Herangehensweise ist erfrischend, denkt sie das Thema doch vom Ergebnis her und stellt den unternehmerischen Erfolg in den Mittelpunkt. Dabei kommt vor allem Sprengers systemisches und menschliches Führungsbild zum Tragen: Die Einsicht, dass man seine Mitarbeiter und sich selbst mögen muss; dass Erfolg am besten zu erreichen ist, wenn Menschen mit Herz und Verstand zu eigenverantwortlicher Arbeit geführt werden.

Dann arbeitet Sprenger die fünf Kernaufgaben der Führung heraus. Erstens: Zusammenarbeit organisieren. Kooperation im Unternehmen ist, wo immer möglich, zu fördern und Hindernisse sind aus dem Weg zu räumen. Zweitens: Transaktionskosten senken. Unternehmen sollten um den Kunden kreisen und nicht um sich selbst; interner Wettbewerb ist zu vermeiden. Kundenorientierung und Vertrauenskultur gehören zum Programm. Drittens: Konflikte entscheiden. Mitarbeiter sollten dialogisch in Entscheidungen eingebunden werden, nur im Zweifel muss die Führungskraft selbst entscheiden. Viertens: Zukunftsfähigkeit sichern. Stetige Selbsterneuerung ist für das langfristige Überleben eines Unternehmens unabdingbar. Und fünftens: Mitarbeiter führen. Sprengers knackige Formel: Finden Sie die Richtigen, fordern Sie sie heraus, sprechen Sie oft miteinander, vertrauen Sie ihnen, bezahlen Sie gut und fair und dann: Gehen Sie aus dem Weg!

Sprengers eingängiger und pointierter Stil, seine klugen Ausführungen und die reduzierte Konzentration auf das Wesentliche schaffen ein rundes Lesevergnügen. Wer bereit ist, Führung aus neuen Perspektiven zu betrachten, sein eigenes Führungsbild zu hinterfragen und weiterzuentwickeln, dem sei das Buch wärmstens empfohlen. Ein Management-Betthupferl der Extraklasse.

Florian Potschka



Reinhard K. Sprenger

Radikal führen

296 Seiten, 24,99 Euro
Campus, 2012
ISBN 978-3-593-39462-6

Realisierung von Mixins mittels AOP (AspectJ) oder Quellcodemodifikation (JaMoPP)

Mixins mit Java-Bordmitteln

Unter Mixins versteht man in der objektorientierten Programmierung eine definierte Menge an Funktionalität, die man einer Klasse hinzufügen kann. Ein wesentlicher Aspekt ist, dass man sich damit bei der Entwicklung mehr auf die Eigenschaften eines bestimmten Verhaltens als auf Vererbungsstrukturen konzentriert. In Scala findet man z. B. eine Variante der Mixins unter dem Namen „Traits". Obwohl Java keine direkte Unterstützung für Mixins bietet, lässt sich diese leicht mit ein paar Annotationen, Interfaces und etwas Toolunterstützung nachrüsten.

von Jendrik Johannes und Michael Schnell

Gelegentlich ist in einigen Artikeln im Internet zu lesen, dass mit Version 8 Mixins Einzug in Java halten werden. Dies ist leider nicht der Fall. Ein Feature des Projekts Lambda (JSR-335 [1]) sind die so genannten „Virtual Extension Methods" (VEM). Diese sind zwar den Mixins ähnlich, haben aber einen anderen Hintergrund und sind deutlich limitierter in Bezug auf den Funktionsumfang. Die Motivation für die Einführung der VEM ist die Problematik der Abwärtskompatibilität bei Einführung von neuen Methoden in Interfaces [2]. Da also in der nahen Zukunft nicht mit „echten" Mixins im Java-Sprachumfang zu rechnen ist, soll in diesem Artikel aufgezeigt werden, wie man auch schon heute mit einfachen Mitteln Mixin-Unterstützung in Java-Projekten schaffen kann. Dafür diskutieren wir zwei Ansätze: Per AOP mit AspectJ [3] und per Quellcodemodifikation mit JaMoPP [4].

Warum nicht einfach Vererbung?

James Gosling, der Erfinder von Java, soll auf einer Veranstaltung auf die Frage „Was würden Sie an Java ändern, wenn Sie es neu erfinden könnten?" geantwortet haben, „Ich würde die Klassen weglassen". Nachdem sich das Gelächter wieder gelegt hatte, erklärte er, was er damit meinte: Die Vererbung in Java, welche mit der „extends"-Beziehung ausgedrückt wird, sollte – wo immer möglich – durch Interfaces ersetzt werden [5]. Jeder erfahrene Entwickler weiß, was er damit gemeint hat: Mit Vererbung sollte man sparsam umgehen. Es passiert leicht, dass man sie als technisches Konstrukt missbraucht, um Code wiederzuverwenden, und nicht, um damit eine fachlich motivierte Eltern-Kind-Beziehung zu modellieren. Aber selbst wenn man eine solch technisch motivierte Codewiederverwendung als legitim betrachtet, stößt man schnell an Grenzen, da Java keine Mehrfachvererbung erlaubt.

Mixins sind immer dann hilfreich, wenn mehrere Klassen ähnliche Eigenschaften haben bzw. ein ähnliches Verhalten definieren, diese aber über schlanke Vererbungshierarchien nicht ohne Weiteres sinnvoll abbildbar sind. Im Englischen sind Begriffe, die auf „able" enden (z. B. „sortable", „comparable" oder „commentable"), häufig ein Indiz für Einsatzfelder von Mixins. Auch wenn man beginnt „Utility"-Methoden zu schreiben, um eine Codeduplizierung bei der Implementierung von Interfaces zu vermeiden, kann dies ein Hinweis für einen sinnvollen Anwendungsfall sein.

AspectJ und „Inter-type declarations"

AspectJ erweitert Java um die Möglichkeit, Funktionalitäten, die sich quer durch den Code ziehen und damit nicht gut innerhalb der fachlichen Klassenhierarchie abzubilden sind, in eigene Codeteile auszulagern. Häufig genannte Bereiche für solche Aspekte sind „Logging", Behandlung von Fehlern oder Performancemessungen. Der Quellcode von Aspekten wird getrennt von Java in einer eigenen Datei mit der Endung *.aj* gehalten und durch Manipulation des Bytecodes zur Compile- oder Ladezeit in den originalen Java-Code „eingewebt".

Ein weniger bekanntes Feature von AspectJ sind die so genannten „Inter-type declarations" [6]. Damit kann man – unter anderem – neue Instanzvariablen und Methoden zu beliebigen Zielklassen hinzufügen. Neben eigenem Code kann man damit sogar Bibliotheken von Fremdanbietern um neue Funktionen erweitern. Dazu wird entweder das originale JAR entpackt, mit AspectJ modifiziert und wieder in ein neues JAR gepackt oder die Aspekte werden erst später beim Laden der 3rd-Party-Klassen hinzugefügt.

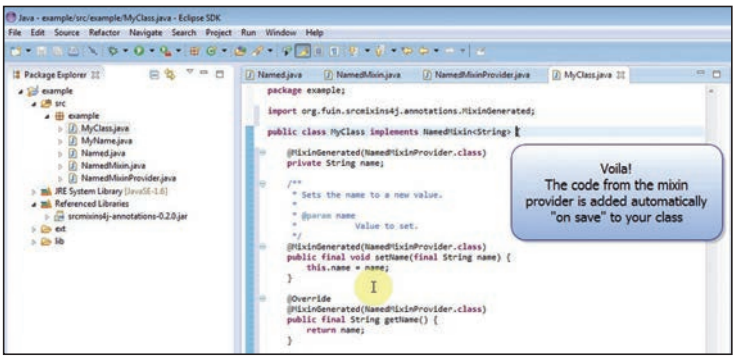


Abb. 1: SrcMixins4J-Eclipse-Plug-in (Screenvideo [10])

Mixins mit AOP

Eine recht einfache Möglichkeit Mixins zu realisieren, bietet das Eclipse-Projekt AspectJ mit den so genannten „Inter-type declarations“ [6]. Damit kann man – unter anderem – neue Instanzvariablen und Methoden zu beliebigen Zielklassen hinzufügen. Dies soll im Folgenden anhand eines kleinen Beispiels in Listing 1 gezeigt werden. Dabei nutzen wir die folgenden Begrifflichkeiten:

- *Basis-Interface*: Beschreibt das gewünschte Verhalten. Klassen, die das Mixin **nicht** nutzen sollen, können dieses Interface verwenden.
- *Mixin-Interface*: Zwischeninterface, welches im Aspekt genutzt und von Klassen implementiert wird, die das Mixin verwenden sollen.
- *Mixin-Provider*: Aspekt, der die Implementierung für das Mixin bereitstellt.

Listing 1

```
/** Basis-Interface */
public interface Named {
  public String getName();
}

/** Mixin-Interface */
public interface NamedMixin extends Named {
}

/** Mixin-Provider */
public aspect NamedAspect {
   private String NamedMixin.name;
   public final void NamedMixin.setName(String name) {
      this.name = name;
   }
   public final String NamedMixin.getName() {
      return name;
   }
}

/** Mixin-User */
public class MyClass implements NamedMixin {
  // Kann weitere Methoden haben und mehrere Mixins implementieren
}
```

- *Mixin-User*: Klasse, die ein oder mehrere Mixin-Interfaces nutzt (implementiert).

Listing 1 zeigt damit ein komplettes AOP-basiertes Mixin-Beispiel. Wenn AspectJ korrekt eingerichtet ist, sollte der folgende Quelltext fehlerfrei kompilieren und ablaufen:

```
MyClass myObj = new MyClass();
myObj.setName("Abc");
System.out.println(myObj.getName());
```

Mit der AOP-Variante kann man schon recht komfortabel arbeiten, aber es gibt auch einige Nachteile, die hier nicht verschwiegen werden sollen. Zunächst können die Inter-type-Deklarationen nicht mit generischen Typen in der Zielklasse umgehen. Das ist in vielen Fällen nicht zwingend notwendig, kann aber ganz praktisch sein. So könnte man das *Named*-Interface auch gut mit einem generischen Typ *<T>* anstelle von *String* definieren. Es würde dann das Verhalten für beliebige Namenstypen definieren. Die verwendende Klasse könnte dann festlegen, wie der Typ des Namens aussehen soll. Ein weiterer Nachteil ist, dass die von AspectJ generierten Methoden einer eigenen Namenskonvention folgen. Dies erschwert z. B. die Untersuchung der Klassen per Reflection, da man mit Methodennamen wie *ajc$interMethodDispatch* ... rechnen muss. Zu guter Letzt: Ohne Unterstützung der Entwicklungsumgebung sieht man den Quellcode in der Zielklasse nicht und ist alleine auf die Interfacedeklaration angewiesen. Dies kann man allerdings auch als Vorteil ansehen, da die nutzenden Klassen weniger Code enthalten.

Auftritt: Java Model Parser and Printer (JaMoPP)

Eine Alternative zur Realisierung von Mixins mit AspectJ bietet der Java Model Parser and Printer (JaMoPP). Vereinfacht gesagt kann JaMoPP Java-Sourcecode einlesen, als Objektgraph im Speicher darstellen und wieder in Text umwandeln, d. h. schreiben. Man kann mit JaMoPP demnach programmatisch Java-Code verarbeiten und so beispielsweise Refaktorings automatisieren oder eigene Codeanalysen implementieren. Technologisch basiert JaMoPP auf dem Eclipse Modeling Framework (EMF) [7] und EMFText [8]. JaMoPP wird gemeinsam von der TU Dresden und der DevBoost GmbH entwickelt und ist als Open-Source-Projekt frei auf GitHub verfügbar.

Mixins mit JaMoPP

Im Folgenden wollen wir das Beispiel aus den AOP-Mixins aufgreifen und etwas erweitern. Dazu definieren wir zunächst einige Annotationen:

- *@MixinIntf*: Kennzeichnet ein Mixin-Interface.
- *@MixinProvider*: Kennzeichnet eine Klasse, welche die Implementierung für ein Mixin bereitstellt. Als einziger Parameter wird das implementierte Mixin-Interface angegeben.

Listing 2

```
/** Basis-Interface (erweitert um generischen Parameter) */
public interface Named<T> {
  public T getName();
}

/** Mixin-Interface */
@MixinIntf
public interface NamedMixin<T> extends Named<T> {
}

/** Mixin-Provider */
@MixinProvider(NamedMixin.class)
public final class NamedMixinProvider<T> implements Named<T> {

  @MixinGenerated(NamedMixinProvider.class)
  private T name;

  @MixinGenerated(NamedMixinProvider.class)
  public void setName(T name) {
    this.name = name;
  }

  @Override
  @MixinGenerated(NamedMixinProvider.class)
  public T getName() {
    return name;
  }

}

/** Spezieller Namens-Typ (als Alternative zu String) */
public final class MyName {
  private final String name;

  public MyName(String name) {
    super();
    if (name == null) {
      throw new IllegalArgumentException("name == null");
    }
    if (name.trim().length() == 0) {
      throw new IllegalArgumentException("name is empty");
    }
    this.name = name;
  }

  @Override
  public String toString() {
    return name;
  }
}
```

- *@MixinGenerated*: Markiert Methoden und Instanzvariablen, die durch das Mixin generiert wurden. Einziger Parameter ist die Klasse des Mixin-Providers.

Wir erweitern im Folgenden die Interfaces und Klassen aus Listing 1 auch gleich noch um einen generischen Typ *<T>* für den Namen. Erst die das Mixin nutzende Klasse legt dann fest, welchen konkreten Typ der Name tatsächlich haben soll.

In der Klasse, die das Mixin nutzen soll, wird nun wieder das Mixin-Interface implementiert, wie in Listing 3 dargestellt. Um die Felder und Methoden, die vom Mixin-Provider definiert werden, in die Klasse *MyClass* „einzumischen", kommt ein Codegenerator zum Einsatz. Dieser modifiziert mithilfe von JaMoPP die Klasse *MyClass* und fügt die vom Mixin-Provider bereitgestellten Instanzvariablen und Methoden hinzu.

Der Codegenerator geht dabei wie folgt vor. Er liest den Quellcode jeder Klasse ein, ähnlich wie es der normale Java-Compiler macht, und untersucht dabei die Menge der implementierten Interfaces. Ist ein Mixin-Interface dabei, d. h. ein Interface mit der Annotation *@MixinIntf*, wird der entsprechende Provider dazu gesucht und die Instanzvariablen sowie die Methoden in die das Mixin implementierende Klasse kopiert.

Um die Generierung des Mixin-Codes anzustoßen, gibt es aktuell zwei Möglichkeiten: Über ein Eclipse-Plug-in direkt beim Speichern oder als Maven-Plug-in im Rahmen des Builds. Eine Installationsanleitung und den Quellcode der beiden Plug-ins findet man auf GitHub im kleinen „SrcMixins4J"-Projekt [9]. Dort ist auch ein Screenvideo abrufbar, welches den Einsatz des Eclipse-Plug-ins demonstriert. Wie die modifizierte Zielklasse dann aussieht, ist in Listing 4 zu sehen.

Wird das Mixin-Interface wieder aus der *implements*-Sektion entfernt, werden automatisch alle Felder und Methoden gelöscht, die mit *@MixinGenerated* des Providers annotiert sind. Generierter Code kann jederzeit überschrieben werden, indem man die *@MixinGenerated*-Annotation entfernt.

Listing 3

```
/** Mixin-User */
public class MyClass implements NamedMixin<MyName> {
  // Kann weitere Methoden haben und mehrere Mixins implementieren
}
```

Listing 4

```
/** Mixin-User */
public class MyClass implements NamedMixin<MyName> {

  @MixinGenerated(NamedMixinProvider.class)
  private MyName name;

  @MixinGenerated(NamedMixinProvider.class)
  public void setName(MyName name) {
    this.name = name;
  }

  @Override
  @MixinGenerated(NamedMixinProvider.class)
  public MyName getName() {
    return name;
  }
}
```

Fazit

Da mit nativer Unterstützung von Mixins im Java-Sprachstandard in absehbarer Zeit nicht zu rechnen ist, kann man sich momentan nur mit etwas AOP oder Quellcodegenerierung behelfen. Welche der beiden Varianten man wählt, hängt im Wesentlichen davon ab, ob man den Mixin-Code lieber getrennt vom eigenen Anwendungscode halten oder ihn direkt in der jeweiligen Klasse haben möchte. Auf jeden Fall erhöht man die Geschwindigkeit der Entwicklung damit deutlich und konzentriert sich weniger auf Vererbungshierarchien als vielmehr auf die Definition fachlichen Verhaltens. Beide Ansätze sind sicher nicht perfekt. Insbesondere werden Konflikte nicht automatisch aufgelöst. So führen Methoden mit gleicher Signatur aus verschiedenen Interfaces, die durch unterschiedliche Mixin-Provider bereitgestellt werden, bei einer Klasse, die beide Mixins nutzt, zu einem Fehler. Wer hier mehr will, muss auf eine andere Sprache mit nativer Mixin-Unterstützung wie z. B. Scala umsteigen.

Jendrik Johannes hat an der TU Dresden promoviert und entwickelt u. a. die Open-Source-Tools EMFText und JaMoPP. Über sein Unternehmen, die DevBoost GmbH, bietet er Unterstützung bei der Modernisierung von Legacy-Systemen sowie Beratung im Bereich MDSD an. Er ist über www.devboost.de erreichbar.

Michael Schnell ist seit mehr als fünfundzwanzig Jahren selbstständig im IT-Bereich tätig, davon die letzten dreizehn Jahre im Java-Enterprise-Umfeld. Er ist in Kundenprojekten als Technischer Projektleiter, Architekt oder Team-Lead im Einsatz.

michael.schnell@future-invent.de

Links & Literatur

[1] http://openjdk.java.net/projects/lambda/

[2] http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v4.pdf

[3] http://www.eclipse.org/aspectj/

[4] http://www.jamopp.org/

[5] http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox.html?post=677

[6] http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html#inter-type-declarations

[7] http://www.eclipse.org/emf/

[8] http://www.emftext.org/

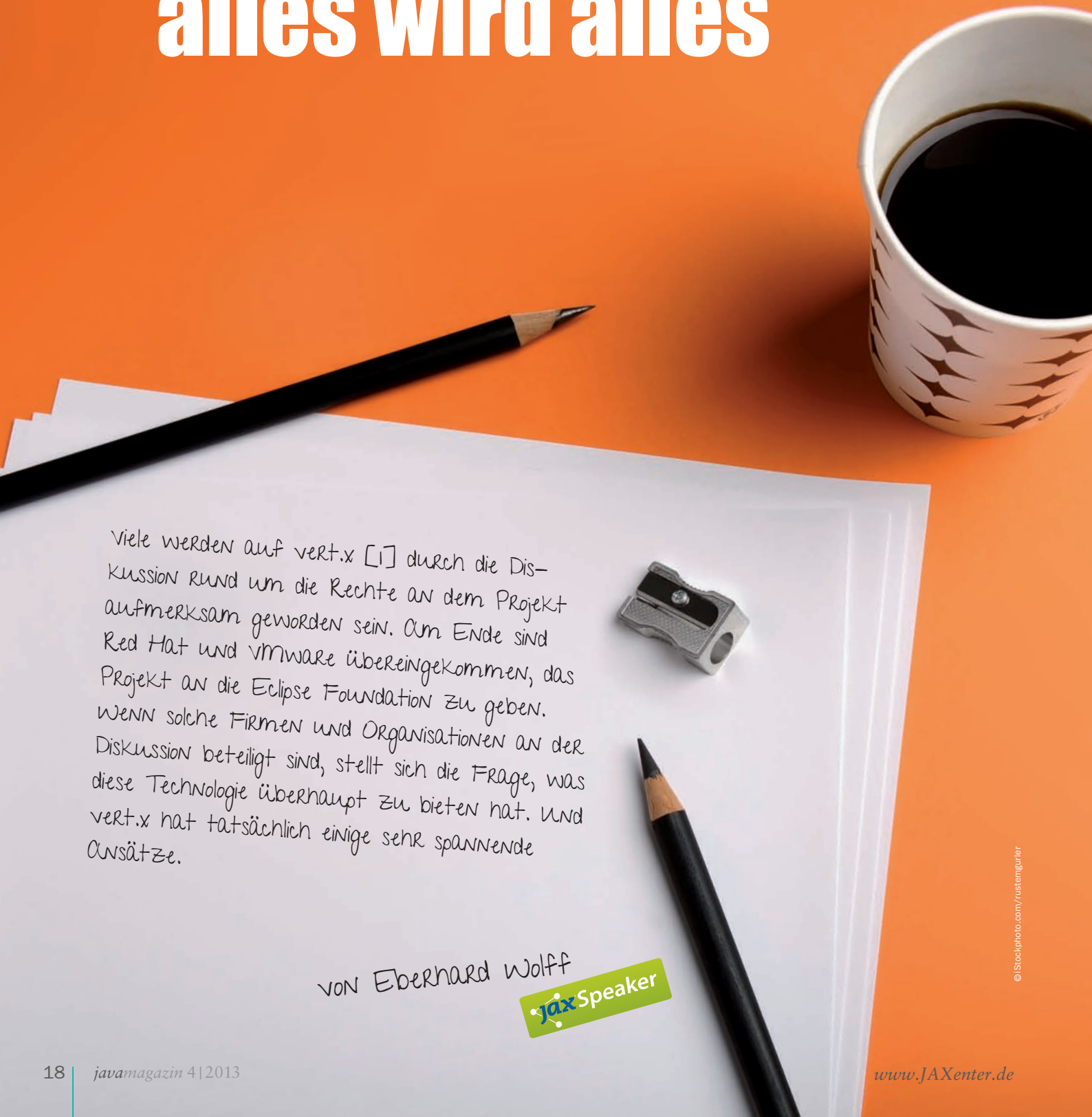
[9] https://github.com/fuinorg/SrcMixins4J/

[10] http://www.fuin.org/files/srcmixins4j-eclipse-example.swf

Anzeige

Polyglott – asynchron – modular

vert.x – alles wird alles



Viele werden auf vert.x [1] durch die Diskussion rund um die Rechte an dem Projekt aufmerksam geworden sein. Am Ende sind Red Hat und VMware übereingekommen, das Projekt an die Eclipse Foundation zu geben. Wenn solche Firmen und Organisationen an der Diskussion beteiligt sind, stellt sich die Frage, was diese Technologie überhaupt zu bieten hat. Und vert.x hat tatsächlich einige sehr spannende Ansätze.

von Eberhard Wolff

jax Speaker

Vorab: vert.x ist nicht *noch* ein Webframework. Es verfolgt vielmehr einen grundlegend anderen Ansatz für die Entwicklung von Anwendungen:

- vert.x ist asynchron. Technologien wie z. B. WebSockets (Kasten: „WebSockets“) stellen völlig neue Anforderungen an Skalierbarkeit. Außerdem müssen sie viele Netzwerkverbindungen handhaben. Ein Modell, bei dem eine Verbindung mit einem Thread versehen wird, wie Servlet-Container das tun, ist dafür nicht ausreichend. Daher setzt vert.x auf ein asynchrones Aktoren-Modell, wie es auch Akka, Node.js oder Erlang tun.
- Polyglotte Programmierung auf der Java Virtual Machine ist natürlich möglich. Dennoch sind die meisten Libraries speziell für Java gedacht und können nur schwer in andere Sprachen integriert werden. Und viele JVM-Sprachen bringen ihre eigenen Bibliotheken mit. vert.x hingegen ist von Anfang an darauf ausgelegt, mit verschiedenen JVM-Sprachen genutzt zu werden.
- Modularisierung ist in Java zwar möglich – beispielsweise durch JARs, aber Module unabhängig voneinander zu deployen, ist nur mit Technologien wie OSGi machbar. Gerade bei Java-EE-Anwendungen sieht man oft Lösungen, bei denen einzelne WARs auf demselben Server über Web Services miteinander kommunizieren. Dann können die einzelnen Module als WARs implementiert und unabhängig voneinander deployt werden. Dabei wird aber den Entwicklern zusätzlicher Aufwand aufgebürdet und gleichzeitig frisst eine solche Lösung Performance durch Socket-Kommunikation und Serialisierung/Deserialisierung. vert.x bietet ein völlig anderes Modell, um mit diesen Problemen umzugehen.

Asynchron

Wie schon erwähnt, wird im klassischen Programmiermodell ein Thread für eine Verbindung genutzt. Eine solche Lösung funktioniert nur bis zu einer bestimmten Grenze, weil das Betriebssystem nur eine begrenzte Anzahl Threads verwalten kann. Bei einer großen Anzahl wird das System auch ineffizient, weil beispielsweise jeder Thread einige Ressourcen alloziert.

vert.x geht daher grundlegend anders vor. Der Entwickler schreibt so genannte Verticles, die dann von vert.x aufgerufen werden, konkret vom Event Loop. Das ist ein Thread, der Verticles aufruft, weil beispielsweise Netzwerkpakete eingetroffen sind. In einer JVM können mehrere Event Loops parallel laufen. Dabei ist die Anzahl der Event Loops typischerweise genauso groß wie die Anzahl der CPU-Cores, sodass die Hardware vollständig ausgenutzt werden kann.

Da ein Thread für viele Verticles zuständig ist, darf ein Verticle auf keinen Fall blockieren, weil es beispielsweise auf I/O wartet, denn dadurch kommen auch

WebSockets

Normalerweise muss der Browser bei einer Webanwendung einen Request schicken, den der Server dann mit einer Response beantwortet. Wie kann ein Server aber beispielsweise Börsenkurse direkt an den Browser schicken? Dazu gibt es zwar einige Lösungen, aber erst WebSockets als Teil von HTML5 hilft wirklich weiter. Es erlaubt den Aufbau von bidirektionalen Verbindungen, durch die der Server dem Client Nachrichten schicken kann. Auf Basis dieser Verbindung können dann Daten ausgetauscht werden. So ist es beispielsweise möglich, dem Browser aktuelle Börsenkurse oder andere aktuelle Informationen zu schicken.

Bei klassischen Webanwendungen muss ein Server für einen Client nur dann eine Netzwerkverbindung offen halten, wenn ein Request bearbeitet werden muss. Bei WebSockets ist das anders: Jeder Client muss jederzeit erreichbar sein und daher eine offene Verbindung haben. Daher muss der Server wesentlich mehr offene Verbindungen ermöglichen, als das bei einfachen Webservern der Fall ist. Und es kann vorkommen, dass viele, eher kleine Nachrichten an eine Vielzahl Clients geschickt werden müssen – beispielsweise die aktuellen Kurse. Auch dazu muss der Server entsprechend ausgelegt sein.

Listing 1: Einfacher HTTP-Server

```
public class ServerBeispiel extends Verticle {

  public void start() {
    vertx.createHttpServer()
     .requestHandler(new Handler<HttpServerRequest>() {
        public void handle(HttpServerRequest req) {
          req.response.headers().put("Content-Type", "text/html; charset=UTF-8");
       req.response.end("<html><body><h1>Hallo von vert.x!</h1></body></html>");
     }
      }).listen(8080);
  }

}
```

Listing 2: WebSockets-Beispiel

```
public class WebsocketsBeispiel extends Verticle {

  public void start() {
    vertx.createHttpServer()
     .requestHandler(new Handler<HttpServerRequest>() {
       public void handle(HttpServerRequest req) {
         if (req.path.equals("/")) req.response.sendFile("websockets/ws.html");
       }
     }).websocketHandler(new Handler<ServerWebSocket>() {
       public void handle(final ServerWebSocket ws) {
    if (ws.path.equals("/myapp")) {
       ws.dataHandler(new Handler<Buffer>() {
         public void handle(Buffer data) {
           ws.writeTextFrame(data.toString());
         }
       });
     } else {
       ws.reject();
     }
  }
    }).listen(8080);
  }
}
```

Listing 3: WebSockets-Beispiel mit JavaScript

```
load('vertx.js')

vertx.createHttpServer().requestHandler(function(req) {
  if (req.uri == "/") req.response.sendFile("websockets/ws.html")
}).websocketHandler(function(ws) {
  ws.dataHandler( function(buffer) { ws.writeTextFrame(buffer.toString()) });
}).listen(8080)
```

alle anderen Verticles des Event Loops zum Stillstand. Stattdessen muss ein Callback registriert werden, der aufgerufen wird, wenn der I/O erfolgt ist. So kann das Verticle die CPU immer vollständig auslasten.

Listing 1 zeigt ein einfaches Beispiel für ein Verticle: Es handelt sich um einen HTTP-Server, wie er auch als Beispiel zu vert.x mitgeliefert wird. Die Klasse erbt von *Verticle* und kann daher mit der Instanz-Variablen *vertx* auf die vert.x-Infrastruktur zugreifen. Dort wird mit der Methode *requestHandler()* als Callback ein Handler installiert, der für jeden HTTP-Request ein HTML-Dokument zurückgibt. Dieses Dokument enthält den notwendigen JavaScript-Code, um mit dem WebSocket-Server zu interagieren. Daraufhin wird der Webserver mit *listen()* gestartet. Die Anwendung kann dann mit *vertx run http/ServerBeispiel.java* gestartet werden. vert.x kümmert sich um das Kompilieren und stellt die Anwendung zur Verfügung. Für solch einen einfachen Fall ist das Modell nicht deutlich anders, als es beispielsweise bei Servlets genutzt wird.

Listing 2 zeigt ein komplexeres Beispiel, nämlich einen WebSockets-Server (Kasten: „WebSockets“). In Listing 2 wird zunächst ein HTTP-Server initialisiert. Er liefert eine statische HTML-Seite aus, die den notwendigen JavaScript-Code für den Browser enthält und dann eine Kommunikation über WebSockets anstoßen kann. Der WebSockets-Handler untersucht, ob die Anfrage an den richtigen Pfad geht. Dann wird ein Buffer-Handler installiert, der auf die jeweiligen Anfragen reagiert. Jeder Client bekommt also seinen eigenen Buffer-Handler, aber alle werden von demselben Event Loop aufgerufen. Bei dem WebSockets-Beispiel wird deutlich, dass die vielen Callbacks in Java durchaus unübersichtlich werden können.

Polyglotte Programmierung

Die JVM ist aber polyglott und unterstützt verschiedene Sprachen. vert.x nutzt das aus und unterstützt Groovy, JavaScript und CoffeeScript mit der JavaScript-Engine Rhino, Python mit Jython und schließlich Ruby mit JRuby. Dabei hat vert.x einen sprachspezifischen Layer für jede der Sprachen. Dadurch fühlt sich die Nutzung von vert.x in diesen Sprachen recht natürlich an. Beispielsweise werden die Callbacks, die in Java mit Inner Classes implementiert werden, auf Funktionen bzw. Closures abgebildet. Listing 3 zeigt denselben WebSockets-Server wie Listing 2 – nur dieses Mal mit JavaScript. Für

den HTTP-Request wird nun eine Funktion als Callback registriert, die das HTML-File zurückgibt. Genauso werden die Handler für WebSockets durch Funktionen implementiert. Dadurch wird der Code wesentlich kürzer und übersichtlicher. Der Code ist den vert.x-Beispielen entnommen – für Ruby, Python oder CoffeeScript sieht der Code recht ähnlich aus.

Modularisierung

Für die Modularisierung bietet vert.x so genannte Modules an. Dabei handelt es sich um ein Verzeichnis, in dem der Code des Moduls abgelegt wird, und eine JSON-Konfigurationsdatei, die definiert, was gestartet werden soll.

Um also den oben gezeigten Code in ein Modul zu verwandeln, muss er in ein Verzeichnis kopiert werden. Per Konvention enthält das Verzeichnis einen Namen, der sich an den Internetdomänennamen orientiert und außerdem die Version enthält – also beispielsweise *com.ewolff.websockets-v0.1*. Die ebenfalls notwendige JSON-Konfiguration zeigt Listing 4. Wie man sieht, wird die entsprechende Datei, die das zu startende Verticle enthält, angegeben. Außerdem wird definiert, dass das Modul erneut deployt werden soll, wenn irgendwelche Dateien aus dem Modul sich ändern. In der Konfiguration können auch andere Module mit *includes* referenziert werden. So kann also ein ganzes System aus einzelnen Modulen aufgebaut werden. Gestartet wird das Modul dann mit *vertx runmod com.ewolff.websockets-v0.1*. Dabei werden auch die abhängigen Module entsprechend geladen. vert.x führt dann die Verticles aus den Modulen aus – im konkreten Fall also den WebSockets-Server.

> Es gibt auch fertige Module, wie z. B. einen Webserver und einen MongoDB Persistor.

Es gibt auch einige fertige Module. [2] stellt das zentrale Repository dar. Unter anderem gibt es einen Webserver und einen MongoDB Persistor. Diese Module können mit *vertx install* direkt in die lokale vert.x-Installation übernommen werden.

Kommunikation

Die Module müssen natürlich miteinander kommunizieren. Dazu enthält vert.x einen Bus, mit dem sich Verticles Nachrichten schicken können. Das funktioniert natürlich auch, wenn die Verticles nicht Bestandteil

eines Moduls sind. Die Nachrichten werden asynchron empfangen und bearbeitet – was dem vert.x-Programmiermodell entspricht. Der Bus unterstützt sowohl Point-to-Point-Kommunikation mit einem Empfänger als auch Publish/Subscribe, wobei die Nachrichten an mehrere Empfänger geschickt werden. Der Bus kann lokal in einer JVM laufen oder auch in einem Cluster, in dem der Bus dann über mehrere Rechner verteilt ist. Die Nachrichten sind transient – also nur im Speicher abgelegt und überleben den Neustart eines Rechners nicht.

Der Bus unterstützt die primitiven Java-Datentypen, Strings, vert.x-Buffer und JSON-Objekte. JSON (JavaScript Object Notation) ist das empfohlene Datenformat, denn damit können auch komplexe Objekte unabhängig von der Sprache dargestellt werden. Listing 5 zeigt ein Beispiel: Das Verticle registriert zunächst einen Handler am Event-Bus. Dabei wird angegeben, auf welche Adresse der Handler reagieren soll. Der Handler liest dann Informationen aus dem JSON-Objekt aus und das Verticle schickt an die Adresse des Handlers ein passendes JSON-Objekt.

Der Bus hat gerade im Zusammenspiel mit Modulen einige Vorteile. Module können so mit JSON oder einfachen Daten miteinander kommunizieren. So können Module Daten austauschen, selbst wenn sie in verschiedenen Sprachen geschrieben sind, da JSON von jeder Sprache interpretiert werden kann. Außerdem haben die Module keine gemeinsamen Klassen, sodass sie unabhängig voneinander deployt werden können. Und natürlich passt der Bus gut zu der asynchronen Kommunikation, die sich durch das gesamte vert.x-System zieht. Als Alternative zu dem Bus bietet vert.x auch an, dass Module sich Speicher teilen können. Das ist zum Beispiel sinnvoll, wenn sich Module einen Cache teilen wollen.

Listing 4: mod.json: Konfigurationsdatei eines Moduls

```
{
  "main": "ws_server_beispiel.js",
  "auto-redeploy" : true
}
```

Listing 5: Beispiel für den Event Bus

```
public class Sender extends Verticle {

  public void start() throws Exception {
    vertx.eventBus().registerHandler("com.ewolff.adress",
      new Handler<Message<JsonObject>>() {
        public void handle(Message<JsonObject> msg) {
          System.out.println("Empfangen : " + msg.body);
          System.out.println("Name : " + msg.body.getString("name"));
          System.out.println("Vorname : " + msg.body.getString("vorname"));
    }
  });
  vertx.eventBus().send(
    "com.ewolff.adress",
    new JsonObject()
    .putString("name", "Wolff")
    .putString("vorname", "Eberhard") );
    }

}
```

Mehr zum Thema

Der Entscheidung, vert.x zur Eclipse Foundation zu bringen, ging eine hitzige Diskussion um die Rechte an dem Projekt und um Open-Source-Software im Allgemeinen voraus. Alle Hintergrundinformationen zur Entwicklung der Diskussion finden sich auf www.jaxenter.de (z. B. unter [4]). Ein Interview über vert.x mit Stuart Williams (VMware) gibt es unter http://bit.ly/Ye1u7J.

Fazit

vert.x führt mehrere Innovationen ein, die vor allem Auswirkungen auf die Strukturierung von Anwendung und die Architektur haben. Dabei könnte es wegweisend sein, weil es Trends wie polyglotte Programmierung und asynchrone Systeme aufgreift – und mit einem guten Modulkonzept kombiniert. Dabei nutzt es die Ansätze, die von anderen Programmiersprachen und Systemen auch genutzt werden. So ergibt sich eine komplett neue Welt, die mit klassischem Java nicht mehr viel zu tun hat. Wer tiefer einsteigen will, dem seien die Beispiele aus der Distribution und die Dokumentation unter [3] ans Herz gelegt. Die Beschäftigung mit vert.x erweitert definitiv den Horizont!

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://vertx.io/

[2] http://github.com/vert-x/vertx-mods

[3] http://vertx.io/manual.html

[4] www.jaxenter.de/news/66526

[5] http://vertx.io/mods_manual.html

Sollte man sich als Java-Entwickler mit Node.js beschäftigen?

Hinter dem Hype

Dieser Artikel gibt Ihnen eine Einführung und einen Überblick über die Node.js-Plattform. Es wird gezeigt, wie Sie einen einfachen Server erstellen, ein Modul schreiben und mit Node.js in einer kommerziellen grafischen Entwicklungsumgebung (IDE) arbeiten.

von Christian Gross

Node.js ist ziemlich hochgejubelt worden, und es lässt sich nicht leugnen, dass verschiedene Leute denken, mit Node.js ließen sich alle Probleme lösen. Sie brauchen nur den ersten Absatz auf der Website nodejs.org zu lesen, um diesen Eindruck zu gewinnen.

„Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices." [1]

Im Buzzword-Bingo würden Sie mit diesem Absatz alle Reihen füllen. Und das an sich ist ein Problem für mich. Denn ich bin noch ein Programmierer aus der guten alten Zeit, als „load ds" tatsächlich noch etwas sehr Wichtiges bedeutete. Wenn Sie diesen Ausdruck in eine Suchmaschine eingeben, sollten Sie ihn in Anführungszeichen setzen und den Begriff „Assembler" hinzufügen. Andernfalls erhalten Sie Informationen zu einem Spiel, auf das ich aber nicht hinweisen wollte.

Was ist Node.js?

Um zu verstehen, was der Absatz zu vermitteln versucht, sehen wir uns Node.js zunächst aus bloßer Technologieperspektive und dann aus einer Architekturperspektive an.

Unter bloßer Technologieperspektive basiert Node.js auf der Programmiersprache JavaScript, die über das JavaScript-Modul von Chrome V8 ausgeführt wird. Das Chrome-V8-JavaScript-Modul ist ein in C++ geschriebenes Open-Source-Projekt von Google. Als Node.js-Programmierer brauchen Sie kein C++-Programmierer zu sein, doch wenn Sie native Erweiterungen schreiben möchten, sind Kenntnisse von C und C++ zweifellos von Vorteil. Zum Einstieg genügt es aber, wenn Sie JavaScript beherrschen.

Bei manchen Leuten löst es Unbehagen aus, wenn sie hören, dass ein Programm in einer dynamischen Sprache wie zum Beispiel JavaScript geschrieben ist, und zwar weil dynamische Programmiersprachen recht langsam sind und ziemlich viel Speicher und Ressourcen belegen. Beim JavaScript-Modul von Chrome V8 sieht die Sache größtenteils anders aus, da es bei der Ausführung JavaScript in nativen Code kompiliert. Natürlich sind keine Wunder zu erwarten, denn dynamische Sprachen mit ihrem typenlosen Datenmodell ziehen trotzdem noch Leistungseinbußen nach sich. Allerdings kann ich mit diesem Kompromiss leben, da typenlose Variablen andere Vorzüge haben. Doch zurück zum eigentlichen Thema des Artikels.

Bevor ich zur Architekturperspektive übergehe, sei eine persönliche Randbemerkung gestattet: Ich halte es für das beste Konzept, Node.js unter Linux zu entwickeln und auszuführen. In der letzten Zeit habe ich Code unter Windows verwendet, entwickle nun aber ausschließlich unter Linux und OS X. Der Vorteil bei Node.js unter Linux liegt darin, dass sich eine Entwicklungsumgebung leichter definieren lässt und es unkomplizierter ist, native Module zu schreiben. Betrachten Sie dies aber bitte als persönliche Beobachtung und nicht als Empfehlung.

Aus dem Blickwinkel der Architektur verwendet Node.js ein Ereignistypprogrammiermodell. Sehen Sie sich den folgenden Code an, wobei die konkreten Funktionsbezeichner momentan keine Rolle spielen. Es geht zunächst nur um die Codestruktur:

```
server.get('/*', function(req, res){
    throw new NotFound;
});
```

Die Variable *server* ist ein Objekt, das über die Funktion *get* verfügt. Die Funktion *get* übernimmt zwei Parameter, eine Zeichenfolge und ein *function*-Objekt. Das Funktionsobjekt ist ein entscheidendes Merkmal in Node.js, da es den Mechanismus realisiert, um Anforderungen zu verarbeiten. Es ist ein ereignisgesteuerter Ansatz, der Closures verwendet, um bestimmte Aufgaben zu behandeln. Vergleichen Sie dieses Konzept mit dem von Java, das eine Konfigurationsdatei und eine Klassenimplementierung voraussetzt (Listing 1 und 2).

Im Java-Ansatz erzeugt die Konfigurationsdatei *web.xml* einen Querverweis zwischen dem URL und dem Code, um die Anforderung zu verarbeiten. Der für die Verarbeitung der Anforderung zuständige Code ist eine Klasse, die eine andere Basisklasse erweitert. Das Java-Konzept entspricht einem klassischen objektorientierten Ansatz, wobei eine Konfigurationsdatei gemeinsam mit Code eine bestimmte Funktionalität umsetzt. Zwar haben Konfigurationsdateien durchaus ihre Berechtigung, doch laufen in vielen Projekten die Konfigurationsdateien aus dem Ruder, da sie zu komplex werden und den Code spröde machen.

Eine Konfigurationsdatei mit der Klassendefinition soll die Trennung von Verantwortlichkeiten ermöglichen. Zum Beispiel sollte ich in der Konfigurationsdatei in der Lage sein, den URL /* oder sogar */irgendetwas-anderes/** zu referenzieren. Doch leider gehen Theorie und Praxis oftmals nicht Hand in Hand. Der Programmierer muss insbesondere darauf achten, dass sein Code zweckmäßig abstrahiert ist und keine ungewollten Abhängigkeiten entstehen. Ich will die objektorientierte Programmierung nicht verteufeln, ich stelle nur Probleme heraus.

Verglichen mit anderen dynamischen Sprachumgebungen weist das Node.js-Prinzip sowohl Gemein-

Listing 1

```
<servlet-mapping>
   <servlet-name>webdav</servlet-name>
   <url-pattern>/files/*</url-pattern>
</servlet-mapping>
```

Listing 2

```
public class ProxyServlet extends HttpServlet {
   public void service(HttpServletRequest req, HttpServletResponse res)
         throws ServletException, IOException {
 }
}
```

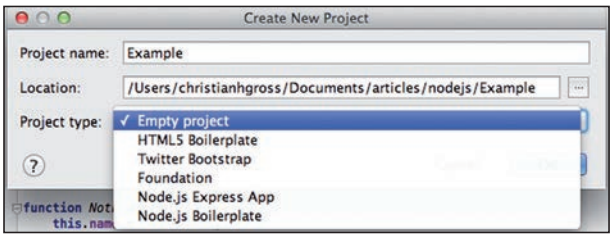


Abb. 1: Ein neues Projekt erstellen

Abb. 2: Zusätzlicher Schritt, den Sie ausführen müssen

samkeiten als auch Unterschiede auf, wobei bestimmte Annahmen und bewährte Programmierstile vorausgesetzt werden.

Die erste Anwendung erstellen und debuggen

Wenn es irgendeine Schwäche in der Node.js-Infrastruktur gibt, dann die, dass standardmäßig keine vollständige Entwicklungsumgebung dabei ist, die auch einen Debugger enthält. Zum Beispiel präsentiert Ihnen die Node.js-Debugging-Seite die in Listing 3 angegebene Lösung.

Als Entwickler bin ich an Umgebungen wie IntellJ, Eclipse oder NetBeans gewöhnt, und wenn ich eine derartige Lösung sehe, kann ich nur den Kopf schütteln und sagen: „Komm wieder, wenn du eine richtige Umgebung hast."

Als alternative Debugging-Lösung bieten sich die Google Chrome Developer Tools for Java an [2]. Interessant für den Node.js-Entwickler ist der Eclipse Debugger, mit dem sich Node.js-Tools debuggen lassen, wie es unter [3] skizziert wird.

Listing 3

```
% node debug myscript.js
< debugger listening on port 5858
connecting... ok
break in /home/indutny/Code/git/indutny/myscript.js:1
  1 x = 5;
  2 setTimeout(function () {
  3   debugger;
debug>
```

Listing 4

```
console.log("hello world")
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

Der zweite Ansatz, der die Chrome-Tools verwendet, ist für eine Entwicklungsumgebung besser geeignet. Ich bevorzuge aber eine andere Lösung, die aus der Firma JetBrains stammt und WebStorm heißt. Obwohl ich normalerweise nach Open-Source-Tools suche, sage ich „Ja" zu einem Closed-Source-Tool, wenn es sich als sehr nützlich erweist und preislich tragbar ist, da mir dieses Tool jede Menge Kummer erspart. Äußerst nützlich bei WebStorm ist, dass es die Entwicklungsumgebung unter Verwendung der üblichen Node.js-Programmierempfehlungen einrichtet.

Wenn Sie WebStorm heruntergeladen und installiert haben, rufen Sie WebStorm je nach Ihrem Betriebssystem entweder über ein Symbol oder ein Befehlszeilenprogramm auf. Starten Sie die Anwendung und erstellen Sie dann ein neues Projekt, was wie in **Abbildung 1** gezeigt aussieht.

Einen Projektnamen können Sie frei festlegen, wichtig ist vor allem, im Kombinationsfeld den Projekttyp *Node.js Boilerplate* auszuwählen. Bei diesem Projekttyp wird eine einfache Node.js-Anwendung erstellt, die die wesentlichen Komponenten (HTML5 Boilerplate, Express, Jade und Socket.IO) herunterlädt und installiert.

Nachdem das Projekt erstellt ist, zeigt WebStorm in der rechten oberen Ecke eine Warnung an, dass Sie noch einen weiteren Schritt ausführen müssen (**Abb. 2**).

Der zusätzliche Schritt ist wirklich notwendig, denn sonst ist Ihre Node.js-Anwendung nicht vollständig. Öffnen Sie also eine Eingabeaufforderung oder ein Terminal und führen Sie den Befehl *initproject.sh* aus. Daraufhin werden die Befehle ausgelöst, um die verschiedenen Bibliotheken in das Node.js-Projekt herunterzuladen und zu installieren, wie die Terminalausgabe in **Abbildung 3** zeigt.

Wenn das Skript fertiggestellt ist, ändert sich die Projektstruktur, wie in **Abbildung 4** gezeigt.

Nunmehr sind Sie bereit, die Node.js-Anwendung zu kodieren, zu debuggen und auszuführen.

Eine Node.js-Anwendung kennt kein *main*-Konzept, wie es Ihnen von Java- oder C++-Programmen her geläufig ist. Die Node.js-Anwendung ist das Skript, das Sie gerade ausführen. Im standardmäßig angelegten Projekt finden Sie eine Datei *server.js*. Sie müssen *node* aufrufen und den Skriptdateinamen als Parameter übergeben, wie die Befehlszeile im folgenden Code zeigt:

```
Chur-Macbook-Air:Example christianhgross$ node server.js
  info  - socket.io started
Listening on http://0.0.0.0:8081
```

Die Ausgabe ist einfach und gibt an, wo ein Webserver an Port 8081 hört. Wenn Sie einen Browser auf dem lokalen Computer ausführen und den URL http://localhost:8081 eingeben, erhalten Sie als Ergebnis eine einfache Sendebestätigung. Das heißt, dass Ihre Anwendung läuft.

Um die Anwendung von WebStorm aus zu debuggen, müssen Sie über das Menü Run | Edit configu-

rations eine Konfiguration erstellen. In der linken oberen Ecke des Dialogfelds finden Sie nebeneinander ein Plus- und ein Minuszeichen. Klicken Sie auf das Pluszeichen. Daraufhin erscheint ein Drop-down-Kombinationsfeld. Wählen Sie hier node.js aus. Damit legen Sie fest, dass Sie für eine Node.js-Anwendung eine Debug-Konfiguration anlegen möchten. Nachdem Sie auf die Schaltfläche geklickt haben, sollte das Dialogfeld aussehen, wie in **Abbildung 5** gezeigt.

Schließlich müssen Sie noch das Feld *Path to Node App JS File* ausfüllen. Geben Sie hier *server.js* ein, d. h. die generierte *server.js*-Datei. Klicken Sie auf OK, und schon sind Sie bereit zum Debugging. Klicken Sie dazu in der Symbolleiste auf das Symbol, das wie eine Wanze aussieht, die in ein Größer-als-Zeichen läuft (**Abb. 6**).

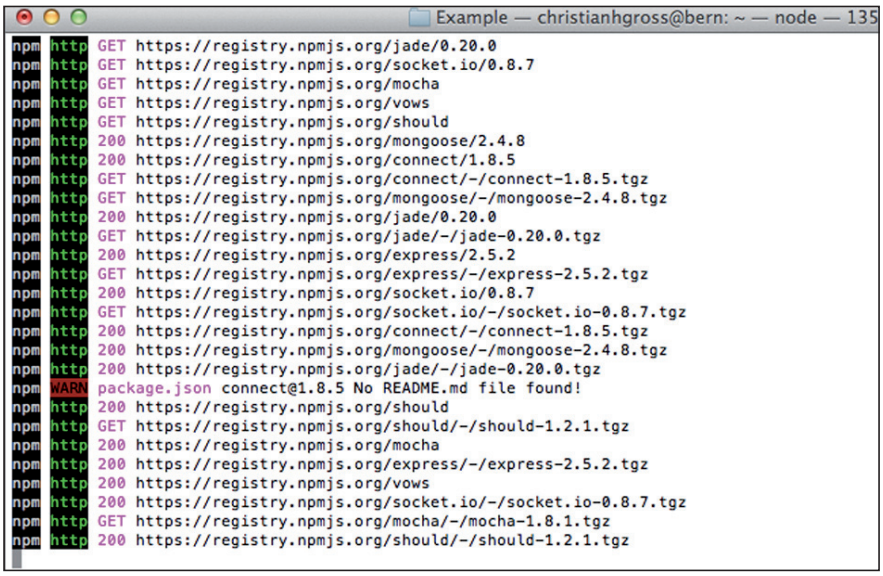


Abb. 3: Terminalausgabe beim Herunterladen zusätzlicher Bibliotheken

Wenn Sie die Schaltfläche drücken, passiert noch nicht viel, da Sie lediglich die Anwendung genau wie von der Befehlszeile ausgeführt haben. Im Unterschied dazu können Sie hier aber die Datei *server.js* im Editor öffnen und einen Haltepunkt setzen, der erreicht wird, wenn Sie die Anwendung beenden und erneut zum Debuggen ausführen. Nun können Sie endlich Code schreiben, der über den generierten Code hinausgeht.

Eine Node.js-Anwendung schreiben

Basierend auf dem Code, den Sie im vorherigen Abschnitt generiert haben, erstellen wir nun eine neue Datei namens *testme.js*. Klicken Sie in WebStorm auf File | New, um eine neue JavaScript-Datei mit dem Namen *testme* anzulegen. In die erstellte JavaScript-Datei geben Sie nun den folgenden Code ein:

```
console.log("hello world")
```

Diesen Code können Sie von der Befehlszeile mit dem Befehl *node testme.js* ausführen oder einen neuen Debug-Konfigurationseintrag mit der Datei *testme.js* als auszuführendes Skript erstellen. In beiden Fällen startet *node* das Skript, gibt „hello world“ aus und beendet es dann.

In diesem Beispiel haben wir etwas erstellt, was das Node.js-Äquivalent von „hello world“ ist. Wir machen dies nun ein wenig komplexer und übernehmen den Code von der Node.js-Homepage per Copy and Paste in *testme.js*. Dieses sehr einfache Codefragment erzeugt einen rudimentären Webserver, der „hello world“ als HTTP-Antwort ausgibt. Listing 4 gibt den Code an.

Dieses Mal haben Sie einen Webserver erstellt, der auf Port 1337 hört. Und er gibt den Text „hello world“ aus. Jetzt machen wir das Ganze noch etwas komplizierter und öffnen zwei Webbrowser mit jeweils demselben URL. Wenn Sie im Browser mehrmals auf Aktualisieren klicken, erscheint jedes Mal ein „hello world“. Um dieses Beispiel abzuschließen, ändern Sie *testme.js* in den Code von Listing 5.

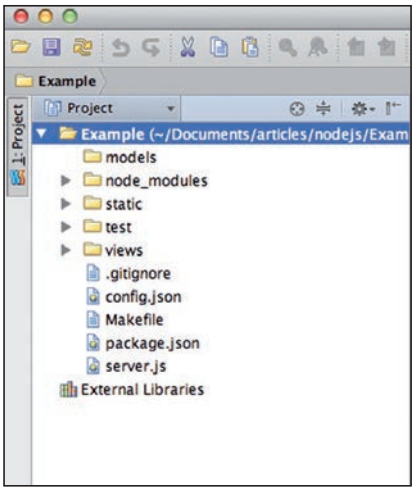


Abb. 4: Aktualisierte Projektstruktur

Der modifizierte Code inkrementiert die Variable *flipFlop*, und wenn ein Vielfaches von 5 erreicht ist, wird ein Jackpot ausgegeben und eine Schleife gestartet. Die Schleife stellt eine längere Berechnung dar, die recht viel Zeit in Anspruch nimmt. Wird kein Vielfaches getroffen, generiert die Anforderung eine Ausgabe und alles sollte in Ordnung sein.

Lassen Sie die beiden Browser diesen Code ausführen und klicken Sie fortwährend auf Aktualisieren. Sehr schnell werden Sie feststellen, dass sich beide Seiten blockieren, und hier wird es verwirrend. Denn spielen wir einmal dieses Szenario durch: Die erste Anforderung ist null, trifft den Jackpot, und noch bevor die lange Berechnung beginnt, wird die Variable *flipFlop* inkrementiert.

Das Inkrementieren der Variablen ist wichtig, denn es bedeutet, dass die nächste Anforderung die lang laufende Berechung nicht ausführen sollte. Somit sollte eine andere Browseranforderung die Daten wie vorher zurückgeben. Doch es passiert nichts. Sie könnten sogar einen Haltepunkt unmittelbar auf die *if*-Anweisung setzen und feststellen, dass die Anforderung niemals das Funktionsobjekt erreicht. Einfach ausgedrückt hat der HTTP-Server die Verarbeitung von Anforderungen eingestellt.

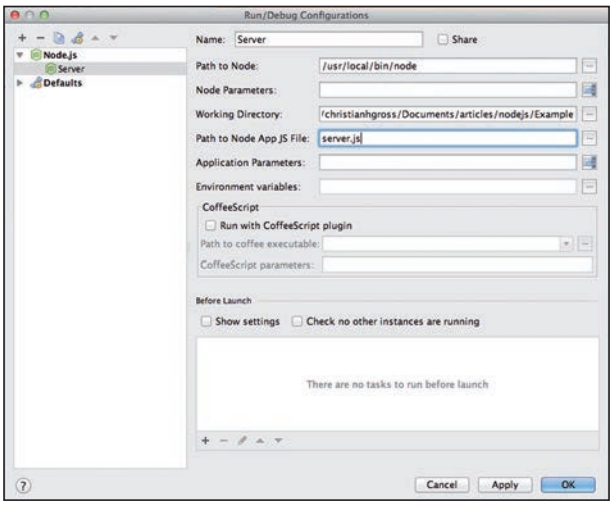


Abb. 5: Debugging-Konfiguration

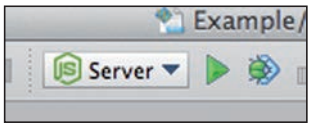


Abb. 6: Die Node.js-Anwendung debuggen

In einem Java-Webserver-Szenario passiert dies niemals, weil jede Anforderung in einem eigenen Thread verarbeitet wird. Dagegen gibt es in einer Node.js-Anwendung nur einen einzigen Thread. Ja, Sie haben richtig gelesen: nur einen einzigen Thread! Viele Node.js-Anhänger würden sagen, dass dies eine gute Idee ist, und sie werden Gründe dafür anführen.

Mein erster Gedanke war allerdings: „Das meinen die nicht ernst." Ich bin mit der Programmierung von UNIX-Boxen aufgewachsen, bevor es Threads gab. In dieser Welt wurde alles in einem Thread abgewickelt, und es war die reinste Folter. Damit eine Anwendung reaktionsfreudig bleibt und in der Lage ist, mehrere Aufgaben auf einmal auszuführen, musste man einen Prozess aufteilen. Das war aufwändig und kompliziert, da immer irgendein prozessübergreifender Kommunikationsmechanismus einzurichten war.

Doch wenn ich aus einem gewissen Abstand über das Design einer Node.js-Anwendung nachdenke, leuchtet mir der Grund ein, und er verlangt von uns, sich noch einmal Listing 7 oder sogar Listing 1 anzusehen. Hier haben wir eine *function*-Methode, die einen Parameter als Funktionsobjekt übernimmt. Dies ist in Node.js ein bestimmendes Architekturkonzept, wobei das Funktionsobjekt als Callback übermittelt wird, wenn ein Ereignis auftritt.

Sehen wir das im Zusammenhang. Trifft eine Anforderung ein, generiert der HTTP-Server ein Ereignis, das einen Aufruf des Funktionsobjekts nach sich zieht. Das Funktionsobjekt soll das Ereignis verarbeiten und die Steuerung an den HTTP-Server zurückgeben. Wird diese Steuerung nicht zurückgegeben, bleibt der Webserver hängen und es kann keine andere Anforderung verarbeitet werden.

Somit liegt die Entscheidung bei Ihnen, wie Sie die Steuerung an den Webserver zurückgeben, ohne die übrige Anwendung zu blockieren. Praktisch möchten Sie in der Lage sein, einen Thread aufzuspannen, wie es ein Java-Programmierer tun würde. Nun sind aber Threads in dem Sinne problematisch, dass Sie damit Deadlocks, Race Conditions und jede Menge anderer Probleme hervorrufen können. Die meisten Programmierer verlassen sich stattdessen auf einen Satz von Bibliotheken auf höherer Ebene, die sich um die Details auf der systemnahen Ebene kümmern. Und dies ist der Weg, den Sie in Node.js einschlagen. Der URL [4] verweist auf eine Anzahl von Ablaufsteuerungsmodulen, mit denen Sie Ihr Node.js in die Lage versetzen, „einen Thread aufzuspannen".

Ein eigenes Modul schreiben

Im letzten Teil dieses Artikels möchte ich darauf eingehen, wie Sie ein eigenes Modul schreiben. In reinem JavaScript würden Sie eine Klasse anlegen und dann diese Klasse füllen. Das Problem dabei ist, dass JavaScript das Konzept von Namespaces nicht kennt. Ohne Namespaces dürften Konflikte mit Klassendefinitionen auftreten, und Sie erzeugen möglicherweise Fehler, die sich nur äußerst schwer aufspüren lassen. Um Node.js erweiterbar und dennoch stabil zu machen, ist darin ein Mo-

Listing 5

```
console.log( "hello world");
var http = require('http');
var flipFlop = 0;
http.createServer(function (req, res) {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   if( (flipFlop % 5) == 0) {
      flipFlop ++;
      var maxCounter = 1000000;
      var val = 0.0;

      for( var counter1 = 1; counter1 < maxCounter; counter1 ++) {
         for( var counter2 = 1; counter2 < maxCounter; counter2 ++) {
            for( var counter3 = 1; counter3 < maxCounter; counter3 ++) {
               for( var counter4 = 1; counter4 < maxCounter; counter4 ++) {
                  val = counter1 + counter2 + counter3 + counter4;
               }
            }
         }
      }
      res.end('Hello World jackpot\n');
   }
   else {
      flipFlop ++;
      res.end('Hello World nope not yet\n');
   }
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

dulkonzept realisiert. Es gibt zwei Arten von Modulen, *node_modules* und einfache Module. Einfache Module werden im Stammverzeichnis gespeichert. Allerdings sind sie nicht zu empfehlen, da zu viele einfache Module einen Projektarbeitsbereich vollstopfen würden.

Deshalb sollten Sie lernen, wie Sie für Ihre Anwendung ein geeignetes Modul erstellen. Mit WebStorm haben Sie schon die halbe Schlacht gewonnen, da das Tool die Verzeichnisstruktur für Sie anlegt. Führen Sie dann über WebStorm die folgenden Schritte aus:

1. Suchen Sie im Projektarbeitsbereich das Verzeichnis *node_modules*.
2. Im Verzeichnis *node_modules* erstellen Sie nun ein Modul namens *mymodule*. Dazu legen Sie ein Verzeichnis *mymodule* an.
3. Im Verzeichnis *mymodule* erstellen Sie ein weiteres Verzeichnis namens *lib*.
4. Im Verzeichnis *mymodule* legen Sie die Datei *index.js* an.
5. Im Verzeichnis *lib* legen Sie die Datei *mymodul.js* an.

Die Verzeichnisstruktur sollte dann wie in **Abbildung 7** aussehen.

Die Verzeichnisstruktur ist wichtig, denn es ist eine bewährte Methode, ein Modul namens *mymodule* und darin eine Bibliothek für JavaScript-Dateien zu erzeugen, die die Funktionalität von *mymodule* implementieren.

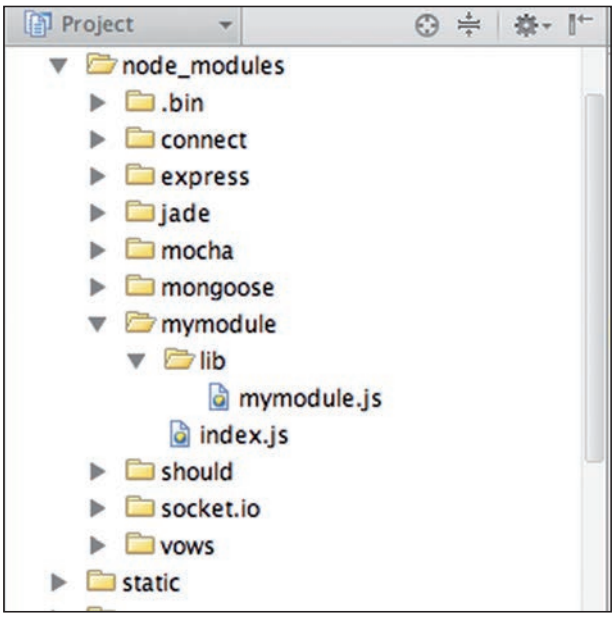


Abb. 7: Die Node.js-Verzeichnisstruktur für das Modul „mymodule“

Wenn Sie auf ein formales Node.js-Modul verweisen, sucht Node.js nach diesem Modul im Verzeichnis *node_*

modules. Dann wird im Stammverzeichnis des Moduls (*mymodule*) die Datei *package.json* geladen. Diese Datei enthält die Beschreibung, wie das Modul strukturiert ist, wer es geschrieben hat, welche Abhängigkeiten bestehen, die Versionsnummer und jede Menge anderer Dinge. Das Beispiel verzichtet auf *package.json*, da es möglichst einfach bleiben soll. Wenn *Node.js* eine fehlende Datei *package.json* ermittelt, wird versucht, zuerst *index.js* und bei fehlender Datei *index.js* die Datei *index.node* zu laden.

In *index.js* ist es lediglich erforderlich, die geeigneten JavaScript-Dateien zu laden, die für die Funktionalität des Moduls zuständig sind. In einem anderen Beispiel heißt das, dass die Datei *mymodul.js* zu laden ist. Der Code hierfür ist folgender:

```
module.exports = require('./lib/mymodule.js');
```

Die Syntax ist sehr wichtig und muss eingehalten werden. Der Variablen *module.exports* wird die Ausgabe des *require*-Funktionsaufrufs zugewiesen. Die Funktion *require* lädt die Teile, aus denen das Modul besteht, in diesem Fall *mymodule.js*. Hätte ich *mymodule.js* weggelassen und nur das Verzeichnis *lib* referenziert, würden alle Dateien im Verzeichnis *lib* geladen. Es stellt sich die Frage, ob alle JavaScript-Dateien im *lib*-Verzeichnis gespeichert sein müssen. Die Antwort ist: nein. Diese Vorgehensweise hat sich lediglich bewährt, da man in *index.js* definiert, wo sich die Dateien für das Modul befinden.

In der Datei *mymodule.js* implementieren Sie nun die Funktionalität. In diesem Fall handelt es sich um eine Variablenzuweisung und einen Funktionsaufruf.

```
exports.answer = 42;

exports.callMe = function( inputVal) {
    console.log( "[Your message => " + inputVal + "]");
}
```

Alles, was Sie exportieren möchten, weisen Sie der Variablen *exports* zu, die Node.js implizit definiert. Jede Variable, die Sie nicht *exports* zuweisen, ist für jeden Konsumenten Ihres Moduls zugänglich. In diesem Sinne haben Sie eine Art Deklaration von öffentlichen und privaten Klassen-Membern. Das Beispiel definiert die Variable *answer* und die Funktion *callMe*.

Damit ist die Moduldefinition vollständig, und wir müssen nun das Modul als Client aufrufen. Dazu zapfen Sie den Loader mithilfe des *require*-Methodenaufrufs an, der in der Datei *index.js* verwendet wird. Die eigentliche Aufgabe von *require* besteht darin, eine JavaScript-Datei zu laden, sie als Modul zu definieren und alles miteinander über implizite Variablen zu verbinden.

```
var mymodule = require('mymodule')
mymodule.callMe( "what am I calling?");
console.log(mymodule.answer);
```

Die erste Zeile lädt *mymodule* und weist es der Variablen *mymodule* zu. An diesem Punkt wird alles, was ich oben an *exports* zugewiesen habe, an *mymodule* zugewiesen. Somit kann ich direkt auf die Member-Variable *answer* und die Funktion *callMe* verweisen. Node.js-Entwickler verwenden oftmals diesen Mechanismus als generische Fabrik, um die Objekte zu instanziieren, die die eigentliche Arbeit übernehmen.

Fassen wir zusammen

Node.js ist keine Patentlösung. Es ist ein brauchbares Konzept, wenn Sie JavaScript-Entwickler sind und mehr aus der Sprache JavaScript herausholen möchten. Als Java-Entwickler werden Sie angesichts dieser Lösung denken: „Ich weiß nicht, ob das wirklich etwas bringt." Und ehrlich gesagt liegen Sie damit gar nicht so daneben. Ich denke bei Node.js eher an eine schnelle Umsetzung von Webproblemen. So kann ich mir leicht vorstellen, die Webanwendung mit Node.js zu erstellen, und dann mit Java als Rückgrat die Verarbeitung über Webdienste vorzunehmen (entweder XML oder JSON).

In Wahrheit haben wir Unmengen von Java-Frameworks, um Webseiten zu generieren, und jedes erzeugt eine große Anzahl von Konfigurations- und Codedateien. Uns wird Flexibilität vorgegaukelt, doch sind sie in meinen Augen anfällig wie eh und je. Mit Node.js erhalte ich ein schnelles Framework, da sowohl die Client- als auch die Serverseite mit JavaScript arbeitet. Die Dinge sind einfacher, da Sie nicht darüber grübeln müssen, was ein Objekt wie realisiert. Ich bin hier nicht auf die verschiedenen Frameworks zur Webgenerierung wie Jade eingegangen, doch sind sie zweifellos eine Erkundung wert.

Insgesamt bin ich der Meinung, dass Sie als Hardcore-Webentwickler zumindest einmal einen Blick auf Node.js werfen sollten.

Aus dem Englischen von Frank Langenau.

Christian Gross ist Quant Developer. Er tätigt Investitionen, indem er Algorithmen schreibt. Als Mitglied der Trading-Community schrieb er die Website timegeist.li, die Tradern den Ideenaustausch, Slice-and-Dice-Datenanalysen und die Einschätzung des Investmentmarkts insgesamt erleichtert. Die Website ist – wen wundert's – in Java geschrieben, mit Groovy- und Node.js-Anteilen. In einem früheren Leben war Christian einmal Entwickler für Entwickler. Er beriet Teams bei ihren Softwareentwicklungsprozessen und -technologien. Noch immer steht er Kunden, die ihn um Hilfe bitten, mit Rat und Tat zur Seite.

Links & Literatur

[1] http://nodejs.org/

[2] http://code.google.com/p/chromedevtools/

[3] https://github.com/joyent/node/wiki/using-eclipse-as-node-applications-debugger

[4] https://github.com/joyent/node/wiki/modules#wiki-async-flow

Anzeige

WebSocket-Portabilität auf der JVM

Atmosphere

Mit dem Atmosphere-Framework lassen sich portierbare Anwendungen in Groovy, Scala und Java schreiben. Neben einer JavaScript-Komponente enthält Atmosphere mehrere Serverkomponenten, die alle wichtigen Java-basierten Webserver unterstützen. Ziel ist es, die Entwicklung von Anwendungen dadurch zu erleichtern, dass das Framework selbstständig den besten Kommunikationskanal zwischen Client und Server findet – und das codeunabhängig.

von Jeanfrançois Arcand

Mit Atmosphere im Einsatz ist es zum Beispiel möglich, dass eine Anwendung das WebSocket-Protokoll nutzt, solange sie mit einem Browser oder Server mit WebSocket-Unterstützung verwendet wird. Andernfalls kehrt das Framework codeunabhängig zu HTTP zurück. Mit dem Internet Explorer 6, 7, 8 und 9 läuft eine Atmosphere-Anwendung problemlos mit HTTP; im Zusammenspiel mit dem Internet Explorer 10 wird auf WebSocket umgestellt. Um die Mächtigkeit dieses Frameworks noch besser zu demonstrieren, wollen wir eine einfache Chat-Anwendung aufsetzen. Wir gehen davon aus, dass diese nur einen Chatroom unterstützt, um die Logik einfach zu halten. Zunächst schreiben wir die serverseitige Komponente. Atmosphere unterstützt folgende Komponenten:

- **Atmosphere-Runtime**: das Kernmodul von Atmosphere, auf dessen Grundlage alle anderen Module gebaut werden. Es stellt zwei einfache APIs für die Anwendungsentwicklung zur Verfügung: *AtmosphereHandler* und *Meteor*. Der *AtmosphereHandler* ist ein einfaches Interface für Implementierungen. Bei *Meteor* handelt es sich um eine Klasse, die aufgerufen und in Servlet-basierte Anwendungen injiziert werden kann.
- **Atmosphere Jersey**: eine Erweiterung für das Jersey-REST-Framework, die einige zusätzliche Annotationen bereitstellt.
- **Atmosphere GWT**: eine Erweiterung für das GWT-Framework.

Die Serverseite

Im Folgenden werde ich die Atmosphere-Runtime verwenden, um zu demonstrieren, wie einfach es ist, eine einfache asynchrone Anwendung zu schreiben. Beginnen wir mit der Serverkomponente, die einen *AtmosphereHandler* verwendet. Letzterer ist wie in Listing 1 definiert.

Die *onRequest*-Methode wird jedes Mal aufgerufen, wenn ein Request auf dem Pfad abgebildet wird, der mit dem *AtmosphereHandler* in Verbindung steht. Der Pfad wird dadurch definiert, dass eine Implementierung des *AtmosphereHandler*s annotiert wird:

```
@AtmosphereHandlerService(path = "/<path>")
```

In Atmosphere steht eine *AtmosphereResource* für eine physische Verbindung. Mithilfe einer *AtmosphereResource* lassen sich Informationen über einen Request abrufen. Auf die Antwort hin kann eine Aktion ausgeführt werden und – was wichtiger ist – die Verbindung während der *onRequest*-Ausführung aufrechterhalten werden. Ein Webserver benötigt Informationen darüber, wann eine Verbindung für weitere Aktionen offen bleiben soll (z. B. für WebSockets) und wann ein Upgrade zwecks Protokollunterstützung erforderlich ist, damit die HTTP-Verbindung (Streaming, Long-Polling, JSONP oder serverseitige Events) für zukünftige Aktionen offen bleibt.

Die *onStateChange*-Methode (**Abb. 1**) wird in Atmosphere in den folgenden Fällen aufgerufen:

- bei einer Broadcast-Operation, die von einem Broadcaster initiiert wird und eine Aktion erfordert. Der Broadcaster ist eine Art Kommunikationskanal. Eine Anwendung kann viele solcher Kanäle erzeugen und per *BroadcasterFactory*-Klasse auf sie zugreifen. Eine *AtmosphereResource* steht grundsätzlich mit einer oder mehreren Broadcastern in Verbindung. Man kann Broadcaster auch als Event-Queue sehen, in der man auf neue Broadcast-Events lauschen und sich im Falle eines Events benachrichtigen lassen kann.

Listing 1: AtmosphereHandler

```
public interface AtmosphereHandler {

    void onRequest(AtmosphereResource resource) throws IOException;

    void onStateChange(AtmosphereResourceEvent event) throws IOException;

    void destroy();
}
```

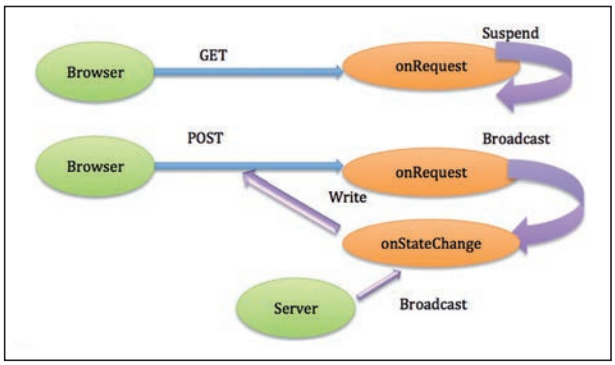


Abb. 1: Die onState-Change-Methode

Broadcast ist durch *onRequest*, *onStateChange* oder überall auf der Serverseite möglich.

- Wenn die Verbindung geschlossen wurde oder ein Time-out aufgetreten ist (d. h. keine Aktivität zu verzeichnen ist).

Das klingt kompliziert? Nun, zum Glück wird das Framework mit *AtmosphereHandlers* ausgeliefert, die in fast allen Szenarien verwendet werden können. So kann sich der Entwickler auf die Anwendungslogik konzentrieren, während sich Atmosphere um den Lebenszyklus der Verbindung kümmert. In Listing 2 wird der *OnMessage<T> AtmosphereHandler* verwendet, um eine Anwendung zu schreiben.

Die Grundidee ist dabei, so viel wie möglich aus dem Lebenszyklus der Verbindung an die einsatzbereite Komponente von Atmosphere zu delegieren. Zunächst versehen wir die *ChatRoom*-Klasse mit der *@AtmosphereHandlerService*-Annotation und definieren den Pfad und die Interceptors. Ein *AtmosphereInterceptor* ist eine Art Filter, der grundsätzlich vor und nach dem *AtmosphereHandler#onRequest* aufgerufen wird. Der *AtmosphereInterceptor* ist für die Bearbeitung von Request und Response nützlich, aber auch für den Lebenszyklus, beispielsweise beim Suspend und beim Broadcast (**Abb. 2**).

Wie oben beschrieben, können zwei Interceptors dazu verwendet werden, zuerst den Request zu verlängern (*AtmosphereResourceLifeCycleInterceptor*) und dann die Daten, die bei jedem POST empfangen werden, per Broadcast zu verteilen. Wir können uns also ausschließlich auf die Anwendungslogik konzentrieren.

Statt also unseren kompletten *AtmosphereHandler* zu schreiben, können wir den *OnMessage<T>*-Handler verwenden, der die Broadcast-Operation an die *onMessage*-Methode delegiert (Zeile 10). Für unsere Chat-Anwendung bedeutet das einfach nur, dass wir schreiben, was wir empfangen haben. Wenn wir 50 verbundene User haben, bedeutet das, dass die *onMessage*-Methode 50-mal aufgerufen wird, damit die 50 User die Nachricht erhalten. Für die Kommunikation zwischen Client und Server verwenden wir JSON. Der Client sendet Folgendes:

```
{"message":"Hello World","author":"John Doe"}
```

Listing 2: OnMessage<T> AtmosphereHandler

```
@AtmosphereHandlerService(
        path="/chat",
        interceptors = {AtmosphereResourceLifecycleInterceptor.class,
                        BroadcastOnPostAtmosphereInterceptor.class})
public class ChatRoom extends OnMessage<String> {

    private final ObjectMapper mapper = new ObjectMapper();

    @Override
    public void onMessage(AtmosphereResponse response, String message)
                                                            throws IOException {
        response.getWriter()
                .write(mapper.writeValueAsString(mapper.readValue(message, Data.class)));
    }
}
```

Listing 3

```
public final static class Data {

    private String message;
    private String author;
    private long time;

    public Data() {
        this("","");
    }

    public Data(String author, String message) {
        this.author = author;
        this.message = message;
        this.time = new Date().getTime();
    }

    public String getMessage() {
        return message;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public long getTime() {
        return time;
    }

    public void setTime(long time) {
        this.time = time;
    }

}
```

Und der Server schickt über die Browser Folgendes zurück:

```
{"message":"Hello World","author":"John Doe","time":1348578675087}
```

Wir verwenden die Jackson-Library, um die Nachricht zu lesen und zusammen mit der Zeit, zu der die Nachricht empfangen wurde, zurückzusenden. Die *Data*-Klasse ist ein einfaches POJO (Listing 3).

Die Clientseite – Atmosphere.js

Um die Clientseite zu schreiben, verwenden wir Atmosphere.js. Werfen wir zunächst einen Blick auf den Code (Listing 4).

Listing 4 enthält eine Menge zusätzlichen Code, wir wollen uns aber auf die wichtigsten Teile von Atmosphere.js beschränken. Als Erstes starten wir eine Verbindung (die im Code *socket* heißt):

```
var socket = $.atmosphere;
```

Im nächsten Schritt definieren wir einige *Callback*-Funktionen. Wir beschränken uns auf eine kleine Gruppe. Als Erstes definieren wir eine *onOpen*-Funktion, die aufgerufen wird, sobald der zugrunde liegende *transport* mit dem Server verbunden wird. Dort zeigen wir nur den *transport* an, der für die Serververbindung benötigt wird. Er wird auf dem Request-Objekt spezifiziert, das wie folgt definiert wird:

```
var request = { url: document.location.toString() + 'chat',
        contentType : "application/json",
        transport : transport ,
        fallbackTransport: 'long-polling'};
```

Hier verwenden wir standardmäßig den WebSocket-*transport* und wechseln zu Long-Polling, falls WebSocket weder vom Browser noch vom Server unterstützt

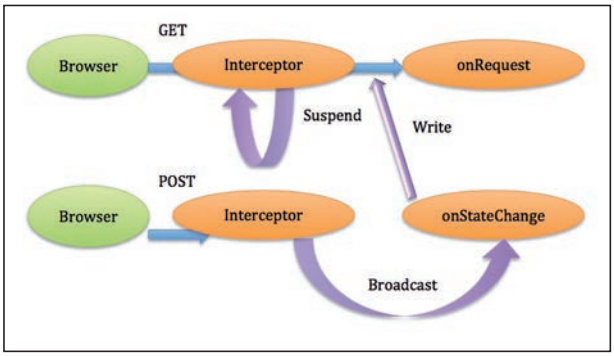


Abb. 2: Suspend und Broadcast

wird. In unserer *onOpen*-Funktion wird angezeigt, welcher *transport* verwendet wird. Übrigens lässt sich der *transport* auch bei einem Ausfall von WebSocket ändern. Dazu wird eine *onTransportFailure*-Funktion hinzugefügt:

```
request.onTransportFailure = function(errorMsg, request) {
    if (window.EventSource) {
        request.fallbackTransport = "sse";
        transport = "see";
    }
```

Um dies zu demonstrieren, suchen wir nach einem *EventSource*-Objekt (HTML5-Server-side-Events). Falls dieses verfügbar ist, stellen wir den *transport* entsprechend um. Das Elegante an dieser Vorgehensweise: Man braucht kein spezielles API. Bei allen *transports* kann man unter Verwendung von Atmosphere.js analog vorgehen.

Als Nächstes definieren wir die *onMessage*-Funktion, die immer dann aufgerufen wird, wenn wir Daten vom Server empfangen:

```
request.onMessage = function (response) {
  .....
}
```

Hier zeigen wir nur die empfangene Nachricht an. Um eine Verbindung aufzubauen und Daten an den Server zu schicken, müssen wir nur folgenden Aufruf starten:

```
subSocket = socket.subscribe(request);
```

Einmal subscribed, können wir Daten empfangen und senden. Dafür verwenden wir das *subSocket*-Objekt, das uns die Subscribe-Operation zurückliefert. Wenn der WebSocket-*transport* in Verwendung ist, referenziert das *subSocket*-Objekt die Verbindung, da das Protokoll bidi-

Listing 4: Atmosphere.js-Clientcode

```
$(function () {
    "use strict";

    var header = $('#header');
    var content = $('#content');
    var input = $('#input');
    var status = $('#status');
    var myName = false;
    var author = null;
    var logged = false;
    var socket = $.atmosphere;
    var subSocket;
    var transport = 'websocket';

    // We are now ready to cut the request
    var request = { url: document.location.toString() + 'chat',
        contentType : "application/json",
        trackMessageSize: true,
        shared : true,
        transport : transport ,
        fallbackTransport: 'long-polling'};

    request.onOpen = function(response) {
        content.html($('<p>', { text: 'Atmosphere connected using '
                                                    + response.transport }));
        input.removeAttr('disabled').focus();
        status.text('Choose name:');
        transport = response.transport;

        if (response.transport == "local") {
            subSocket.pushLocal("Name?");
        }
    };

    request.onTransportFailure = function(errorMsg, request) {
        jQuery.atmosphere.info(errorMsg);
        if (window.EventSource) {
            request.fallbackTransport = "sse";
            transport = "see";
        }
        header.html($('<h3>', { text: 'Atmosphere Chat. Default transport
                            is WebSocket, fallback is ' + request.fallbackTransport }));
    };

    request.onMessage = function (response) {

        // We need to be logged first.
        if (!myName) return;

        var message = response.responseBody;
        try {
            var json = jQuery.parseJSON(message);
        } catch (e) {
            console.log('This doesn\'t look like a valid JSON: ', message.data);
            return;
        }

        if (!logged) {
            logged = true;
            status.text(myName + ': ').css('color', 'blue');
            input.removeAttr('disabled').focus();
            subSocket.pushLocal(myName);
        } else {
            input.removeAttr('disabled');

            var me = json.author == author;
            var date = typeof(json.time) == 'string' ? parseInt(json.time) : json.time;
            addMessage(json.author, json.message, me ? 'blue' : 'black',
                                                        new Date(date));
        }
    };

    request.onClose = function(response) {
        logged = false;
    }

    subSocket = socket.subscribe(request);

    input.keydown(function(e) {
        if (e.keyCode === 13) {
            var msg = $(this).val();
            if (author == null) {
                author = msg;
            }

            subSocket.push(jQuery.stringifyJSON({ author: author, message: msg }));
            $(this).val('');

            input.attr('disabled', 'disabled');
            if (myName === false) {
                myName = msg;
            }
        }
    });

    function addMessage(author, message, color, datetime) {
        content.append('<p><span style="color:' + color + '">'
                                                    + author + '</span> @ ' +
            + (datetime.getHours() < 10 ? '0' + datetime.getHours() :
                                                    datetime.getHours()) + ':'
            + (datetime.getMinutes() < 10 ? '0' + datetime.getMinutes() :
                                                    datetime.getMinutes())
            + ': ' + message + '</p>');
    }
});
```

rektional ist. Bei allen anderen Übertragungsarten wird für jede Push-Operation eine neue Verbindung geöffnet:

```
subSocket.push(jQuery.stringifyJSON({ author: author, message: msg }));
```

Als Nächstes fügen wir Unterstützung für ein besonders nützliches Atmosphere-Feature hinzu: die Möglichkeit, eine Verbindung unter geöffneten Fenstern bzw. Tabs zu teilen. Man braucht bei einem Request nur die geteilte Variable auf *true* zu setzen:

```
var request = { url: document.location.toString() + 'chat',
        contentType : "application/json",
        transport : transport ,
        shared : true,
        fallbackTransport: 'long-polling'};
```

Nun wird die Verbindung jedes Mal, wenn ein neues Fenster oder ein neuer Tab geöffnet sowie dieselbe Seite aufgerufen wird, geteilt. Um eine Benachrichtigung über die „Master"-Tabs bzw. -Fenster zu erhalten, implementiert man folgende Funktion:

```
request.onLocalMessage = function(message) {
  ....
}
```

Tabs bzw. Fenster können über die folgende Funktion auch direkt kommunizieren:

```
subSocket.pushLocal(...)
```

Voll einsatzbereit? Noch nicht ganz...

Wir haben jetzt eine voll funktionsfähige Chat-Anwendung, die allerdings in ihrem jetzigen Zustand noch mit zwei Problemen behaftet ist: Das erste ist Proxy- bzw. Firewall-bedingt: Gelegentlich lassen es Letztere nicht zu, dass eine Verbindung für längere Zeit inaktiv bleibt, weswegen die Verbindung automatisch vom Proxy geschlossen wird. Wenn die Verbindung aufrechterhalten werden soll, muss der Client jedes Mal eine neue Verbindung aufbauen. Lösen lässt sich dieses Problem durch Übertragung einiger Bytes zwischen Client und Server, um einen Verbindungsabbruch zu vermeiden. In unserem Fall muss lediglich der *HeartbeatInterceptor* hinzugefügt werden. Durch ihn bleibt die Verbindung aktiv (Listing 5), und zwar wiederum codeabhängig.

Listing 5: HeartbeatInterceptor

```
@AtmosphereHandlerService(
        path = "/chat",
        interceptors = {AtmosphereResourceLifecycleInterceptor.class,
                        BroadcastOnPostAtmosphereInterceptor.class,
                         HeartbeatInterceptor.class})
public class ChatRoom extends OnMessage<String> {
```

Nun schreibt der *HeartbeatInterceptor* periodisch einige Bytes (Whitespace), um die Verbindung aufrecht zu erhalten. Ausschließen lässt sich allerdings nicht, dass einige Proxies die Verbindung trotz Aktivität unterbrechen oder dass ein Netzwerkproblem auftritt und der Browser sich erneut verbinden muss.

Während eines Reconnects kann es immer sein, dass gerade eine Broadcast-Operation läuft. Der Browser wird den Broadcast in diesem Fall nie empfangen, weil die Verbindung gerade erst aufgebaut wird. So kann es vorkommen, dass der Browser eine Nachricht verpasst. Bei einigen Anwendungen stellt das ein größeres Problem dar.

Zum Glück unterstützt Atmosphere das Prinzip des *BroadcasterCache*. Installiert man einen solchen, verliert oder verpasst der Browser keine einzige Nachricht mehr. Wenn der Browser sich neu verbindet, durchsucht Atmosphere den Cache und stellt sicher, dass alle Nachrichten, die während des erneuten Verbindungsversuchs versendet wurden, zum Browser gesendet werden. Das

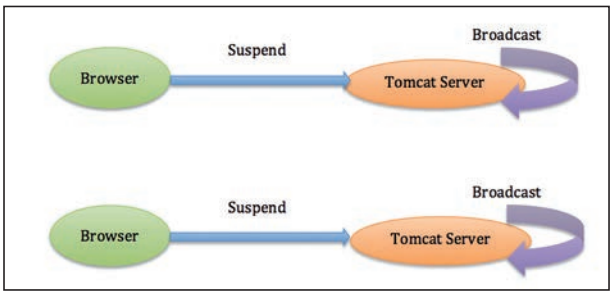


Abb. 3: Server in der Cloud

BroadcasterCache-API lässt sich einbinden, und Atmosphere wird mit einer einsatzbereiten Implementierung ausgeliefert. Für unsere Chat-Anwendung brauchen wir also nur Folgendes zu tun:

```
@AtmosphereHandlerService(
        path = "/chat",
        broadcasterCache = HeaderBroadcasterCache.class,
        interceptors = {AtmosphereResourceLifecycleInterceptor.class,
                        BroadcastOnPostAtmosphereInterceptor.class,
                        HeartbeatInterceptor.class})
  public class ChatAtmosphereHandler extends OnMessage<String> {
```

Unsere Anwendung verliert oder verpasst nun garantiert keine Nachricht mehr.

Das zweite Problem: Je nach Webserver können sich empfangene Nachrichten mischen und der Browser z. B. zwei oder anderthalb Nachrichten auf einmal empfangen. Nehmen wir an, dass wir JSON zum Schreiben unserer Nachricht verwenden. In dem Fall wird der Browser Nachrichten wie die folgenden nicht lesen können:

```
{"message":"Hello World","author":"John Doe","time":1348578675087}
          {"message":"Cool Man","author":"Foo Bar","time":1348578675087}
```

oder

```
{"message":"Hello World","author":"John Doe
```

oder

```
{"message":"Hello World","author":"John Doe","time":1348578675087}
                                        {"message":"Cool Man","author"
```

Wenn der Browser solche Nachrichten empfängt, wird er sie nicht lesen können:

```
var json = jQuery.parseJSON(message);
```

Um Abhilfe zu schaffen, müssen wir den *TrackMessageSizeInterceptor* installieren, der der Nachricht einige Hinweise hinzufügt, damit die *atmosphere.js-onMessage*-Funktion immer mit einer gültigen Nachricht aufgerufen wird (Listing 6). Auf der Clientseite müssen wir noch die *trackMessageLength* im Request-Objekt zu setzen:

```
var request = { url: document.location.toString() + 'chat',
      contentType : "application/json",
      logLevel : 'debug',
      shared : true,
      transport : transport ,
      trackMessageLength : true,
      fallbackTransport: 'long-polling'};
```

Ab in die Cloud!

Wir können jetzt unsere erste Anwendung in der Cloud deployen. Zuerst müssen wir durch ein noch hinzuzufügendes Feature angeben, wie Nachrichten auf die Server verteilt werden, wenn sie in der Cloud deployt werden. Das hierbei entstehende Problem zeigt **Abb. 3**.

Wenn sich in diesem Szenario eine Broadcast-Aktion auf dem Tomcat-Server 1 ereignet, wird der Tomcat-Server 2 diese Nachrichten niemals erhalten. Das ist nicht nur mit Blick auf den Chat problematisch, sondern auch im Fall jeder anderen Anwendung, die in der Cloud deployt werden soll. Es kommt uns daher sehr entgegen, dass Atmosphere Cloud- oder Cluster-fähige Broadcaster unterstützt, mit denen man Nachrichten zwischen Serverinstanzen verbreiten kann. Atmosphere bietet derzeit nativen Support für bewährte Technologien wie Redis Pubsub, Hazelcast, JGroups, JMS oder XMPP (z. B. für Gmail-Server). Wir verwenden Redis Pubsub (**Abb. 4**).

Durch Redis Pubsub ist es möglich, sich mit einer Redis-Instanz zu verbinden und sich für bestimmte Topics zu subscriben. Für unsere Anwendung müssen wir lediglich ein „Chat"-Topic erzeugen und für alle unserer Server eine Subscription starten. Als Nächstes bringen wir unsere Anwendung dazu, den *RedisBroadcaster* statt des normalen Broadcasters zu verwenden. Das ist tatsächlich so einfach wie in Listing 7 dargestellt.

Indem wir den *RedisBroadcaster* hinzufügen, ermöglichen wir Message Sharing zwischen den Servern, so-

Listing 6

```
@AtmosphereHandlerService(
        path = "/chat",
        broadcasterCache = HeaderBroadcasterCache.class,
        interceptors = {AtmosphereResourceLifecycleInterceptor.class,
                        BroadcastOnPostAtmosphereInterceptor.class,
                        TrackMessageSizeInterceptor.class,
                        HeartbeatInterceptor.class})
  public class ChatRoom extends OnMessage<String> {
```

Listing 7

```
@AtmosphereHandlerService(
        path = "/chat",
        broadcasterCache = HeaderBroadcasterCache.class,
        broadcaster = RedisBroadcaster.class,
        interceptors = {AtmosphereResourceLifecycleInterceptor.class,
                        BroadcastOnPostAtmosphereInterceptor.class,
                        TrackMessageSizeInterceptor.class,
                        HeartbeatInterceptor.class})
  public class ChatRoom extends OnMessage<String> {
```

dass unsere Chat-Anwendung durch eine einzige Zeile „Cloud-freundlich“ wird. Clientseitig müssen wir nichts verändern. Wir haben nun eine voll funktionsfähige Anwendung, die

- codeunabhängig alle existierenden Webserver unterstützt
- codeunabhängig alle existierenden Browser unterstützt
- Cloud-/Cluster-fähig ist

Unsere Anwendung wird zunächst den besten Übertragungsweg zwischen Client und Server ausfindig machen. Wenn wir also beispielsweise Jetty 8 verwenden, werden die folgenden Übertragungsarten verwendet:

- Chrome 21: WebSockets
- Internet Explorer 9: Long-Polling
- Firefox 15: Server-side Events
- Safari/iOS 6: WebSockets
- Internet Explorer 10: WebSockets
- Android 2.3: Long-Polling
- Firefox 3.5 : Long-Polling

All das geschieht codeunabhängig, sodass der Entwickler sich auf die Anwendung konzentrieren kann, statt sich mit Übertragungs- und Portierungsproblemen herumschlagen zu müssen.

Fazit und Ausblick

WebSockets und Server-side Events sind auf dem Vormarsch. Beim Schreiben eines Frameworks sollten daher folgende Kriterien unbedingt berücksichtigt werden:

- Ist das API portierbar, d. h. wird es auf allen bekannten Webservern funktionieren?
- Stellt das Framework bereits einen Fallback-Mechanismus bereit? IE 7, 8 und 9 unterstützen z. B. weder WebSockets noch Server-side Events; zu unserem Nachteil sind diese Browser noch vielerorts im Einsatz.
- Ist das Framework Cloud-fähig skalierbar?
- Kann man damit problemlos Anwendungen schreiben? Hat sich das Framework gut etabliert?

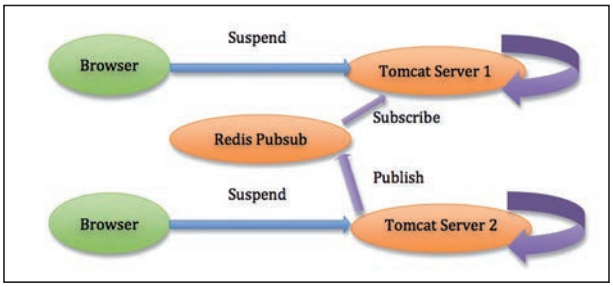


Abb. 4: Redis Pubsub

In Bezug auf Atmosphere lassen sich alle vier Fragen positiv beantworten. Noch immer nicht überzeugt? Dann gehen Sie zum Wall Street Journal [1], öffnen Sie die Seite und sehen Sie nach dem Wordnik-Logo: mehr als 60 Millionen Requests pro Tag – powered by Atmosphere!

Von der Redaktion des Java Magazins aus dem Englischen übersetzt.

Jeanfrançois Arcand ist seit achtzehn Jahren in der Softwareindustrie tätig. Er studierte vorher reine Mathematik und arbeitete für eine kanadische Forschungseinrichtung. Dort entwarf er mathematische Modelle mit C++, bis ihm jemand Java vorstellte. Jeanfrançois arbeitete fast zehn Jahre für Sun Microsystems, schrieb Grizzly, eines der ersten NIO-Frameworks, und Grizzly Comet, eines der frühen Frameworks zur Implementierung asynchroner Webanwendungen. Danach begann er mit der Arbeit am Atmosphere-Framework [2].

@jfarcand

Links & Literatur

[1] http://wsj.com

[2] http://github.com/Atmosphere/atmosphere

Webentwicklung mit dem Play-Framework

Play – jetzt wird's ernst!

Nachdem wir uns im ersten Teil unseres Tutorials mit den Grundlagen einer Play-Applikation beschäftigt haben, geht es nun weiter mit Fixtures, Bootstrapping, Forms und deren Validierung und Security.

von Yann Simon und Remo Schildmann

In Teil 1 haben wir uns mit den Grundlagen einer Play-Applikation vertraut gemacht: Struktur, Funktionsweise, Konfiguration, Routing und Templates. Darüber hinaus bietet Play aber noch eine Reihe weiterer interessanter Möglichkeiten, denen wir uns in diesem Teil zuwenden wollen. Unsere Webapplikation (To-do-Liste) implementieren wir weiter und werden am Ende dieses Teils eine zumindest teilweise gebrauchsfähige Anwendung haben. Der interessierte Leser wird festgestellt haben, dass an dem Tag, an dem Sie den ersten Teil unseres Tutorials in den Händen halten konnten (06.02.2013) die aktuelle Version 2.1 des Play-Frameworks veröffentlicht wurde. Dies wollen wir zum Anlass nehmen und in diesem Teil auf einige Anpassungen für Play 2.1 hinweisen. So wurden die Druckerpressen noch einmal für Sie angehalten und wir – die Autoren – haben die Möglichkeit bekommen, aktuell zu sein. Wir haben aus diesem Grund auch ein wenig die Themen der beiden letzten Teile variiert, was Sie uns bitte nachsehen mögen.

Fixtures

Sie erinnern sich sicher noch an die letzten Schritte. Wir haben unsere Modellklassen und entsprechende Testklassen geschrieben. Die eigentlichen Testdaten haben wir dann mittels Instanziierung unserer Modellklassen und deren Persistierung in unsere In-Memory-Datenbank eingefügt. Dies mag für einige wenige Testdaten noch annehmbar sein, für eine große Menge an Testdaten aber eher nicht. Ein weitaus eleganterer Weg ist die Verwendung von Fixtures in Play. Fixtures ermöglichen die Injektion von Daten in eine DB. In Play beschreiben wir diese Daten in YAML-Dateien. Aus diesen Daten werden dann Java-Objekte. Wichtig ist in diesem Zusammenhang der YAML-Operator „*!!*“. Dieser spezifiziert die Java-Klasse, die mit den Daten befüllt werden soll. Erstellen Sie im Verzeichnis *conf* eine YAML-Datei *test-data.yml* (Listing 1).

Anschließend wollen wir diese Daten in unseren Tests verwenden. Bei der Gelegenheit erweitern wir unsere Modellklassen (*User.java* und *ToDo.java*) um eine Methode zum Ermitteln der Anzahl an Datensätzen (Listing 2). Anschließend passen wir die Testklasse *UserTest.java* für die Verwendung der YAML-Daten an (Listing 3).

Geladen werden die Testdaten durch den Aufruf *Yaml.load()*. Den Rückgabewert casten wir auf *java.util.List* und persistieren diese mittels *Ebean.save()*. Da die Testdaten im Allgemeinen von allen Testmethoden verwendet werden, laden wir diese in unserer *setUp()*-Methode. Der Test selbst ist analog zu den bestehenden Tests.

Bootstrapping

Der Mechanismus der Dateninjektion ist für die eigentliche Applikationsentwicklung aber ebenso hilfreich. Wenn wir unsere Applikation entwickeln wollen, sollten wir so schnell wie möglich „klicken“ können und mit sinnvollen Daten arbeiten. Wenn wir aber erst die Übersichtsseite unserer Daten umsetzen (auflisten), existiert die Seite zur Dateneingabe noch nicht. In diesem Fall ist es hilfreich, wenn es bereits Testdaten gibt. Das ist mit dem Laden von YAML-Daten zur Startzeit unserer Play-Applikation möglich. Dazu hängen wir uns in den Start-up-Prozess ein. In Play geschieht dies durch eine eigene Klasse, *Global.java*, im Root Package (*app*).

Artikelserie

Teil 1: Grundlagen
Teil 2: Fixtures, Bootstrap-Klassen, Security, Validierung, Anpassungen für Play 2.1
Teil 3: JavaScript Routing, Styling, Tests, Play in Produktion

Unsere Klasse muss die Klasse *play.GlobalSettings.java* erweitern und die Methode *onStart()* implementieren (Listing 4).

Wir benötigen natürlich noch die referenzierte YAML-Datei *initial-data.yml*. Diese erstellen wir ebenfalls im *conf*-Verzeichnis. Sie können hier eine neue Datei erstellen oder die Datei *test-data.yml* kopieren. Wie Sie in Listing 4 gesehen haben, befüllen wir unsere Datenbank nur, wenn es noch keine Daten gibt. Außerdem bietet uns Play die Möglichkeit, die Umgebung zu ermitteln, in der wir unsere Applikation gestartet haben. Da wir in einer Produktionsumgebung (Kommando *play start*) normalerweise keine Daten auf diesem Weg in unserer Datenbank persistieren wollen, prüfen wir hier explizit auf unsere Entwicklungsumgebung (*play run*).

Listing 1

```
# Users

- &bob !!models.User
    email:       bob@example.com
    firstName:   Bob
    lastName:    Razowski
    password:    secret

- &jane !!models.User
    email:       jane@example.com
    firstName:   Jane
    lastName:    Krautrock
    password:    secret

# ToDos

- !!models.ToDo
    description:  Fix the documentation
    done:         false
    assignedUser: *bob

- !!models.ToDo
    description:  Prepare the beta release
    done:         false
    assignedUser: *bob

- !!models.ToDo
    description:  Buy some milk
    done:         true
    dateOfDone:   2013-01-04
    assignedUser: *bob

- !!models.ToDo
    description:  Roll to continuous deployment
    done:         true
    dateOfDone:   2012-12-25
    assignedUser: *jane

- !!models.ToDo
    description:  Deploy to cloud
    done:         false
    assignedUser: *jane
```

Unsere erste Seite

Nachdem wir unsere Applikation mit initialen Daten befüllt haben, wollen wir diese auf einer ersten einfachen Seite anzeigen. Wir erweitern dazu unsere Komponenten für die Anzeige unserer *ToDo*s. In der *routes* haben wir unsere GET-Aufrufe an die Methode *index()* unseres *Application*-Controllers geleitet. Dieser rendert dann die Seiten für die Anzeige. Für die Anzeige unserer *ToDo*s erweitern wir nun unsere *index()*-Methode.

Listing 2

```
package models;

import javax.persistence.Entity;
import javax.persistence.Id;

import play.db.ebean.Model;

@Entity
public class ...

  public static int count() {
    return find.findRowCount();
  }
}
```

Listing 3

```
package models;

import static org.junit.Assert.*;
import static play.test.Helpers.*;

import java.util.List;

import org.junit.Before;
import org.junit.Test;

import play.libs.Yaml;

import com.avaje.ebean.Ebean;

public class UserTest {
  @Before
    public void setUp() {
      start(fakeApplication(inMemoryDatabase()));

      Ebean.save((List<Object>) Yaml.load("test-data.yml"));
    }
...
  @Test
    public void canAuthenticateUserLoadedByYamlTest() {
        assertEquals(2, User.count());

        assertNotNull(User.authenticate("bob@example.com", "secret"));
        assertNull(User.authenticate("invalid@exemple.de", "secret"));
        assertNull(User.authenticate("bob@example.com", "wrong"));
    }
}
```

Abb. 1: ToDos-Übersichtsseite

Diese übergibt als Parameter für das Rendering der *index.scala.html*-View (im *views*-Verzeichnis) die Liste der *ToDo*s (Listing 5). Aktuell werden „festverdrahtet" die *ToDo*s des Users Bob angezeigt. Dies werden wir später anpassen, wenn wir Login und Authentifikation implementiert haben.

Die *index.scala.html* passen wir an das Argument (Liste von *ToDo*s) an (Listing 6).

Sie erinnern sich? Mit „@" wird Scala-Code eingeleitet. Die Anweisung in der ersten Zeile deklariert also den Parameter *todos* als Liste von *ToDo*-Objekten. Anschließend rufen wir die *main.scala.html*-View (Template-Komposition) auf und übergeben einen String (Titel) sowie ein HTML-Fragment (Content). In diesem iterieren wir über unsere Liste von *ToDo*s. Da Scala statisch typsicher ist, gestalten sich die Verwendung der Liste und der Zugriff auf unsere Objekte sehr einfach. Wichtig in diesem Zusammenhang (und um die Angst vor dem Erlernen-Müssen einer weiteren Sprache zu nehmen): Nur wenige Sprachelemente von Scala werden wir im Allgemeinen in Play-Templates verwenden. Sie müssen also keine neue Sprache lernen, sollten sich aber mit einigen Sprachelementen vertraut machen [1]. In **Abbildung 1** sehen Sie den aktuellen Stand unserer Webapplikation. Nicht überwältigend, aber noch haben wir unsere Applikation auch nicht gestylt.

Listing 4

```
import java.util.List;

import models.User;
import play.Application;
import play.GlobalSettings;
import play.Logger;
import play.libs.Yaml;

import com.avaje.ebean.Ebean;

public class Global extends GlobalSettings {

    @Override
    public void onStart(Application app) {
        // Check if the database is empty
        // and we are running in development mode
        if (app.isDev() && User.count() == 0) {
            Logger.debug("Bootstrapping with default data");
            Ebean.save((List) Yaml.load("initial-data.yml"));
        }
    }
}
```

Listing 5

```
package controllers;

import models.ToDo;
import play.mvc.Controller;
import play.mvc.Result;
import views.html.index;

public class Application extends Controller {

  public static Result index() {
    return ok(index.render(
        ToDo.findTodosByUserEMail("bob@example.com")
        ));
  }
}
```

Forms

Natürlich wollen wir unsere *ToDo*s schützen. Weder sollen diese öffentlich sichtbar noch manipulierbar sein. Ein authentifizierter *User* soll nur seine *ToDo*s anlegen, anzeigen und löschen können. Hierfür benötigen wir zuerst einen Eintrag in unsere *routes* (Listing 7), eine Controller-Klasse (Listing 8) und eine Login-Seite (Listing 9).

In der *routes* definieren wir drei Pfade. Mit einem *GET* fordern wir die Login-Seite an, mit einem *POST* (*Submit* des Login-Buttons) senden wir unsere Daten an den Controller (*controllers.Credential*). Zusätzlich wollen wir uns noch abmelden können.

Unser Controller hat für alle drei definierten Routen eine *public-static-Result*-Methode. Zusätzlich deklarieren wir noch eine statische Klasse *Login*, die die zwei für ein Login notwendigen Attribute *email* (Nutzer) und *password* besitzt. Play bietet ein *Forms-API*, das uns das Rendering, das Dekodieren und die Validierung dieser Forms abnimmt. Mit der statischen Klasse *Login* haben Sie eine solche Form angelegt. In der *login()*-Methode

Listing 6

```
@(todos: List[ToDo])

@main("Welcome to Play 2.0") {
  <h1>TODOS</h1>
  <ul>
    @for(todo <- todos) {
      <li data-todo-id="@todo.id">
        <h4>@todo.description</h4>
      </li>
    }
  </ul>
}
```

Listing 7

```
# Login
GET    /login          controllers.Credential.login
POST   /login          controllers.Credential.authenticate
GET    /logout         controllers.Credential.logout
```

des *Credential*-Controllers wird diese Form nun an das Template (*login.scala.html*) übergeben.

Unser Template muss die von der *login()*-Methode übergebene Form akzeptieren (Listing 9, 1. Zeile). Im *Body* rendern wir unsere Form. *@helper.form(Controller.methode())* wird durch Play aufgelöst zu *<form action="/login" method="POST" >*. Play beherrscht auch ein Reverse-Routing, sodass die in der *routes* definierten Routen in die andere Richtung verwendet werden können. In diesem Fall erfolgt der Methodenaufruf zum definierten URL (*/login*). Sie erinnern sich sicher noch: Die *routes* ist eine Art API für unsere Applikation. Änderungen hier müssen somit nicht an verschiedenen Stellen im Code nachgezogen werden.

Im Form-Bereich fordern wir zum Login auf (*<h1>Sign in</h1>*) und sehen einen Bereich für Fehlermeldungen und einen für Erfolgsmeldungen vor. Letztlich haben wir noch zwei Felder, einen für *email* (Username) und einen für *password*. Diese werden durch Play an unsere Form *controllers.Credential.Login* über die Namen gebunden. Der SUBMIT-Button schickt die Daten via *POST* an die *authenticate()*-Methode des *Credential*-Controllers.

Validierung

Zum Validieren einer Form können die Attribute dieser Form durch entsprechende *Constraints* annotiert werden. Dafür bringt Play die Klasse *play.data.validation.Constraints* mit, die eine Vielzahl solcher Annotationen zum Validieren enthält [2]. Alternativ kann eine Form auch

Listing 8

```
package controllers;

import models.User;
import play.data.Form;
import play.mvc.Controller;
import play.mvc.Result;
import views.html.login;

public class Credential extends Controller {

    public static class Login {

        public String email;
        public String password;

        public String validate() {
            if (User.authenticate(email, password) == null) {
                return "Invalid user or password";
            }
            return null;
        }
    }

    public static Result login() {
        return ok(
            login.render(form(Login.class))
        );
    }

    public static Result authenticate() {
        Form<Login> loginForm = form(Login.class).bindFromRequest();
        if (loginForm.hasErrors()) {
            return unauthorized(login.render(loginForm));
        } else {
            session().clear();
            session("email", loginForm.get().email);
            return redirect(
                routes.Application.index()
            );
        }
    }

    public static Result logout() {
        return TODO;
    }
}
```

Listing 9

```
@(form: Form[Credential.Login])

<!DOCTYPE html>

<html>
  <head>
    <title>Login</title>
    <link rel="stylesheet" media="screen"
                    href="@routes.Assets.at("stylesheets/main.css")">
    <link rel="shortcut icon" type="image/png"
                     href="@routes.Assets.at("images/favicon.png")">
  </head>
  <body>
    @helper.form(routes.Credential.authenticate) {
      <h1>Sign in</h1>
      @if(form.hasGlobalErrors) {
        <p class="error">
          @form.globalError.message
        </p>
      }
      @if(flash.contains("success")) {
        <p class="success">
          @flash.get("success")
        </p>
      }
      <p>
        <input type="email" name="email" placeholder="Email"
                                        value="@form("email").value">
      </p>
      <p>
        <input type="password" name="password"
                                            placeholder="Password">
      </p>
      <p>
        <button type="submit">Login</button>
      </p>
    }
  </body>
</html>
```

die Methode *public String validate()* besitzen (in unserer *Login*-Klasse), die ebenfalls beim Binden der Form-Attribute ausgeführt wird. Diese gibt im Fall einer fehlerhaften Validierung einen String mit dem Fehlertext zurück und im Erfolgsfall *null*. Wir prüfen in dieser Methode, ob die Login-Daten korrekt sind. Sind sie es nicht, geben wir eine Fehlermeldung zurück. In unserer *authenticate()*-Methode wird die Form also (beim Binden) validiert. Im Fehlerfall zeigen wir die Login-Seite erneut (mit dem Status 401 – *unauthorized*) an. Wir übergeben die Form mit den aktuellen Daten und der Fehlermeldung, sodass diese Daten im Template aus der Form gelesen und angezeigt werden (**Abb. 2**). Ist die Formvalidierung erfolgreich, so speichern wir den *User* (die E-Mail-Adresse) an der Session (Erklärung *Play-Session*) und routen auf unsere *ToDo*-Seite weiter. Das in der Session abgelegte Attribut benötigen wir zum weiteren Ermitteln des angemeldeten *User*-Objekts.

Jetzt wollen wir unsere neuen Funktionalitäten natürlich testen. Erstellen Sie dazu im *test/controllers*-Package eine Klasse *CredentialTest.java*. Diese hat zwei Testmethoden, eine für eine erfolgreiche und eine für eine fehlgeschlagene Anmeldung (Listing 10). Grundlage ist die Datei *test-data.yml*, die wir am Anfang geschrieben haben.

Wie bereits erwähnt, haben wir in Play die Möglichkeit des Reverse Routing. In unserer Testklasse nutzen wir diese und lassen uns eine Referenz auf unsere Controller-Methode geben (*controllers.routes.ref.Credential.authenticate()*). Weiter müssen wir noch einen *play.test.Helpers.fakeRequest()* erzeugen, dem wir eine Form mit *email* und *password* übergeben. Die *authenticate()*-Methode können wir dann mit dem *fakeRequest* per *callAction()* ausführen. Letztlich verwenden wir noch die Helper *play.test.Helpers.status()* und *play.test.Helpers.session()*, um den Status und die Informationen aus der Session (dem Session-Cookie) des *Result* zu bekommen. Dabei greifen wir nicht direkt auf das *Result* zu. Der Grund dafür ist, dass ein Aufruf bspw. asynchron sein kann. Für den Zugriff auf so ein *Result* benötigt Play die Helper. An dieser Stelle sei noch erwähnt, dass es weitere Helper gibt, die beim Testen hilfreich sind.

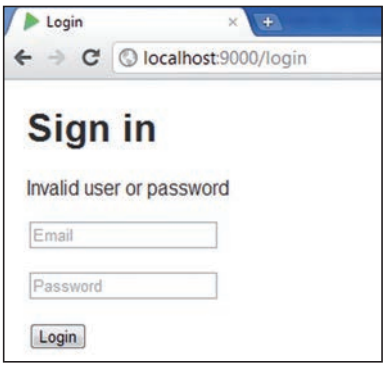


Abb. 2: Login-Seite nach fehlerhafter Anmeldung

Authentication

Nachdem wir die Möglichkeit implementiert haben, dass sich ein User anmeldet, müssen wir nun noch die *Action*-Methoden unserer Controller absichern. Aktuell können wir nämlich durch direkte Eingabe eines URLs die korrespondierende Controller-Action ausführen, ohne angemeldet zu sein – probieren Sie es aus. Play bie-

Listing 10

```
package controllers;

import static org.junit.Assert.*;
import static play.test.Helpers.*;

import java.util.List;

import org.junit.*;

import play.libs.Yaml;
import play.mvc.Result;
import play.test.FakeApplication;

import com.avaje.ebean.Ebean;
import com.google.common.collect.ImmutableMap;

public class CredentialTest {

    private FakeApplication fakeApplication;

    @Before
    public void setUp() {
        fakeApplication = fakeApplication(inMemoryDatabase());
        start(fakeApplication);
        Ebean.save((List) Yaml.load("test-data.yml"));
    }

    @Test
    public void authenticateSuccess() {
        Result result = callAction(
                controllers.routes.ref.Credential.authenticate(),
                fakeRequest().withFormUrlEncodedBody(ImmutableMap.of(
                        "email", "bob@example.com",
                        "password", "secret"))
        );
        assertEquals(303, status(result));
        assertEquals("bob@example.com", session(result).get("email"));
    }

    @Test
    public void authenticateFailure() {
        Result result = callAction(
                controllers.routes.ref.Credential.authenticate(),
                fakeRequest().withFormUrlEncodedBody(ImmutableMap.of(
                        "email", "bob@example.com",
                        "password", "badpassword"))
        );
        assertEquals(401, status(result));
        assertNull(session(result).get("email"));
    }

    @After
    public void tearDown() {
        stop(fakeApplication);
    }
}
```

Play-Session

Eine Play-Applikation ist im Gegensatz zu einer klassischen JEE-Anwendung standardmäßig *stateless* (zustandslos). Der Zustand wird nicht serverseitig gespeichert. Damit kann eine Play-Applikation problemlos in einem Cluster ohne *session affinity (sticky session)* laufen (*scale out*). Generell gibt es verschiedene Möglichkeiten, einen Zustand zu speichern, bspw. durch Datenbankpersistierung oder clientseitig (JavaScript, HTML5-Webdatenbank). Play ermöglicht auf einfache Weise die Speicherung des Zustands in einem Cookie. Die Applikation behält somit ihren Stateless-Charakter. Dieser Cookie ist signiert, sodass Manipulationen am Inhalt unmöglich sind. Man sollte aber nicht zu viele Informationen in den Cookie legen, da letzterer nur eine begrenzte Menge an Daten speichern kann. Für reine Cache-Zwecke bietet Play ein eigenes Cache-API (*play.cache.Cache*).

tet uns zum Absichern die Möglichkeit der Action-Komposition. Action-Komposition heißt hier, dass mehrere Action-Methoden hintereinander aufgerufen werden. *Request* und *Result* können also mehrere dieser Methoden durchlaufen. Jede davon kann den *Request* oder das *Result* verändern. Darüber hinaus können diese Methoden auch eigenständig statt den *Request* weiterzuleiten ein *Result* erzeugen und dieses zurückgeben.

Play bringt eine *Authenticator*-Klasse mit (*play.mvc.Security.Authenticator*). Diese erweitern wir, um unsere Authentifikationslogik zu implementieren. Dazu legen Sie eine Klasse *Secured.java* im Package *app/controllers* an (Listing 11).

Wir haben zwei Methoden implementiert. Die Methode *getUsername()* gibt den angemeldeten *User* zurück (wir lesen dazu den in der Session gespeicherten Wert aus, den wir dort bei einem erfolgreichen Login hineingeschrieben haben). Gibt diese Methode einen Wert zurück, so leitet unsere *Authenticator*-Klasse den Request weiter. Gibt diese Methode *null* zurück, so geht die *Authenticator*-Klasse davon aus, dass kein *User* angemeldet ist. In diesem Fall blockt die *Action*-Methode den aktuellen *Request* und ruft stattdessen die Methode *onUnauthorized()*. Dies ist die zweite Methode, die wir implementiert haben. An dieser Stelle machen wir ein *Redirect* zu unserer Login-Seite. Jetzt wollen wir unsere *Authenticator*-Klasse noch verwenden. Wir können ganze Klassen oder einzelne Methoden annotieren, sodass eine Action-Komposition mit unserer *Authenticator*-Klasse stattfindet. Sichern Sie die *index()*-Methode Ihrer *Application*-Klasse mit der Annotation *@Security.Authenticated(Secured.class)* ab (Listing 12).

Jeder Request an die *index()*-Methode wird nun zuvor an die *getUsername()*-Methode unserer *Secured*-Klasse geleitet. Je nach Ergebnis wird dann die Seite mit den *ToDo*s angezeigt oder auf die Login-Seite umgeleitet.

Natürlich implementieren wir auch für unsere Authentifizierungsmethodentests. Dazu erweitern Sie die Klasse *test.controllers.CredentialTest.java*. Auch hier implementieren wir wieder zwei Tests, einen für den Positiv- und einen für den Negativfall (Listing 13).

Wenn wir die *index()*-Methode mit einem in der *Session* gespeicherten *User* aufrufen, dann bekommen wir

Listing 11

```
package controllers;

import play.mvc.Http;
import play.mvc.Result;
import play.mvc.Security;

public class Secured extends Security.Authenticator {

    @Override
    public String getUsername(Http.Context ctx) {
        return ctx.session().get("email");
    }

    @Override
    public Result onUnauthorized(Http.Context ctx) {
        return redirect(routes.Credential.login());
    }
}
```

Listing 12

```
public class Application extends Controller {

  @Security.Authenticated(Secured.class)
  public static Result index() {
...
```

Listing 13

```
public class CredentialTest {

  ...

  @Test
  public void authenticated() {
      Result result = callAction(
              controllers.routes.ref.Application.index(),
              fakeRequest().withSession("email", "bob@example.com")
      );
      assertEquals(200, status(result));
  }

  @Test
  public void notAuthenticated() {
      Result result = callAction(
              controllers.routes.ref.Application.index(),
              fakeRequest()
      );
      assertEquals(303, status(result));
      assertEquals("/login", header("Location", result));
  }

  ...
}
```

die angeforderte Seite als Ergebnis. Der HTTP-Status ist also 200: „ok“. Rufen wir die *index()*-Methode ohne einen angemeldeten *User*, so werden wir auf die Login-Seite weitergeleitet. In diesem Fall ist der HTTP-Status 303, und wir bekommen den URL für den *Redirect* im Header der Antwort geliefert.

Bestimmen des aktuell angemeldeten Nutzers

Für unsere Applikation ist es nicht nur wichtig, dass ein Nutzer sich autorisieren muss, sondern auch, welcher Nutzer angemeldet ist. Jedem Nutzer sollen nur „seine“ *ToDo*s angezeigt werden, wir müssen also den aktuell angemeldeten Nutzer ermitteln und dann die für diesen Nutzer relevanten Daten filtern. Den angemeldeten Nutzer ermitteln wir am *Request* mit *request().username()*. Durch die Annotation *@Security.Authenticated(Secured.class)* (der *index()*-Methode) wird der Methodenaufruf an die Methode *public Result call(Context ctx)* der Klasse *Security.AuthenticatedAction* geleitet. In dieser wird die Methode *getUsername()*unserer *Secured*-Klasse gerufen, also der *User* aus der *Session* gelesen. In der *AuthenticatedAction* wird auf unsere Login-Seite weitergeleitet, wenn es keinen *User* in der *Session* gibt. Es wäre also grundsätzlich möglich, den angemeldeten Nutzer direkt aus der *Session* zu lesen (statt mittels *request().username()*).

Im *Application*-Controller ermitteln wir zurzeit nur die *ToDo*s für den *User* Bob (Listing 5). Implementieren Sie nun die Filterung der *ToDo*s nach dem angemeldeten Nutzer (Listing 14).

Wir ermitteln nun die *ToDo*s anhand des angemeldeten *User*-Objekts und übergeben diese Liste. Zusätzlich ermitteln wir das *User*-Objekt selbst und geben dieses ebenfalls zurück (für den Zugriff muss das Attribut *find* in der Klasse *models.User.java* die Sichtbarkeit *public* bekommen, oder eine entsprechende *getUser()*-Methode implementiert werden). Das ermöglicht uns eine Anzeige der Daten des angemeldeten Nutzers (Name, Vorname etc.). Dazu müssen Sie noch die *index.scala.html* anpassen (Listing 15). Diese akzeptiert nun den zweiten Parameter *user* und gibt ihn an die *main.scala.html* weiter, die Sie ebenfalls erweitern (Listing 16).

Vor- und Nachname des angemeldeten Nutzers werden nun im Header angezeigt. Ebenfalls platzieren wir einen Link zum Abmelden (**Abb. 3**).

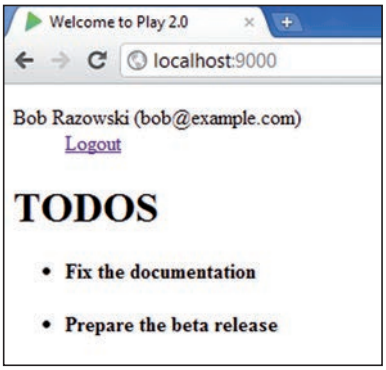


Abb. 3: ToDos mit Header-Zeile: Namen und Logout-Link

Logout

In Teil 3 dieses Tutorials werden Sie noch die Funktionalität zum Abmelden eines angemeldeten Nutzers implementieren. Den Link dafür haben Sie soeben im Header unserer HTML-Seiten eingebaut. Die *routes*-Datei ist bereits vollständig, und auch die *logout()*-Methode ist im *Credential*-Controller vorhanden. Diese können Sie jetzt fertigstellen (Listing 17).

Listing 14

```
@Security.Authenticated(Secured.class)
public static Result index() {
      String email = request().username();
   return ok(index.render(
            ToDo.findTodosByUserEMail(email),
            User.find.byId(email)
));
 }
```

Listing 15

```
@(todos: List[ToDo], user: User)

@main("Welcome to Play 2.0", user) {
   ...
```

Listing 16

```
@(title: String, user: User)(content: Html)

<!DOCTYPE html>

<html>
   <head>
      <title>@title</title>
      <link rel="shortcut icon" type="image/png"
            href="@routes.Assets.at("images/favicon.png")">
   </head>
   <body>
      <header>
         <dl id="user">
          <dt>@user.firstName @user.lastName
            <span>(@user.email)</span>
          </dt>
          <dd>
            <a href="@routes.Credential.logout()">Logout</a>
          </dd>
         </dl>
      </header>
      <div class="container">
         @content
      </div>
   </body>
</html>
```

Listing 17

```
public static Result logout() {
   session().clear();
   flash("success", "You've been logged out");
   return redirect(
         routes.Credential.login()
   );
}
```

Play-API

Play unterstützt sowohl Java-, als auch Scala-Code. Das API für Java ist unter dem Package *play.** verfügbar. Das API für Scala ist für Scala reserviert unter *play.api.** und sollte nicht im Java-Code verwendet werden [3]. So gibt es bspw. die Klasse *Result* sowohl unter *play.mvc.Result* (für Java) als auch unter *play.api.mvc.Result* (für Scala).

Zum Abmelden löschen wir den angemeldeten Nutzer aus der Session. Anschließend leiten wir auf die Login-Seite weiter. Vor dem Weiterleiten nutzen wir eine andere in Play verfügbare Möglichkeit, den *Flash-Scope*. Dieser speichert Informationen bis zum nächsten *Request*. Wir legen den Hinweis über ein erfolgreiches Abmelden unter dem Schlüssel "*success*" in den *Flash-Scope*. Diesen lesen wir in unserer Login-Seite aus und zeigen die Informationen über der Anmelde-Form an (Listing 9), *@if(flash.contains("success")) {*.

Anpassungen an Play 2.1.0

Für den Fall, dass Sie zum Zeitpunkt der Verfügbarkeit von Play 2.1.0 mit der Entwicklung beginnen, können Sie gleich diese Version verwenden. Für die von uns, die mit der Verion 2.0.4 gestartet sind und die ebenfalls auf 2.1 migrieren wollen, soll es an dieser Stelle einige Hinweise hierfür geben. Darüber hinaus gibt es zwischen den beiden Versionen auch einen Unterschied in unseren implementierten Klassen, den wir ebenfalls beschreiben werden.

Für eine Migration müssen die drei Dateien im *project*-Verzeichnis angepasst werden. In der *build.properties* ändern Sie die *sbt.version* von 0.11.3 auf 0.12.2. In der *Build.scala* muss ein Import angepasst werden. Zusätzlich gibt es in den *appDependencies* drei neue Einträge. Letztlich muss die *main*-Zuweisung ebenfalls angepasst werden (Listing 18). Anschließend ändern Sie die *plugins.sbt*. Hier ändert sich die konfigurierte Versionsnummer *addSbtPlugin* von 2.0.4 auf 2.1.0. Um einen sauberen Projektstand zu erhalten leeren Sie noch das *target*-Verzeichnis.

Damit haben wir einen Stand, der dem nach dem Anlegen eines neuen Projekts der Version 2.1.0 entspricht. Das Projekt muss noch für unsere IDE (Eclipse) vorbereitet bzw. aktualisiert werden. Auch hier gibt es eine Änderung zu der Version 2.0.4. Das Kommando lautet nun *play eclipse* (und nicht mehr *play eclipsify*). Unser Projekt ist damit für die (Weiter)entwicklung mit Play 2.1 bereit.

Jetzt müssen wir zwei unserer bestehende Controller-Klassen für Play 2.1 anpassen. Es ist noch ein zusätzlicher Import *import play.data.Form;* in der Klasse *app.controllers.ToDos.java* notwendig. Weiterhin haben wir keinen direkten Zugriff mehr auf *form*, sondern per *Form.form*. Beheben Sie die (in der IDE angezeigten) Fehler in den Klassen *app.controllers.Credential.java* (2 Fehler) und *app.controllers.ToDos.java* (1 Fehler) (exemplarisch für *ToDos.java* in Listing 19). Mit dieser letzten Änderung haben Sie die aktuelle Webapplikation erfolgreich nach Play 2.1 migriert.

Damit endet der zweite Teil, und die Autoren hoffen, dass Ihnen auch dieser gefallen hat. Das Styling werden wir in Teil 3 behandeln (mit LESS CSS-Dateien erstellen und mit CoffeeScript JavaScript erzeugen lassen). Und natürlich werden wir unsere Applikation vervollständigen: Hinzufügen von *ToDo*s und ihre Erledigung. Das Thema „Vorbereitung für die Produktion“ soll dann damit das Tutorial als Ganzes abrunden.

Listing 18

```
import PlayProject._
import play.Project._

val appDependencies = Seq(
  // Add your project dependencies here,
  javaCore, javaJdbc, javaEbean
)

val main = play.Project(appName, appVersion,
                                        appDependencies).settings(
  // Add your own project settings here
)
```

Listing 19

```
...
import play.data.Form;

@Security.Authenticated(Secured.class)
public class ToDos extends Controller {

    public static Result add() {
        String description =
                        Form.form().bindFromRequest().get("description");
...
```

Yann Simon ist Softwareentwickler bei leanovate in Berlin und begeistert sich für Open-Source- und Webtechnologien.

simon_yann http://bit.ly/Y9fBik

Remo Schildmann ist Software Engineer/Softwarearchitekt bei der adesso AG in Stralsund. Er ist seit mehr als dreizehn Jahren in der Java-Entwicklung tätig und hat bereits einige Projekte mit dem Play-Framework umgesetzt.

Links & Literatur

[1] http://www.playframework.org/documentation/2.0.4/JavaTemplates

[2] http://www.playframework.org/documentation/2.0.4/JavaForms

[3] http://www.playframework.org/documentation/2.0/JavaHome

Anzeige

Anzeige

Anzeige

Anzeige

Wicket 6 und JSR 303 – Bean Validations

Aber nicht die Bohne!

Wicket 6.4.0 ist veröffentlicht und ein Geschenk an die Wicket-Community. Hinter diesem vermeintlich kleinen Versionssprung versteckt sich eine kleine Perle. Wicket bietet ab 6.4.0 Out-of-the-Box-Support für JSR 303 Bean Validations.

von Martin Dilger

JSR 303 Bean Validations wurde 2009 final freigegeben. Mithilfe von Bean Validations lassen sich Domänenobjekte mit Metainformationen annotieren und validieren. Hier lohnt sich ein genauer Blick. Zunächst erzeugen wir uns ein einfaches Maven-Archetype-Projekt, um Bean Validations mit Wicket in Aktion zu erleben (Listing 1).

Der Maven Archetype generiert uns bereits ein vollständiges Wicket-Projekt mit einer Startklasse, die einen Embedded-Jetty-Server startet. Im Wicket-Projekt ist für den Bean-Validation-Support ein neues Modul entstanden. Dieses Modul deklarieren wir als Abhängigkeit in der Maven *POM* (Listing 2). Die Versionierung folgt noch nicht dem Wicket-Standard, da das Modul derzeit als experimental deklariert ist.

Zusätzlich brauchen wir eine Implementierung der JSR-303-Spezifikation. Wir verwenden hierfür den Hibernate Validator (Listing 3).

Um Bean Validations in unserer Wicket-Anwendung verwenden zu können, ist zunächst nur eine einzige Zeile Code in der *init*-Methode der *WicketApplication*-Klasse notwendig. Diese Zeile registriert eine Instanz der neu hinzugekommenen *BeanValidationConfiguration* in den Metadaten der Applikation (Listing 4).

Jetzt brauchen wir noch ein annotiertes Domänenobjekt, das es zu validieren gilt (Listing 5). Konstruktoren und Getter sind der Übersichtlichkeit halber weggelassen. Mit den hier definierten Constraints stellen wir sicher, dass Name, E-Mail und Geburtsdatum gesetzt sein müssen (*@NotNull*), dass die E-Mail ein korrektes Format hat (*@Pattern*), die Telefonnummer nur aus Ziffern besteht (*@Pattern*) und das Geburtsdatum in der Vergangenheit liegt (*@Past*).

In der Datei *HomePage.html* ersetzen wir das von Maven generierte Body Tag durch das Markup in Listing 6.

In der Klasse *HomePage* implementieren wir ein einfaches Formular (Listing 7).

Für jedes Attribut im Trainer-Objekt erzeugen wir ein Textfeld mit einem entsprechenden *PropertyModel*. Die einzige Besonderheit ist, dass wir jedem Textfeld zusätzlich einen *PropertyValidator* hinzufügen. Der *PropertyValidator* stammt aus dem Wicket-Bean-Validation-Modul und sorgt dafür, dass die JSR-303-Annotationen

Listing 1

```
mvn archetype:generate -DarchetypeGroupId=org.apache.wicket
-DarchetypeArtifactId=wicket-archetype-quickstart
-DarchetypeVersion=6.4.0 -DgroupId=de.effectivetrainings
-DartifactId=bean-validations -DarchetypeRepository=https://repository.
apache.org/ -DinteractiveMode=false
```

Listing 2

```
<dependency>
  <groupId>org.apache.wicket</groupId>
  <artifactId>wicket-bean-validation</artifactId>
  <version>0.5</version>
</dependency>
```

Listing 3

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>4.3.0.Final</version>
</dependency>
```

Listing 4

```
@Override
public void init()
{
  super.init();
  new BeanValidationConfiguration().configure(this);
}
```

am Domänenobjekt ausgelesen und validiert werden. Startet man die Anwendung und schickt das Formular testweise ab, ergibt sich das Bild in **Abbildung 1**.

Listing 5

```
public class Trainer implements Serializable {

@NotNull
private String name;

@Pattern(regexp = "^[_A-Za-z0-9-]+(\\.[_A-Za-z0-9-]+)*
                          @[A-Za-z0-9-]+(\\.[A-Za-z0-9-]+)*((\\.[A-Za-z]{2,}){1}$)")
@NotNull
private String email;

@Pattern(regexp = "[0-9]+")
private String phone;

@Past
@NotNull
private Date birthDay;
}
```

Listing 6

```
<body style="padding: 25px">
      <div wicket:id="feedback"/>
 <form wicket:id="form">
         <input type="text" wicket:id="name"/> <br/>
         <input type="text" wicket:id="email"/> <br/>
         <input type="text" wicket:id="phone"/> <br/>
         <input type="text" wicket:id="birthDay"/> <br/>
   <input type="submit"/>
 </form>
</body>
```

Listing 7

```
public HomePage(final PageParameters parameters) {
  super(parameters);
  IModel<Trainer> trainerModel = Model.of(new Trainer());
  add(new FeedbackPanel("feedback"));
  Form<Trainer> form = new Form<Trainer>("form", trainerModel);
  form.add(new TextField("name", new PropertyModel(trainerModel, "name"))
.add(new PropertyValidator()));
  form.add(new TextField("email", new PropertyModel(trainerModel, "email"))
.add(new PropertyValidator()));
  form.add(new TextField("phone", new PropertyModel(trainerModel, "phone"))
.add(new PropertyValidator()));
  form.add(new TextField("birthDay", new PropertyModel(trainerModel, "birthDay"))
.add(new PropertyValidator()));
  add(form);
}
```

Der Hibernate Validator verrichtet seine Arbeit. Die Constraints werden validiert. Bisher werden aber nur die Standardfehlermeldungen ausgegeben. Für den Fall *@NotNull* ergeben diese meistens noch Sinn. Im Fall der invaliden E-Mail mit der *@Pattern*-Annotation wird es für den normalen User schon völlig unverständlich.

Feedback Messages

Die erste und einfachste (aber auch unflexibelste) Möglichkeit wäre, die Fehlermeldungen direkt an den Constraint-Annotationen über das Attribut *message* zu hinterlegen (Listing 8).

Startet man die Anwendung erneut und schickt das Formular mit absichtlich falschen Werten ab, ergibt sich das Bild aus **Abbildung 2**.

Ideal ist das aber nicht, denn die Benutzeroberfläche sollte festlegen, wann in welchem Kontext welche

Listing 8

```
@NotNull(message = "Bitte geben Sie Ihren Namen ein.")
private String name;

@Pattern(message = "Die E-Mailadresse ist nicht gültig",
regexp = "^[_A-Za-z0-9-]+(\\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+
                          (\\.[A-Za-z0-9-]+)*((\\.[A-Za-z]{2,}){1}$)")
@NotNull(message = "Bitte geben Sie Ihre E-Mailadresse ein")
private String email;

@Pattern(message = "Ihre Telefonnummer sollte aus Ziffern
                                        bestehen",regexp = "[0-9]+")
private String phone;

@Past(message = "Sie können nicht in der Zukunft Geburtstag haben")
@NotNull(message = "Bitte geben Sie Ihr Geburtsdatum ein.")
private Date birthDay;
```

Listing 9

```
@NotNull
private String name;

@Pattern(message = "{email.invalid}",
regexp = "^[_A-Za-z0-9-]+(\\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+
                          (\\.[A-Za-z0-9-]+)*((\\.[A-Za-z]{2,}){1}$)")
@NotNull
private String email;

@Pattern(message = "{phone.invalid}",regexp = "[0-9]+")
private String phone;

@Past(message = "{birthDay.invalid}")
@NotNull
private Date birthDay;
```

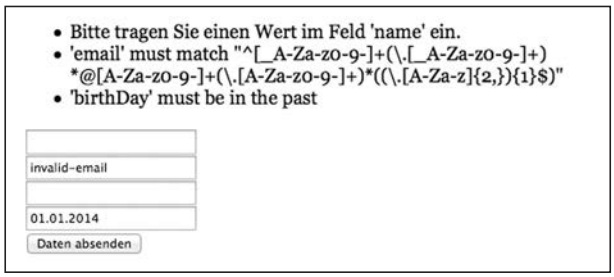


Abb. 1: JSR-303-Standardfehlermeldungen – lesbar, aber nicht gut

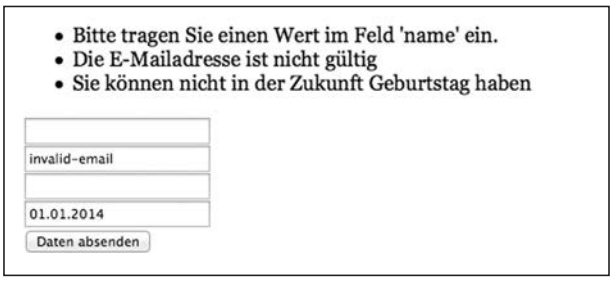


Abb. 2: Meldungen direkt an Annotationen

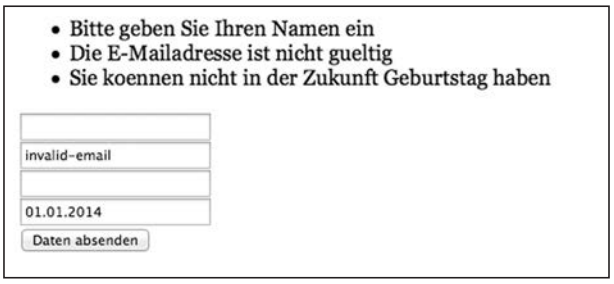


Abb. 3: Meldungen aus Platzhaltern

Fehlermeldung angezeigt wird. Das Domänenobjekt *Trainer* ist so nicht ohne Weiteres kontextübergreifend wiederverwendbar. Viel besser ist die Verwendung der in der JSR-303-Spezifikation definierten Platzhalter (Listing 9).

Die JSR-303-Spezifikation definiert, dass Fehlermeldungen mit Platzhaltern befüllt sein können. Platzhalter haben das Format *{ value }*. Wir verwenden die Platzhalter einfach als Schlüssel in die in Wicket üblicherweise verwendeten Property-Dateien für Texte.

Zuletzt müssen wir nur noch die Datei *HomePage.properties* im Package der Klasse *HomePage.java* definieren (Listing 10).

Wieso aber beispielsweise *email.Required*? Wir haben keinen Platzhalter mit diesem Namen definiert. Attribute, die mit *@NotNull* annotiert sind, werden automatisch als Pflichtfelder (mittels *setRequired*) markiert. Der Schlüssel für Validierungsfehlermeldungen von Pflichtfeldern ist standardmäßig *<Komponenten-ID>.Required*. Es ist also gar nicht notwendig, einen eigenen Schlüssel zu definieren, denn für diesen Fall reichen die Wicket-Bordmittel. Startet man die Anwendung erneut, ergibt sich das Bild aus **Abbildung 3**.

Ganz so einfach ist es leider nicht

Die bisherige Umsetzung war sehr einfach, deckt aber nicht alle Use Cases ab. Gerade für Formulare bietet sich zur Vereinfachung die Verwendung von *CompoundPropertyModels* an. Eine Umsetzung des Formulars mit einem *CompoundPropertyModel* ist in Listing 11 zu sehen.

Dadurch, dass wir der Formkomponente ein *CompoundPropertyModel* zuweisen, ist es nicht mehr notwendig, jeder Formularkomponente innerhalb der Form ein eigenes *PropertyModel* zuzuweisen. Die Auflösung geschieht automatisch über die Wicket-ID der einzelnen Komponenten.

Listing 10

```
name.Required=Bitte geben Sie Ihren Namen ein
email.Required=Bitte geben Sie Ihre E-Mailadresse ein.
email.invalid = Die E-Mailadresse ist nicht gueltig
phone.invalid=Ihre Telefonnummer sollte aus Ziffern bestehen
birthDay.invalid=Sie koennen nicht in der Zukunft Geburtstag haben
birthDay.Required=Bitte geben Sie Ihr Geburtsdatum ein
```

Die Anweisung *form.add(new TextField("name"))* ist also äquivalent mit der Anweisung *form.add(new TextField("name", new PropertyModel(trainerModel, "name"))*. Starten wir die Anwendung erneut, kommt zunächst ein böses Erwachen, denn wir sehen nicht das Formular, sondern die folgende Fehlermeldung:

Listing 11

```
public HomePage(final PageParameters parameters) {
        super(parameters);
        IModel<Trainer> trainerModel = Model.of(new Trainer());
        add(new FeedbackPanel("feedback"));
        Form<Trainer> form = new Form<Trainer>("form",
                                        new CompoundPropertyModel<Trainer>(trainerModel));
        form.add(new TextField("name")
                .add(new PropertyValidator()));
        form.add(new TextField("email")
                .add(new PropertyValidator()));
        form.add(new TextField("phone")
                .add(new PropertyValidator()));
        form.add(new TextField("birthDay")
                .add(new PropertyValidator()));
        add(form);
    }
```

Listing 12

```
public HomePage(final PageParameters parameters) {
        super(parameters);
        IModel<Trainer> trainerModel = Model.of(new Trainer());
        add(new FeedbackPanel("feedback"));
        Form<Trainer> form = new Form<Trainer>("form",
                                        new CompoundPropertyModel<Trainer>(trainerModel));
        form.add(new TextField("name")
                .add(new PropertyValidator(new Property(Trainer.class,"name"))));
        form.add(new TextField("email")
                .add(new PropertyValidator(new Property(Trainer.class,"email"))));
        form.add(new TextField("phone")
                .add(new PropertyValidator(new Property(Trainer.class,"phone"))));
        form.add(new TextField("birthDay")
                .add(new PropertyValidator(new Property(Trainer.class,"birthDay"))));
        add(form);
    }
```

Listing 13

```
public enum License {
    NONE,EBAY,CERTIFICATION;

    public boolean isValid(){
        return EBAY != this;
    }

    public boolean isAvailable(){
        return NONE != this;
    }
}
//Trainer.java
    @LicenseValid(message = "{license.invalid}“)
    @LicenseAvailable(message = "{license.unavailable}")
    @NotNull
    private License license;
```

```
Could not resolve Property from component: [TextField [Component id =
name]]. Either specify the Property in the constructor or use a model that
works in combination with a IPropertyResolver to resolve the Property
automatically
```

Ein JSR-303-kompatibler Validator benötigt für die Validierung die Klasse des zu validierenden Objekts, den Namen des Attributs, den zu validierenden Wert und gegebenenfalls die zugeordneten Validierungsgruppen, die wir später noch betrachten werden. Der Validator muss zur Laufzeit die JSR-303-Annotationen laden und den Wert des jeweiligen Attributs gegen den definierten Constraint validieren (beispielseise das E-Mail-Attribut gegen die @*Pattern*-Annotation).

Um die notwendigen Informationen zur Laufzeit auswerten zu können, benötigen wir ein Model vom Typ *IPropertyReflectionAwareModel* (also beispielsweise ein *PropertyModel*). In unserem aktuellen Fall haben die Formularkomponenten jedoch gar kein Model, sondern basieren auf dem *CompoundPropertyModel* der Formkomponente. Der *PropertyValidator* beschwert sich also zu Recht.

Listing 14

```
//LicenseValid.java
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR,
                                                        PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {LicenseValidator.class})
public @interface LicenseValid {

    String message() default
                        "{de.effectivetrainings.LicenseValid.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default {};
}

//LicenseAvailable.java
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR,
                                                        PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {LicenseAvailableValidator.class})
public @interface LicenseAvailable {

    String message() default
                        "{de.effectivetrainings.LicenseAvailable.message}";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default {};
}
```

Es ist aber glücklicherweise sehr einfach, dem *PropertyValidator* eine kleine Starthilfe zu geben. Wir teilen ihm einfach selbst mit, worum es sich bei dem zu validierenden Feld handelt. Hierfür übergeben wir dem *PropertyValidator* wie in Listing 12 einfach ein Objekt vom Typ *Property* mit dem Typ des zu validierenden Objekts und dem Namen des Attributs.

Wir können uns wieder entspannt zurücklehnen. Sobald wir die Anwendung erneut starten, sehen wir wieder das Formular, und alles funktioniert wie zuvor. Es ist übrigens sehr interessant und lehrreich, wie das Auflösen der Typinformation in Wicket implementiert ist. Bei Interesse lege ich Ihnen die Klasse *DefaultPropertyResolver* von Wicket ans Herz.

Gruppenzwang mit Validation Groups und eigenen Constraints

Ein letztes wichtiges Konzept aus der JSR-303-Spezifikation sind die Validation Groups. Über Validation Groups ist es möglich, Feldvalidierungen zu gruppieren. Ein typischer Use Case ist die Unterscheidung zwischen einfachen und komplexen Validierungen. Beispielsweise kann es Felder geben, die für die Validierung auf externe Systeme zugreifen müssen, was langsam und teuer sein kann. Diese Zugriffe möchten wir nur machen, wenn alle einfachen Validierungen bereits erfolgreich waren.

Nehmen wir an, jeder Trainer muss sich lizensieren lassen. Die Überprüfung, ob eine Lizenz gültig ist, ist teuer, da ein externer Service-Provider angefragt werden muss. Wir möchten also sicherstellen, dass der Lizenzcheck erst gemacht wird, wenn alle einfachen Überprüfungen erfolgreich waren. Wir implementieren hierfür eine neue *enum License* (Listing 13) und weisen diese dem Trainer zu.

Eine Lizenz ist entweder nicht vorhanden, gültig oder ungültig, wenn sie beispielsweise bei einer Onlineauktion ersteigert wurde. Um die Lizenz zu validieren, implementieren wir zwei eigene Constraint-Annotationen, *@LicenseValid* und *@LicenseAvailable* (Listing 14).

Listing 15

```
//LicenseValidator.java
public class LicenseValidator implements
                                ConstraintValidator<LicenseValid,License>{
    @Override
    public void initialize(LicenseValid constraintAnnotation) {
    }
    @Override
    public boolean isValid(License license,
                                    ConstraintValidatorContext context) {
        //external systems are so slow...
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            return false;
        }
        return license.isValid();
    }
}
//LicenseAvailableValidator.java
public class LicenseAvailableValidator implements
                               ConstraintValidator<LicenseAvailable,License>{

    @Override
    public void initialize(LicenseAvailable constraintAnnotation) {
    }

    @Override
    public boolean isValid(License license,
                                    ConstraintValidatorContext context) {
        return license.isAvailable();
    }
}
```

Die *@LicenseValid*- und *@LicenseAvailable*-Annotationen sind einfache JSR-303-Annotationen. Sie legen über das Attribut *validatedBy* bereits fest, welcher Constraint Validator für die Validierung zuständig ist. Wir implementieren jeweils eigene Constraint Validators (Listing 15), um die Lizenz zu validieren. Eine Lizenz ist nicht vorhanden, wenn sie den Typ *NONE* hat. Diese Überprüfung ist schnell und einfach. Eine Lizenz ist ungültig, wenn sie auf eBay ersteigert wurde. Diese Überprüfung ist langsam und teuer.

Da Trainer ein ehrlicher Menschenschlag sind, gestatten wir über ein Drop-down die Auswahl der Lizenzierungsquelle in unserem Formular (Listing 16).

Starten wir die Anwendung, sehen wir sofort, dass die Lizenzvalidierung greift und das „externe“ System genauso langsam ist wie befürchtet. Höchste Zeit, unsere Constraints zu gruppieren. Hierfür definieren wir das Marker-Interface in Listing 17.

Das Interface *ExtendedValidation* dient uns nur dazu, Con-

straints zu markieren, die so spät wie möglich validiert werden sollen. Jede JSR-303-Annotation ermöglicht eine Zuordnung zu einer bestimmten *ValidationGroup* über das Attribut *groups*.

Der *PropertyValidator* erwartet optional eine Liste von *ValidationGroup*-Interfaces der Gruppen, die validiert werden sollen. Würden wir hier für die Validierung der Lizenzierung einfach das *ExtendedValidation*-Interface angeben, würde die *@LicenseAvailable*-Validierung gar nicht mehr laufen, da diese nicht der richtigen Gruppe zugeordnet ist. Geben wir keine Gruppe an, würde die *@LicenseValid*-Validierung nicht laufen, da diese nicht der Default-Gruppe zugeordnet ist.

Listing 16

```
//HomePage.java
form.add(new DropDownChoice<License>("license",
            Arrays.asList(License.values()))
            .add(new PropertyValidator(
                  new Property(Trainer.class,"license"))));
//HomePage.html
<select wicket:id="license"/> <br/>
//HomePage.properties
license.invalid=Leider koennen wir keine Lizenzen von eBay akzeptieren
license.unavailable=Sie brauchen schon eine Lizenz, um als Trainer zu
arbeiten.
```

Listing 17

```
//ExtendedValidations.java
public interface ExtendedValidation {
}
```

Listing 18

```
//TrainerGroupSequence.java
@GroupSequence(value = {Default.class,ExtendedValidation.class})
public interface TrainerGroupSequence {
}
 //Trainer.java
 @LicenseValid(
   message = "invalid.license",
   groups = ExtendedValidation.class)
    @LicenseAvailable(message = "{license.unavailable}")
    @NotNull
    private License license;

    //HomePage.java
    form.add(new DropDownChoice<License>("license",
            Arrays.asList(License.values()))
            .add(new PropertyValidator(
                  new Property(Trainer.class,"license"),
                                    TrainerGroupSequence .class)));
```

Was wir brauchen, ist eine Liste von Gruppen, die in der richtigen Reihenfolge abgearbeitet werden müssen. Hierfür kann die *@GroupSequence*-Annotation verwendet werden. Wir definieren uns ein weiteres Marker-Interface (Listing 18).

Über die *TrainerGroupSequence* legen wir fest, dass zuerst die Default- und anschließend die *ExtendedValidation*-Gruppe validiert werden soll. Die Default-Gruppe ist im Hibernate-Validator-Modul definiert und enthält alle Constraints, die nicht explizit einer anderen Validierungsgruppe zugeordnet sind. Schlägt die Validierung der Default-Gruppe fehl, werden die teuren Validierungen gar nicht erst gemacht. Dieser Effekt lässt sich übrigens sehr schön beobachten, wenn man die Reihenfolge der Validation Groups einfach umdreht.

Die *TrainerGroupSequence* weisen wir direkt dem *PropertyValidator* zu. Starten wir die Anwendung jetzt ein letztes Mal, sehen wir, dass der langsame *LicenseValidator* nur dann läuft, wenn eine Lizenz ausgewählt (*@NotNull*) und diese nicht *NONE* ist (*@LicenseAvailable*).

Was übrigens leider nicht funktioniert, ist, die teure Validierung nur dann auszuführen, wenn feldübergreifend alle schnellen Validierungen bereits erfolgreich waren. Aber hier könnte beispielsweise ein eigener Form-Validator Abhilfe schaffen. Das wäre Stoff für einen weiteren Artikel.

Fazit

Die Integration von JSR 303 Bean Validations ist denkbar einfach und schön umgesetzt. Bean Validations funktionieren auch wunderbar parallel zum bereits bestehenden Validierungsframework in Wicket. Ich hatte sehr viel Freude damit und hoffe, dass ich Sie auch für Bean Validations begeistern konnte. Der Sourcecode ist auf GitHub gehostet [1].

Martin Dilger ist freiberuflicher Software-Consultant und Wicket-Trainer. Er beschäftigt sich seit Jahren intensiv mit der Entwicklung von Webanwendungen im Enterprise-Java-Umfeld.

martin@effectivetrainings.de

Links & Literatur

[1] https://github.com/dilgerma/wicket6-bean-validations

Anzeige

Implementierung eines OAuth-2.0-Servers mit Spring Security

Spring ins OAuth-Vergnügen!

Im dritten und letzten Teil der OAuth-2.0-Serie wechseln wir die Seiten: vom Client hin zum Server. Wenn Sie selbst ein API anbieten und Clients per OAuth 2.0 autorisieren wollen, so müssen Sie das Protokoll entweder selbst implementieren oder ein vorhandenes Framework verwenden. Wir entschieden uns für die letztere Option und stellen Ihnen vor, wie Sie einen standardkonformen OAuth 2.0 Authorization Server per Spring Security und dessen OAuth-2.0-Modul realisieren können.

von Sven Haiges

Wenn Sie einen Blick auf die Website www.oauth2.net/2 werfen, so entdecken Sie direkt auf der Homepage im unteren Teil eine Reihe von Implementierungen – sortiert nach Client und Server. Sofort fällt auf, dass es mehr Client-Libraries gibt. Im Bereich der Serverimplementierungen und gerade im Bereich Java sah es lange Zeit mau aus. Mittlerweile gibt es drei auf Java basierende Serverimplementierungen, die hier genannt werden:

- Apache Amber
- Apis Authorization Server
- Spring Security for OAuth

Apache Amber unterstützt aktuell die Revision 22 des OAuth-2.0-Protokolls, die mit September 2011 datiert ist. Der Apis Authorization Server ist laut der Dokumentation auf dem aktuellsten Stand (am 8. Dezember 2012 war das die v31) und mit Sicherheit einen Blick wert, wenn Sie keine ansonsten auf Spring basierende Systeme haben. Da Spring Security weit verbreitet und das Spring-Security-Modul für OAuth 2.0 noch dazu auf dem aktuellsten Stand ist, fiel unsere Entscheidung bei der hybris GmbH auf dieses Modul.

Artikelserie

Teil 1: Authorization Code Grant
Teil 2: Weitere OAuth 2.0 Grants
Teil 3: Implementierung eines OAuth-2.0-Servers anhand des OAuth-2.0-Moduls von Spring Security

Wir gehen im Folgenden nicht auf die Details zu Spring Security selbst ein. Dazu wurde und wird viel geschrieben. Wir gehen davon aus, dass Sie eine auf Spring-Security-basierende Java-Webapplikation auf OAuth 2.0 „upgraden“ wollen. In vielen Fällen bedeutet dies, HTTP Basic Authentication endlich herauszunehmen und auf OAuth 2.0 zu setzen.

Installation

Der Code des OAuth-2.0-Moduls wird auf GitHub verwaltet und ständig an neuere Revisionen des OAuth-2.0-Protokolls angepasst. Auch wenn sich OAuth 2.0 auf der Zielgeraden zur fertigen und empfohlenen Spezifikation befindet, gibt es meist alle paar Tage kleinere Änderungen. Sie können entweder das Git Repository klonen und alle paar Wochen die neuesten Änderungen ziehen, oder Sie nehmen einfach die unter den Downloads bereitgestellten *zip*- oder *tar.gz*-Archive. Nach dem Entpacken oder Klonen wechseln Sie in das Verzeichnis *spring-security-oauth2* und führen dort *mvn package* aus, um die Quelldateien zu kompilieren und im *target*-Verzeichnis ein fertiges JAR-File vorzufinden. Fügen Sie das Spring OAuth 2.0 JAR-File Ihrem *WEB-INF/lib*-Verzeichnis hinzu (oder passen Sie entsprechend Ihre Maven-Konfiguration an), und es kann losgehen.

Tonr und Sparklr

An dieser Stelle sei auch auf Tonr und Sparklr [1] hingewiesen, die beiden OAuth-2.0-Beispielapplikationen direkt von Spring Security. Die beiden Webapplikationen geben Beispiele für einen (Spring-)OAuth-2.0-Client und -Server. Ebenso wie das OAuth-2.0-Modul finden Sie diese beiden Apps im *samples*-Verzeichnis des Spring Security OAuth 2.0 Repositories. Stellen Sie sicher, dass

Sie die Beispiele für OAuth2 betrachten, da sich diese natürlich sehr stark von OAuth 1.0 unterscheiden.

Spring Security OAuth 2.0 konfigurieren

Die vorzunehmende Spring-Konfiguration für das OAuth-2.0-Modul besteht aus folgenden Teilen:

- Vorbereitung der bestehenden Spring-Konfigurationsdateien
- Konfiguration der OAuth-2.0-Clients, die auf Ihr API Zugriff haben sollen
- Konfiguration des Tokenmanagements (Access/Refresh-Tokens) sowie des Token-Endpoints
- Konfiguration und Erstellung des Authorization Servers inklusive der Konfiguration der Authorization und Token-Endpoints
- Absicherung der vorhandenen APIs, sprich: des Resource-Servers

Vorbereitung der Spring-Konfiguration

Um die speziellen Spring-OAuth-2.0-Tags in Ihrer Konfiguration nutzen zu können, müssen Sie zunächst die Spring-Konfiguration um den OAuth2 Namespace erweitern (Listing 1).

Der Namespace *oauth* wurde eingeführt, und ebenso haben wir das XML Schema dafür angegeben. In Eclipse und anderen Entwicklungstools haben Sie dadurch den Vorteil, Auto Completion nutzen zu können.

OAuth-2.0-Clients

Alle OAuth-2.0-Clients, die auf Ihr API oder Ihre Webapplikation generell zugreifen dürfen, müssen im *ClientDetailsService* eingetragen sein. Der *ClientDetailsService* informiert über die OAuth-2.0-Clients, die mit *ClientDetails*-Objekten verwaltet werden. Wenn Sie ein API planen, das über eine Clientregistrierung jederzeit neue OAuth-2.0-Clients verwalten muss, so sollten Sie das *ClientDetailsService*-Interface (besser: die Klasse *BaseClientDetails*, die dieses Interface implementiert) von Spring überschreiben. Sie können dann die verfügbaren OAuth-2.0-Clients selbst verwalten, beispielsweise in einer Datenbank. In diesem Fall handelt es sich bei Ihrem API um ein Public API. In den meisten Fällen reicht für private APIs eine statische Konfiguration. Bei privaten APIs ist die Anzahl der Clients (nicht Nutzer!) meist recht gering, und vor allem kommt nicht andauernd ein neuer Client hinzu. Spring bietet für diese Fälle auch eine In-Memory-Implementierung an, die wir hier vorstellen.

Im Beispiel in Listing 2 wurden zwei OAuth-2.0-Clients konfiguriert. Beim Ersten handelt es sich um einen Client, der nur den *implicit* Grant Type für clientseitige Webapplikationen benutzen darf. Versucht dieser Client, einen anderen Grant Type als den Authorization Code Flow zu benutzen, so wird dieser Versuch scheitern. Da beim Implicit Grant kein Client-Secret verwendet wird, muss natürlich auch keines in der Konfiguration angegeben werden.

Der zweite OAuth-2.0-Client ist etwas umfangreicher ausgestattet. Er kann den klassischen OAuth2 Authorization Code Flow, Resource Owner Password Flow und den Client Credentials Flow benutzen. Ebenso darf dieser Client den abgelaufenen Access-Token per Refresh-Token erneuern. Im zweiten *<oauth:client>*-Tag wurde nun auch ein Secret in der Konfiguration angegeben, da diese Flows *client_id* und *client_secret* voraussetzen, um den Client zu authentifizieren.

Beide Clients haben einen Redirect-URI konfiguriert. Nach erfolgter Autorisierung des Clients durch den Resource Owner wird der Browser an diesen URI weitergeleitet. Wie Sie bereits in den ersten beiden Teilen gesehen haben, wird dieser URI vom Server zusätzlich noch auf Übereinstimmung geprüft. Die URIs in den obigen Beispielen zeigen zurück auf einen lokalen Server, der zum Testen der OAuth 2.0 Flows verwendet wurde.

Die Authorities (siehe entsprechendes Attribut) des OAuth-2.0-Clients sind die Rollen, die dem Client selbst zugewiesen werden. Diese Berechtigungen gehören dem Client selbst. Nicht zu verwechseln ist dies mit den Berechtigungen, die ein Nutzer des APIs mit einem Access-Token erhält. Die im *<oauth:client> authorities*-Attribut zugewiesenen Berechtigungen sind also vor allem für den Client Credentials Flow wichtig, da sich dadurch der OAuth-2.0-Client selbst autorisieren kann.

Tokenmanagement

Sämtliche mit Access-Tokens in Verbindung stehende Services werden in den *AuthorizationServerTokenServices* zur Verfügung gestellt. Spring OAuth 2.0 liefert eine fertige Implementierung, die *DefaultTokenServices*, mit, die direkt eingesetzt werden kann. Für die

Listing 1

```
<beans xmlns="http://www.springframework.org/schema/beans"
  ...
  xmlns:oauth="http://www.springframework.org/schema/security/oauth2"
  xsi:schemaLocation="http://www.springframework.org/schema/aop
    ...
    http://www.springframework.org/schema/security/oauth2
      http://www.springframework.org/schema/security/spring-security-oauth2-1.0.xsd">
    ...
```

Listing 2

```
<oauth:client-details-service id="clientDetails">
  <oauth:client client-id="client-side" resource-ids="myresource"
    authorized-grant-types="implicit" authorities="ROLE_CLIENT"
    redirect-uri="http://localhost:8080/oauth2_implicit_callback" />
  <oauth:client client-id="mobile_android" resource-ids="myresource"
    authorized-grant-types="authorization_code,refresh_token,password,client_credentials"
    authorities="ROLE_CLIENT" secret="ganzgeheim"
    redirect-uri="http://localhost:8080/oauth2_callback" />
</oauth:client-details-service>
```

Speicherung der Tokens liefert Spring ebenso eine In-Memory-Implementierung mit, den *InMemoryTokenStore*. Gerade in einer produktiven Umgebung sollten Sie es in Betracht ziehen, wenigstens den *InMemoryTokenStore* durch eine persistente Variante zu ersetzen. Dafür hält Spring die Implementierung in *JdbcTokenStore* bereit, die die Tokens in einer konfigurierten Datenbank speichern kann. Die persistente Speicherung ist sehr sinnvoll, da ein Herunterfahren des Servers ansonsten sämtliche Erinnerungen an bereits bekannte Access- und vor allem Refresh-Tokens löscht. Während dies für Access-Tokens gerade noch annehmbar ist (typischerweise eine Gültigkeit von 24 Stunden, ebenso der Standardwert), so ist dies äußerst problematisch bei den Refresh-Tokens, die typischerweise bis zu mehreren Monaten gültig sind. Sobald der Authorization Server jedoch ein Refresh-Token nicht mehr validieren kann (da der *InMemoryTokenStore* dieses Token nicht mehr kennt), hat dies wenig komfortable Auswirkungen auf die Nutzer: sie müssen nun erneut autorisiert werden. Listing 3 zeigt die beispielhafte Konfiguration des *DefaultTokenServices* inkl. der Nutzung des *InMemoryTokenStores*.

Listing 3

```
<bean id="tokenStore"
  class="org.springframework.security.oauth2.provider.token.InMemoryTokenStore" />

<bean id="tokenServices"
  class="org.springframework.security.oauth2.provider.token.DefaultTokenServices">
  <property name="tokenStore" ref="tokenStore" />
  <property name="supportRefreshToken" value="true" />
  <property name="refreshTokenValiditySeconds" value="2592000" /> <!-- 60*60*24*30
                                                                    = 30d -->
  <property name="accessTokenValiditySeconds" value="43200" /> <!-- 60*60*12 =
                                                                  12h -->
</bean>
```

Während die Gültigkeit der Refresh- und Access-Tokens im obigen Beispiel mit den bereits vorhandenen Standardwerten überschrieben wurde, werden die Refresh-Tokens per Standardwert ausgeschaltet. Per Property *supportRefreshToken* kann man dieses Feature einschalten.

Listing 4

```
<oauth:authorization-server
  client-details-service-ref="clientDetails" token-services-ref="tokenServices" >
  <oauth:authorization-code />
  <oauth:implicit />
  <oauth:refresh-token />
  <oauth:client-credentials />
  <oauth:password />
</oauth:authorization-server>
```

Der Authorization Server

Nachdem nun die OAuth-2.0-Clients und die Token-Services konfiguriert sind, kann der Authorization Server aufgebaut werden. Für das Parsen dieser Konfiguration ist die Klasse *AuthorizationServerBeanDefinitionParser* zuständig, die bei Detailfragen oftmals hilfreich sein kann. Da hier viele Konventionen (wie z. B. für die Token- und Authorization Endpoints) benutzt werden, kommt die Definition des OAuth 2.0 Authorization Servers selbst recht schlank daher (Listing 4).

Über die **-service-ref*-Attribute werden zunächst der *ClientDetailsService* sowie die *TokenServices* konfiguriert. Die unterstützten Grant Types werden dann per Tags innerhalb des *<oauth:authorization-server>*-Tags aufgelistet. Fehlt beispielsweise der *<oauth:password>*-Tag, so wird dieser Grant Type auch nicht unterstützt. Falls erwünscht, kann sich jeder einzelne Grant Type auch explizit per *disabled=true* abschalten. Die wichtigen Endpoints, der Authorization und Token-Endpoint, sind per Konvention vorbelegt:

- Der Standard-Token-Endpoint ist */oauth/token*
- Der Standard-Authorization-Endpoint ist */oauth/authorize*

Wenn Ihnen daran gelegen ist, dass Ihr per OAuth 2.0 gesichertes API möglichst standardkonform ist, so sollten Sie diese Werte auch beibehalten. Nutzer von OAuth-2.0-APIs haben diese Endpoints bereits verinnerlicht. Die Konfiguration der Endpoints setzt freilich auch voraus, dass ein Standard-Spring-MVC-Gerüst bereits vorhanden ist. Während der Token-Endpoint

Listing 5

```
@Controller("accessConfirmationController")
@SessionAttributes(types = AuthorizationRequest.class)
public class OAuth2AccessController
{

  private ClientDetailsService clientDetailsService;

  @RequestMapping(value = "/oauth/confirm_access", method = RequestMethod.GET)
  public ModelAndView getAccessConfirmation(@ModelAttribute final
                                    AuthorizationRequest clientAuth) throws Exception
  {
    final ClientDetails client =
                        clientDetailsService.loadClientByClientId(clientAuth.getClientId());
    final TreeMap<String, Object> model = new TreeMap<String, Object>();
    model.put("auth_request", clientAuth);
    model.put("client", client);
    return new ModelAndView("access_confirmation", model);
  }

  ...
}
```

jedoch komplett von Spring erstellt und zur Verfügung gestellt wird, muss der Authorization Endpoint, eine Webseite, von Ihnen selbst erstellt werden. Genauer genommen ist dies die User Approval Page, auf die per Forward an */oauth/confirm_access* weitergeleitet wird.

Die User-Approval-Seite ist dafür verantwortlich, vom Resource Owner die Zustimmung einzuholen. Damit stimmt der Resource Owner, der Nutzer, also zu, dass der anfragende Client in seinem Namen das API benutzen darf. Um diese Zustimmung einzuholen, muss die Approval-Seite jedoch zunächst den Resource Owner authentifizieren. Dies geschieht typischerweise über die Standard Basic Authentication, d. h. der Resource Owner muss seinen Username und sein Passwort angeben.

Gehen wir jedoch Schritt für Schritt vor. Zunächst benötigen wir die Approval-Seite. Der folgende Controller ist auf den Pfad */oauth/confirm_access* gemappt und zeigt letztlich die JSP-Seite *access_confirmation.jsp* an (Listing 5).

Vom Authorization Server bekommt die *getAccessConfirmation*-Methode einen *AuthorizationRequest* mitgeliefert. Es werden die *ClientDetails* mithilfe des *ClientDetailsService* geladen und zusätzlich zusammen mit dem *AuthorizationRequest* der View zur Verfügung gestellt.

Die *access_confirmation.jsp*-Seite erzeugt ein simples UI, aus dem für den Resource Owner ersichtlich wird, welcher Client welche Berechtigungen anfordert. Der Resource Owner kann diesen Berechtigungen dann zustimmen oder den Request ablehnen. In der *access_confirmation.jsp*-Seite kommen einige Expressions zum Einsatz, die auch überprüfen, ob der aktuelle User authentifiziert ist (Listing 6).

In diesem Fall muss der aktuelle User die Rolle *ROLE_CUSTOMERGROUP* innehaben, ansonsten hat er auf dieser Seite nichts zu suchen. Diese Absicherung muss freilich mit normaler Spring Security konfiguriert werden. Ohne hier ins Detail zu gehen und die restliche Spring-Konfiguration aufzuführen, muss dafür grob das in Listing 7 Gezeigte in den Spring-Konfigurationsdateien vorhanden sein.

Der User Approval Page wird hiermit ein übliches, HTML-Form-basiertes Login-Verfahren vorangestellt. Auch der Token-Endpoint muss speziell konfiguriert werden. Eine Beispielkonfiguration sehen Sie in Listing 8.

Zunächst wird für den Token-Endpoint */oauth/token* per Referenz ein *ClientAuthenticationManager* gesetzt. Dieser hat letztlich Zugriff auf die *ClientDetails* und kann damit die Client-Authentication durchführen. Die meisten OAuth 2.0 Grants setzen eine Client-Authentication voraus, d. h. hier fragt der Client mittels *client_id* und *client_secret* für den User nach einem Access-Token und wird dabei auch selbst authentifiziert. Beachten Sie auch, dass der anonyme Zugriff für den Token-Endpoint explizit abgeschaltet ist. Ebenso setzen wir für den Zugriff auf den Authentication Endpoint den HTTPS-Channel voraus. Während der Entwicklung kann es zwar sinnvoll sein, auf HTTP zu wechseln (aufgrund selbst erstellter Zertifikate), jedoch sollte in Produktion nur HTTPS erlaubt sein. Da der Client Credentials Flow die Client-Authentication auch per Basic Authentication Header senden kann, wird hier noch die Basic Authentication konfiguriert. Der *clientCredentialsTokenEndpointFilter* ist dann notwendig, falls *client_id* und *client_secret* per Request-Parameter gesendet werden. Für den Client Credentials Grant und den Resource Owner Password Grant sieht die OAuth-2.0-Spezifikation vor, dass die Client-Authentication (*client_id* und *client_secret*) entweder per Basic Authentication Header oder als Request-Parameter gesendet werden kann. Die Resource Owner und deren Usernames und Passwords werden durch den *userDetailService* konfiguriert. Die Spring-Beispiel-App zeigt hier eine statische Konfiguration für die user *marissa* und *paul* (Listing 9). Dieser Teil muss natürlich durch eine passende, eigene Implementierung abgelöst werden und ist nur der Vollständigkeit wegen hier erwähnt.

Listing 6

```
...
<authz:authorize ifAllGranted="ROLE_CUSTOMERGROUP">

  <h2 class="form-authorize-heading">Please confirm</h2>
  <p>You hereby authorize <strong><c:out value="${client.clientId}"/></strong> to access
                                                          your protected resources.</p>

  <form id="denialForm" name="denialForm" action="<%=request.getContextPath()%>/
                                                      oauth/authorize" method="post">
  <input name="user_oauth_approval" value="false" type="hidden"/>
  <input name="deny" class="btn btn-large" value="Deny" type="submit">
  </form>
  <form id="confirmationForm" name="confirmationForm"
            action="<%=request.getContextPath()%>/oauth/authorize" method="post">
  <input name="user_oauth_approval" value="true" type="hidden"/>
  <input name="authorize" class="btn btn-large btn-primary" value="Authorize"
                                                                  type="submit">
  </form>

</authz:authorize>
...
```

Listing 7

```
<http access-denied-page="/login.jsp?authorization_error=true"
  disable-url-rewriting="true" xmlns="http://www.springframework.org/schema/security">
  <intercept-url pattern="/oauth/**" access="ROLE_CUSTOMERGROUP" />

  <form-login authentication-failure-url="/login.jsp?authentication_error=true"
    default-target-url="/index.jsp" login-page="/login.jsp"
    login-processing-url="/login.do" />
  ...
</http>
```

Listing 8

```
<http xmlns="http://www.springframework.org/schema/security"
  pattern="/oauth/token" create-session="stateless"
  authentication-manager-ref="clientAuthenticationManager"
  entry-point-ref="oauthAuthenticationEntryPoint" >
  <intercept-url pattern="/oauth/token" access="IS_AUTHENTICATED_FULLY"
                                                    requires-channel="https" />
  <anonymous enabled="false" />
  <http-basic entry-point-ref="oauthAuthenticationEntryPoint" />
  <!-- include this only if you need to authenticate clients via request parameters -->
  <custom-filter ref="clientCredentialsTokenEndpointFilter"
                                                  before="BASIC_AUTH_FILTER" />
  <access-denied-handler ref="oauthAccessDeniedHandler" />
</http>

<authentication-manager xmlns="http://www.springframework.org/schema/security"
  id="clientAuthenticationManager">
  <authentication-provider user-service-ref="clientDetailsUserService" />
</authentication-manager>

<bean id="clientDetailsUserService"
  class="org.springframework.security.oauth2.provider.client.
                                                  ClientDetailsUserDetailsService">
  <constructor-arg ref="clientDetails" />
</bean>

<bean id="oauthAuthenticationEntryPoint"
  class="org.springframework.security.oauth2.provider.error.
                                                OAuth2AuthenticationEntryPoint">
  <property name="realmName" value="hybris" />
</bean>

<bean id="clientCredentialsTokenEndpointFilter"
  class="org.springframework.security.oauth2.provider.client.
                                              ClientCredentialsTokenEndpointFilter">
  <property name="authenticationManager" ref="clientAuthenticationManager" />
</bean>

<bean id="oauthAccessDeniedHandler"
  class="org.springframework.security.oauth2.provider.error.OAuth2AccessDeniedHandler"
/>
```

Listing 9

```
<authentication-manager alias="authenticationManager"
                        xmlns="http://www.springframework.org/schema/security">
  <authentication-provider>
    <user-service id="userDetailsService">
      <user name="marissa" password="koala" authorities="ROLE_USER" />
      <user name="paul" password="emu" authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

Konfiguration des Resource Servers, API-Access

Zuletzt muss nun das API per OAuth 2.0 abgesichert werden. Erneut ist dies eigentlich keine OAuth2-Besonderheit, sondern zu großem Teil Standard-Spring-Security. Unsere App hat einen Teil dieser Konfiguration in der Spring-Konfiguration und einen Teil per Spring Security Annotations direkt in den Spring Controllern. Listing 10 zeigt zunächst die Spring-Konfiguration.

Die Security-Konfiguration betrifft das API unter dem Pfad */v1/***. Wichtig ist hierbei, zumindest bei einem produktiven System, den HTTPS-Channel per *requires-channel=https* vorzuschreiben. Sollte ein Client nun per HTTP auf das API zugreifen, erfolgt zunächst ein automatischer Redirect (302 Location Header) an den HTTPS-URL durch Spring Security. Des Weiteren wurde ein OAuth 2.0 Access Denied Handler konfiguriert, der die OAuth-2.0-Zugriffsverletzungen abdeckt. Ein Controller, der in Ihrer Spring-Web-Applikation konfiguriert wurde, kann nun einzelne Methoden (oder auch den gesamten Controller) per Annotations absichern. Ein OAuth-2.0-Client hat per Client-Details-Konfiguration die Rolle *ROLE_CLIENT* bekommen und wird folgendermaßen abgesichert (Listing 11).

Listing 10

```
<http pattern="/v1/**" entry-point-ref="oauthAuthenticationEntryPoint"
  access-decision-manager-ref="accessDecisionManager"
  xmlns="http://www.springframework.org/schema/security">
  <anonymous username="anonymous"
                                granted-authority="ROLE_ANONYMOUS" />
  <session-management
    session-authentication-strategy-ref="fixation" />

  <intercept-url pattern="/**/cart/address/**" requires-channel="https"
/>
  <intercept-url pattern="/**/customers" requires-channel="https" />


  <custom-filter ref="resourceServerFilter" before="PRE_AUTH_FILTER" />
  <access-denied-handler ref="oauthAccessDeniedHandler" />

</http>
```

Listing 11

```
@Secured("ROLE_CLIENT")
@RequestMapping(method = RequestMethod.POST)
@ResponseBody
@ResponseStatus(value = HttpStatus.CREATED,
                                          reason = "Customer created.")
public void registerUser(@RequestParam final String login,
                                  @RequestParam final String password
    ...) throws DuplicateUidException
{
  ...
}
```

Für den Zugriff auf die Ressourcen eines Resource Owners durch einen OAuth-2.0-Client sieht es ganz ähnlich aus. Passend zum Beispiel des In-Memory-*UserDetailsServices* kann ein Zugriff auf Nutzerdaten folgendermaßen kontrolliert werden:

```
@Secured("ROLE_USER")
@RequestMapping(value = "/current", method = RequestMethod.GET)
@ResponseBody
public CustomerData getCurrentCustomer()
                                    throws InconsistentUseridException
{
  ...
}
```

Zusammenfassung

Im letzten Teil der OAuth-2.0-Serie haben wir Ihnen die Konfiguration eines OAuth 2.0 Authorization und Resource Servers mithilfe des Spring-Security-OAuth-2.0-Moduls erläutert. Insofern bereits eine Spring-basierte Nutzerverwaltung vorhanden ist, lässt sich mit recht wenig Aufwand ein standardkonformer OAuth 2.0 Authorization Server hinzufügen. Die Einrichtung beginnt mit der Vorbereitung der bestehenden Konfigurationsdateien, z. B. dem Entfernen der Basic Authentication. Es folgt die Einrichtung der OAuth-2.0-Clients, die auf das API nach Autorisierung der Resource Owner Zugriff haben sollen. Für viele private APIs reicht hier mit Sicherheit eine statische In-Memory-Konfiguration aus. Es folgt dann die Einrichtung der notwendigen Token-Infrastruktur, die Spring Security ebenso dank Standardimplementierungen auf wenige Zeilen XML reduziert. Es folgt die Konfiguration des OAuth 2.0 Authorization Servers – darunter ist vor allem die Konfiguration der unterstützten OAuth 2.0 Flows, die Einbindung der bestehenden Nutzerverwaltung sowie die Erstellung einer Authorization-Seite zu verstehen. Letztlich muss das API mittels Spring abgesichert werden, beispielsweise per Spring Annotations direkt in den Spring Controllern.

Sven Haiges arbeitet als Technology Strategist bei der hybris GmbH in München. Neben Groovy and Grails beschäftigt er sich dort derzeit auch mit HTML5 und Android. Sven lebt mit seiner Familie in München. Ihm kann gerne unter @hansamann auf Twitter gefolgt werden.

Links & Literatur

[1] https://github.com/SpringSource/spring-security-oauth/tree/master/samples/oauth2

Leichtgewichtige Webanwendungen mit AngularJS und Java EE 6

Das dynamische Web

JavaScript und HTML5 sind in aller Munde. Vom „Skript-Kid“ bis zum eingefleischten Java-Entwickler, für den dynamische Typisierung eindeutig Teufelszeug ist: Wer moderne Webanwendungen schreibt, kommt ohne JavaScript nicht aus. Auch Frameworks wie GWT, JSF, Wicket oder Tapestry helfen da wenig. Häufig erschweren sie das manuelle Eingreifen und die Implementierung eigener Features in JavaScript sogar durch unnötige Abstraktionen und übermäßig komplizierte APIs. Stellt sich also die Frage, ob es nicht besser ist, im Frontend gleich ganz auf Java zu verzichten und stattdessen auf bewährte Webtechnologien zu setzen. Wie das aussehen kann, wollen wir in diesem Artikel anhand einer kleinen CRUD-Anwendung (Create, Read, Update, Delete) zeigen. Im Backend setzen wir dazu komplett auf Java EE 6, während wir im Frontend Twitter Bootstrap und AngularJS verwenden.

von Nils Preusker

Bevor wir uns gleich auf die Implementierung stürzen, hier noch ein kurzer Blick auf die Technologien. HTML5 steht bekanntlich für HTML plus JavaScript plus CSS3. Dazu kommen Features wie Offline Storage oder WebSockets. Wer heute mit einem modernen Browser wie Chrome oder Firefox im Web surft, erlebt damit einen Featurereichtum, der früher nur mit Plug-ins wie dem Flash Player oder Silverlight zu erreichen war. Das Spannende daran ist, dass wir als Entwickler keine proprietären Sprachen mehr lernen oder teure Lizenzen kaufen müssen, um uns diesen Featurereichtum zu erschließen. Stattdessen können wir auf frei verfügbare Standards und Entwicklertools setzen und mit JavaScript und CSS3 atemberaubende interaktive Webanwendungen schreiben. Der Kreativität sind also wirklich keine Grenzen mehr gesetzt. Ähnlich sieht es im Backend aus. Mit Java EE 6 steht endlich ein wirklich schlanker Standard für die Anwendungsentwicklung zur Verfügung. Und mittlerweile gibt es auch die dazugehörigen Laufzeitumgebungen in Form von schlanken und frei verfügbaren Application-Servern wie JBoss AS 7 oder GlassFish 3. Natürlich beinhaltet der Standard nach wie vor Dinge, um die man besser einen Bogen macht. Mein persönlicher Favorit ist da JSF 2.0 … Aber wer sich die Zeit nimmt, um die richtigen Technologien herauszupicken, und um die Schwachstellen einen angemessen großen Bo-

gen macht, hat mit Java EE 6 einen wirklich schlanken und produktiven Technologiestack an der Hand.

Nun stellt sich nur noch die Frage, wie wir die zwei Welten – JavaScript im Frontend und Java EE 6 im Backend – zusammenbringen. Insbesondere, wenn wir bewusst auf JSF und Co. verzichten wollen. Die Antwort lautet REST (Representational State Transfer). Das bedeutet, dass wir jede Entität unserer Anwendung unter einem eindeutigen HTTP-URL als JSON-(JavaScript-Object-Notation-)Repräsentation veröffentlichen. Rufen wir also beispielsweise http://localhost:8080/crm-demo/rest/customer/1 auf, so erhalten wir einen JSON-String (z. B. *{id:1, firstName:Markus, lastName:Mustermann, …}*) als Antwort. Um Entitäten zu erzeugen, können wir einen HTTP-POST-Request an den URL http://localhost:8080/crm-demo/rest/customer senden und erhalten eine Antwort mit dem HTTP-Code „201 Created“ und dem Location-Header http://localhost:8080/crm-demo/rest/customer/2 zurück. Analog dazu werden HTTP PUT zum Aktualisieren und HTTP DELETE zum Löschen verwendet, aber dazu später mehr. Als Media-Type für die Repräsentation der Entitäten wählen wir JSON, da es sich in JavaScript sehr einfach verarbeiten lässt. Der Vollständigkeit halber sei aber erwähnt, dass der Media-Type bei REST-Architekturen nicht auf JSON beschränkt ist. Im Gegenteil gehört Content-Negotiation, also die Verhandlung zwischen Client und Server über den zu liefernden Media-Type, sogar zu den wichtigsten Grundprinzipien von REST-Architekturen. Auch das Protokoll ist übrigens nicht auf HTTP beschränkt. Wer sich ausführlicher mit dem Thema befassen möchte, dem sei das Buch „REST in Practice“ [1] empfohlen.

Aber zurück zu unserem Beispiel. Java bietet mit JAX-RS (Java API for RESTful Web Services) eine Programmierschnittstelle für REST und der Application Server bringt die Implementierung gleich mit, sodass wir keine weiteren Bibliotheken einbinden müssen. Durch ein paar Annotationen können wir so unsere Entitäten über HTTP für das Frontend bereitstellen. Nebenbei kümmert sich JAX-RS auch gleich transparent um die Transformation der Java-Objekte zu JSON und umgekehrt.

Der fachliche Kontext

Und damit sind wir fast soweit, dass wir mit der Entwicklung loslegen können. Unser Ziel ist eine einfache CRUD-Anwendung zur Verwaltung von Kunden. Kundenobjekte können einer Firma zugeordnet werden, zur Anzeige gibt es eine Übersichtsseite mit Suchfunktion

Listing 1

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>crm-demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <name>CRM Demo</name>

  <dependencies>
    <!-- Java EE 6 Specification -->
    <dependency>
      <groupId>org.jboss.spec</groupId>
      <artifactId>jboss-javaee-web-6.0</artifactId>
      <type>pom</type>
      <scope>provided</scope>
      <version>3.0.1.Final</version>
      <exclusions>
        <exclusion>
          <artifactId>xalan</artifactId>
          <groupId>org.apache.xalan</groupId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>

  <repositories>
    <repository>
      <id>jboss-public-repository</id>
      <name>JBoss Repository</name>
      <url>http://repository.jboss.org/nexus/content/groups/public</url>
    </repository>
  </repositories>

  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.1</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.12</version>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.1.1</version>
        <configuration>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

und Kundentabelle, über einen Link lassen sich die Details bestehender Kunden anzeigen und bearbeiten. Außerdem können neue Kunden angelegt und bestehende Kunden gelöscht werden.

Das Backend

Bevor wir die Entitätsklassen anlegen können, brauchen wir zunächst die Laufzeitumgebung und ein Grundgerüst für unsere Anwendung. Dazu laden wir einen JBoss AS 7.1.1 Final herunter [2] und legen ein Maven-Projekt mit der Java-EE-6-Spezifikation als Abhängigkeit an (Listing 1). Ich habe dazu Eclipse und das m2e-Plug-in verwendet, alternativ lassen sich *pom.xml* und die entsprechenden Verzeichnisse aber natürlich auch per Hand oder mit den entsprechenden Tools in jeder anderen Java-IDE anlegen. Der Quellcode der fertigen Anwendung ist auf GitHub unter [3] zu finden. Als Nächstes benötigen wir die Entitäten *Customer* und *Company* (Listing 2). Dabei ist zu beachten, dass wir die Entitätsklassen in dieser Anwendung sowohl als Datenbankentitäten als auch als DTO (Data Transfer Object) zur Kommunikation mit dem Client verwenden. Für die Persistenz versehen wir die Klassen mit JPA-Annotationen und definieren so das Mapping auf die entsprechenden Datenbankfelder. Für die Kommunikation mit dem Client wollen wir wie eingangs erwähnt JSON verwenden. JBoss AS 7.1 bringt standardmäßig RESTEasy [4] als JAX-RS-Implementierung mit, wo das Data Binding zwischen POJOs und JSON transparent von Jackson [5] erledigt wird. Wir müssen also nichts weiter an unseren Entitätsklassen tun, als einen Default-Konstruktor zu implementieren und Getter- und Setter-Methoden für die Attribute anzubieten.

Für die Persistenz müssen wir außerdem eine *persistence.xml*-Datei im Verzeichnis *src/main/resources/META-INF* anlegen. Hier referenzieren wir die im JBoss AS 7 standardmäßig vorkonfigurierte ExampleDS als DataSource. Wer sich das genauer ansehen möchte, der findet die Details in der Konfigurationsdatei des Servers (*standalone/configuration/standalone.xml*), für uns ist nur wichtig, dass hier eine In-Memory-H2-Datenbank verwendet wird, die der Server Out of the Box mitbringt. Um die Datenbank später durchstöbern zu können, bietet es sich übrigens an, die H2-Konsole als Webanwendung zu deployen. Die Konsole gibt es fertig verpackt für das Deployment auf JBoss AS 7 unter [6]. Das *.war*-Archiv muss dann einfach in das Verzeichnis *standalone/deployments* gelegt werden, wonach die Konsole unter http://localhost:8080/h2console erreichbar ist. Zum Login gibt man den JDBC-URL *jdbc:h2:mem:test* und als Benutzername und Passwort jeweils *sa* ein. Die Tabellen für unsere Entitäten werden dann beim ersten Deployment der Anwendung automatisch angelegt. Zum Zugriff auf die Datenbank schrei-

Listing 2

```
package com.mycompany.entity;

import java.util.Date;
import java.util.Locale;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Customer {

 @Id
 @GeneratedValue
 private Long id;
 private String firstName;
 private String lastName;
 private String email;
 private String phone;
 private String fax;
 private Sex sex;
 private String country;
 private Locale locale;
 private Date createDate;
 @ManyToOne(cascade = { CascadeType.ALL })
 private Company company;

 /**
  * Default-Konstruktor für JAX-RS (Objekt <> JSON Serialisierung)
  */
 public Customer() {
 }
 // ... (Konstruktor und Getter und Setter)
}

package com.mycompany.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Company {

 @Id
 @GeneratedValue
 private Long id;
 private String name;

 /**
  * Default-Konstruktor für JAX-RS (Objekt <> JSON Serialisierung)
  */
 public Company() {
 }
 // ... (Konstruktor und Getter und Setter)
}
```

ben wir die Klasse *CustomerService*, in der wir einen EntityManager „injecten“ und die klassischen CRUD-Funktionen implementieren.

Let's REST...

Nun, da wir das Datenmodell und die Persistenz abgedeckt haben, können wir uns der Veröffentlichung der Entitäten als REST-Ressourcen widmen. Dazu benötigen wir zunächst eine Application-Klasse, die einen Pfad unterhalb des Contextpfads der Anwendung an die REST-Ressourcen bindet. Das klingt komplizierter als es ist, im Endeffekt müssen wir nur an irgendeiner Stelle den Einstiegspunkt für die REST-URLs bestimmen. Wir verwenden *@ApplicationPath("/rest")* und stellen so alle REST-Ressourcen unter http://localhost:8080/crm-demo/rest bereit. Alternativ ließe sich dieses Mapping übrigens auch in einer *web.xml*-Datei definieren, wir haben uns aber hier für den seit Java EE 6 möglichen Weg entschieden, um komplett ohne *web.xml* auszukommen. Darüber, ob es sich lohnt, eine Klasse nur für dieses Mapping zu schreiben, lässt sich aber sicher streiten.

Als Nächstes benötigen wir eine Ressource für die Customer-Entität (Listing 3). Mit der *@Path*-Annotation auf Klassenebene legen wir den Basispfad fest, während sie auf Methodenebene dazu verwendet werden kann, den Pfad um weitere Elemente wie z. B. eine ID zu erweitern. Wir benötigen fünf Methoden, um die Funktionalität des *CustomerService* über REST zu veröffentlichen. Dabei kann die *find*-Methode einen „Searchstring“ entgegennehmen, um die Ergebnisse einzuschränken, *findById* und *delete* erwarten eine Kundennummer und *save* und *update* benötigen das jeweilige Objekt als JSON-String, wobei *save* ohne und *update* mit ID aufgerufen wird. Bei der Implementierung lohnt es sich, auf die Rückgabe der jeweils korrekten HTTP-Status-Codes und Header zu achten. So geben wir zum Beispiel bei *find* bzw. *GET* ein HTTP 404 „Not Found“ zurück, wenn eine nicht-existente ID angefragt wird. Beim HTTP-POST-Request zum Anlegen einer neuen Ressource wird entsprechend der HTTP-Code „201 Created“ mit dem URL der neuen Ressource im Location-Header zurückgegeben.

Nebenbei ist noch darauf hinzuweisen, dass wir die Ressource mit der *@Stateless*-Annotation als Stateless Session Bean markieren, um die Methodenaufrufe automatisch in eine Transaktion zu hüllen. Geht also beispielsweise nach dem Speichern einer neuen Entität etwas schief, wird die Datenbanktransaktion automatisch zurückgerollt. Außerdem verwenden wir in der Ressource den *CustomerService*, der durch die *@Inject*-Annotation von der CDI-Runtime automatisch beim Erzeugen der *CustomerResource*-Instanz gesetzt wird. Damit CDI in unserer Anwendung aktiv ist, müssen wir noch eine leere *beans.xml*-Datei im *src/main/webapp/WEB-INF*-Verzeichnis anlegen.

Frontend

Und damit sind wir beim Frontend. Als Frameworks wollen wir hier auf AngularJS und Twitter Bootstrap setzen. AngularJS soll uns dabei helfen, die Architektur der JavaScript-Anwendung zu strukturieren, sowie das Data Binding zwischen View und Controllern drastisch zu vereinfachen, und Bootstrap nimmt uns in erster Linie viel Arbeit beim Layout ab.

Als Startpunkt verwenden wir angular-seed [7], ein Projekt, das ein Grundgerüst für AngularJS-Anwendungen liefert. Für unsere Zwecke reicht es, angular-seed auszuchecken und daraus das *app*-Verzeichnis in unser Projekt unter *src/main/webapp* zu kopieren. Für Neugierige lohnt es sich auch, einen Blick in die anderen Ordner zu werfen, insbesondere in Sachen Testing bietet angular-seed interessanten Beispielcode. Zusätzlich werden wir Twitter Bootstrap [8] und jQuery [9] benötigen. Beide Skripte legen wir in das Verzeichnis *src/main/webapp/app/lib*, jeweils in einen Unterordner namens *bootstrap* bzw. *jquery*. Wir benötigen jQuery hier in erster Linie wegen eines Drop-down-Menüs zum Anlegen neuer Kunden. Während AngularJS eine abgespeckte jQuery-Version namens jQLite mitliefert, ist Twitter Bootstrap für bestimmte Features auf das Vorhandensein von jQuery angewiesen.

Im *src/main/webapp/*-Verzeichnis legen wir außerdem eine *index.html*-Datei an, die jedoch lediglich die grundlegende Seitenstruktur beschreibt und die benötigten Skripte lädt. Der eigentliche Ausgangspunkt der Anwendung befindet sich in der Datei *app.js* im *js*-Verzeichnis.

Listing 3

```
package com.mycompany.boundary;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.List;

import javax.ejb.Stateless;
import javax.inject.Inject;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;
import javax.ws.rs.core.UriInfo;

import com.mycompany.control.CustomerService;
import com.mycompany.entity.Customer;

@Path("/customer")
@Stateless
public class CustomerResource {

  @Inject
  private CustomerService customerService;

  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public Response findCustomers(@QueryParam("searchString")
                                                    String searchString,
      @QueryParam("name") String name) {
    List<Customer> customers;
    if (searchString != null) {
      customers = customerService.findCustomers(searchString);
    } else {
      customers = customerService.findAllCustomers();
    }
    return Response.ok(customers).build();
  }

  @POST
  public Response saveCustomer(@Context UriInfo uriInfo,
                                                    Customer customer)
      throws URISyntaxException {
    customerService.saveCustomer(customer);
    return Response.created(
        new URI(uriInfo.getRequestUri() + "/" + customer.getId())).build();
  }

  @GET
  @Path("/{customerId}")
  @Produces(MediaType.APPLICATION_JSON)
  public Response findCustomerById(@PathParam(value = "customerId")
                                                    String customerId) {
    Customer customer = customerService.findCustomerById(Long
        .parseLong(customerId));

    if (customer != null) {
      return Response.ok().entity(customer).build();
    } else {
      return Response.status(Status.NOT_FOUND).build();
    }
  }

  @PUT
  @Path("/{customerId}")
  public Response updateCustomer(Customer customer) {
    customerService.updateCustomer(customer);
    return Response.status(Status.ACCEPTED).build();
  }

  @DELETE
  @Path("/{customerId}")
  public Response deleteCustomer(@PathParam("customerId")
                                                    Long customerId) {
    customerService.deleteCustomer(customerId);
    return Response.ok().build();
  }
}
```

Routing

Hier wird das *CrmDemo*-Modul mit seinen Abhängigkeiten deklariert und der so genannte *RouteProvider* zur Navigation innerhalb der Anwendung konfiguriert. Die Navigation basiert auf URL-Fragmenten, also dem Teil der URL, der hinter dem #-Zeichen folgt. Mit dem RouteProvider können wir nun für die Übersichtsseite das Fragment */list* und für die Detailseite */edit/:customerId* angeben, wobei der durch einen Doppelpunkt eingeleitete Teil (hier also *:customerId*) den Namen eines Parameters angibt, der an den Controller weitergereicht wird. Des Weiteren konfigurieren wir für jede Route ein eigenes HTML-Template und einen JavaScript Controller. Für die Übersichtsseite verwenden wir die Datei *app/partials/customer-list.html* als Template und *CustomerListCtrl* als Controller und für die Detailseite *app/partials/customer-details.html* und *CustomerDetailCtrl*. Schließlich geben wir durch Aufrufen der *otherwise*-Methode eine Default-Route an. Dadurch erreichen wir beim initialen Aufruf von http://localhost:8080/crm-demo/ eine Weiterleitung auf http://localhost:8080/crm-demo/#/list.

Controller und Services

Die Controller, in denen die Anwendungslogik beschrieben wird, finden sich in der Datei *controllers.js*. Der Datenzugriff, also hauptsächlich die Kommunikation mit dem Backend, wird in der Datei *services.js* beschrieben. Hier stellen wir eine Factory für Customer-Objekte bereit, die intern den AngularJS-*$resource*-Service verwendet, um auf das REST-Backend zuzugreifen. Da der *$resource*-Service fast die ganze Arbeit für uns erledigt, müssen wir lediglich den Basis-URL der Kundenobjekte angeben und die *update*-Methode konfigurieren. Standardmäßig stellt der *$resource*-Service die Methoden *get*, *save*, *query*, *remove* und *delete* zur Verfügung. Da wir aber zwischen Speichern und Aktualisieren unterscheiden wollen, erweitern wir die „Customer"-Ressource um die Methode *update* und geben *PUT* als HTTP-Methode an. So sorgen wir dafür, dass neue Objekte durch HTTP POST erzeugt, und bestehende durch HTTP PUT aktualisiert werden.

Aufbau der Seite

Wird die Anwendung nun aufgerufen, leitet der RouteProvider den Aufruf automatisch an den URL http://localhost:8080/crm-demo/#/list weiter. Durch die vorher beschriebene Konfiguration in *app.js* wird nun der Controller *CustomerListCtrl* instanziiert und das HTML-Fragment *customer-list.html* geladen. AngularJS übergibt dem Controller die *$scope*-Variable und eine Instanz der Customer-Ressource. Nun kann der Controller die Liste aller Customer-Objekte laden. Die *$scope*-Variable ist dabei das Bindeglied zwischen Controller und View, wir fügen also die geladenen Customer-Objekte dem *$scope* als Attribut namens *customers* hinzu. Im View können wir dann über *ng-repeat* auf die Liste der Kundenobjekte zugreifen und über ihre Elemente iterieren. Das Interessante dabei ist, dass Änderungen am Modell automatisch zum View propagiert werden. Sobald der Aufruf ans Backend die Customer-Objekte zurückliefert, tauchen die Daten wie durch Zauberhand im View auf, ohne dass wir uns um die asynchrone Abarbeitung des HTTP-Requests kümmern müssen.

Innerhalb der mit *ng-repeat* definierten Schleife können wir nun mit doppelten geschweiften Klammern auf die Elemente der Kundenliste zugreifen. So sorgt *{{ customer.lastName }}* beispielsweise für die Ausgabe des Nachnamenattributs des aktuellen Kundenobjekts. Zur bidirektionalen Bindung, also um beispielsweise Änderungen in Formularfeldern zum Modell zu propagieren, stellt AngularJS das *ng-model*-Attribut bereit. Zum Aufrufen von Methoden im Controller kann das *ng-click*-Attribut verwendet werden, indem als Wert der Methodenname (z. B. *save()*) angegeben wird. Um ein besseres Gefühl für das Arbeiten mit AngularJS zu bekommen und die genannten Attribute besser zu verstehen, bietet es sich an, den Quellcode der Beispielanwendung in Ruhe anzuschauen und evtl. ein wenig damit herumzuspielen. Wie wäre es zum Beispiel mit einem Bestätigungsdialog vor dem Löschen von Kunden oder einer weiteren Seite zur Verwaltung von Firmen?

Fazit

Ich hoffe, diese kleine Beispielanwendung hat Lust auf mehr gemacht. AngularJS, REST und Java EE 6 bilden in meinen Augen einen sehr attraktiven Technologiestack. Sowohl Java EE 6 als auch AngularJS nehmen uns viel Entwicklungsarbeit ab, was weniger Code bedeutet und uns damit mehr Zeit für das Wesentliche gibt – die Features. Und die resultierenden Anwendungen sind dank guter Testbarkeit und Verzicht auf Server-State robust und skalierbar. Vielleicht lohnt sich also doch ein Ausflug aus der statisch typisierten Java-Welt, um sich vom erstaunlich gut funktionierenden Chaos des größten verteilten Systems der Welt – des Internets – inspirieren zu lassen.

Nils Preusker ist Softwareentwickler und technischer Berater bei der camunda services GmbH. Neben den Themen Prozessmodellierung und -automatisierung interessiert er sich für alles rund um die Mensch-Maschine-Interaktion und ist begeisterter Anhänger der agilen Softwareentwicklung.

Links & Literatur

[1] Webber, Jim; Parastatidis, Savas; Robinson, Ian: „REST in Practice – Hypermedia and Systems Architecture", O'Reilly Media, 2010, ISBN: 978-0-596-80582-1

[2] http://www.jboss.org/jbossas/downloads/

[3] https://github.com/nilspreusker/crm-demo

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

[5] http://jackson.codehaus.org/

[6] https://github.com/jboss-jdf/jboss-as-quickstart/tree/master/h2-console

[7] https://github.com/angular/angular-seed

[8] http://twitter.github.com/bootstrap/assets/bootstrap.zip

[9] http://code.jquery.com/jquery-1.9.0.min.js

Vorteile mehrstufiger Integration und hoher Automatisierung

Continuous Integration in großen Projekten

Continuous Integration ist ein bewährter Ansatz moderner Softwareentwicklung, doch in großen Projekten kann ein zentraler Build schnell instabil werden. Ein instabiler Build senkt die Produktivität der Entwickler und bringt die Projektziele in Gefahr. Release und Deployment sind grundlegende Aufgaben, die durch unzureichende Automatisierung und suboptimale Zusammenarbeit zwischen Entwicklungsteam und IT-Betrieb schnell ein Projekt ins Stolpern bringen können. Was sind die Gründe für diese Probleme und mit welchen Ansätzen kann man sie vermeiden?

von Kai Spichale

Continuous Integration (CI) ist ein anerkannter und weitverbreiteter Ansatz moderner Softwareentwicklung, bei dem jedes Teammitglied regelmäßig seine Arbeitsfortschritte in eine gemeinsame Codebasis integriert. Jede Integration führt zu einer neuen Version, die durch einen automatisierten Build gebaut und getestet wird. Ziel dieses Ansatzes ist es, fehlerhafte Komponenten und Integrationsprobleme so schnell und so früh wie möglich entdecken und beheben zu können. Martin Fowler beschreibt in einem Artikel [1] CI anhand der Entwicklung eines neuen Features für eine Anwendung. Als wichtige Bausteine von CI nennt er die Automatisierung von Build, Test und Deployment, und den Einsatz eines Versionsverwaltungssystems, eines CI-Servers sowie einer Testumgebung, die der tatsächlichen Produktivumgebung ähnelt. Das neue Feature wird in der lokalen Arbeitskopie eines Entwicklers umgesetzt. Alle Änderungen werden mindestens einmal täglich in den Trunk integriert und anschließend vom CI-Server gebaut. Vor dem Commit sollten alle zwischenzeitlich gemachten Änderungen im Trunk mit der Arbeitskopie gemergt und der Build erfolgreich lokal ausgeführt worden sein. Falls trotz aller Sorgfalt der zentrale CI-Build fehlschlägt, ist es Aufgabe des Entwicklers bzw. des gesamten Teams, den Fehler schnellstmöglich zu beheben.

Die von Fowler beschriebene CI-Spielart funktioniert hervorragend für kleine Teams, doch mit zunehmender Teamgröße treten Probleme auf, die die Produktivität der Entwickler reduzieren und Fortschritte blockieren können. Ein wichtiges Prinzip von CI ist der regelmäßige Commit der gemachten Änderungen ins gemeinsame Repository, denn im Allgemeinen sind häufige Integrationen mit kleinen Änderungen einfacher zu beherrschen, als selten durchgeführte Integrationen mit umfangreichen Änderungen. Doch wenn die Anzahl der Commits mit der Anzahl der Entwickler wächst, können folgende Probleme entstehen:

- **Instabilität des Builds**: Je mehr Entwickler beteiligt sind und je häufiger diese committen, desto mehr Build-Fehler treten auf. Angenommen alle Entwickler arbeiten höchst professionell und nur zwei Prozent ihrer täglichen Commits führen zu einem Build-Fehler, dann wäre die Wahrscheinlichkeit, dass 100 Commits nicht zu einem Build-Fehler führen mit 0,13 trotzdem relativ gering. Nach der Jenkins-Terminologie [2] werden Builds mit Kompilationsfehlern als fehlgeschlagen bezeichnet. Builds ohne Kompilationsfehler bezeichnet man als erfolgreich. Instabile Builds sind erfolgreiche Builds, bei denen andere Probleme, wie das Nichtbestehen eines Komponententests, festgestellt wurden. Stabile Builds sind Builds, bei denen keine Probleme festgestellt wurden.
- **Permanente Veränderung**: In großen Projekten konzentriert sich die Arbeit der meisten Entwickler auf eine oder wenige Komponenten, die sie lokal selber bauen. Die Artefakte der anderen Komponenten des Systems werden ihnen beispielsweise durch ein Maven Repository zur Verfügung gestellt. Die Artefakte im Repository werden durch die CI-Builds aktualisiert, sodass sich das System im Rhythmus des CI-Builds verändert. Das ist auch der Rhythmus, mit dem die Entwickler integrieren müssen, auch wenn sie aufgrund ihres Arbeitsfortschritts temporär nicht dazu bereit sind. Ein anderes Problem entsteht, wenn zu viele Entwickler auf der gleichen Codelinie arbeiten. Das ständige Aktualisieren der Arbeitskopie kostet Zeit und kann schnell frustrieren. Anstatt

täglich in Summe eine Stunde mit dem Mergen von vielen kleinen Änderungen zu verbringen, könnte ein anderer Rhythmus mit gesammelten Änderungen Zeit sparen helfen.

- **Mehrere Änderungen pro Build**: Wenn der zentrale Build in Schieflage gerät, aber jeder Commit einzeln gebaut wird, kann man leicht feststellen, mit welcher Änderung das Problem erstmalig auftrat. Die Logfiles des CI-Servers in Kombination mit der Änderungshistorie des Versionsverwaltungssystems können sehr hilfreich bei der Fehleranalyse sein. In großen Teams mit hoher Commit-Rate ändert sich das System jedoch ständig, sodass aus Zeitgründen nicht jeder Commit einzeln gebaut werden kann. Alternativ könnte man deswegen im 15-Minuten-Takt prüfen, ob Änderungen vorliegen und gegebenenfalls den Build starten. Falls aber dann ein Fehler auftritt und zwischen zwei Builds mehrere Änderungen von vielleicht sogar unterschiedlichen Entwicklern durchgeführt worden sind, wird die Fehleranalyse deutlich komplexer.
- **Blockierung durch Fehleranalysen**: Eine einzelne defekte Komponente kann das Deployment und den Test eines Teilsystems oder sogar des Gesamtsystems unmöglich machen. In einer derartigen Situation wird die Arbeit des ganzen Entwicklungssystems behindert, bis das Problem behoben ist. In schwierigen Fällen kann die Korrektur nicht innerhalb eines Arbeitstages behoben werden. Eventuell müssen Änderungen rückgängig gemacht werden, um den Build-Zyklus fortsetzen zu können.
- **Frustration statt Sorgfalt**: In großen Projekten mit instabilen Builds entsteht der Eindruck, dass ständig irgendetwas kaputt ist. Entsprechend der Broken-Window-Theorie [3] sinkt in solchen Situationen die Sorgfalt der Entwickler sogar noch weiter. Beispielsweise ist das Committen auf einen gebrochenen Build ein Anti-Pattern, denn wenn der Build anschließend erneut fehlschlägt, können unbemerkt weitere Fehler hinzukommen. Doch Zeitdruck und häufige Build-Fehler senken die Hemmschwelle, dieses Anti-Pattern zu beachten.

Build-Fehler eingrenzen

Wenn ein Entwickler ein neues Feature entwickelt oder einen Fehler behebt, dann verändert er eine oder mehrere Dateien. Zwischen allen Änderungen existiert eine starke Koppelung, sodass der Entwickler die Änderungen allein ausführt. Erst wenn die Arbeit abgeschlossen oder zumindest ein brauchbarer Zwischenstand erreicht ist, werden die Änderungen committet und sichtbar für das restliche Team. Das heißt, der Entwickler schützt die anderen Teammitglieder während dieses Umbaus vor inkonsistenten und nicht getesteten Änderungen. Umgekehrt sollte in dieser Zeit der Entwickler auch unabhängig von den Änderungen seiner Kollegen sein. Das heißt, der Entwickler sollte erst dann äußere Änderungen integrieren, wenn er dazu bereit ist.

Das gleiche Prinzip nur auf anderer Ebene wird für 3rd-Party-Bibliotheken und Produkte wie beispielsweise Datenbanken, virtuelle Maschinen und Applikationsserver eingesetzt. Es handelt sich hierbei um Software, die von anderen Teams bzw. anderen Unternehmen mit eigenen Release-Plänen und Build-Zyklen entwickelt werden. Nur bestimmte Release- und Produktversionen werden von Kunden eingesetzt. Die anderen unzähligen Versionen, die während der Entwicklung entstehen, werden nie das Team bzw. das Unternehmen verlassen.

Dieser Ansatz kann auch auf große Softwareprojekte angewendet werden, indem Teams auf separaten Codelinien arbeiten und diese mit eigenen CI-Builds bauen und testen, um die Kopplung zwischen den Teams zu verringern.

Mehrstufiges CI

Abbildung 1 zeigt schematisch einen mehrstufigen CI-Ansatz. Die Builds der ersten Stufe führen die Ent-

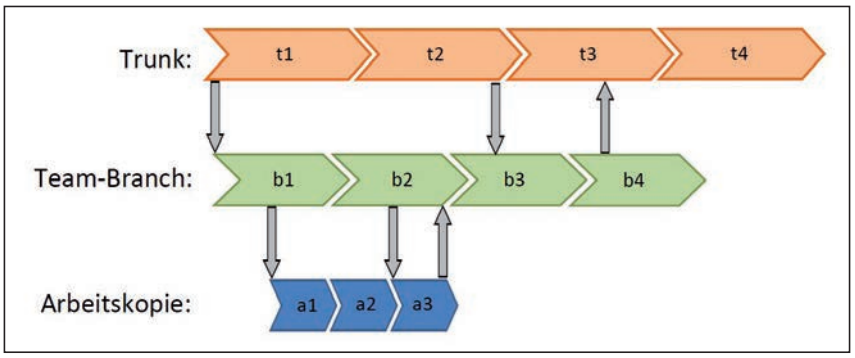


Abb. 1: Mehrstufige Integration

wickler lokal aus, um die Änderungen in ihrer privaten Arbeitskopie zu bauen und zu testen. Diese Builds finden im Minutentakt statt. Die zweite Stufe bilden die Builds auf dem Team- bzw. Feature-Branch. Jedes Team arbeitet auf einem separaten Branch und kann diesen kontinuierlich auf einem CI-Server bauen. Die Integrationen werden im Stundentakt durchgeführt. Die letzte Integrationsstufe sind die täglichen Builds auf dem Trunk, der Hauptentwicklungslinie. Von unten nach oben nehmen Reifegrad und Stabilität zu. Der lokale Build des Entwicklers ist am instabilsten, denn hier findet die eigentliche Arbeit statt. Entwickler brauchen Freiheit zum Experimentieren und Fehler müssen auf dieser Stufe tolerierbar sein. Bei testgetriebener Entwicklung gehören temporäre lokale Build- bzw. Testfehler sogar zum methodischen Vorgehen. Die Entwickler integrieren ihren Code in den Branch ihres Teams. Falls daraufhin der CI-Build fehlschlägt, hat das keinen Einfluss auf die anderen Teams. Wenn der Team-Branch stabil ist, wird dieser mit dem Tunk zusammengeführt. Der Team-Branch ist einerseits eine zusätzliche Stufe und bedeutet Mehraufwand bei der Integration. Auf der anderen Seite werden durch diese Stufe die Teams voneinander entkoppelt.

Die zusätzliche Stufe verlangsamt die kontinuierliche Integration. Sie sollte aber trotzdem oft und regelmäßig durchgeführt werden, weil ansonsten wieder die Integrationsprobleme auftreten, die man ursprünglich mit der Einführung von CI vermeiden wollte. Jez Humble und David Farley [4] empfehlen deswegen die Änderungen der Hauptentwicklungslinie täglich in die Branches zu integrieren. Ein Ansatz mit Branches ist empfehlenswert, wenn die Aufwände für Merging überschaubar sind. Das ist beispielsweise dann der Fall, wenn die Teams auf unterschiedlichen Komponenten arbeiten und teamübergreifende Konflikte beherrschbar bleiben.

Build-Pipelines

Ein häufiges Problem großer und komplexer Projekte ist die Build-Dauer. Ein CI-Build wird im Laufe eines Projekts hunderte oder tausende Male ausgeführt. Daher ist es wichtig, dass der CI-Build möglichst schnell Feedback über Erfolg oder Misserfolg der gemachten Änderung liefert. Aber was kann man machen, wenn der Build mit Integrationstest dreißig Minuten oder länger dauert? Eine Lösung ist das Aufteilen des Builds in mehrere kleinere Abschnitte, die in definierter Reihenfolge ausgeführt werden. Nur wenn ein Abschnitt erfolgreich ist, stößt er seinen Nachfolger an. Das Durchführen der kompletten Build-Pipeline kann, wenn keine Builds parallelisiert werden können, sogar noch zeitaufwändiger als ein einzelner umfassender Build sein, aber durch die stärkere Strukturierung kann beispielsweise die Reihenfolge der Tests gesteuert werden, sodass früher Feedback verfügbar ist. Beginnen könnte man eine Build-Pipeline mit einem schnellen Commit-Build, der den Quellcode kompiliert und wichtige Funktionen durch Unit Tests verifiziert. Im nächsten Schritt könnten umfassendere Integrationstests folgen. Diese Tests sind relativ zeitaufwändig, falls hierfür die Anwendung in eine Testumgebung deployt werden muss. Eventuell wird auch eine Datenbank verwendet, die zuvor konfiguriert und mit Testdaten gefüllt werden muss. Anschließend können Oberflächentests und nichtfunktionale Tests für Performance oder Hochverfügbarkeit durchgeführt werden. Das Erstellen von Codequalitätsmetriken hat i.d.R. geringere Priorität und kann am Ende durchgeführt werden.

Staging

Die Build-Pipeline kann mit dem Deployment der zuvor getesteten Software auf Staging-Umgebungen für weitere System- und Akzeptanztests zur Deployment Pipeline ausgebaut werden. In Projekten, in denen das Deployment sehr zeitaufwändig und kompliziert ist, wird das Testteam durch Tätigkeiten des Build-Teams immer wieder blockiert. Empfehlenswert ist der Einsatz mehrerer Testumgebungen, um diese abwechselnd mit der neuesten Version bespielen zu können. Das heißt, wenn auf einer Umgebung das Deployment durchgeführt wird, steht eine weitere für Systemtests zur Verfügung. Mehrere Testumgebungen sind auch dann notwendig, wenn die Software auf verschiedenen Plattformen getestet werden muss. Nur mit Staging-Umgebungen, die den späteren Produktivsystemen ähneln, können aussagefähige Testergebnisse erzielt werden. Staging-Systeme können für interne, externe, automatisierte und manuelle Abnahmetests verwendet werden.

Continuous Delivery und Automatisierung

Mit einem hohen Automatisierungsgrad kann die Deployment Pipeline für kontinuierliche Deployments bzw. Continuous Delivery eingesetzt werden, um mit geringem manuellen Overhead schnell, verlässlich und reproduzierbar neue Softwareversionen mit Erweiterungen oder Fehlerkorrekturen auszuliefern. Um dieses Ziel zu erreichen, sollte mit dem Deployment in produktivsystemähnlichen Umgebungen möglichst früh während der Entwicklung der Software begonnen werden. Während die Software Stück für Stück entsteht, können die Skripte für Deployment und Konfigurationsmanage-

ment kontinuierlich erweitert und angepasst werden. Manuelles Deployment und manuelles Konfigurationsmanagement für Produktivsystemumgebungen sind bekannte Anti-Pattern [4] und sollten vermieden werden. Manuelle Vorgänge sind fehleranfällig und nicht reproduzierbar. Es besteht ständig die Gefahr, dass Dokumentationen unbemerkt veralten, wenn beispielsweise unter Zeitdruck das Deployment oder die Konfiguration aufgrund einer neuen Abhängigkeit zu einem externen System oder durch eine neue Komponente verändert wird. Das Wissen über solche kleinen Details verstreut sich im Projektverlauf auf viele Köpfe, die nicht jederzeit zur Verfügung stehen.

Ein Nebeneffekt der Automatisierung ist Standardisierung. Manuell durchgeführte Tätigkeiten können sich von Person zu Person und von Mal zu Mal unterscheiden, dabei besteht immer die Gefahr, dass unbemerkt Fehler gemacht werden. Teammitglieder müssen die Tätigkeiten mit allen Details lernen und Änderungen müssen kommuniziert werden.

Fehlende Automatisierung kann als technische Schuld (Englisch: technical debt) [5] angesehen werden. Die Zinsen sind immer dann zu zahlen, wenn ein manueller Vorgang mit hohem Ressourcenaufwand durchgeführt werden muss. Wenn das Build-Team zu sehr auf Pump lebt, wird es allmählich handlungsunfähig, denn es ist zunehmend mit der Zinstilgung, also mit dem Leisten von wiederkehrenden manuellen Aufgaben, beschäftigt und das verbleibende Budget reicht nicht aus für neue Aufgaben. Angenommen ein Projekt setzt einen Applikationsserver ein, dessen Konfiguration verändert werden muss. Die Änderung ist trivial, doch durch die fehlende Automatisierung wird das Anpassen aller Testumgebungen und aller lokalen Installationen der Entwickler in großen Projekten sehr aufwändig.

Idealerweise werden virtualisierte Testumgebungen mit automatisierter Provisionierung eingesetzt, um deren Anzahl flexibel nach aktuellem Bedarf des Projekts anpassen zu können. Falls beispielsweise der Testaufwand unterschätzt wurde und zusätzliche Testumgebungen notwendig werden, können diese schnell bereitgestellt werden. Auch für Performancetests bietet der Einsatz von virtuellen Maschinen Vorteile, wenn diese mit unterschiedlich starker Hardware gestartet werden können.

DevOps

Viele Probleme in Projekten sind nicht technischer, sondern soziologischer Natur [6]. Das fällt vor allem beim Deployment von Software in Test- und Produktionsumgebungen auf. In kleinen Projekten oder kleinen Organisationen ist das Entwicklungsteam (Dev) selbst für Infrastrukturaufgaben und IT-Betrieb (Ops) verantwortlich. In großen Organisationen oder großen Projekten werden jedoch Entwicklung- und Infrastrukturaufgaben durch verschiedene Abteilungen oder Teams ausgeführt. Während das Entwicklungsteam das Ziel hat, die Software schnellstmöglich auszuliefern, ist das Infrastrukturteam an Stabilität und störungsfreiem Betrieb interessiert. Infolgedessen kann es zu Spannungen zwischen beiden Gruppen kommen. Beide Seiten versuchen sich zu schützen und sicherzustellen, dass sie für eventuell auftretende Probleme nicht verantwortlich sind. DevOps [7] ist ein Ansatz, um die Kommunikation und Zusammenarbeit von Softwareentwicklung und IT-Betrieb zu verbessern. Zwischen beiden Arbeitsbereichen existieren wechselseitige Abhängigkeiten und Überschneidungen, sodass nur durch intensive Kollaboration Build, Qualitätskontrolle, Release und Bereitstellung aus Sicht beider Stakeholder erfolgreich durchgeführt werden können.

Fazit

Wichtige Erfolgsfaktoren großer Projekte sind stufenweise durchgeführte kontinuierliche Integrationen, ein hoher Grad an Automatisierung und effektive Kollaboration von Entwicklungs- und Infrastrukturteams. Regelmäßige Integrationen sind einfacher zu managen als seltene und verbessern die Kommunikation innerhalb des Entwicklungsteams. Jedoch sind häufige Build-Fehler störend, sodass eine Erweiterung des klassischen CI-Ansatzes um zusätzliche Integrationsstufen von Vorteil sein kann, weil die Integration der Änderungen einzelner Teams besser gesteuert wird. Die Automatisierung von Vorgängen ist wichtig für Projekte und Organisationen, unabhängig von ihrer Größe. Je komplexer und aufwändiger Build, Test, Konfiguration und Bereitstellung werden, desto wichtiger ist deren Automatisierung, um diese Aufgaben schnell, verlässlich und reproduzierbar durchführen zu können. In großen Projekten und Organisationen findet eine stärkere Spezialisierung und Aufgabenteilung zwischen Dev und Ops statt, als in kleineren Teams. Daher ist es wichtig, gezielt in großen Projekten durch eine DevOps-Initiative die Kollaboration zwischen beiden Seiten zu fördern.

Kai Spichale ist Senior Software Engineer beim IT-Dienstleistungs- und Beratungsunternehmen adesso AG. Sein Tätigkeitsschwerpunkt liegt in der Konzeption und Implementierung von Softwaresystemen auf Basis von Java EE und Spring.

Links & Literatur

[1] http://martinfowler.com/articles/continuousIntegration.html

[2] http://wiki.hudson-ci.org/display/HUDSON/Terminology

[3] Hunt, Andrew; Thomas, David: „The Pragmatic Programmer: From Journeyman to Master“, Addison-Wesley Professional, 1999

[4] Humble, Jez; Farley, David: „Continuous Delivery“, Addison-Wesley Signature, 2010

[5] http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx

[6] DeMarco, Tom; Lister, Timothy: „Peopleware: Productive Projects and Teams“, Dorset House, 1999

[7] Peschlow, Patrick: „Die DevOps-Bewegung“, in Java Magazin, 1.2012, S. 32–40

Kolumne: EnterpriseTales

von Lars Röwekamp und Arne Limburg

Wie frag' ich die Datenbank? Typsichere dynamische Abfragen mit JPA

Die Java Persistence Query Language (JPQL) ist sehr gut geeignet, statische Abfragen, die schon zur Entwicklungszeit feststehen, zu formulieren, sodass der Persistenz-Provider daraus SQL-Abfragen generieren kann. Wie sieht es aber in Situationen aus, in denen sich die Abfrage in Abhängigkeit von Laufzeitbedingungen ändert? Welche Möglichkeiten gibt es, dynamische JPA-Abfragen zu formulieren und dennoch bereits in der IDE und zur Compile-Zeit sicher zu sein, keinen Syntaxfehler eingebaut zu haben? Jeder kennt wohl den Use Case: Es gibt eine Suchmaske mit mehreren Feldern, z. B. „Vorname" und „Nachname". Gesucht werden soll die zugehörige Person. Eine Abfrage gegen die Datenbank muss man in Abhängigkeit davon absetzen, ob eines der Felder befüllt ist oder es beide sind. Wie kann man einen solchen Use Case mit JPA umsetzen, ohne auf Typsicherheit zu verzichten?

Für statische Abfragen ist in der heutigen Toolwelt eine recht gute Unterstützung zur Vermeidung von Tipp- und/oder Semantikfehlern vorhanden. Die meisten IDEs können mittlerweile JPQL-Abfragen bereits zur Entwicklungszeit analysieren und dann nicht nur direkt auf Syntaxfehler hinweisen, sondern finden sogar Unstimmigkeiten mit dem verwendeten JPA-Modell (falsche Entity-Namen, falsch verwendete Attribute etc.).

Auch wenn einige IDEs noch Schwierigkeiten haben, einen JPQL-String zu erkennen, wenn er direkt an *entityManager.createQuery(...)* übergeben wird, gelingt es mit den entsprechenden Plug-ins mittlerweile jeder IDE *@NamedQueries(...)* zu parsen und dort Fehler zu erkennen. In Eclipse z. B. muss man dazu nur das JPA-Facet aktivieren. Schon alleine deshalb lohnt es sich, Named Querys den Vorzug vor Querys zu geben, die man direkt über *entityManager.createQuery(...)* ausführt.

Vor diesem Hintergrund könnte man auf die Idee kommen, auch obigen Use Case mit Named Querys abzubilden. Man bräuchte ja nur drei Stück: eine, die nur nach dem Vornamen sucht, eine, die nur nach dem Nachnamen sucht, und eine, die nach Vor- und Nachnamen sucht.

Wenn es sich bei den optionalen Parametern nur um Vor- und Nachnamen handelt, mag das ja noch funktionieren. Man stößt mit diesem Vorgehen aber recht schnell an Grenzen. Spätestens ab dem dritten optionalen Parameter erhöht sich die Anzahl der benötigten Abfragen auf sieben, und man benötigt eine andere Technik, um die Abfrage zu definieren.

Auch der Ansatz, eine solche Anforderung über String-Konkatenation zu lösen, ist auf Dauer zum Scheitern verurteilt, weil die Wartbarkeit enorm leidet. Ganz zu schweigen davon, dass man damit in der Regel alle oben genannten Syntax- und Semantikchecks der IDE sofort aushebelt. Wie also soll dann mit einer solchen Situation umgegangen werden?

Das Criteria API

Mit der Version 2.0 von JPA ist die Antwort des Standards auf diese Frage das Criteria API [1]. Mit ihm ist

Porträt

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

es tatsächlich möglich, Query-Konstrukte dynamisch aufzubauen, ohne dass String-Konkatenation verwendet werden muss (Listing 1). Ab JPA 2.1 wird man auch Update- und Delete-Statements mit dem Criteria API formulieren können.

Etwas merkwürdig mutet beim Criteria API allerdings der Zugriff auf Attribute eines Objekts an: *person.get(Person_.firstName)*. Die Klasse *Person_* (nein, der Unterstrich ist kein Tippfehler) ist dabei eine automatisch generierte Klasse des ebenfalls mit JPA 2.0 eingeführten Metamodells, die typsicheren Zugriff auf eben diese Attribute bieten soll. Der leichter zu lesende Zugriff über die Angabe des Attributnamens als String, also *person.get("firstName")* ist zwar auch möglich und auch deutlich lesbarer, schützt aber nicht vor Tippfehlern.

Der etwas komplizierte Zugriff auf Attribute einer abzufragenden Entität über das Metamodell ist auch der erste Grund, warum Criteria Querys recht schnell unübersichtlich werden. Während man in JPQL für den Zugriff auf das Attribut *firstName* der Entität *Person* einfach *person.firstName* schreiben kann, muss man im Criteria API besagten Umweg über die Metadatenklasse, nämlich *person.get(Person_.firstName)* verwenden. Dies erscheint bei einem so einfachen Beispiel zwar nur unwesentlich länger. Wenn man aber, wie es normalerweise der Fall ist, viele solcher Konstrukte in einer Abfrage verwenden muss, wächst der Code schnell an und wird damit schlechter wartbar. Der zweite Grund, warum das Criteria API schwer zu lesen ist, ist die Entscheidung des JPA Specification Committees, für das Erstellen von Query-Konstrukten den *CriteriaBuilder* einzuführen. Er erfordert die Verwendung einer funktio-

Listing 1

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Person> personQuery = cb.createQuery(Person.class);
Root<Person> person = personQuery.from(Person.class);
List<Predicate> condition = new ArrayList<Predicate>();
if (firstName != null) {
  condition.add(cb.equal(person.get(Person_.firstName), firstName));
}
if (lastName != null) {
  condition.add(cb.equal(person.get(Person_.lastName), lastName));
}
return entityManager.createQuery(
    personQuery
      .select(person)
      .where(condition.toArray(new Predicate[0])))
  .getResultList();
```

nalen Syntax, bei der das Prädikat vorangestellt ist. Dies entspricht weder dem deutschen noch dem englischen Satzbau, weshalb sich solche Query-Konstrukte deutlich schlechter lesen lassen als ihr JPQL-Pendant. Schreibt man in JPQL z. B. *person.firstName = :firstName*, so lässt sich das leichter lesen als die Criteria-Variante: *cb.equal(person.get(Person_.firstName), firstName)*. Auch hier gilt wieder, dass man in einer Criteria-Abfrage in der Regel nicht nur *ein* solches Konstrukt vorfindet, sondern gleich eine ganze Reihe, was eine Criteria Query in der Summe deutlich unleserlicher macht als das zugehörige JPQL.

Querydsl

Genau diese Kritikpunkte geht das Open-Source-Framework Querydsl [1] an. Wie das Criteria API basiert es auf der Generierung eines Metadatenmodells. Im Gegensatz zur JPA-Standardvariante sind die entstehenden Abfragen durch ein Fluent-API aber deutlich leserlicher. Der bereits oben erwähnte Vergleich des Vornamens, der im Criteria API recht kompliziert aussah, nämlich *cb.equal(person.get(Person_.firstName), firstName)*, lässt sich mit Querydsl deutlich leserlicher formulieren: *person.firstName.eq(firstName)*. Allein dieser kleine API-Unterschied zwischen dem Criteria API und Querydsl macht dynamische Abfragen mit Querydsl deutlich leserlicher und damit wartbarer, als es im JPA-Standard möglich ist.

Einen kleinen Nachteil hat Querydsl dabei: Das implizite Joinen über Many-to-One-Beziehungen, wie es in JPQL z. B. mit *person.address.street* und mit dem Criteria API (etwas unleserlicher) mit *person.get(Person_.address).get(Address_.street)* möglich ist, geht in Querydsl nicht. Hier muss ein Join immer explizit formuliert werden (Listing 3).

Über die Unterstützung von JPA hinaus bietet Querydsl noch Adapter zum Bauen von nativem SQL, eine Anbindung von JDO und von nativem Hibernate. Letzteres dürfte vor allem für diejenigen interessant sein, die über den JPA-Standard hinaus Hibernate-spezifische Features nutzen, die sie auch in den dynamischen Abfragen nicht missen wollen. Zu guter Letzt bietet Querydsl sogar ein Modul für Abfragen gegen die NoSQL-Datenbank MongoDB.

Listing 2

```
QPerson person = QPerson.person;
List<Predicate> condition = new ArrayList<Predicate>;
if (firstName != null) {
  condition.add(person.firstName.eq(firstName));
}
if (lastName != null) {
  condition.add(person.lastName.eq(lastName));
}
return new JPAQuery(entityManager)
  .from(person)
  .where(condition.toArray(new Predicate[0]))
  .list(person);
```

Listing 3

```
QPerson person = QPerson.person;
QAddress address = QAddress.address;
person.join(person.address, address);
return new JPAQuery(entityManager)
  .from(person)
  .leftJoin(person.address, address)
  .where(address.street.eq(street))
  .list(person);
```

Einsatz in Spring Data

Wie wir im vorherigen Abschnitt gesehen haben, versucht Querydsl mit der großen Auswahl an Zielplattformen einen Abfragemechanismus zur Verfügung zu stellen, der deutlich über JPA hinausgeht. Damit passt er hervorragend in den Datenzugriffsableger des Spring Frameworks: Spring Data. Auch dieses versucht, vom Java-EE-Standard zu abstrahieren und eine gemeinsame Zugriffsschicht für JPA, JDO, JDBC, MongoDB u. a. zu bieten. Daher ist es naheliegend, dass Querydsl ein First-Class-Citizen in Spring Data ist. Mit Spring Data lassen sich Abfragen in Querydsl direkt absetzen, ohne Rücksicht auf die darunterliegende Persistenztechnologie. Das eigene Spring-Data-Repository muss nur das Interface *org.springframework.data.querydsl.QueryDslPredicateExecutor* implementieren, und schon kann man Querydsl-Prädikate direkt an das Repository übergeben, das daraus JPA-Abfragen erzeugt.

Fazit

Für dynamische Datenbankabfragen bietet JPA das Criteria API, das allerdings vor allem für komplexe Abfragen schnell recht unübersichtlich wird. Mit Querydsl haben wir hier eine interessante Alternative vorgestellt, mit der es möglich ist, deutlich lesbare dynamische Abfragen zu definieren.

Wer seine Datenzugriffsschicht bereits von Spring Data generieren lässt, muss nur ein weiteres Interface definieren, um mit dem Repository Querydsl-Abfragen absetzen zu können. Aber auch für alle anderen, die in ihren Projekten komplexe, dynamische JPA-Abfragen benötigen, lohnt sich ein Blick in Querydsl, um die Wartbarkeit der dynamischen Abfragen im Vergleich zum Criteria API deutlich zu erhöhen.

Links & Literatur

[1] http://www.querydsl.com/

[2] http://docs.oracle.com/javaee/6/tutorial/doc/gjitv.html

[3] http://blog.springsource.org/2011/04/26/advanced-spring-data-jpa-specifications-and-querydsl/

Entwicklung einer Energiedatenmanagement-Anwendung mit Java EE 6

Java EE meets Smart Metering

Die Diehl-Metering-Systemsoftware [1] zur Erfassung von Verbrauchsdaten basiert auf der Java-EE-6-Technologie. Dieser Standard erlaubt es, den Fokus auf die Entwicklung der Geschäftslogik zu legen, da die Komplexität auf technologischer Ebene durch standardisierte Konzepte abstrahiert wird. Die dadurch verkürzten Entwicklungszyklen sollen einen schnelleren Return-on-Investment garantieren. Anhand der Kernanforderung, des Imports der Messdaten in die Datenbank, werden nachfolgend einige interessante Aspekte vorgestellt, die sich bei der Entwicklung ergeben haben.

von Andreas Bauer, Dino Tsoumakis und Tobias Zeck

Die Vorverarbeitung von Messdaten erfolgt im System über eine in C++ geschriebene Bibliothek. Bedient wird diese über Lua-Skripte, die in einer von der Bibliothek bereitgestellten Skripting-Umgebung definiert und ausgeführt werden. Ergebnisse der Skripte werden als JSON-String an den Aufrufer zurückgeliefert. Die Integration der Bibliothek erfolgt durch eine Resource-Adapter-Implementierung auf Basis der Java EE Connector Architecture 1.6 (JSR 322). Grundlage dieser Implementierung ist ein relativ einfaches Java Native Interface zur C++-Bibliothek. Über das Interface lassen sich Skriptumgebungen erzeugen, zurücksetzen oder zerstören. Des Weiteren bietet das Interface Methoden an, um Lua-Skripte in der Skriptumgebung zu definieren und auszuführen. Bei der Implementierung des Resource Adapters wurde folgende Strategie verfolgt:

- Eine Verbindung zu einem Legacy-System entspricht einer Skriptumgebung.
- Der Aufbau einer Verbindung entspricht der Instanzierung einer Skriptumgebung.
- Das Zurücksetzen einer Verbindung entspricht dem Initialisieren einer Skriptumgebung.
- Der Abbau einer Verbindung entspricht dem Zerstören einer Skriptumgebung (bzw. dem Freigeben des allozierten Speichers für eine Skriptumgebung).

Die Implementierung ist in drei Teilprojekte aufgeteilt:

- Ein Client-Interface
- Die eigentliche Resource-Adapter-Implementierung
- Integrationstests für die Resource-Adapter-Implementierung

Das Client-Interface wird in den Klassenpfad einer Enterprise-Applikation eingebunden. Über dieses Interface kann die Applikation Skriptumgebungen anfordern, mit diesen arbeiten und sie nach getaner Arbeit wieder zurückgeben. Im Applikationscode wird der Resource Adapter an den benötigten Stellen über das CDI-Framework injiziert.

Die eigentliche Resource-Adapter-Implementierung enthält neben den Klassen, die die Spezifikation implementieren, auch die C++-Bibliothek und die Klasse, die das Java Native Interface auf die Bibliothek bereitstellt. Da die Bibliothek im vorliegenden Anwendungsfall nur als passive Bibliothek verwendet wird und die Enterprise-Applikation immer der Akteur ist, ist der Resource Adapter als „outbound only“ implementiert. Beim Deployment des Resource Adapters wird die Bibliothek aus dem Archiv extrahiert und über die *System.load()*-Methode in die JVM geladen. Nachdem in der Domäne des Application Server der Connector-Connection-Pool eingerichtet und die Connector-Ressource definiert ist, können Enterprise-Applikationen den Resource Adapter nutzen. Als Connector erweitert der Resource Adapter den Application Server und stellt damit das Tor in die Systemlandschaft dar. Der Application Server verwaltet die Verbindungen zum Resource Adapter und übernimmt damit das Erstellen, Pooling und Zerstören von Skriptumgebungen. Die Enterprise-Applikation ist daher vollkommen zustandslos („stateless“) entwickelt und ist auf diese Weise skalierbar.

Die Integrationstests im dritten Projekt stellen die Funktionen der JNI-Klasse, den Bootstrap-Algorithmus und die Thread-Sicherheit der Skriptumgebungen sicher.

Transaktionale Verarbeitung

Der über den Resource Adapter bereitgestellte Pool an Skriptumgebungen wird über die Annotation *@Resource* in EJBs eingebunden:

```
@Resource(name = "jca/scripting")
ScriptingPool scriptingPool;
```

Da der JCA-Connector Transaktionen unterstützt und das verwendete Datenbank-Backend ebenfalls eine Transaktion verlangt, wäre der Einsatz von XA-Transaktionen nötig, um eine Transaktion über mehrere Ressourcen zu ermöglichen. Die Verarbeitung lässt sich in diesem Fall aber in zwei Schritte mit jeweils eigener Transaktion zerlegen und so der Einsatz von XA-Transaktionen vermeiden. Hierzu wird die EJB, die den Resource Adapter injiziert bekommt, mit *TransactionAttributeType.REQUIRES_NEW* annotiert und so für den JCA-Connector eine neue Transaktion gestartet. Die gesamte Transaktionssteuerung kann mithilfe der EJB-3-Annotationen deklarativ erfolgen. Programmatische Transaktionen über *begin*, *commit* und *rollback* sind nicht notwendig. Wichtig bei der Verwendung des JCA-Connectors ist ein sauberes Exception Handling, um Pool-Elemente auch im Fehlerfall wieder freizugeben und die Transaktion ordnungsgemäß zu beenden:

```
try {
  ScriptingEnvironment s = scriptingPool.getScriptingEnvironment();
} finally {
  try {s.close();} catch (Exception e) {}
}
```

JSON Parsing mit JAXB

Die Rückgabewerte des Resource Adapters werden als JSON-String bereitgestellt. Die im GlassFish 3.1 enthaltene JAXB-Implementierung erlaubt durch die eingebundene *jettison*-Bibliothek neben XML auch das Parsen von JSON-Strings. Dies ermöglicht es, auf weitere externe Libraries zu verzichten und sich auch hier an den JEE-6-Standard zu halten:

```
JAXBContext jc = JAXBContext.newInstance(Json.class);
Configuration config = new Configuration();
MappedNamespaceConvention con =
                                  new MappedNamespaceConvention(config);
Unmarshaller unmarshaller = jc.createUnmarshaller();

XMLStreamReader reader =
            new MappedXMLStreamReader(new JSONObject(jsonResult), con);
Json json = (Json) unmarshaller.unmarshal(reader);
```

Da die JSON-Strings im vorliegenden Anwendungsfall mehrere Megabytes groß werden können, ist die mit der Größe steigende Verarbeitungsdauer ein Problem. Die Lösung bringt hier das Zerlegen der großen JSON-Strings in mehrere kleinere, die sich unabhängig voneinander parallel verarbeiten lassen. Das Aufteilen übernimmt dabei ein einfaches *split()*, und die asynchronen Verarbeitungsmöglichkeiten seit EJB 3.1 durch die Annotation *@Asynchronous* machen die parallele Weiterverarbeitung sehr einfach.

CDI Events vs. Future Interface

CDI Events bieten hervorragende Möglichkeiten, um eine möglichst lose Kopplung zwischen Event Producer und Event Consumer zu erreichen. Der Event Consumer arbeitet aber zunächst immer noch im gleichen Thread wie der Event Producer, also synchron. Auch hier hilft die Annotation *@Asynchronous* weiter und ermöglicht eine parallele Verarbeitung durch den Event Consumer:

```
@Asynchronous
public void processNotification(@Observes ProgressEvent event) { ... }
```

Die Herausforderung ist, dass man mit dieser Fire-and-Forget-Methode nur wenig Kontrolle über die Ausführung hat und die Skalierbarkeit eingeschränkt ist, da viele wartende Events das System belasten können. Darüber hinaus können Events durch das Queuing bei einem Absturz verloren gehen. Für komplexe Businesslogik wie das parallele JSON-Parsen eignet sich das Future-Interface daher besser, um die Verarbeitung zu überwachen und die Skalierbarkeit durch Throttling zu verbessern:

Listing 1

```
@WebServlet( name = "ImportStateRegistration", urlPatterns = {"/import/*"},
          asyncSupported = true)
public class ImportStateRegistration extends HttpServlet {

   @Inject
   private Event<ImportProcessListener> importProcessController;

   @Override
   protected void doGet( HttpServletRequest request,
                  HttpServletResponse response)
        throws ServletException, IOException {
      // make this request asynchronous
      // -> this method completes but the response was not sent
      AsyncContext asyncContext = request.startAsync(request, response);
      // create a new import listener that will handle the response later
      // fire the import listener (in this case the ImportEventListener
      // is the observer)
      importProcessController.fire(
                     new ImportProcessListener(asyncContext));
   }
}
```

```
@Asynchronous
public Future<Result> processJson(String json) {
  ...
  return new AsyncResult<Result>(result);
}
```

Performance Monitoring mit Aspekten

Die Performance eines Systems ist immer ein wichtiger Punkt. Mithilfe von Interceptoren gibt es seit EJB 3 eine sehr elegante Methode, um den Programmfluss zu überwachen, ohne den Businesscode zu verändern. Hierzu verwenden wir einen Aspekt und annotieren die Methode mit *@AroundInvoke*:

```
@AroundInvoke
public Object measure(InvocationContext ic) throws Exception {
  try {
    ic.proceed();
  } finally {
    onMethodCompletion(...);
  }
}
```

Der Aspekt kann nun über die Annotation *@Interceptors(Aspect.class)* an eine Klasse gebunden werden, wodurch automatisch jeder Methodenaufruf durch den Aspekt umhüllt wird. Dieser wird somit immer aufgerufen, sobald eine Methode betreten wird.

Real Time Events – Echtzeit-Monitoring mit Comet

In der Applikation bildet der Import von neuen Daten und deren Persistierung die Kernaufgabe. Ein Import wird dabei von einem Client aus angestoßen (**Abb. 1**). Hierfür wurde eine REST-Schnittstelle in die serverseitige Kernanwendung implementiert. Der Client sendet ein Paket mit definierten Importparametern (enthält u. a. den URL zu den Daten) an den Server und startet so einen Importvorgang. Dem Import wird eine eindeutige ID zugewiesen. Unter Angabe dieser Import-ID kann sich der Client jederzeit über den Status seines Imports informieren. Er stellt dazu eine Anfrage an die genannte REST-Schnittstelle und registriert sich so als Listener. Ändert sich der Status eines Imports bzw. tritt ein HTTP-Time-out ein, werden alle registrierten Listener benachrichtigt. Da der Server erst bei neuen Ereignissen die Response sendet und sich der Client in der Regel sofort wieder mit dem Server verbindet, spricht man hierbei von einem Comet-Programmiermodell für Webapplikationen. Es basiert auf Ajax mit Long Polling und erlaubt ein Pushen von Informationen vom Server zu den Clients in Echtzeit.

Jegliche Importereignisse (Statusänderungen) werden serverseitig in einem Puffer gehalten. Hierbei werden sie aufsteigend nach ihrer Aktualität durchnummeriert. Der Client merkt sich den Index des jeweils letzten Events, zu dem er informiert wurde, und meldet diesen beim HTTP-Request mit an den Server. Der Server kann ihn nun über alle noch ausstehenden Statusänderungen informieren. Die zu übermittelnde Datenmenge wird so möglichst gering gehalten, und ein Verlust an Import-Progress-Paketen ist ausgeschlossen.

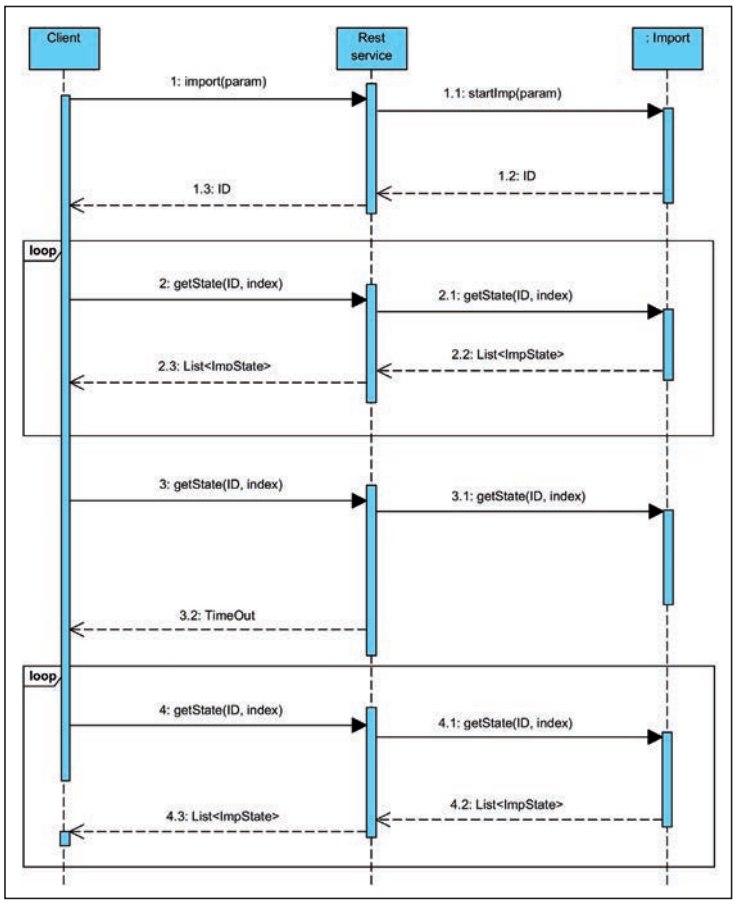


Abb. 1: Client-Server-Importinterface

Grundlage für das oben beschriebene Verfahren ist das *HttpServlet*, das den Request entgegennimmt, jedoch aufgrund des asynchronen Verhaltens nicht sofort eine Response sendet. Die Verbindung wird offen gehalten und erlaubt somit die geforderte Echtzeitbenachrichtigung der Clients (Listing 1).

JPA-Zugriff

Eine weitere Herausforderung, die im Rahmen des Projekts auftrat, stellte das Persistieren der Messwerte dar. Dabei wurde das Datenbankmodell aus einer bestehenden Applikation vorgegeben und sollte aus Kompatibilitätsgründen beibehalten werden (*Legacy-Datenbank-Modell*). Für den Einsatz im Java-EE-Umfeld wurde auf das Java Persistence API (JPA 2.0) gesetzt. Das bestehende Datenbankmodell wurde somit auf JPA-Entities gemappt.

Vom Datenbankschema vorgegeben, werden für das Persistieren der Daten zusätzliche Informationen benötigt. Diese Informationen befinden sich bereits in Form einer eigenen Tabelle in der Datenbank selbst. Architekturbedingt sind diese Zusatzinformationen jedoch nicht statisch und können daher auch nicht fest bei den Entities hinterlegt werden. Sie müssen bei jedem Applikationsstart aus der Datenbank gelesen und in Form ei-

nes Caches den Entities zur Verfügung gestellt werden (Hinweis: Eine Entity selbst darf nicht lesend auf die Datenbank zugreifen. Voraussetzung hierfür wäre, dass sie Zugriff auf einen Entity-Manager hätte. Dies ist jedoch nach Java-EE-Standard nicht erlaubt, [2]). Die zusätzlichen Informationen werden optimalerweise über ein Singleton Pattern zur Verfügung gestellt. Dabei erlaubt dank CDI die mit *@Produces* annotierte *expose()*-Methode das Injecten der reinen Zusatzinformationen selbst (Listing 2).

Das eigentliche Laden der Informationen aus der Datenbank erfolgt über die *loadPersistInfo()*-Methode. Wird diese beispielsweise in einer mit *@PostConstruct* annotierten Methode eines Start-up-Singleton aufgerufen, erfolgt das Laden der Zusatzinformationen bei jedem Hochfahren der Applikation automatisch (Listing 3).

Im nächsten Schritt müssen die Informationen den Entities zur Verfügung gestellt werden. Um Threading-Probleme zu vermeiden, arbeitet idealerweise jeder Thread auf einer eigenen Kopie des Caches. Der Einsatz von *java.lang.ThreadLocal<T>* liegt damit nahe. Er erzeugt pro Thread eine lokale, unabhängig initialisierte Kopie einer Variablen. Um das Initialisieren der Variablen selbst muss sich die Applikation allerdings noch kümmern. Zum Einsatz kommen hierbei Interceptoren. Diese erfüllen den Zweck, den bereits gelesenen Cache in die Thread-lokale Kopie zu speichern. Bindet man nun alle Klassen an diesen Interceptor an, die Daten in der Datenbank persistieren könnten, so ist sichergestellt, dass pro Thread immer ein gefüllter, lokaler Cache zur Verfügung steht, wenn dieser gebraucht wird. Wichtig ist hierbei noch anzumerken, dass das Aufräumen dieser Thread-lokalen Kopien keinesfalls übersehen werden darf. Es können sonst sehr schwer auffindbare Memory Leaks entstehen.

Listing 2

```
@Singleton
public class PersistInfoSingleton {
   ...
   private Map<String, Long> persistInfo = null;

   @Produces
   public Map<String, Long> expose() {
      return this.persistInfo;
   }

   public void loadPersistInfo(){...}
   ...
}
```

Listing 3

```
@Singleton
@Startup
public class PersistInfoCacheStarter {
   @Inject
   PersistInfoSingleton persistInfoCache;

   @PostConstruct
   public void populateCache() {
      persistInfoCache.loadPersistInfo();
   }
}
```

Fazit

Die importierten Smart-Metering-Messwerte werden auf einer webbasierten Oberfläche dargestellt. Diese basiert auf dem JavaServer-Faces-Framework PrimeFaces, mit dem in Sachen Darstellung und Handhabbarkeit sehr gute Erfahrungen gemacht werden konnten.

Durch den Einsatz von Java-Enterprise-Technologie konnte die Gesamtperformance des Importvorgangs im Vergleich zur bisher verwendeten Java-SE-Anwendung um ca. 400 Prozent gesteigert werden. Der äußerst geringe Arbeitsspeicherverbrauch des GlassFish-Applikationsservers erlaubt eine unproblematische Integration, auch wenn nur wenig Speicher zur Verfügung steht. Nach dem Start belegt der GlassFish nur etwa 30 MB Arbeitsspeicher. Mit laufender Applikation werden auch unter Stress kaum mehr als 300 MB Arbeitsspeicher benötigt.

Die in Smart-Metering-Anwendungen stetig wachsenden Datenmengen stellen die Datenhaltung (Big Data) vor eine große Herausforderung. Um die Performanz weiter zu steigern, wird derzeit der Einsatz von NoSQL-Datenbanken geprüft. Der Einsatz der Java-EE-6-Technologie hat sich in den Unternehmenseinheiten der Diehl Metering bewährt.

Andreas Bauer ist Systemarchitekt und Software Engineer bei Diehl Metering. Über die C/C++-Programmierung, insbesondere mit dem Qt-Framework, fand er seinen Weg hin zu Java und Java EE. Er beschäftigt sich derzeit mit der Entwicklung von Anwendungen im Java-Enterprise-Umfeld. Seine Schwerpunkte liegen in den Bereichen Deployment, Big-Data-Architekturen und JPA.

Dino Tsoumakis ist Systemarchitekt und Software Engineer bei Diehl Metering und beschäftigt sich bereits seit zehn Jahren mit Java und Datenbanken im Client-Server-Umfeld. Zu den Schwerpunkten zählen insbesondere mobile Datenkommunikation, Big-Data-Architekturen und Performanzoptimierung von Java-Enterprise-Anwendungen.

Tobias Zeck arbeitet als Systemarchitekt und Software Engineer bei Diehl Metering in Nürnberg. Er verfügt über mehrjährige Erfahrung im Bereich der Java-Entwicklung. Sein Schwerpunkt ist aktuell die Entwicklung von Anwendungen im Java-Enterprise-Umfeld. Des Weiteren beschäftigt er sich mit Big-Data-Architekturen.

Links & Literatur

[1] http://www.diehl-metering.com

[2] http://java.net/jira/browse/JPA_SPEC-24

Git in Kombination mit anderen Systemen

Ein wahrer Teamplayer

Softwareanwendungen können heutzutage nicht ohne Versionsverwaltung leben. Beim Großteil der Anwendungen werden dafür Systeme wie SVN oder CVS verwendet. Aber auch Git ist wegen seiner Flexibilität und vieler anderer Vorteile inzwischen sehr populär geworden. Besonders interessant wird der Einsatz von Git in der Verknüpfung mit den Features von SVN oder CVS in einem Projekt. Der folgende Beitrag veranschaulicht, wie bestehende Applikationen, die bereits ein Versionskontrollsystem nutzen, gleichzeitig mit Git verbunden werden können und welche Vorteile sich daraus ergeben.

von Walid El Sayed Aly

Git hat sich mittlerweile zum starken Konkurrenten sowohl aller zentralen (SVN, CVS und RCS) als auch lokaler Versionsverwaltungssysteme wie Mercurial und Bazaar entwickelt. Im Februar 2010 wurden im Rahmen einer Umfrage [1] die wichtigsten Versionierungssysteme miteinander verglichen. Dabei stellte sich heraus, dass Git eine beachtliche Bedeutung in diesem Umfeld innehat: Es erhielt die höchste Punktzahl in der Bewertung. Es gibt zahlreiche Einführungen in die Arbeit mit Git. Auch wurde bereits ausgiebig darüber referiert, welche Vorteile User genießen, die statt SVN oder CVS Git verwenden. Doch bisher ist noch niemand auf die Idee gekommen, ein zentrales oder lokales Versionsverwaltungssystem wie SVN „ohne *git-svn*“ oder CVS „ohne *git-cvsimport*“ mit Git in ein und demselben Projekt zu kombinieren, um die Vorteile aller Features zu bündeln und den größtmöglichen Nutzen daraus zu ziehen. Diese innovative Methode soll hier näher vorgestellt werden. Darüber hinaus wird gezeigt, wie man Git als Cloud-Lösung verwenden kann.

Git versus zentrale Versionsverwaltung

Im Gegensatz zu zentralen Versionsverwaltungssystemen verwendet ein verteiltes Versionsverwaltungssystem

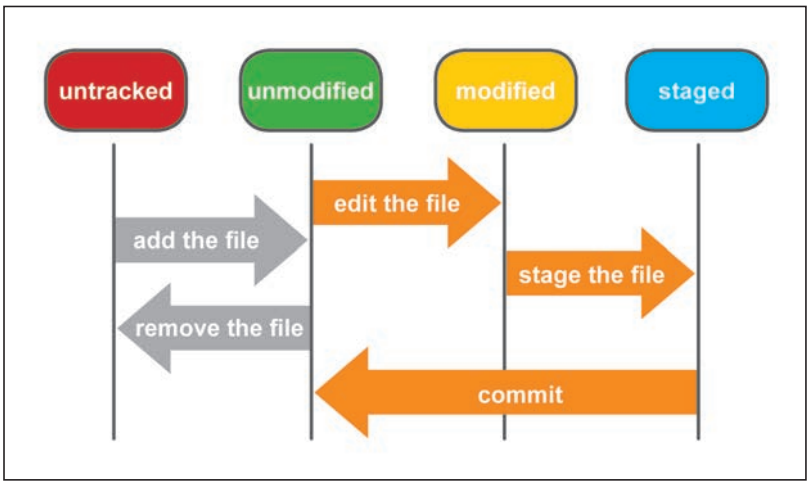


Abb. 1: Lifecycle-Status eines Files mit Git

kein zentrales Repository. Jedoch verleiht gerade dieses Merkmal einem verteilten Versionsverwaltungssystem mehr Stärke, da das System vom Netzwerk unabhängig ist. Git zählt zu den verteilten Versionsverwaltungssystemen. Es greift ausschließlich beim Abgleich mit Branches auf anderen Repositories auf den Netzwerkbefehl zurück. Die lokalen Git-Daten sind immer ein vollwertiges Repository für das Projekt, in dem es angewendet wird. Bei Git besitzt jeder User ein eigenes Repository, in das er seine Änderungen einpflegen kann. Git ist zudem ein extrem schnelles System. Das Austauschen von Änderungen kann enorm vereinfacht werden, zum Beispiel bei der Erstellung vollwertiger Clones auf GitHub und Google Code, da Git unabhängig vom Netz ist.

Trotz seiner zahlreichen Vorzüge soll Git hier aber nicht in den Vordergrund gestellt werden. Vielmehr soll das Augenmerk auf der Kombination von Git mit einem Versionierungssystem liegen sowie auf den Vorteilen, die Git als Cloud-Lösung bietet.

Lifecycle-Status einer Datei mit Git

Abbildung 1 zeigt die einzelnen Status eines Git-Files. Es gibt im Grunde genommen vier Zustände [2]: Eine Datei kann „untracked“ sein (Zustand 1) bzw. nicht versioniert, „unmodified“ (Zustand 2) – was bedeutet, dass die Datei schon versioniert ist, aber noch nicht verändert. Der dritte Zustand ist „modified“. Die Datei ist also schon versioniert, aber noch nicht „staged“. Befindet sich die Datei schließlich im Stage, also im vierten Zustand, ist sie noch nicht „committet“.

Git ist ein unkompliziertes, sehr effizientes System und als solches besonders geeignet für User, die es lieben, immer und überall zu programmieren. Sei es im Zug, beim Auslandsaufenthalt oder sogar am Strand – Git macht's möglich, auch wenn das heimische Firmennetzwerk nicht verfügbar ist. Projekte können auf diese Art viel schneller und effektiver bearbeitet und abgeschlossen werden. Dies spart also nicht nur Zeit, sondern auch Geld.

Bei Git gibt es die Möglichkeit, die einzelnen Änderungen zu historisieren und diese dann gemeinsam und gleichzeitig zu SVN oder CVS hochzuladen. Aber was machen Softwareentwickler, wenn sie z. B. auf Reisen oder am Wochenende zu Hause mit einem SVN- oder CVS-Projekt beschäftigt sind? Sie möchten dann natürlich auch Änderungen, die sie fernab vom Schreibtisch gemacht haben, historisieren, damit sie nicht verloren gehen. Die Antwort liegt auf der Hand: Git verwenden! Allerdings – wie geht das mit SVN oder CVS?

Git und SVN

Zunächst stellt sich die Frage, wie sich die Historisierung der Daten steuern lässt. Programmierer, die beim Entwickeln eine IDE wie Eclipse, IntelliJ oder auch Visual Studio verwenden, wissen, dass all diese IDEs eine Funktion zur Historisierung von Dateiänderungen bieten. Diese birgt aber auch einige Nachteile. So ist die Historisierung in den IDEs nicht für einen längeren Zeitraum möglich. Auch kann man Änderungen nicht wie in einem System effektiv anschauen und darstellen. Ein weiteres Handicap ist, dass die IDEs keine lokalen Branches oder Tags erstellen können. Infolgedessen kann man die Historisierung mithilfe einer IDE nicht zweckmäßig abschließen.

Welche Möglichkeiten gibt es nun, Git mit einem SVN-Projekt zu kombinieren? Im Gegensatz zu den zentralen Versionsverwaltungssystemen werden bei den verteilten Versionsverwaltungssystemen alle Änderungen lokal committet. Danach kann man diese Änderungen in der Cloud bzw. im Netz committen. Die naheliegendste Methode, Git mit SVN nutzen zu können, ist die Verwendung des Prozesses Git-SVN. Diese Funktion bietet Git von Haus aus, denn sie ist eine Standardmethode, wenn Git als Konsolensystem für Linux bzw. Mac OS installiert wird. Aber leider fehlt dieser Prozess in vielen Git-GUI-Systemen – ebenso wie viele Git-Plug-ins für die IDEs. Um ein vorhandenes SVN-Projekt mit Git kombinieren zu können, muss zuerst dieses Projekt mit Git geklont werden. Git-SVN bietet dafür eine Möglichkeit:

```
git svn clone --stdlayout http://svnProjectPath
```

Die Option *stdlayout* beinhaltet, dass Git auch Dateien aus Branches und Tags mit herunterlädt. SVN kennt das Konzept Git/branch und Git/tags nicht. Ist dies geschehen, kann man mit Git normal weiterarbeiten. Die meisten Git-Funktionen wie *add*, *commit*, *branch* usw. können dabei verwendet werden. Möchte man dann die Änderungen auch in Git committen, verwendet man *commit*. Zum Beispiel:

```
$git commit -am 'Fixed some Issues in index.html'
[master 6e40d02] Fixed some Issues in index.html
 1 files changed, 1 insertion(+), 1 deletion(-)
 rewrite index.html (100%)
```

Wenn man danach die Änderung auch in SVN committen möchte:

```
$ git svn dcommit
Committing to http://svnProjectPath
  M index.html
Committed r79
  M index.html
```

Vor- und Nachteile der Git-SVN-Verwendung

Git-SVN ist das perfekte Werkzeug, um mit Git zu beginnen, ohne radikale Veränderungen an der Organisation durchführen zu müssen. Bedauerlicherweise bringt dieses Tool aber viele Nachteile mit sich. Das Committen und Auschecken von Projekten beansprucht sehr viel Zeit, was besonders bei umfangreichen Projekten, die lange in einem SVN-Repository existierten, nicht sehr hilfreich ist. Darüber hinaus können beim Mergen Konflikte entstehen. Nur um einen Eindruck davon zu vermitteln: Es gibt mehr als 11 000 Einträge bei stackoverflow.com zum Suchbegriff „git-svn Problem“. Außerdem funktioniert das Werkzeug ausschließlich in SVN-Projekten. Das bedeutet: Wenn man ein CVS-Projekt hat und dieses parallel mit Git verwenden möchte, ist das leider nicht mit dem Git-SVN-Tool möglich.

Es bedarf zudem eines gewissen Know-hows, um dieses Tool zu nutzen. Die meiste Arbeit wird auf der Konsole erledigt, da Git-SVN nicht von vielen GUI-Tools und Plug-ins angeboten wird. Unter [3] findet man eine Liste grafischer Frontends und Tools.

Einen der größten Nachteile hat Scott Chacon in seinem Buch „Pro Git“ [4] gezeigt, nämlich, dass bei den Git-SVN-Commits bzw. bei der Verwendung von „git svn dcommit“ für jedes Commit ein Subversion Commit gemacht und dann das lokale Git umgeschrieben wird, um einen spezifischen Identifier zu inkludieren. Das bedeutet, dass sich alle SHA-1 Checksums der Commits eines Programmierers ändern. Daran erkennt man, dass das gleichzeitige Arbeiten mit Git-basierten Remote-Versionen in den Projekten eines Entwicklers und einem Subversion-Server nicht sinnvoll ist. Schaut man sich das letzte Commit an, sieht man, dass die neue Git-SVN-ID hinzugekommen ist. Die SHA Checksum, die z. B. ursprünglich mit 97031e5 beim Committen begann, startet nun mit 938b1a5. Will man sowohl zu einem Git-Server als auch zu einem Subversion-Server pushen, muss man zuerst zum Subversion-Server pushen (dcommit), weil diese Aktion die Commit-Daten ändert.

Git und CVS

Für all die schönen Projekte, bei denen immer noch CVS verwendet wird, gibt es auch die Möglichkeit, mit Git zu arbeiten. Mithilfe des „git-cvsimport“-Tools kann man CVS-Projekte importieren, um sie mit Git zu klonen. Ein CVS-Projekt wird folgendermaßen mit Git initialisiert:

```
$git cvsimport -d $CVSROOT -C dir_to_create -r cvs –k -A /path/to/authors/
                                                    file cvs_module_to_checkout
```

Das *A* aus dem *cvsimport* ist optional, aber es hilft dabei, die Historisierung mit Git harmonisieren zu können und mehr nach Git auszusehen.

Die erste Erstellung nimmt noch vergleichsweise viel Zeit in Anspruch. Nach der Initialisierung wird ein „master“ erstellt. Dabei gibt es bestimmte Konfigurationen, die hilfreich sind, wenn man mit Git und CVS arbeitet:

```
$ git config cvsimport.module cvs_module_to_checkout
$ git config cvsimport.r cvs
$ git config cvsimport.d $CVSROOT
```

Hierbei werden die Konfigurationen für die CVS-Module, den CVS-Import und den Root festgelegt, sodass man sie nicht jedes Mal neu eingeben muss. Um die Änderungen wieder in CVS spiegeln zu können, verwendet man das *git cvsexportcommit*.

Die Benutzung stellt den Programmierer jedoch vor einige Herausforderungen. Wie bei Git-SVN funktioniert dies nur in einem ausschließlich CVS-fähigen Projekt. GUI-Tools oder Plug-ins für Eclipse oder IntelliJ existieren leider nicht. Und zu guter Letzt kann es nach dem Klonen Probleme mit den Standard-Git-Funktionen geben.

Was passiert, wenn man innerhalb eines Subversion Directory Git initialisiert?

Tipps zur Verwendung von Git mit anderen Systemen

Nachdem nun Für und Wider der Kombination eines SVN- oder CVS-Systems mit Git und die Verwendung von *git-svn* sowie *svn-cvs-import* erörtert worden sind, wollen wir uns meiner konkreten Alternativlösung zuwenden. Es handelt sich dabei um eine eigentlich simple Idee, die aber eine höchst effiziente Wirkung hat: die Git-Initialisierung jedes beliebigen SVN- oder auch CVS-Projekts ohne Verwendung zusätzlicher Tools.

Hierfür ist „Metadata“ das Schlüsselwort. Bei Git gibt es das Repository *Metadata .git*. Alle Projekte bzw. Dateien, die mit einem Versionsverwaltungssystem verbunden sind, sehen nur diese Metadaten. Ein Subversion-Projekt beispielweise speichert seine Historie und auch die gesamte Verwaltung in einem Verzeichnis namens *.svn*. Subversion kontrolliert den gesamten Tree anhand dieses Verzeichnisses. Was passiert, wenn man innerhalb eines Subversion Directory Git initialisiert?

```
$git init
Initialized empty Git repository in /user/walid/svnProject/.git/
```

```
total 0
drwxr-xr-x   6 walid  staff  204 29 Jan 21:04 .
drwx------+  6 walid  staff  204 29 Jan 21:03 ..
drwxr-xr-x  10 walid  staff  340 29 Jan 21:03 .git
drwxr-xr-x   2 walid  staff   68 29 Jan 21:04 .svn
-rw-r--r--   1 walid  staff    0 29 Jan 21:04 pom.xml
drwxr-xr-x   2 walid  staff   68 29 Jan 21:04 src
```

Abb. 2: Strukturdarstellung eines GIT-SVN-Projekts

Nun hat man plötzlich ein Git-Repository innerhalb eines SVN-Directory erstellt. Dabei kann man alle Git-Funktionalitäten und die damit verbundenen Vorteile verwenden. Man kann Branches und Tags erstellen, Dateiänderungen committen und auch diese Veränderungen nachvollziehen, ohne SVN in Betracht ziehen zu müssen.

Allerdings gibt es einen Trick, den es zuvor anzuwenden gilt: Man muss die SVN-Metadata in Git ignorieren. Das geht einfach, indem man eine Datei mit dem Namen *.gitignore* im Hauptverzeichnis des Projekts erstellt und die *Metadata .svn* eingibt. Daraufhin wird Git die gesamten Metadaten des SVNs ignorieren. Man kann auch diese Bemerkung in *.git/info/exclude.git/info/exclude* erstellen. Hat man Git bereits unter Linux bzw. Mac OS installiert, erstellt es automatisch im aktuellen Arbeitsverzeichnis eine Datei mit dem Titel *.gitignore_global*. Dort kann man auch alle denkbaren ignorierbaren Dateien hinterlegen. **Abbildung 2** zeigt ein SVN-Projekt, das nachträglich mit Git initialisiert wurde.

> Git ist schnell, unabhängig, zeitgemäß und sehr sicher. Man kann es praktisch überall verwenden.

Nach diesem Prinzip kann man alle zentralen Verwaltungssystemprojekte zusammen mit Git steuern. Ein großer Vorteil liegt darin, dass Entwickler immer unabhängig vom Netz arbeiten können. Sie verwenden die Benefits von Git und sind somit frei von den Nachteilen von SVN-Git. Ob am Strand, im Zug oder im Central Park am Ententeich – man hat stets sein eigenes Repository dabei. Ein kleiner Nachteil ist, dass SVN diese Änderungen nicht mitbekommt und sie nicht zentral gespeichert werden. Aber sie gehen selbstverständlich nicht direkt verloren, denn man kann sie nachträglich zur SVN committen und zentral speichern.

Wir haben bis jetzt über Sourcecode, SVN, CVS und Git gesprochen – aber was ist mit Doc? Man kann Git auch als Cloud-Lösung verwenden, um Dateien wie Doc und Excel speichern und zentral verwalten zu können. Auf Google Docs oder Dropbox kann verzichtet werden, wenn man einen virtuellen Server schafft, um Git für die Verwaltung der einzelnen Dokumente benutzen zu können. Zum Beispiel so: Im Remote-Server muss ein leeres Repository erstellt werden mit

```
$git init --bare
```

Danach klont man dieses Repository:

```
$git clone me@vserver.com:/docs
```

So kann man letztendlich seine Dateien immer nachvollziehen und stets zentral und sicher von jedem System aus verwalten. Ein kleiner Nachteil, wenn man ihn überhaupt als solchen bezeichnen will, ist, dass man die Änderungen selbst committen und verwalten muss, weil es kein Tool dafür gibt, das sie im Hintergrund batcht. Es gibt aber auch dafür einen Trick: Anstatt immer mit *git add* und dann mit *git commit* zu arbeiten, gibt es die Möglichkeit, diese in einem Schritt ausführen zu können. Dafür muss man nur ein Alias erstellen:

```
git config --global alias.ac '!git add -A && git commit'
```

Danach kann man *git add* und auch *git commit* wie folgt ausführen:

```
git ac -m "message"
```

Fazit

Man kann Git praktisch überall verwenden. Es ist schnell, unabhängig, zeitgemäß und sehr sicher. Allerdings braucht es ein gewisses Know-how, um sich zu Beginn damit zurechtzufinden. Wenn man es einmal beherrscht, macht es große Freude, damit zu arbeiten. In vielen Bereichen verschafft es Erleichterung und Zeitersparnis. Auch wenn es nicht immer einen reibungslosen Ablauf gewährleistet, wiegen die vielen Vorteile, die Git bietet, kleinere Schwierigkeiten allemal auf.

Walid El Sayed Aly (walid.elsayedaly@gmail.com) ist IT-Consultant bei der 7P Solution & Consulting AG in Frankfurt am Main, Member of SEVEN PRINCIPLES. Seine Schwerpunkte liegen bei den agilen Softwarearchitekturen und Integrationstechnologien (Middleware). Nebenbei beschäftigt er sich u. a. mit der Entwicklung mobiler Technologien.

parmag.com **@wparmag**

Links & Literatur

[1] http://martinfowler.com/bliki/VcsSurvey.html

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

[3] https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools

[4] http://bit.ly/UeowOh, S. 207

[5] http://www.kernel.org/pub/software/scm/git/docs/git-cvsimport.html

Anzeige

Anzeige

Load Testing

Java Magazin Zeitreise

von Bernhard Löwenstein

JAX Speaker

Lasttests waren Titelthema der Ausgabe 8.2005. Bei einem Lasttest versucht man festzustellen, wie ein System unter Last reagiert. Solche Tests sollten für jedes nicht-triviale Projekt vorgesehen werden. Die Realität sieht leider oft anders aus, denn die beiden Themen Performance und Skalierbarkeit werden geradezu stiefmütterlich behandelt. So gibt es Unternehmen, in denen Lasttests nach dem Motto eingeplant werden: „Wenn am Ende des Projekts noch Zeit übrigbleibt". Komischerweise bleibt bei einer derartigen Projektplanung aber niemals Zeit über. Auch die Devise, dass vorab auf Lasttests verzichtet wird und beim Feststellen von produktiven Lastproblemen spontan schnellere Hardware gekauft wird, löst bei größeren Geschäftsanwendungen nur selten das Problem. Ebenfalls wenig sinnvoll ist das manuelle Testen eines Systems durch viele Benutzer gleichzeitig. Hierfür hat sich scherzhafterweise der Begriff „Friday Night Pizza Party" etabliert. Wer einen sinnvollen Lasttest durchführen will, der benötigt ein passendes Werkzeug. Mit den Kriterien zur Auswahl eines solchen Tools beschäftigte sich einer der Artikel, der zweite stellte Apache JMeter vor, das nach wie vor das herausragende Open-Source-Tool auf diesem Gebiet ist. Beide Beiträge lesen sich auch heute noch recht interessant – und der erste Artikel bringt die Sache bereits im Titel auf den Punkt: „A Fool with a (Loadtest-)Tool is still a Fool!".

Auch Eberhard Wolffs Beitrag, in dem er darüber philosophiert, ob Java in naher Zukunft das gleiche Schicksal wie COBOL erleiden wird, ist immer noch lesenswert. Als Voraussetzungen für das Ablösen einer alten Programmiersprache durch eine neue muss ihm nach ein großer Leidensdruck bei der täglichen Arbeit vorherrschen und die neue Sprache bei Kenntnis der alten einfach zu erlernen sein. Mit seinen technologischen Vorhersagen lag er absolut richtig: Die beiden damaligen Trends MDA und AOP ergänzten tatsächlich nur die klassischen Programmiersprachen und ersetzten diese nicht, und funktionalen Sprachen ist trotz ihrer kompakteren Syntax immer noch nur ein Nischendasein bestimmt. Interessant ist auf alle Fälle aber, dass mit Lambda Expressions nun ein funktionales Konzept in Java SE 8 Einzug halten wird. Auch sieben Jahre später lässt sich feststellen: Java gehört noch lange nicht zum alten Eisen.

Wie bereits erwähnt, lag AOP seinerzeit voll im Trend. Passend dazu fand sich ein Interview mit dem AspectJ-Projektleiter Adrian Colyer im Magazin. Die Grundidee hinter AOP ist, die typischen Querschnittsbelange, z. B. Logging, in zentral verwaltete und gepflegte Aspekte auszulagern und den eigentlichen Applikationscode frei von diesen Dingen zu halten. Er sollte dadurch übersichtlicher bleiben und einfacher zu warten sein. Einzug in Java Enterprise hielten die Aspekte schließlich mit den Interceptoren. Wer häufig mit Sourcen unterschiedlicher Entwickler zu tun hat, stellt allerdings schnell fest, dass dieses elegante Konzept längst nicht so oft eingesetzt wird, wie es möglich wäre. Der Mensch denkt nun einmal in Sequenzen, weshalb er sich beim Programmieren leichter tut, wenn er den Code in sequenzieller Form anschreibt, bzw. sich beim Verstehen leichter tut, wenn sich die Codeteile genau an den Stellen wiederfinden, wo sie dann während der Programmausführung tatsächlich ausgeführt werden.

Mit BPEL wurde dem Leser eine weitere Technologiehoffnung nähergebracht. Die Idee bestand darin, eine SOA auf Basis von Web Services aufzubauen und BPEL dabei als „Klebstoff" zu verwenden. Konkret sollten die einzelnen Geschäftsprozesse mittels der XML-basierten Sprache definiert werden. Größere Verbreitung fand BPEL allerdings nur in akademischen Kreisen. Generell schaffte es dieser Themenkreis bisher nicht, sich in der Softwareindustrie zu etablieren. Daran änderte auch die Ersetzung des Begriffs „Workflow" durch „Businessprozess" wenig.

Abschließend möchte ich noch auf den Artikel verweisen, der mit Knopflerfish eine konkrete Implementierung des OSGi-Frameworks präsentierte und eine gute Einführung in OSGi gab. Projektleiter Erik Wistrand äußerte dabei im begleitenden Interview die Hoffnung, dass OSGi eines Tages in die JVM integriert werden würde. Wie wir heute wissen, wird das nicht der Fall sein. Die modulare Entscheidung fiel zugunsten des Jigsaw-Projekts aus. Langfristig könnte dies eventuell das Ende für OSGi bedeuten. Konkretes wissen wir möglicherweise in weiteren sieben Jahren. Bis dahin wird es ja hoffentlich eine Java SE mit integriertem Modulsystem geben.

Bernhard Löwenstein (bernhard.loewenstein@java.at) ist als selbstständiger IT-Trainer und Consultant für javatraining.at und weitere Organisationen tätig. Als Gründer und ehrenamtlicher Obmann des Instituts zur Förderung des IT-Nachwuchses führt er außerdem altersgerechte Roboterworkshops für Kinder und Jugendliche durch, um diese für IT und Technik zu begeistern.

Warum Sie in Interviews nie die ganze Wahrheit erfahren

Fragen und Antworten

In der Softwareentwicklung stehen wir immer wieder vor der Frage: „Was will der Kunde (oder Anwender) denn eigentlich?“. Ohne eine ausreichende Auseinandersetzung mit dieser Fragestellung laufen wir Gefahr, ein Produkt zu entwickeln, das eigentlich niemand braucht und erst recht keiner kaufen will. Es müssen also die Anforderungen an das zu entwickelnde System erhoben werden. Hierfür steht uns eine Vielzahl von Ermittlungstechniken zur Verfügung. Allerdings sind nicht alle Techniken für alle Wissensarten (bewusstes, unterbewusstes, unbewusstes Wissen) gleichermaßen geeignet. Mit dem häufig gewählten Interview lässt sich beispielsweise sehr gut bewusstes Wissen ermitteln, wohingegen es für unbewusstes und unterbewusstes Wissen weniger geeignet ist.

von Chris Rupp und Dirk Schüpferling

In der letzten Ausgabe des Java Magazins haben wir die Notwendigkeit des Einsatzes mehrerer Ermittlungstechniken gezeigt. Bei der Einordnung des Wissens in verschiedene Wissensebenen haben wir gesehen, dass wir durch eine reine Befragung nur einen Teil der Anforderungen unserer Stakeholder ermitteln können. Alle Anforderungen bekommen wir also nur mit dem richtigen Mix aus verschiedenen Ermittlungstechniken. In diesem zweiten Teil möchten wir verschiedene Ermittlungstechniken mit deren Stärken und Schwächen vorstellen. Mit Befragungstechniken können vor allem Leistungsfaktoren (vgl. Kano-Modell aus dem ersten Teil) des Systems ermittelt werden, da diese dem Stakeholder bewusst sind und somit bei ihm erfragt werden können. Der bekannteste Vertreter dieser Kategorie ist das Interview.

Das Interview

Das Interview kann man salopp beschreiben als Technik, bei der dem Stakeholder Fragen gestellt werden und dieser sie beantwortet. Ziel ist es, Hintergründe und Zusammenhänge ans Licht zu bringen, aber nicht ein repräsentatives Ergebnis zu erzielen. Allerdings steckt bei einem guten Interview mehr dahinter. Grundsätzlich teilt sich ein Interview in drei aufeinander aufbauende Phasen auf – Vorbereitung, Durchführung und Nachbereitung.

Ein wichtiger Aspekt in der Vorbereitungsphase ist die inhaltliche Vorbereitung des Interviews selbst. Der Interviewer sichtet die Informationen, welche er im Vorfeld bereits hat, und überlegt sich, zu welchem Thema der Stakeholder überhaupt befragt werden soll. Der Interviewer erstellt sich einen groben Leitfaden mit einem Katalog aus Fragen, welche er vom Stakeholder beantwortet haben möchte. Der größere Teil der Vorbereitungsphase sollte dem Stakeholder gewidmet sein, da dieser letztlich befähigt werden muss, Informationen zu explizieren. Wichtig ist es, sich als Requirements Engineer möglichst gut auf ihn einzustellen. Dazu gehört auch, den Stakeholder auf den aktuellen Stand des Projekts zu bringen oder die beiderseitigen Annahmen bzgl. des Interviews abzuklären. Zu welchem Thema wird ein Beitrag erwartet? Welche Informationen soll der Stakeholder mitbringen? Wie läuft der Freigabeprozess des Interviewprotokolls und was passiert dann mit den Informationen? Und vieles mehr.

In der Phase der Interviewdurchführung gilt es vor allem den roten Faden des Interviews, also den aufgestellten Leitfaden, nicht aus den Augen zu verlieren. Um dies zu erreichen, ist es notwendig, das Interview mittels Fragetechniken zu steuern. So muss z. B. Verstandenes wiederholt werden, um sich abzusichern, dass man den Stakeholder auch richtig verstanden hat, nachgehakt werden, um weiter in die Tiefe oder Breite zu fragen, zusammengefasst werden, um Sinneinheiten abzuschließen und zu Protokoll zu bringen. Weiterhin muss Wichtiges von Unwichtigem getrennt oder ganze Teile für eine spätere Bearbeitung zurückgestellt werden. Das Ergebnis einer Interviewdurchführungsphase ist das im Interview entstandene Interviewprotokoll.

In der Nachbereitungsphase wird das Protokoll ggf. neu strukturiert und auf einen sauberen, lesbaren Stand gebracht. Anschließend geht es, mit der Bitte nach Feedback bis zu einem bestimmten Zeitpunkt abzugeben, an den Stakeholder. Erst nach seiner Freigabe können die Informationen aus dem Interview weiterverwendet werden, um daraus z. B. Anforderungen abzuleiten.

Kreativitätstechniken

Kreativitätstechniken kommen vor allem dann zum Einsatz, wenn etwas Neues, Innovatives ermittelt werden soll. Entsprechend eignen sie sich vor allem für die Begeisterungsfaktoren aus dem Kano-Modell (vgl. erster Artikelteil).

Brainstorming und Brainstorming paradox

Brainstorming ist eine weithin bekannte Kreativitätstechnik und dient vor allem dazu, Ideen mehrerer Teilnehmer zu sammeln. Dabei werfen die Teilnehmer ihre

Ideen zu einem vorgestellten Thema in den Raum und ein Moderator schreibt diese für alle sichtbar auf. Im Vordergrund steht dabei die strikte Trennung von der Ideenfindung und der Diskussion der Ideen. Denn während die Teilnehmer ihre Ideen einwerfen, dürfen diese nicht diskutiert werden. Dies ist erst möglich, nachdem alle Ideen gesammelt wurden, beziehungsweise eine vorgegebene Zeit abgelaufen ist.

Das Brainstorming paradox ist bezüglich der Vorgehensweise gleich dem normalen Brainstorming. Allerdings wird beim Brainstorming paradox das Thema umgekehrt, also das gegenteilige Thema abgefragt. Zum Beispiel könnte im Rahmen der Neuentwicklung der Homepage eines Unternehmens das Thema lauten: „Wie muss die Homepage gestaltet sein, damit möglichst wenige Menschen diese besuchen?“. Ziel dieser Art des Brainstormings ist es, die Ursachen von Problemen zu ergründen und herauszufinden, welche Ergebnisse in Zukunft vermieden werden sollen.

Legende:

-	nicht empfohlen
0	kein Einfluss=> ist anwendbar
+	empfohlen
++	sehr empfohlen

	Brainstorming	Brainstorming Paradox	Methode 6-3-5	Wechsel der Perspektive	Walt Disney Methode	Analogietechnik (Bionik/Bisoziation)	Osborn Checkliste	Feldbeobachtung	Apprenticing	Contextual Inquiry	Fragebogen	Interview	Selbstaufschreibung	On-Site-Customer	Systemarchäologie	Reuse - Wiederverwendung	Perspektivenbasiertes Lesen
Menschliche Einflussfaktoren																	
Geringe Motivation der Stakeholder (aktiv mitzuwirken)	-	-	-	-	-	-	-	+	-	0	0	+	-	-	++	++	-
Schlechte kommunikative Fähigkeiten	-	-	-	-	-	-	-	++	++	0	-	+	-	-	++	++	++
Geringes Abstraktionsvermögen	-	-	-	-	-	0	-	++	++	+	0	+	-	-	++	++	++
Viele veschiedene Meinungen	+	+	++	+	0	+	+	++	++	++	+	0	0	-	0	0	+
Machtgefälle zwischen beteiligten Personen	-	-	+	-	0	0	0	0	0	0	0	0	0	0	0	0	++
Problematische Gruppendynamik	-	-	+	+	+	0	0	0	0	0	0	0	0	0	0	0	+
Organisatorische Einflussfaktoren																	
Entwicklung für den komplexen Markt	++	+	+	+	++	+	+	-	-	-	++	0	0	0	+	0	0
Fixiertes, knappes Projektbudget	++	++	++	+	+	-	+	+	-	-	-	+	+	+	-	++	+
Hohe örtliche Verteilung der Stakeholder	-	-	0	-	0	-	0	0	0	0	++	0	+	-	0	0	+
Schlechte zeitliche Verfügbarkeit der Stakeholder	+	+	+	-	-	-	+	+	-	+	+	++	-	-	++	++	-
Hohe Anzahl der Stakeholder	+	+	-	+	-	0	0	0	-	0	++	0	-	-	0	0	-
Fachlich/inhaltliche Einflussfaktoren																	
Hohe Kritikalität des Sachverhalts	0	0	0	+	0	0	0	++	-	+	+	+	+	++	++	+	-
Großer Systemumfang	0	0	0	0	0	-	-	+	-	0	-	+	-	+	+	+	0
Keine Erfahrung im Fachgebiet	0	0	0	0	0	0	0	-	+	+	-	-	0	+	+	+	0
Grobe Anforderungen gesucht	++	++	++	+	+	+	++	+	0	0	+	++	+	+	-	0	0
Detaillierte Anforderungen gesucht	+	+	+	+	-	0	0	+	++	++	-	+	+	+	++	+	0
Nicht funktionale Anforderungen gesucht	0	0	0	0	0	+	+	0	+	+	-	+	0	++	+	+	+
hohe Komplexität des Sachverhalts	0	0	0	0	0	+	-	-	-	0	-	+	-	+	+	+	0
Part of the syllabi for CPRE Foundation Level (*), Advanced Level Elicitation & Consolidation (**)	*,**	*,**	**	**		*,**	**	*,**	*,**	**	*,**	*,**			*,**	*,**	*,**

Abb. 1: Entscheidungsmatrix für Ermittlungstechniken

Erwarten Sie aber nicht, dass mit den Brainstormingmethoden sämtliche Anforderungen gefunden werden. Vielmehr dient das Brainstorming dazu, Optimierungspotenziale zu finden und einen ersten Überblick über ein Thema zu erlangen. Mit dem Brainstorming paradox werden vor allem Gefahren und Risiken herausgearbeitet. Durch die Kombination beider Techniken (also einmal Brainstorming normal und dann paradox) werden beide Sichtweisen genutzt, um das Thema möglichst umfassend zu beleuchten.

Methode 6-3-5

Die Methode 6-3-5 unterscheidet sich vom Brainstorming vor allem in der Tatsache, dass jeder Teilnehmer für sich seine Ideen aufschreibt und zunächst nicht vor den anderen nennt.

Diese Ermittlungstechnik ist ursprünglich für sechs Teilnehmer gedacht. Jeder schreibt zu einem vorher festgelegten Thema drei Ideen auf ein Blatt Papier. Danach reicht jeder Teilnehmer sein Blatt an den nächsten Kollegen weiter. Dieser fügt seinerseits wiederum Ideen hinzu. So werden reihum immer mehr Ideen auf dem Blatt gesammelt. Dabei kann es durchaus vorkommen, dass Ideen mehrmals auftauchen. Der große Vorteil gegenüber dem Brainstorming ist, dass die Ideenfindung sehr gut auf die verschiedenen Stakeholder verteilt wird. Das heißt, jeder schreibt für sich Ideen auf, während man beim Brainstorming die Ideen vor allen anderen Teilnehmern vortragen muss, was mitunter zu Hemmungen führen kann. Häufig hat man in einer Gruppe Leute, die viel reden, und ein paar zurückhaltende, ruhige Leute. In einem Brainstorming kommen die Ruhigen seltener zu Wort, wodurch das Ergebnis negativ beeinflusst wird. Das Problem tritt bei der 6-3-5-Methode nicht auf. Interessant ist diese Ermittlungstechnik auch, wenn es schwierig ist, die Teilnehmer zu einer Zeit an einen Ort zu bringen. Denn die 6-3-5-Methode ist auch per Mail durchführbar.

Beobachtungstechniken

Wenn es um das Ermitteln der Basisfaktoren des Kano-Modells eines Systems geht, ist man mit Beobachtungstechniken am besten aufgestellt. Diese sind zum Beispiel die Feldbeobachtung, das Contextual Inquiry und das Apprenticing.

Die Feldbeobachtung

Bei der Feldbeobachtung ist der Anforderungsermittler in der Rolle des Beobachters und der Stakeholder in der Rolle des Beobachteten. Der Anforderungsermittler beobachtet den Stakeholder bei der Arbeit z. B. beim Umgang mit einem Vorgängersystem. Dabei notiert er sich sorgsam alles, womit er beobachten kann: Was macht der Stakeholder, in welcher Reihenfolge macht er das etc. Aus diesen Beobachtungen kann der Wissensermittler entsprechend Basisfaktoren, bzw. Hinweise auf Basisfaktoren ableiten. Aus der Tatsache, dass sich der Stakeholder im Auto hinsetzt, kann der Analytiker den entsprechenden Basisfaktor „Sitzmöglichkeit vorhanden“ ableiten.

Ein Problem bei dieser Vorgehensweise ist, dass der Analytiker die Basisfaktoren hauptsächlich durch Rück-

schlüsse aus den Beobachtungen herleiten muss. Dabei können durchaus falsche Schlüsse gezogen werden. Daher muss das Beobachtete genauer hinterfragt werden. Demnach muss einer Feldbeobachtung immer eine weitere Technik (z. B. ein Interview) folgen. Es gibt allerdings eine Technik, welche die Feldbeobachtung und das Interview vereint, und zwar das Contextual Inquiry.

Contextual Inquiry

Hierbei handelt es sich um eine Mischung aus Beobachtungs- und Befragungstechnik. Hier wird nicht wie bei der Feldbeobachtung dem Stakeholder ausschließlich zugesehen und das Beobachtete dokumentiert, sondern zusätzlich das Beobachtete besprochen. Somit wird die Gefahr falscher Rückschlüsse gebannt. Das bedeutet also, dass der Stakeholder seine Arbeitsabläufe ausführt und der Analytiker beobachtet und Annahmen formuliert. Danach hat der Analytiker die Möglichkeit, dem Stakeholder diese Annahmen zu explizieren. Er kann Verständnisfragen stellen, aber auch das Beobachtete hinterfragen. Das gibt ihm die Möglichkeit herauszufinden, ob die gesehenen Abläufe notwendig sind, oder nur aufgrund der bisher existierenden Lösungen sich so wie gesehen darbieten. Wichtig beim Contextual Inquiry ist, dass es nicht zu einer reinen Beobachtung oder zu einem reinen Interview wird. Die Mischung ist wichtig. Der Stakeholder soll die Abläufe vorführen und nicht erzählen.

Das Apprenticing/In die Lehre gehen

Beim Apprenticing wird im Gegensatz zur Feldbeobachtung das Machtverhältnis zwischen Beobachter und Beobachteten umgekehrt. Das bedeutet, der Anforderungsermittler übernimmt die Aufgabe des Stakeholders und der Stakeholder sieht zu und hilft im Bedarfsfall. Dadurch erhöht sich die Aktivität des Ermittlers, er hört und sieht nicht nur den Sachverhalt, sondern er erlebt ihn selber. Dies ermöglicht eine stärkere Vertiefung der Materie. Vor allem, da nicht alle Informationen zu dem Sachverhalt beobachtbar, aber erlebbar sind. Nehmen wir als Beispiel den Schaltvorgang im Auto. Das richtige „kommen lassen“ der Kupplung lernt man nur durch Selber-Ausprobieren.

Dokumentenzentrierte Techniken

Dokumentenzentrierte Techniken basieren darauf, dass bereits vorhandene Dokumente oder Systeme als Anforderungsquelle herangezogen werden. Das könnte ein Anforderungsdokument aus einem Vorgängerprojekt sein oder ein existierendes System, das abgelöst werden soll. Folglich wird man mit Techniken aus dieser Kategorie keine „neuen“ Anforderungen ermitteln können. Lediglich die bereits vorhandenen Informationen kommen hier zutage. Wenn wir das auf das Kano-Modell übertragen, werden wir sehen, dass wir mit diesen Techniken nur Basis- und Leistungsfaktoren ermitteln werden. Das würde den Ebenen bewusstes und unterbewusstes Wissen entsprechen.

Die Systemarchäologie

In diesem Artikel werden wir nur einen Vertreter dieser Techniken vorstellen, die Systemarchäologie. Dabei werden zum Ermitteln der Anforderungen bestehende Artefakte eines vorhandenen Systems herangezogen. In unserem Beispiel bedeutet das, dass ein Benutzerhandbuch des Autos oder das Auto selber als Quelle für die Anforderungen dient. Man würde sich das Auto ansehen, damit herumspielen und auch fahren. Aus den daraus gewonnenen Informationen können Anforderungen abgeleitet werden. Wichtig dabei ist, dass man hier erst einmal nur mögliche Anforderungen identifiziert hat. Denn man sieht ja nur was im Altsystem vorhanden ist, aber nicht ob dies auch benötigt, beziehungsweise gefordert wird. Es könnte also leicht passieren, dass unnütze Dinge als Ergebnis anfallen. Folglich sollte diese Technik niemals als einzige Ermittlungstechnik verwendet werden. Vielmehr sollten die gewonnenen Ergebnisse mit anderen Techniken, zum Beispiel einem Interview, überprüft werden.

Eine Entscheidungshilfe

Zum Abschluss möchten wir Ihnen noch eine Entscheidungshilfe mit auf den Weg geben. Diese haben wir in Form einer Matrix zusammengefasst (**Abb. 1**). In dieser Matrix ist eine Reihe von Einflussfaktoren aufgelistet und diese werden verschiedenen Ermittlungstechniken gegenüber gestellt. Diese Symbole zeigen, welche Technik bei welchen Einflussfaktoren gut geeignet (++) oder nicht empfohlen (-) wird. Nicht in der Matrix enthalten ist die Unterscheidung zwischen den verschiedenen Wissensebenen. Hier gilt nach wie vor die einfache Regel: Bewusstes Wissen mit Fragetechniken, unterbewusstes Wissen mit beobachtungs- oder dokumentenzentrierten Techniken und unbewusstes Wissen mit Kreativitätstechniken ermitteln. Nicht alle in dieser Matrix gezeigten Ermittlungstechniken haben wir in diesem Artikel erläutert. Eine Beschreibung zu jeder der Techniken finden Sie in [1], [2]. Die Matrix ist auch als Download auf unserer Webseite erhältlich [3].

Chris Rupp (www.sophist.de/chris.rupp) liefert durch Publikationen immer wieder wichtige Impulse für die Bereiche Requirements Engineering und Objektorientierung. Erfindungen von ihr legten die Basis des modernen Requirements Engineering. Chris ist Geschäftsführerin der SOPHIST GmbH.

Dirk Schüpferling (www.sophist.de/dirk.schuepferling) ist bereits seit 2001 als Berater und Trainer bei den SOPHISTen. Er forscht auf dem Gebiet der Dokumentationstechniken und löst hier Problemstellungen aus der Praxis, indem er Neuerungen und Anpassungen an etablierten Tools und Dokumentationsarten vornimmt.

Links & Literatur

[1] Chris Rupp & die SOPHISTen: „Requirements Engineering und Management“, Hanser Verlag, 2009

[2] Rupp, Chris; Pohl, Klaus: „Basiswissen Requirements Engineering“, dpunkt.Verlag, 2011

[3] http://www.sophist.de/infopool/downloads/offener-downloadbereich/

Fünf MDM-Anbieter im Vergleich

Mobile Geräte verwalten

Immer mehr Firmen entscheiden sich dafür, mobile Geräte in den Arbeitsalltag mit einzubinden. Mit einem Mobile-Device-Management-System lassen sich diese Geräte sicher und zuverlässig verwalten. Auf dem Markt bieten mittlerweile zahlreiche Hersteller derartige MDM-Lösungen an. Drei der großen Mitspieler sind AirWatch, Symantec und MobileIron aus den USA. Wir nehmen die Lösungen aus Übersee genauer unter die Lupe und vergleichen sie mit den europäischen Produkten Bunya und datomo. Dabei werfen wir auch einen direkten Blick in das jeweilige MDM-System.

von Lukas Holzamer, Mario Tonelli und Christine König

Mobile Device Management (MDM) liegt im Trend. Viele Firmen wollen mittlerweile von Smartphones und Tablets profitieren und diese gezielt im Unternehmensalltag einsetzen. Mit MDM-Systemen werden mobile Geräte von einem Administrator zentral verwaltet. Für Unternehmen wie United Airlines bringt dies viele Vorteile mit sich. So spart der Fluggigant dank elektronischer Flughandbücher jährlich rund 16 Millionen Seiten Papier ein [1]. MDM-Systeme sind als SaaS (Software as a Service) oder als On-Premise-(In-House-)Variante verfügbar. Bei On-Premise wird das Framework vom firmeneigenen Applikationsserver bereitgestellt. Anfallende Daten verbleiben auf dem Firmenserver und sind dadurch sicher vor Dritten geschützt. SaaS dagegen stellt dem Unternehmen lediglich die Nutzung des Services bereit. Die Applikation selbst sowie anfallende Daten werden auf Servern des MDM-Anbieters bereitgestellt und gespeichert. Die Verwaltung erfolgt remote über gesonderte Zugänge zum Anbieter. Sowohl bei SaaS als auch bei On-Premise können bestimmten Mitarbeitern über das MDM-System ausgewählte Applikationen zur Verfügung gestellt werden. So lassen sich ausgewählten Arbeitsgruppen oder -bereichen die benötigten Apps zuteilen.

Bereitstellen kann der Arbeitgeber seinen Mitarbeitern mobile Geräte entweder selbst oder sie bringen ihre eigenen mit (Bring your own Device, kurz: BYOD).

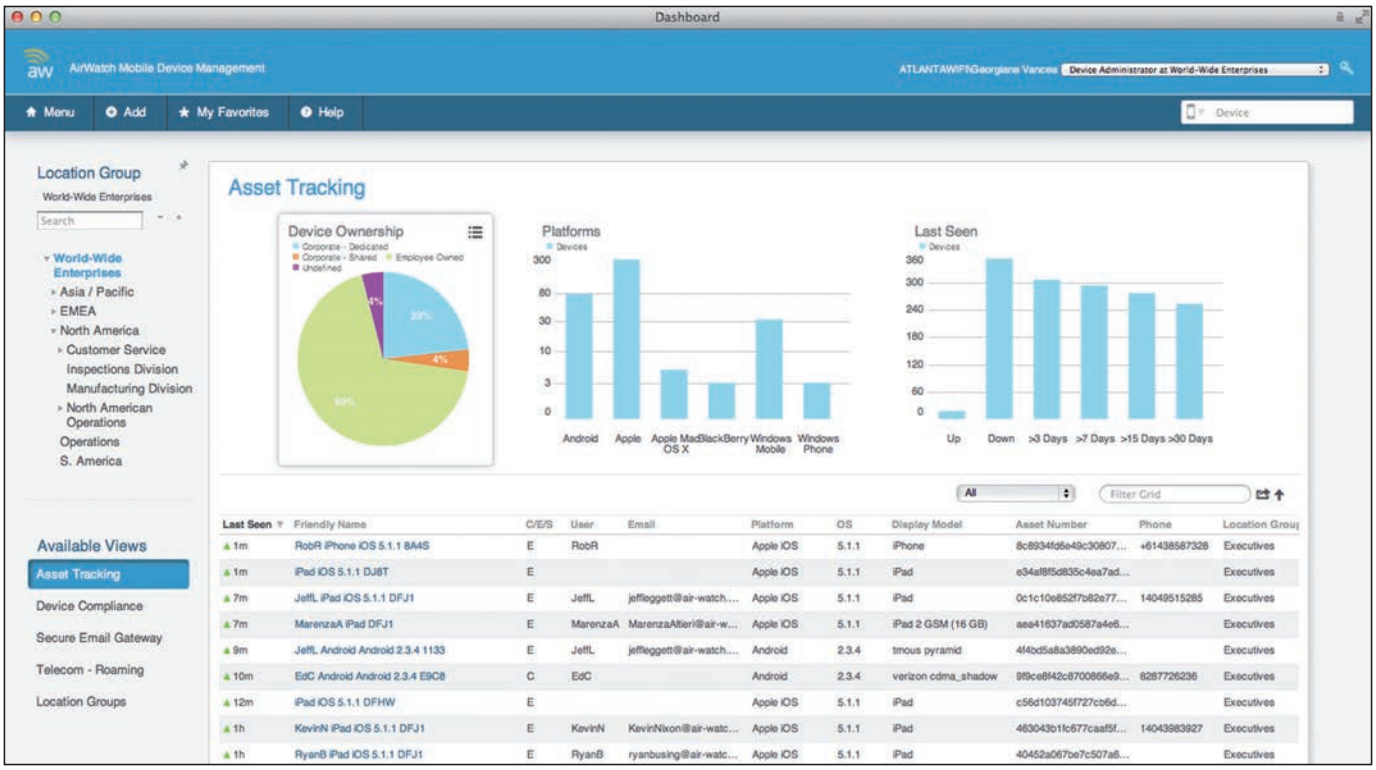


Abb. 1: Web-UI von AirWatch

Dies zeigt auf, dass moderne MDM-Systeme nicht nur im Trend liegen. Vielmehr sind sie notwendig geworden, um den Administrator aktiv bei der Sicherung von Kompatibilität, Konnektivität und Datensicherheit zu unterstützen. Denn sämtliche mobile Geräte verschiedener Hersteller werden zentral und einfach in einer Verwaltungsoberfläche administriert.

Die verschiedenen MDM-Anbieter kümmern sich ebenfalls um die Bereitstellung mobiler Apps. Einige Hersteller gestatten den uneingeschränkten Download von Apps und führen eine Restriktion nur über eine „Black List“ durch. Andere bieten die Apps im eigenen In-House App Store an oder erlauben die Nutzung einer zentralen Plattform im Internet. Wieder andere verwalten die Apps zentral und geben diese nur nach Berechtigung frei. Einige haben sich für eine hybride Lösung entschlossen, in der die Bereitstellung von Apps auf verschiedene Arten zulässig ist. Doch welche dieser Varianten erhält man bei welchem Hersteller? Für welche Geräte ist das jeweilige MDM-System überhaupt geeignet? Und kann BYOD mit dem jeweiligen MDM realisiert werden? Fragen wie diesen sind wir in unserem Vergleich verschiedenster MDM-Anbieter auf den Grund gegangen.

AirWatch

AirWatch ist ein auf Enterprise Mobility Management (EMM) spezialisierter Anbieter von MDM, MAM (Mobile Application Management), MCM (Mobile Content Management), E-Mail- und BYOD-Management [1].

Die Administrationsweboberfläche von AirWatch lässt sich beliebig aufteilen. Einzelnen Ebenen können dabei verschiedene Rechte zugeteilt werden. So können auch weitere Mitarbeiter – neben dem Hauptverwalter – eine administrative Rolle einnehmen. AirWatch verfügt außerdem über einen eigenen Browser (Secure Browser) mit Proxy-Funktion. Websites können über Black und White Lists verwaltet werden.

Im Bereich Mobile Content Management (MCM) erfolgt die Verwaltung von Daten in einem Container (Content Locker) mit Multifaktor- und Directory-Services-Authentifizierung. Es lassen sich Dokumente verschlüsselt auf bestimmte Geräte pushen, bei Bedarf nur für einen festgelegten Zeitraum. Auf diese Weise kann festgelegt werden, dass Mitarbeiter auf gewisse Dokumente nur innerhalb eines bestimmten Zeitraums zugreifen können.

Das E-Mail-Management erfolgt entweder durch den nativen E-Mail-Client oder durch Integration mit einem Drittanbieter in Containern. E-Mails können also nur abgerufen werden, wenn das Gerät über AirWatch verwaltet wird. Befinden sich unerwünschte Apps auf dem Gerät, kann der Zugriff auf die E-Mails automatisch gesperrt werden. Regelkonformität nimmt bei AirWatch folglich eine bedeutende Stellung ein. Ist ein Gerät nicht regelkonform, kann dem Benutzer ein Zeitlimit zur Behebung gesetzt werden. Bereinigt er bis dahin sein Gerät nicht nach den entsprechenden Richtlinien, kann der Administrator den Gerätezugang sperren oder sogar löschen [2].

Eine Besonderheit von AirWatch MDM ist das Geofencing. Mitarbeiter können damit nur innerhalb eines festgelegten Gebiets auf mobile Apps und Daten zugreifen. Außerhalb dieses Bereichs ist der Zugriff ge-

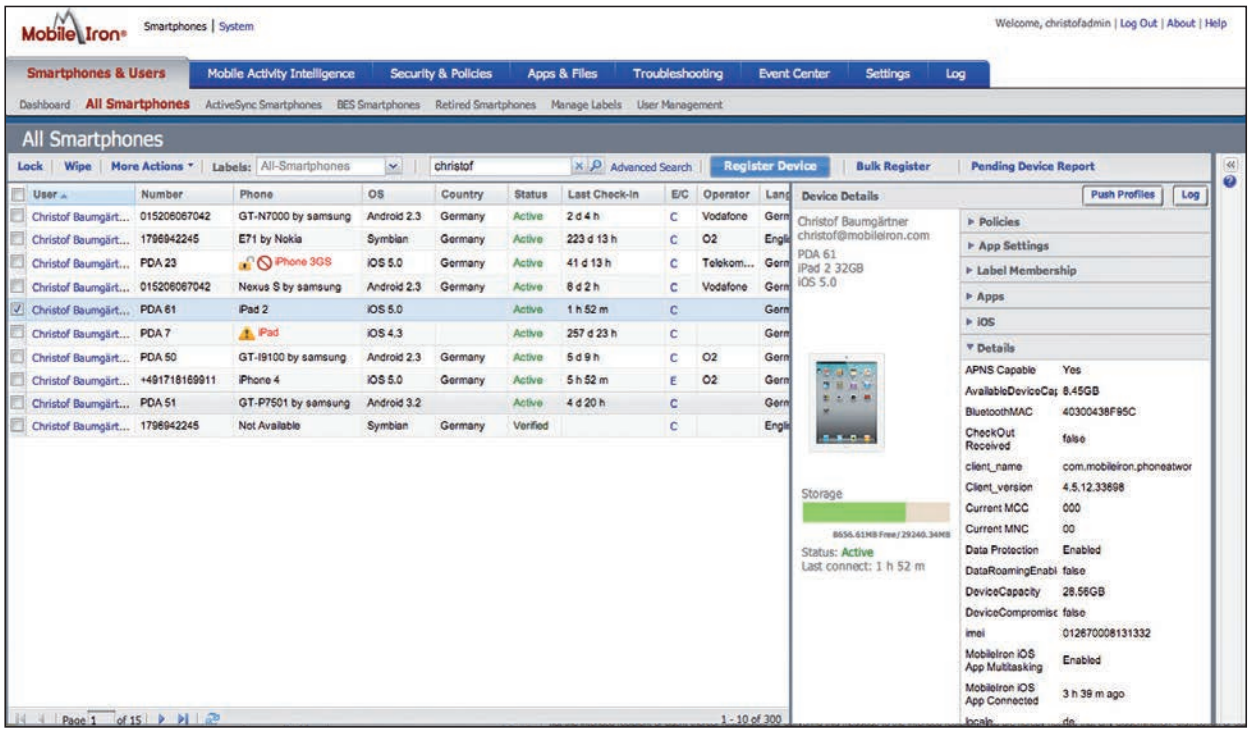


Abb. 2: Web-UI von MobileIron

sperrt, um Unternehmensdaten zu sichern [2]. Darüber hinaus zeichnet sich AirWatch durch eine hochgradige Skalierbarkeit aus, sodass 10 bis 100 000 Geräte gleichzeitig verwaltet werden können. Außerdem ist eine Notfallwiederherstellung möglich. Über ein Remote-Rechenzentrum kann die Software aktiviert werden [3].

Im Bereich BYOD erfolgt die Unterscheidung von privaten und unternehmensgebundenen Geräten während der Geräteregistrierung, die eine Registrierung als privates, geschäftliches oder geteiltes Gerät ermöglicht. Durch Trennung privater und geschäftlicher Daten können Geschäftsdaten sicher gelöscht werden, während persönliche Benutzerdaten bestehen bleiben [2].

MobileIron

MobileIron ist ein amerikanisches Unternehmen zur Bereitstellung und Verwaltung mobiler Geräte im Unternehmen. Es liefert eine vollständige Plattform für mobile IT im Unternehmen, die Funktionalitäten für MDM, MAM, MCM, MDLP (Mobile Data Loss Prevention) und MobileSecurity enthält [4].

MobileIron bietet Funktionen, die die vollständige Trennung von privaten und geschäftlichen Daten gewährleisten. Docs@Work gewährleistet durchgängige Sicherheit für ActiveSync-E-Mail-Anhänge und SharePoint-Inhalte. Es sorgt für einen sicheren Transport, sichere lokale Speicherung und eine sichere Darstellung und Datenlöschung. E-Mail-Anhänge können auf dem Transportweg zum Mobilgerät verschlüsselt werden. Verschlüsselte Inhalte und Anwendungen können dann nur mit Docs@Work entschlüsselt und angezeigt werden. SharePoint-Inhalte lassen sich direkt über Docs@Work darstellen, um sie auf dem mobilen Endgerät sicher zu speichern und ansehen zu können. SharePoint und ActiveSync-E-Mail integriert sich nahtlos in die MobileIron-Richtlinien und Active-Directory-Gruppen. Unter Nutzung der nativen E-Mail-Apps kann MobileIron verhindern, dass Attachments per „Öffnen in“ in unerwünschten Apps (z. B. Dropbox) geöffnet werden [5].

AppConnect ist eine Technologie für Android und iOS, die Geschäftsanwendungen in virtuelle Container schleust. Diese schafft eine sichere Umgebung für mobile Business-Apps und verhindert Datenverluste, ohne dass Benutzer auf die gewohnte App-Umgebung verzichten müssen. Sie wird von allen Android-Geräten sowie -Betriebssystemversionen und iOS unterstützt. Über AppConnect für Android wird jede geschäftliche App in einen sicheren Container gepackt, der über die Konsole verwaltet wird. Der Container verschlüsselt Daten, trennt sie und bietet eine Single-Sign-On-Funktion. So wird sichergestellt, dass nur vertrauenswürdige, authentifizierte Apps Daten untereinander austauschen können. Private Apps sind von dieser Kommunikation ausgeschlossen [6].

Verfügbar sind die Apps im In-House App Store. Hier werden sowohl eigens entwickelte Apps, die nicht im nativen App Store erhältlich sind, als auch normale App-Store-Apps zum Download angeboten. Sollten Apps gegen die Richtlinien verstoßen, wird das Gerät im System durch ein kleines Symbol und rote Schrift gekennzeichnet. Zudem wird der User benachrichtigt, dass er sein Gerät wieder den Richtlinien anzupassen hat oder es werden Compliance Actions ausgeführt. Dabei lässt sich beispielsweise auch der Zugang zu E-Mails blockieren [4].

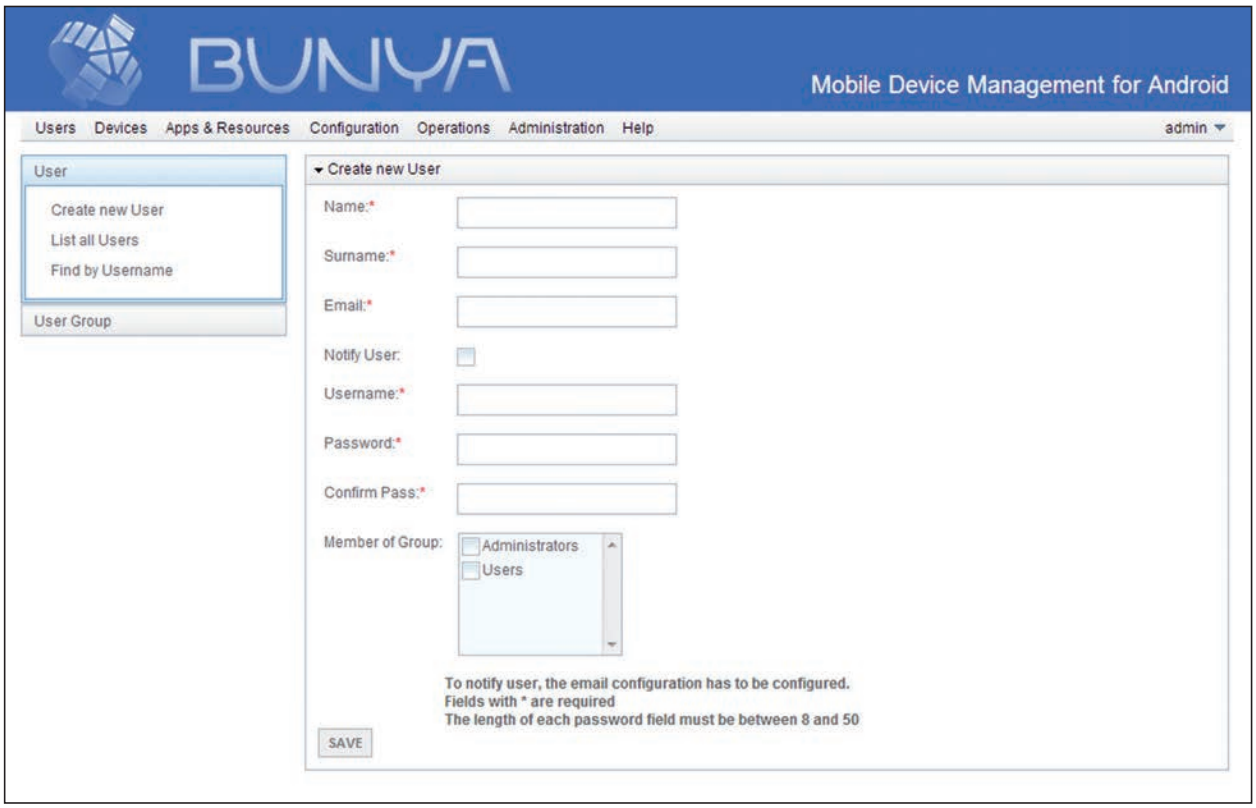


Abb. 3: Web-UI von Bunya

Symantec Mobile Management (SMM)

Symantec Corporation ist ein amerikanisches Unternehmen für Sicherheitssoftware mit rund 20 000 Mitarbeitern. Zu seinen Produkten gehört u. a. auch Symantec Mobile Management (SMM) – eine skalierbare MDM-Plattform mit Mehrinstanzenfähigkeit, rollenbasiertem Zugriff und automatisiertem Workflow. Im Symantec Mobile Management können mobile Geräte zugelassen, abgesichert und über eine einzige Konsole verwaltet werden [7].

Bei SMM wird alternativ zum normalen Webinterface für den Admin ein Helpdeskzugang geboten, der eine direkte Einwahl auf dem Gerät des Mitarbeiters ermöglicht. Einmal gestartet läuft die Clientanwendung stets im Hintergrund auf den Endgeräten. Den ersten Zugang zum System erhält man, nachdem ein Link an das Gerät gesendet wurde und eine Nutzervalidierung über Benutzername und Kennwort erfolgt ist. Dieser Link kann per E-Mail an das Gerät geschickt oder auf einem Webportal für das Endgerät bereitgestellt werden [8].

Im Vordergrund von SMM steht die ganzheitliche Sicherheit, d. h. der Schutz von Informationen, Infrastruktur und Identität. Das Angebot reicht hier von der Authentifizierung bis hin zur Verschlüsselung sowie der Vermeidung von Datenverlust. Zudem sind erweiterte Sicherheitseinstellungen durchführbar. So lassen sich mobile Geräte unabhängig vom Besitzer schützen. Beispiele hierfür sind Kennwörter, Fernsperrung und -löschung sowie die Einschränkung von Anwendungen, Ressourcen und Content. Darüber hinaus schützt Symantec Mobile Security Android- und Windows-basierte mobile Geräte vor Spams, Malware, Viren etc. [7].

Sämtliche Inhalte wie Apps, Bilder, Videos etc. werden bei SMM zudem sicher in Containern gespeichert und über einen Firmen-App-Store bereitgestellt. Diese Container werden schließlich an den Nutzer geliefert. Zusätzlich erhalten Nutzer eine oder auch mehrere Berechtigungen, so genannte Policies, und können sich entsprechende Apps oder Inhalte vom Server holen bzw. bekommen diese zugewiesen. Die Nutzung des Apple App Store ist geräteabhängig, pauschal oder auch nutzerabhängig regulierbar [8].

Auch bei Symantec ist BYOD möglich. Indem Apps per App-Wrapping mit einer Sicherheitsrichtlinie verbunden werden, können Geschäftsdaten sicher gelöscht werden, während persönliche Benutzerdaten bestehen bleiben [9].

datomo MDM

Die Pretioso GmbH ist ein Anbieter und Distributor für mobile Applikationen. Sie unterstützt vor allem mittelständische Unternehmen bei der Abbildung von mobilen Prozessen. Neben Produkten wie datomo Mobile Forms oder datomo Mobile Password, findet sich auch datomo Mobile Device Management im Portfolio des deutschen Unternehmens – dessen Server sich übrigens ebenfalls in Deutschland befinden [10].

datomo MDM unterstützt alle gängigen Smartphones und verfügt über eine einheitliche Verwaltungsoberfläche für sämtliche mobile Plattformen. Großer Vorteil ist, dass es bereits in kleinen Stückzahlen eingeführt werden

Anzeige

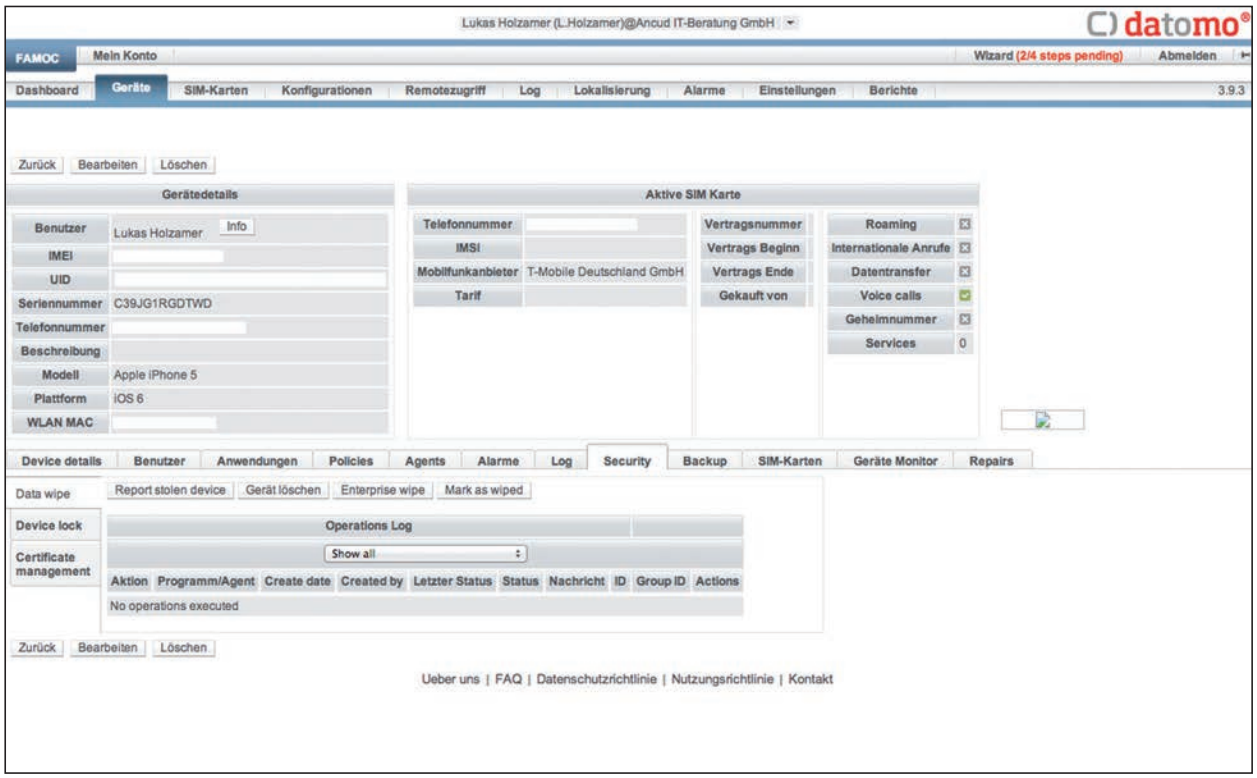


Abb. 4: UI von datomo

kann. So lässt sich das MDM-System erst in einem kleineren Teilbereich des Unternehmens einsetzen, bevor es schließlich flächendeckend verwendet wird. Zudem ist der Private Use Of Company Equipment (PUOCE) möglich. Der Mitarbeiter kann also – unter zuvor festgelegten Rahmenbedingungen – das vom Unternehmen bereitgestellte mobile Gerät auch privat nutzen [11].

Alleinstellungsmerkmal von datomo MDM ist die Mandantenfähigkeit. Im MDM-System kann der Administrator also eine unbegrenzte Summe von verschiedenen Strukturen mit unterschiedlichen Berechtigungen nebeneinander verwalten. Dies bedeutet, dass sich beispielsweise verschiedene Niederlassungen auf einem MDM-System managen, verschiedene Hierarchieebenen mit mehrfach gekapselten VIP-Strukturen problemlos verwalten und verschiedene LDAP/AD-Strukturen, Proxys und BlackBerry Enterprise Server in das MDM-System einbinden lassen [10].

Apps können mit datomo MDM sowohl auf White als auch auf Black Lists vermerkt werden [12]. Letztere enthalten alle Apps, die für das Unternehmen keinen Nutzen haben (z. B. Spiele) oder als gefährlich eingestuft wurden. Diese Apps, den App Store sowie den Webbrowser kann der Administrator des MDM-Systems blocken.

Darüber hinaus besteht die Möglichkeit, Apps zu einem bestimmten, vorher festgelegten Zeitpunkt zu pushen. Dies bedeutet, dass der Administrator bestimmten Usern (bzw. Usergruppen) spezifische Apps temporär zuteilt. Auch so genannte „App-Packages“ (Pakete) können geschnürt werden. Diese lassen sich auf die Geräte gewisser Benutzergruppen pushen [13]. Auf diese Weise wird gewährleistet, dass bestimmte User (z. B. einzelne Abteilungen) über die gleichen Apps verfügen. Zudem haben User auf sämtliche für sie relevante Apps Zugriff, ohne dass sie diese selbst zusammenstellen müssen.

Sollten die Daten einmal nicht mehr sicher sein, z. B. bei Verlust des Geräts, besteht die Möglichkeit zum Geräte-Wipe. Hierbei wird auch die SD-Karte gelöscht [12].

Bunya

Hyderu ist ein europäisches Unternehmen für Mobilität, Sicherheit und Android-Lösungen mit Sitz in Malta und beschäftigt rund zehn Mitarbeiter. Gegründet wurde Hyderu im Jahr 2010 als Schwestergesellschaft von Ricston, einem IT-Beratungsunternehmen mit mehrjähriger Erfahrung im Bereich Open Source Business Intelligence.

Ein Produkt von Hyderu ist Bunya [24] – eine lizenzpflichtige, proprietäre MDMS-Lösung. MDMS steht für Mobile Device Management Security und hat zum Ziel, die maximale Sicherheit und Kontrolle über ein Gerät zu gewährleisten. Mit dem Zusatz Security verzichtet Bunya innerhalb des Systems auf die Nutzung jeglicher Plattformen für den Download von Apps oder Content. Dies macht Bunya zu einem reinen Arbeitswerkzeug, für das die Sicherheit des Unternehmens höchste Priorität hat. Damit der Nutzer weiterhin Dinge privater Natur mit seinem Gerät erledigen kann, muss erst die Bunya-Applikation beendet werden.

Bunya ist als SaaS- sowie On-Premise-Variante verfügbar und unterstützt ausschließlich Android-Geräte. Dies gewährleistet eine bessere Kompatibilität der Ge-

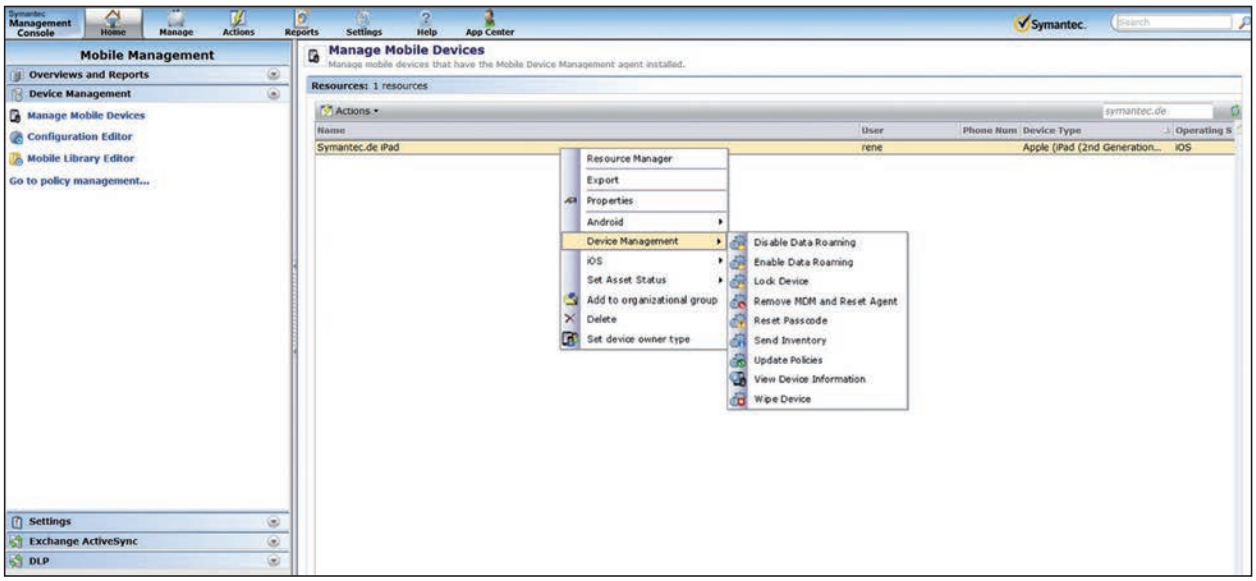


Abb. 5: Web-UI von Symantec

räte untereinander. Auch fällt der Aufwand für die App-Entwicklung dadurch etwas geringer aus. Dank der engen Zusammenarbeit mit GoTek, einem Hardwarelieferant für Tablet und Smartphones, ist eine Logoanbringung auf den Geräten möglich. Darüber hinaus kann auf Kundenwunsch auch eine zusätzliche Hardwaresicherung, wie etwa ein Sensor für Fingerprints, verbaut werden. Das Konzept von Bunya sieht zwar vor, die Endgeräte sowie das dazugehörige Verwaltungstool aus einer Hand zu beziehen, jedoch besteht kein Zwang zum Kauf genau dieser Endgeräte.

Bunya bietet eine schlichtgehaltene, webbasierte Administrationsschnittstelle, eine eigene Rubrik für Tracking, eine Wipe-Funktion sowie Messageservice und Reset des Userpassworts. Zudem können statische Ressourcen, wie Dokumente, Bilder, Videos etc. verwaltet werden. Des Weiteren kann im Falle eines Serverausfalls der Gerätebetrieb über eine Offlinenutzung aufrecht gehalten werden. Dieses Feature ist pro Ressource frei einstellbar. Um die Performance den aktuellen Gegebenheiten anzupassen, sind die Intervalle für die Synchronisation mit dem Server frei skalierbar.

Bei Bunya erfolgt die Rechtevergabe einmalig in einer Dreikomponentenauthentifikation nach Rollen, Ressourcen und Geräten. Jedem Nutzer wird eine Rolle vergeben, ein bestimmtes Gerät zugewiesen und fest definierte Ressourcen bereitgestellt. Die Nutzung fremder Geräte, Ressourcen oder Identitäten ist somit ausgeschlossen. Bunya verzichtet auf den Einsatz von Policies für Ressourcen oder Nutzer. Diese sind hier nicht zwingend notwendig, weil der Nutzer bei Bunya von vorn herein kein Wahlrecht auf Ressourcen hat.

MDM-Systeme im direkten Vergleich

Beim direkten Blick in das jeweilige MDM-System sind sowohl Vor- als auch Nachteile zu erkennen. Tabelle 1 gibt einen kurzen Überblick über einzelne Charakteristika, die nachfolgend erläutert werden.

	AirWatch	Bunya	datomo	MobileIron	Symantec
Lizenzmodell	SaaS/On-Premise	SaaS/On-Premise	SaaS/On-Premise	SaaS/On-Premise	SaaS/On-Premise
Geräte	Android, iOS, BB, Windows, Symbian	Android	Android, iOS, BB, Windows, Symbian, Bada, HP WebOS	Android, iOS, BB, Windows, Symbian, HP WebOS	Android, iOS, BB, Windows, Symbian
Administrations-Interface	Webportal, App	Webportal	Webportal, App, Helpdesk	Webportal, App, Helpdesk	Webportal, App, Helpdesk
Web-UI	ansprechend, klassisch, tabellarisch	klassisch, tabellarisch	komplex, klassisch, tabellarisch	klassisch, tabellarisch	komplex, tabellarisch, Windows-affin
BYOD	ja	nur Android	ja	ja	ja
Bereitstellung	App Store, App-Katalog, Push	Push	App Store, App-Katalog, Push	App Store, App-Katalog, Push	App-Katalog, Push
Gerätesicherheit	Passwort, zweistufige Benutzer-authentifizierung	Benutzernamen, Passwort	Benutzernamen, Passwort	Passwort, Geräte-ID, Zertifikate	VPN-Clients für Mobiltelefone, Passwort

Tabelle 1: Die fünf MDM-Anbieter im Überblick

Lizenzmodell

Alle Anbieter stellen ihr MDM als SaaS oder On-Premise bereit. Auch bei der Lizenzierung können keine Unterschiede festgestellt werden. Eine Lizenz wird in aller Regel für ein Jahr vergeben, wobei jeweils pro Serverapplikation und je Endgerät lizenziert wird. Bei der SaaS-Variante entfällt zumeist eine Serverlizenz, stattdessen wird eine monatliche Gebühr für das Hosting fällig. Genaue Preise sind bei den Herstellern auf Anfrage zu erhalten. Lediglich datomo und MobileIron verzichten auf Serverlizenzen.

Auch was den Support anbelangt, gibt es keine nominalen Unterschiede. Alle Hersteller bieten verschiedene Supportlevels an, die von Onlineanfragen bis hin zu 24/7-Support in den Enterprise-Versionen reichen.

Geräte

Bis auf eine Ausnahme werden von allen Anbietern die gängigen Betriebssysteme (Android, iOS, BlackBerry, Windows) unterstützt. Darüber hinaus heben sich MobileIron, datomo und Symantec besonders hervor, da sie auch ältere Systeme wie Symbian unterstützen. Mit datomo und MobileIron können Exoten wie HP webOS verwaltet werden. Mit Symantec lassen sich sogar Laptops in die Verwaltung einbinden. Lediglich Bunya konzentriert sich auf die marktführende Plattform Android.

Administrationsinterface

In die Administration gelangt man bei allen Anbietern über ein Webportal. AirWatch, datomo und MobileIron ermöglichen dem Nutzer eines Endgeräts darüber hinaus, einfache administrative Einstellungen selbst via App vorzunehmen. Symantec, MobileIron und datomo stellen wahlweise einen Helpdeskzugang bereit. Alle MDM-Systeme warnen den Administrator, sobald ein Endgerät

Mit datomo und MobileIron können sogar Exoten wie HP webOS verwaltet werden.

nicht regelkonform genutzt wird oder unerlaubte Apps (bzw. Software) auf den jeweiligen Endgeräten installiert werden. Lediglich Bunya kommt ohne diese Überwachungsmechanismen aus, weil hier eine strikte Trennung von privater und geschäftlicher Nutzung erfolgt. Der Nutzer erhält kategorisch keine Berechtigung zur Veränderung im geschäftlich genutzten Bereich.

Web-UI

Bei der Gestaltung des Administrationsinterfaces haben sich alle Hersteller mehr oder weniger an ein klassisches Layout gehalten – das bedeutet eine Navigationsleiste on top und einen tabellarischen Aufbau für das Listing der Geräte bzw. Einstellungen. Eingaben und Änderungen erfolgen bei den Anbietern ebenfalls in tabellarischen Formularen, die teilweise mit Drag-down-Feldern für eine vereinfachte Eingabe ergänzt werden. Klar und übersichtlich präsentieren sich hier AirWatch (**Abb. 1**) und MobileIron (**Abb. 2**). Auch Bunya zeigt sich einfach und übersichtlich (**Abb. 3**).

Bei datomo wird ein natives Erkennen etwas erschwert. Grund dafür ist, dass zwei horizontale Menüleisten den Bildschirm aufteilen (**Abb. 4**). Jeder Menüpunkt der unteren Leiste ist zudem noch einmal durch seitlich angebrachte Reiter unterteilt. Bei Symantec ist das Web-UI sehr komplex aufgebaut. Im Großen und Ganzen handelt es sich zwar wie bei den meisten MDM-Anbietern um ein tabellarisches, in Frames un-

terteiltes webbasiertes Interface. Viele Einstellungen lassen sich allerdings nur in Pop-up-Fenstern vornehmen (**Abb. 5**). Der geübte Windows-Benutzer wird sich in dieser Struktur schnell beheimatet fühlen. Wer das Arbeiten mit rein webbasierten Anwendungen gewohnt ist, sollte eine Schulung besuchen.

BYOD

Bring your own Device ist bei allen Anbietern umsetzbar. Die Ausprägung ist jedoch stark von dem Gerät selbst und dem eingesetzten MDM abhängig. Die größte Flexibilität, vor allem für die Endnutzer, bieten MobileIron, AirWatch, datomo und Symantec. Hier werden die gängigsten Geräte und teilweise sogar Exoten wie Symbian, HP Palm oder Bada für mobile Endgeräte unterstützt. Bunya bleibt sich an dieser Stelle treu. Entwickelt für den reinen Unternehmenseinsatz, lässt sich BYOD hier nur realisieren, wenn alle Mitarbeiter mit Android-Geräten arbeiten.

Bereitstellung

Die Bereitstellung der Apps und die Synchronisation der Endgeräte erfolgt bei allen Anbietern over the Air. Zudem besteht bei allen Anbietern die Möglichkeit, Apps auf die Endgeräte zu pushen. Unterschiede liegen hier in der Verteilung von Apps. Bei Bunya verteilt lediglich der Admin die Apps, damit nur validierte Inhalte und Anwendungen auf die Endgeräte kommen. Hingegen bieten MobileIron, AirWatch und datomo zusätzlich einen In-House App Store an, in dem jeder Mitarbeiter auf Wunsch validierte Apps abrufen kann. Darüber hinaus besteht bei MobileIron, AirWatch, datomo und Symantec die Möglichkeit, Apps aus öffentlichen Plattformen herunterzuladen. Diese werden anschließend über Black und White Lists validiert.

Gerätesicherheit

Bei allen Anbietern wird der Zugang zum MDM über einen Nutzernamen und ein Passwort gebildet. Die eindeutige Identifikation der Geräte findet parallel über die Geräte-ID statt. Zudem ist in allen Fällen eine zusätzliche Sicherung per Soft- oder Hardtoken möglich. Dies ist zwar bei keinem Anbieter standardmäßig im Paket enthalten, wird aber unterstützt. Einzige Ausnahme bildet Bunya. Dank der Zusammenarbeit mit einem Hardwarehersteller ist bei Bezug der Endgeräte über Bunya der Einbau von Fingerprintern möglich.

Ausblick

Der Markt für mobile Endgeräte verzeichnet jährlich wachsende Umsatzzahlen. Längst sind Smartphones, Tablets und Co. nicht mehr nur rein funktionale Nutzgegenstände, sondern sprechen durch groß angelegte Werbekampagnen der Hersteller den Nutzer individuell an. Firmengeräte dagegen sind traditionell auf Funktionalität und Wirtschaftlichkeit geeicht. Dies stößt bei vielen Mitarbeitern auf Unverständnis und bewegt sie dazu, ihre persönlichen Geräte auch für betriebliche Arbeiten herzunehmen. Jedoch sorgt dieses Verhalten für eklatante Sicherheitslücken in den Firmennetzwerken. MDM-Hersteller sind bemüht, beide Parteien bestmöglich zu bedienen. Der Mitarbeiter soll die Möglichkeit haben, sein lieb gewonnenes Gerät zu nutzen. Und die Firma soll vor externen Angriffen geschützt werden.

Alle fünf vorgestellten MDM-Hersteller bieten sowohl für Kleinunternehmen als auch für globale Konzerne Lösungen an. Für welchen Anbieter man sich letztlich entscheidet, ist stark von der Unternehmensgröße und dem Einsatzgebiet abhängig. Dabei spielt die Eigenverantwortlichkeit des jeweiligen Mitarbeiters ebenfalls eine maßgebliche Rolle. Denn bei allen Sicherungsmethoden – egal ob Black Lists, regionale Begrenzungen oder Wipe-Funktionen – ist ein verantwortungsbewusster Umgang mit der neuen Technik die beste Voraussetzung für den Schutz von Firmendaten.

MDM bleibt also ein interessantes Thema mit vielen offenen Fragen. Wie wird sich der MDM-Markt wohl zukünftig entwickeln? Welche Lösungen erfahren größte Akzeptanz? Und welche neuen Einsatzmöglichkeiten wird es für MDM geben? Es bleibt sicher spannend, die Entwicklung des noch jungen Markts weiter zu beobachten.

Lukas Holzamer ist seit 2012 bei der Ancud IT-Beratung GmbH beschäftigt. Seine Aufgabenschwerpunkte liegen in den Bereichen Mobile Device Management und Webgestaltung.

Mario Tonelli arbeitet seit 2012 für die Ancud IT-Beratung GmbH. Er ist mit den Themen Mobile Device Management und Java-Programmierung betraut.

Christine König ist seit Oktober 2012 als Marketing-Managerin bei der Ancud IT-Beratung GmbH beschäftigt. Von 2010 bis 2012 war sie dort bereits als freie Mitarbeiterin in Presse und Marketing tätig.

Links & Literatur

[1] AirWatch PDF: Der Secure Content Locker von AirWatch

[2] AirWatch-Webinar vom 30.11.2012

[3] http://www.air-watch.com/de/warum-airwatch

[4] MobileIron-Webinar vom 14.12.2012

[5] http://bit.ly/KeMh41

[6] http://bit.ly/WmNGpl

[7] https://symantec.com

[8] Symantec-Webinar vom 11.01.2013

[9] http://www.symantec.com/app-center-enterprise-edition

[10] http://datomo.de

[11] datomo-Whitepaper: datomo Mobile Device Management

[12] http://www.datomo.de/de/mobile-device-management/features.html

[13] datomo-Webinar vom 22.11.2012

[14] http://www.hyderu.com/index.php/bunya/what-is-bunya

Anzeige

Nexus 4 und Nexus 10 genauer unter der Lupe

We are family

Die Neuerungen des Android-4.2-Release, Jelly Bean, aus Nutzersicht wurden in der letzten Ausgabe in einem ersten Übersichtsartikel dargestellt. Nun möchten wir uns mit den neuen Nexus-Referenzgeräten beschäftigen, die inzwischen traditionsgemäß ein Android-Release begleiten. Ein Novum ist hier sicherlich, dass mit Android 4.2 gleich drei Geräte in den Größen 4, 7 und 10 Zoll die Bezeichnung „Nexus" bekamen. Sind dies reine Entwicklerreferenzgeräte oder eignen sie sich auch für Otto Normalnutzer? Wie sind die ersten Erfahrungen in der Praxis? Sind sie für die Entwicklung auch komplett „hackable"? Diese Fragen beantwortet der zweite Teil dieser Serie.

von Christian Meder

Seitdem Google mit dem Nexus One zum ersten Mal die Nexus-Marke eingeführt und damit dem „Google-Telefon" einen eigenen Namen gegeben hat, ist schon einige Zeit vergangen. Was aber zeichnet Geräte der Nexus-Marke eigentlich aus?

Die Nexus-Marke

Erstens werden Nexus-Geräte mit einer nativen Android-Oberfläche ohne herstellerspezifische Anpassungen (skins) ausgeliefert. Zu Zeiten des Nexus One (Android 2.1, Eclair) war das eine Seltenheit – es gab nur sehr wenige andere Smartphones, die ohne herstellerspezifische Anpassungen verfügbar waren. Sicherlich war dies auch der Tatsache geschuldet, dass die Originaloberflächen der älteren Android-Releases und ihrer Apps nicht gerade mit einem herausragenden Design glänzten. Insofern zeigen Nexus-Geräte aber immer das unverfälschte Android.

Zweitens werden die Updates der Nexus-Geräte direkt von Google bereitgestellt und nicht durch die Gerätehersteller. Gerade durch die herstellerspezifischen Modifikationen benötigen die Gerätehersteller nach einem neuen Android-Release oft noch viele Monate Zeit, um ein Update auf die aktuelle Android-Version für ihre Smartphones und Tablets bereitzustellen – wenn es denn überhaupt ein Update gibt. Plattformupdates für Nexus-Hardware direkt durch Google erfolgen meist parallel bzw. kurz nach Freigabe eines neuen Android-Releases.

Drittens lassen sich auf Nexus-Geräten sehr einfach Root-Rechte freischalten (unlocking, rooting). Ein komplettes Systemabbild kann einfach mithilfe der Quellen von source.android.com gebaut werden. Damit ist es Entwicklern oft leichter als bei anderen Geräten möglich, mit Modifikationen an der Android-Plattform zu experimentieren.

Viertens werden die Nexus-Geräte von wechselnden Hardwareherstellern in enger Kooperation mit der Android-Abteilung von Google entworfen. Nexus-Geräte sind also immer das Ergebnis einer engen Verzahnung der Android-Spezialisten von Google und der Produktspezialisten des Hardwareherstellers.

Nexus 4

Der jüngste Spross in der Historie der Nexus-Smartphones ist das Nexus 4, das diesmal von LG gebaut wird. Von den reinen Hardwaredaten braucht es sich beileibe nicht vor der gegenwärtigen Konkurrenz zu verstecken. Es wird mit einem 4,7 Zoll großen Bildschirm ausgeliefert, der sich in 1280 x 768 Pixel auflöst, also 320 ppi (Pixels per Inch) darstellt. Zum Vergleich: Das iPhone 5 hat einen 4-Zoll-Bildschirm mit 1136 x 640 Auflösung (326 ppi), das Samsung Galaxy S III einen 4,8-Zoll-Bildschirm mit 1 280 x 720 Auflösung (306 ppi), das HTC One X einen 4,7-Zoll-Bildschirm mit 1 280 x 720 Auflösung (312 ppi) und das Nokia Lumia 920 einen 4,5-Zoll-Bildschirm mit 1280 x 768 Auflösung (332 ppi). Wie die Konkurrenten verwendet das Nexus 4 Corning Gorilla Glass 2. Die aktuelle Generation von Top-Smartphones schenkt sich also nicht wirklich viel in puncto Bildschirmgröße und -auflösung, vielleicht abgesehen von der Bildschirmgröße des iPhone.

Das Nexus 4 verwendet die Qualcomm-Snapdragon-S4-Pro-Plattform, einen 1,5-GHz-Vier-Kern-Krait-Prozessor mit einer Adreno-320-Grafikeinheit, unterstützt durch üppige 2 GB RAM. Vorbei die Zeiten, in denen Ein-Kern-CPUs sogar für Desktoprechner ausreichend waren. Schaut man sich reine Benchmarkergebnisse an [1], so spielt das Nexus 4 eher im Mittelfeld der Top-Smartphones. Gerade im Bereich 3-D-Grafik und Browserbenchmarks liegen das iPhone 5 und das iPad noch

Artikelserie zu Android 4.2

Teil 1: Die Nutzerseite
Teil 2: Nexus-Hardware
Teil 3: Die Entwicklersicht

immer vorne. Die Tests der Qualcomm-Snapdragon-S4-Pro-Plattform [2] lassen aber vermuten, dass beim Nexus 4 in Sachen Performanz noch Luft nach oben ist.

Die weitere Ausstattung ist im üblichen Rahmen der aktuellen Smartphone-Generation: Kamera mit 8 MP, Frontkamera mit 1,3 MP, NFC, GPS, Bluetooth, Kompass, Beschleunigungsmesser, Barometer etc. Die Kamera stellt mit ihren 8 MP einen großen Schritt nach vorne dar, gerade im Vergleich zu den 5 MP des Vorgängers, kommt aber nicht an die Qualität der Konkurrenz heran. Bemerkenswert ist, dass das Nexus 4 induktiv geladen werden kann. Zwar glänzt die spezielle Ladestation (Orb) noch durch Nichtverfügbarkeit, was sich aber ab Mitte Februar ändern könnte (Stand: Anfang Februar). Interessant ist auch, dass das Nexus MyDP (Mobility Display Port) unterstützt. Dabei kann über den Micro-USB-Anschluss DisplayPort, HDMI und VGA für externe Bildschirme übertragen werden. Der Akku ist nicht wechselbar, und die Batterielaufzeit siedelt sich eher im Mittelfeld vergleichbarer Smartphones an. Hier sind wahrscheinlich noch durch Updates Verbesserungen zu erwarten. Auch gewichtsmäßig ist es mit seinen 139 g etwa so leicht wie die Android-Konkurrenz.

Das Nexus 4 beherrscht nur HSDPA, eine LTE-Version gibt es nicht. Der Speicher ist nicht erweiterbar. Geliefert wird das Gerät entweder mit 8 oder 16 GB Speicherplatz zu Preisen von 299 Euro bzw. 349 Euro direkt von Google.

Nexus 10

Das Nexus 10 Tablet wird von Samsung hergestellt und hat als Besonderheit eine sehr hohe Auflösung von 2560 x 1600 Pixeln auf seinem 10,055 Zoll großen Bildschirm zu bieten (300 ppi). Das übertrifft selbst das vielbeschworene Retina-Display des aktuellen iPad mit seinen 2048 x 1536 Pixeln (264 ppi).

Prozessortechnisch ist es mit der Samsung-Exynos-Plattform, mit einem Zweikern-Cortex-A15-Prozessor mit 1,7 GHz und einer Mali T604 GPU ausgestattet. Die Benchmarkergebnisse zeigen, dass es sich trotz der hohen Auflösung in fast allen Bereichen gegen die Android-Konkurrenz durchsetzen kann und sich nur gegenüber den neuesten iOS-Geräten geschlagen geben muss [1].

Das Nexus 10 besitzt außerdem eine 5-MP-Kamera und eine 1,9-MP-Frontkamera, 2 GB Hauptspeicher, NFC, Mikro-HDMI, einen Dock-Anschluss, Bluetooth, GPS, Kompass etc. In Sachen Konnektivität wird nur Dualband-WLAN unterstützt, eine HSDPA-Variante ist nicht vorgesehen. Dafür ist das Nexus 10 mit seinen 603 g für ein 10-Zoll-Tablet angenehm leicht. Preislich liegt es bei 399 Euro mit 16-GB-Speicher, 499 Euro mit 32-GB-Speicher im Google Play Store.

Das Nexus 7 wurde bereits an anderer Stelle ausführlich betrachtet [3]. In der jüngsten Version ist vor allem eine Verdopplung des Speichers auf 16 bzw. 32 GB erfolgt sowie eine Version mit HSDPA hinzugekommen. Mit 32-GB-Speicher und HSDPA liegt das Nexus 7 bei 299 Euro.

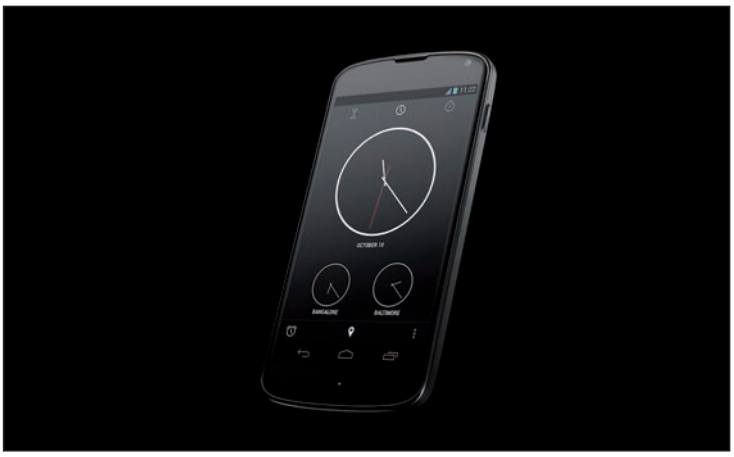


Abb. 1: Nexus 4 (Quelle: Google)

Abb. 2: Nexus 10 (Quelle: Google)

Erste Eindrücke in der Praxis

Nach einer einmonatigen Testphase kann ich subjektiv bestätigen, was auch die ersten Tests aus dem amerikanischen Sprachraum anmerkten: Das Nexus 4 fühlt sich elegant an, es liegt gut in der Hand und hinterlässt mit seiner Glasrückseite mit dem ganz dezenten Glitzereffekt einen sehr wertigen Eindruck. Allerdings tendiert das Gerät schon bei geringstem Gefälle einer Oberfläche zu rutschen. Man hat daher ständig Angst, dass es herunterfällt. Glasbruch ist hier leider eine sehr reelle Gefahr. Irritierend ist auch, dass das Nexus 4 sich bei intensiverer Nutzung recht schnell erwärmt. Die Glasrückseite ist dann sehr gut als Taschenwärmer zu verwenden. Die oft genannte Beschränkung auf HSDPA statt LTE habe ich hier in Deutschland bisher nicht wirklich als Einschränkung empfunden: Selbst für hochauflösende YouTube-Videos ist HSDPA völlig ausreichend. Im Gegensatz zu den großen Tablet-Geschwistern muss ich beim Nexus 4 immer etwas auf die Batterie achten, damit ich mit einer Ladung durch den Tag komme. Das ist zwar vergleichbar mit den anderen Smartphones, aber von einem Nexus 7 bin ich dann doch etwas verwöhnt in puncto Laufzeit. Generell ist das Gerät aber immer sehr schnell und flüssig bedienbar gewesen.

Das Nexus 10 hat mich mit seiner Leichtigkeit positiv überrascht: Vom Gewicht her ist es inzwischen näher an einer Zeitschrift als an einem Buch. Die hohe Auflösung

Abb. 3: Photosphäre

ist nur bei vergrößertem Text oder Bildbearbeitung zu sehen und fällt mir bei einer typischen Benutzung mit Webseiten oder auch Filmen nicht wirklich auf. Die Bedienung war ebenfalls wie beim Nexus 4 immer sehr schnell und flüssig möglich.

Das Nexus 7 in der HSDPA-Version spart den Schritt des Tetherings über das Smartphone, vereinfacht damit die Nutzung nochmals und hat auch einen besseren Empfang im Vergleich zum Nexus 4. Wie verhalten sich die Android-4.2-Features nun auf den Nexus-Geräten?

Jelly Bean 4.2 auf den Nexus-Geräten

Die Schnelleinstellungen sind auf allen drei Geräten sehr praktisch und hilfreich. Während die Benachrichtigungsleiste und die Schnelleinstellungen beim Nexus 7 und 10 auf die linke und rechte obere Leiste verteilt sind, gibt es beim Nexus 4 eine gemeinsame Leiste, die umgeschaltet werden kann.

Die Multi-User-Funktionalität wurde intensiv mit bis zu fünf Konten auf dem Nexus 7 und Nexus 10 mithilfe der gesamten Familie getestet. In diesem Szenario ist es ein echtes Killerfeature. Die Benachrichtigungen, der Home Screen, die installierten Apps sind sauber getrennt, und zwischen den gleichzeitigen Nutzern kann schnell umgeschaltet werden. Ähnlich einer virtuellen, persönlichen Zeitschrift liegt das Tablet jederzeit griffbereit auf dem Wohnzimmertisch.

Die neue Kamera-App ist gut zu bedienen, aber vor allem auf dem Nexus 4 interessant. Bilder mit einem 10-Zoll-Tablet zu machen, ist zugegeben etwas gewöhnungsbedürftig [4].

Photosphären kann man sowohl mit dem Nexus 4 als auch dem Nexus 10 mit passablem Ergebnis erstellen, handlich ist es aber nur auf dem Nexus 4 – der erste Versuch ist in **Abbildung 3** zu sehen. Die neu gestaltete Bildergalerie und Bildbearbeitung ist auf allen drei Geräten ein Fortschritt, von dem aber vorrangig Tablet-User profitieren, denn Bilder ansehen und Bildbearbeitung auf einem 4-Zoll-Smartphone ist nur eine Notlösung.

Die Daydream-Funktionalität, etwa mit Fotos oder Nachrichten, ist schön und praktisch auf dem Nexus 7 und 10, krankt aber aktuell noch an den schwer erhältlichen Docks. Die Lock Screen Widgets sind da schon einfacher nutzbar, und neben Kalender und Nachrichten aus sozialen Netzwerken gibt es auch die ersten Fernbedienungs-Apps für Media Center (in diesem Fall XBMC), die Lock Screen Widgets als Fernbedienungsunterstützung einsetzen.

Miracast funktioniert prinzipiell auf dem Nexus 4 [5], aber die Miracast-kompatiblen Adapter sind in Deutschland noch nicht zu bekommen. Vom neuen Virenscanner war in den vergangenen zwei Monaten des Testens noch nichts zu sehen. Das mag ein gutes Zeichen sein, allerdings ist die aktuelle Effizienz des Scanners auch umstritten [6].

Für alle Nexus-Geräte sind seit Dezember 2012 bereits erste nächtliche Builds von Cyanogenmod 10.1 verfügbar. Das Prädikat „hackable“ kann daher aus Entwicklersicht ohne Vorbehalte vergeben werden.

Fazit und Ausblick

Das Nexus 4 ist ein enormer Fortschritt gegenüber dem Galaxy Nexus und spielt mit der aktuellen Riege der Top-Smartphones auf Augenhöhe. Es reiht sich dabei nur ein, übertrifft die Konkurrenz technisch nicht, unterbietet die anderen Smartphones preislich aber deutlich. Dadurch ist es für mehrere Zielgruppen sehr interessant: für App-Entwickler, die von Google und der Community unterstützte Android-Referenz schlechthin, für Endnutzer – als gut designter und preislich sehr attraktiver Einstieg in die Welt der Top-Smartphones – und für Handyberater (sind wir das nicht alle?) als günstiges, schickes, schnelles und längerfristig mit Updates versorgtes Handy. Bliebe da noch die Verfügbarkeit. Hier hat sich Google nicht gerade mit Ruhm bekleckert. 2012 war das Nexus 4 nur weniger als eine Stunde überhaupt im Google Play Store in Deutschland verfügbar. Und in dieser Zeit konnte man viele verschiedene Varianten studieren, wie ein elektronischer Kaufvorgang auch bei Google scheitern kann. Aktuell scheint es ab Februar 2013 wieder Geräte zu geben, und hoffentlich wird sich die Verfügbarkeitsmisere dann auch entspannen.

Das Nexus 10 hat mich positiv überrascht, und als gemeinsames Familien- und Medien-Tablet hat es sei-

nen Charme. Es ist angesichts der Ausstattung preislich attraktiv, aber mit einem Kaufpreis von 500 Euro in der größeren Version kein Einsteiger-Tablet. Durch die Multi-User-Funktionalität kann es einfach gemeinsam genutzt werden und ist so das ideale Tablet für den Wohnzimmertisch inklusive Nachrichtenticker und Fotoalbum per Daydream. Allerdings gibt es dafür sicher auch bald hardwaretechnisch zwar abgespeckte, aber günstigere Varianten mit Jelly Bean 4.2.

Damit bleibt noch das Nexus 7 mit Mobilfunkunterstützung. Einige Neuerungen von Jelly Bean 4.2 sind dem Nexus 7 verwehrt: Es hat keine Kamera, profitiert also nicht von der neuen Kamera-App, und Miracast-Unterstützung wird es wohl nicht für das Nexus 7 geben. Dennoch ist es das Gerät, das ich am wenigsten vermissen möchte. Es hat das Nexus 4 die meiste Zeit zu einem reinen Telefon mit Kamera degradiert. Gelesen, gesurft, gesehen, geschrieben und gehört wird nur selten auf dem Nexus 4. Das Nexus 7 ist nun immer dabei – als Notizbuch, Taschenbuch, Taschenkino, Walkman, Zeitung, Spielekonsole etc. In den seltenen Fällen, in denen das Nexus 7 zu unhandlich ist, kommt das Nexus 4 zum Einsatz. Wenn ich zufällig in der Nähe des Nexus 10 bin, verwende ich dieses. Aber das Nexus 7 ist immer dabei.

In weiteren Artikeln werden wir auf einzelne 4.2-Features aus der Entwicklerbrille eingehen, etwa die Entwicklung von Daydreams und Lock Screen Widgets oder die Nutzung der Mehrbildschirmfunktionalität.

PS: Dieser Text wurde inhaltlich komplett auf einem Nexus 7 mithilfe einer Bluetooth-Tastatur erstellt.

Christian Meder ist CTO bei der inovex GmbH in Pforzheim. Dort beschäftigt er sich vor allem mit leichtgewichtigen Java- und Open-Source-Technologien sowie skalierbaren Linux-basierten Architekturen. Seit mehr als einer Dekade ist er in der Open-Source-Community aktiv.

Links & Literatur

[1] http://www.anandtech.com/show/6425/google-nexus-4-and-nexus-10-review

[2] http://www.anandtech.com/show/6112/qualcomms-quadcore-snapdragon-s4-apq8064adreno-320-performance-preview

[3] Meder, Christian: „Nexus 7: Das Taschenbuch unter den Tablets“, Mobile Technology 4.2012

[4] https://plus.google.com/+LinusTorvalds/posts/Qj5WnLJXLXX

[5] https://plus.google.com/115465922685935745669/posts/dv7CoXD4phS

[6] http://www.techradar.com/news/phone-and-communications/mobile-phones/research-shows-android-4-2-malware-scanner-has-paltry-detection-rate-1118294

Den Android Emulator effektiv nutzen

Emulierte Roboter

Bei vielen Entwicklern dürfte der Android Emulator gemischte Gefühle erzeugen. Einerseits ist er ein mächtiges Werkzeug und unterstützt den Entwicklungsprozess von Apps. Auf der anderen Seite verärgert er durch unzureichende Performance und verwirrende Konfiguration. Höchste Zeit, den Emulator von einer anderen, besseren Seite kennenzulernen und einige seiner Geheimisse zu lüften.

von Dominik Helleberg

Als Google im November 2007 das Android-Projekt der Öffentlichkeit präsentierte, war der Emulator die einzige Möglichkeit, sich einen ersten Eindruck von der Plattform zu verschaffen. Bei aller Euphorie zu diesem Zeitpunkt war das Erlebnis, mit dem Emulator zu arbeiten, allerdings damals schon recht ernüchternd. Quälend lange Startzeiten waren an der Tagesordnung. Und schon damals fragte man sich: Woran liegt das eigentlich?

Emulation vs. Simulation

Diese beiden Herangehensweisen lassen sich an zwei recht populären Beispielen praxisnah miteinander vergleichen, denn Android verwendet einen Emulator, iOS einen Simulator. Der Emulator verfolgt den Ansatz, eine bestimmte Hardware (in diesem Fall ein Smartphone oder Tablet) zu emulieren und dem darin laufenden System alle nötigen Hardware- und Softwarekomponenten virtuell bereitzustellen. Im Android Emulator startet also ein komplettes Android-Betriebssystem, und die darum liegende Emulationssoftware (in diesem Fall ein erweiterter QEMU [1]) simuliert Prozessor, Speicher, Netzwerkschnittstellen etc. Vergleichbar ist dies mit bekannten Virtualisierungstechnologien im Rechenzentrum oder auf dem Desktop.

Der Simulator hingegen emuliert keine Hardware. Es wird auch kein eigenes Betriebssystem gestartet. Im Simulator sorgt eine Software dafür, die benötigten APIs bereitzustellen und simuliert eine Laufzeitumgebung und z. B. bestimmte Ereignisse, die sonst vom Betriebssystem erzeugt würden.

Beide Ansätze haben ihre Vor- und Nachteile. Im Android Emulator bewegt man sich mit seiner App in einer ziemlich realistischen Umgebung, denn das komplette Android-Betriebssystem läuft genau wie auf echter Hardware. Somit stehen alle APIs, Events, Komponenten etc. zur Verfügung, es fehlt kaum etwas, und die Schnittstellen werden vom selben Code implementiert wie im realen Szenario. Speicher- und Prozessorarchitektur können entsprechend den realen Geräten emuliert werden. Somit können auch schwierig zu simulierende Situationen wie ein voller Speicher oder zu wenig Arbeitsspeicher realistisch getestet werden. Der Nachteil liegt in der Performance. Denn letztendlich müssen alle Speicherzugriffe sowie die CPU virtuell in Software umgerechnet werden, und das kostet Zeit.

Hier punktet der Simulator. Da die Simulationssoftware nativ auf dem Betriebssystem des Rechners läuft, kommt es hier zu keinerlei Performanceproblemen. Dafür muss das komplette Verhalten des zu simulierenden Betriebssystems nachgebildet werden. Hier kann es zu inkonsistentem Verhalten kommen, teilweise fehlen auch einfach Implementierungen. Die Prozessor- und Speicherarchitektur wird nicht abstrahiert. Deshalb werden iOS-Applikationen für den Simulator gegen die x86-Architektur und die finale App dann gegen die ARM-Architektur kompiliert.

Der realitätsnähere Ansatz des Emulators wird leider mit Performance bezahlt, aber auch hier gibt es zum Glück schon Abhilfe, wie wir im Absatz „Emulator-Tuning“ zeigen werden.

AVDs im Dateisystem

Für jedes AVD werden Dateien auf der lokalen Festplatte angelegt. Manchmal ist es sinnvoll, direkt auf diese Dateien zu schauen, um z. B. Backups von AVDs zu erstellen. Sie befinden sich in der Regel im *Home*-Verzeichnis des Benutzers im versteckten Ordner *.android*. Hier liegt für jedes AVD eine *.ini*-Datei sowie ein Unterverzeichnis. Im Unterverzeichnis befinden sich verschiedene Konfigurationsdateien (*.ini*) sowie die jeweiligen Partitionen als *.img*-Dateien. In den *.ini*-Dateien finden sich Konfigurationsparameter für den QEMU und Android-spezifische Erweiterungen. Die *.img*-Dateien spiegeln die Partitionen eines Android-Systems wider. Das sind in der Regel: *cache.img*, *userdata.img* sowie *sdcard.img*. Die Systempartition wird je nach gewählter Konfiguration referenziert und befindet sich im Android-SDK-Verzeichnis unter *platforms/android-XX/images/*, wobei *XX* dem Android Target (z. B. 10) entspricht.

Konfiguration

Mit dem Android Emulator lassen sich nahezu beliebige Gerätekonfigurationen emulieren. Dabei steht dem Entwickler eine Vielzahl von Parametern zur Verfügung, die pro Gerät unterschiedlich konfiguriert werden können. Die wichtigsten Parameter sind dabei:

- Android-Betriebssystem-Version (z. B. 2.3.3, 3.0, 4.1 etc.)
- Anzahl der Pixel des Displays (z. B. 1 280 x 800)
- Physische Größe des Displays (z. B. 7 Zoll)
- Größe des Arbeitsspeichers (z. B. 1 024 MByte)
- Größe des Festspeichers, eventuell der SD-Karte (z. B. 512 MByte)
- CPU-Architektur (z. B. ARM, x86, MIPS)
- Vorhandene Hardwarekomponenten wie Kameras, GPS, Beschleunigungssensor etc.
- Vorhandene Eingabekomponenten wie Keyboard (physisch/virtuell), Buttons (physisch/virtuell), Trackball, D-Pad etc.
- Vorhandene Netzwerkschnittstellen (z. B. GSM, WiFi etc.)

Diese (und weitere) Parameter werden in so genannten AVD-(Android-Virtual-Device-)Konfigurationen verwaltet. Jedes AVD stellt dabei ein eigenständiges Gerät mit eigenem internem Speicher und getrennter SD-Karte dar. AVDs lassen sich komfortabel über den AVD-Manager erstellen und verwalten. Dieser wird entweder über ein Icon aus der Eclipse Toolbar gestartet oder über die Kommandozeile mit dem Aufruf:

```
android avd
```

Seit ADT (Android Developer Tools) Version 21 gibt es vordefinierte Gerätekonfigurationen für z. B. ein Nexus S, Nexus 7 oder ein typisches 10-Zoll-Tablet. Auf Basis dieser Konfigurationen können ganz einfach eigene AVDs erstellt werden.

Tipp: Beim Anlegen von AVDs sollte man die Größe der simulierten SD-Karte nicht zu groß ansetzen, da für jede emulierte SD-Karte eine Image-Datei mit entsprechender Größe angelegt wird, siehe auch Kasten „AVDs im Dateisystem“. Hat man sein eigenes AVD konfiguriert, kann man es über den AVD-Manager oder über die Kommandozeile mit dem Befehl starten:

```
emulator -avd  [avdname]
```

Eine Liste aller verfügbaren AVDs zeigt der Befehl:

```
android list avd
```

Ist der Emulator gestartet, kann mit ihm über die adb (Android Debug Bridge) interagiert werden, genauso wie mit einer realen, über USB angeschlossenen Hardware. Hier spielt der Emulator seine Stärken aus, denn er benötigt keine eigenen Tools, sondern verwendet dieselbe Software, die auch auf realen Geräten zum Einsatz kommt.

Listing 1

```
@local:~ $ adb devices
List of devices attached
emulator-5554  device

@local:~ $ telnet localhost 5554
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Android Console: type 'help' for a list of commands
OK
power ac off
OK
power capacity 5
OK
network delay gprs
OK
network speed gsm
OK
```

Eigene Emulator-Images erstellen

Wie zu Beginn des Artikels beschrieben, startet im Emulator das komplette Android-Betriebssystem. Das hat u.a. den Vorteil, dass man Änderungen am System selbst im Emulator testen kann. Checkt man den Android-Sourcecode aus, sind im Build-System zwei so genannte *lunch-Combos* für den Emulator hinterlegt: *full-eng* und *full_x86-eng*. Wählt man eine der beiden Optionen und kompiliert den Android-Sourcecode, kann man die generierten *.img*-Dateien (*ramdisk.img system.img* und *userdata.img*) direkt im Emulator starten, wie hier im Beispielaufruf gezeigt:

```
emulator -system system.img -ramdisk ramdisk.img -data userdata.img
```

Tipp: Netzwerktraffic im Emulator mitprotokollieren

Manchmal hilft es, während der Entwicklung oder beim Testen den Netzwerkverkehr zwischen der App im Emulator und dem Internet einzusehen. Man kann für den Emulator beim Start von der Kommandozeile aus einen Proxy konfigurieren. Eine weitere Möglichkeit bietet das Telnet-Interface: Über die Kommandos

```
network capture start out.dmp
network capture stop
```

startet bzw. stoppt man einen Tcpdump in die Datei *out.dmp*, die auf dem Host-PC des Emulators gespeichert wird. Diese kann man z. B. mit dem Tool Wireshark [3] analysieren.

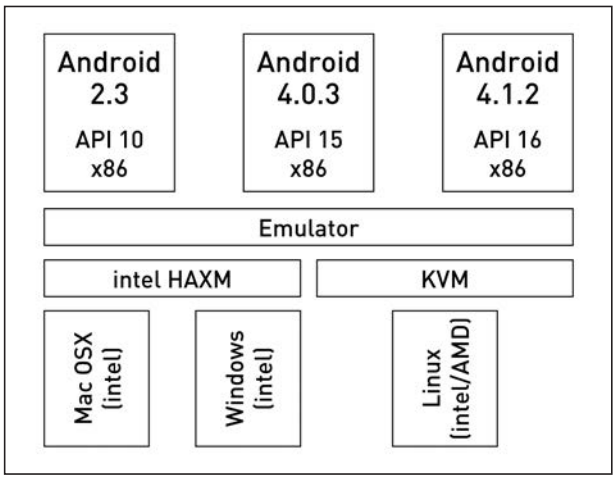


Abb. 1: Virtual Machine Acceleration

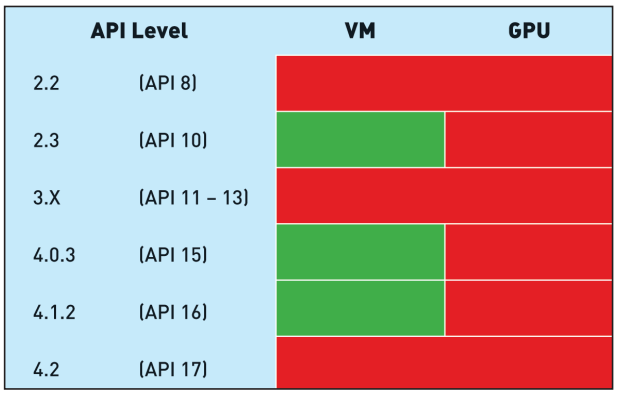


Abb. 2: VM- und Grafikbeschleunigung pro Android-Version

Simulation und Testen von Apps im Emulator

Wie schon zu Beginn erwähnt, hat der Android Emulator unter anderem die Aufgabe, dem Betriebssystem gegenüber bestimmte *Hardwarekomponenten* zu virtualisieren. Dazu zählen die aus der Server- oder Desktopvirtualisierung bekannten Bereiche wie CPU, Speicher oder Grafikchip. Da wir aber ein mobiles Endgerät virtualisieren, kommen noch weitere Komponenten wie GSM-Modem, GPS, Beschleunigungssensor, Gyroskop, Kamera oder Akku hinzu. Ein Vorteil der Virtualisierung ist, dass diese Komponenten gegenüber dem Betriebssystem beliebige Zustände annehmen können – und das quasi auf Knopfdruck. Das hilft beim Testen von Apps, denn hiermit lassen sich Zustände reproduzierbar simulieren, die normalerweise nur in realen Szenarien unter schwierigen Bedingungen auftauchen, z. B. ein extrem langsames oder kein vorhandenes GSM-Netz oder ein fast leerer Akku. Apps sollten auf diese Zustände entsprechend reagieren und der dazugehörige Code auch getestet werden. Zu diesem Zweck bietet der Emulator die Möglichkeit, die verschiedenen Komponenten des virtuellen Geräts zu konfigurieren. Hierzu bedient man sich der so genannten Emulator-Konsole [2]. Sie ist über ein Telnet-Interface verfügbar. Jeder Emulator öffnet einen Port, der im Namen der Emulator-Instanz vermerkt ist. Mittels des Befehls

```
adb devices
```

erhält man eine Übersicht über die erkannten Geräte und Emulatoren. Der Standardport des ersten gestarteten Emulators ist 5554. Über

```
telnet localhost 5554
```

verbindet man sich mit der Emulator-Konsole. Über einfache Kommandos kann man nun die virtualisierten Komponenten des Emulators beeinflussen. In einem Beispiel (Listing 1) simulieren wir folgende Schritte:

1. Ladegerät ausschalten
2. Kapazität der Batterie auf 5 Prozent setzen
3. Emulator in das 2G-Netz einbuchen
4. Bandbreite verringern und Latenz erhöhen

Zu empfehlen ist auch ein Test der App bei nicht vorhandener Netzabdeckung, um hier den Benutzer nicht mit kryptischen Fehlermeldungen oder Abstürzen zu belästigen. Mit dem Emulator lassen sich also – im Gegensatz zu einer realen Hardware – sehr einfach und reproduzierbar Ereignisse und Zustände simulieren und testen.

Emulator-Tuning

Wie schon im ersten Absatz beschrieben, ist der größte Nachteil bei der kompletten Emulation von Hardware die Performance. Dies dürfte auch der größte Kritikpunkt der Entwicklergemeinde am Android Emulator sein. Doch auch hier gibt es erste, vielversprechende Lösungsansätze. Jeder davon zielt dabei auf ein dediziertes Performanceproblem ab, das wir hier getrennt behandeln werden. Dabei werden auch wieder Parallelen zur Virtualisierung auf dem Desktop und Server sichtbar.

Snapshots: Startet man einen neuen Emulator, muss zunächst das komplette Betriebssystem gestartet werden, inklusive Initialisierung der Hardware, Ausführen des Bootloaders, Laden des Kernels, Starten des Betriebssystems usw. Da kann es schon mal ein paar Minuten dauern, bis man den Emulator wirklich verwenden kann. Hier hilft das Aktivieren von Snapshots, die beim Schließen des Emulators den aktuellen Stand von Hard- und Software in eine Datei schreiben, um diesen Stand beim nächsten Starten wiederherzustellen. Dadurch spart man die vorher beschriebenen Vorgänge und kann schneller wieder mit dem Emulator arbeiten. Snapshots werden über den AVD-Manager pro AVD aktiviert. Sollte es ein Problem beim Laden eines Snapshots geben, kann der Emulator über die Kommandozeile mit dem Parameter *-no-snapshot-load* gestartet werden.

Virtual Machine Acceleration: Mehrfach wurde hier schon die Parallele zu Virtualisierungstechnologien auf dem Desktop oder im Rechenzentrum gezogen. Auf diesem Feld wurde in das Thema Performance schon viel Energie investiert. Folglich gibt es gute Lösungen, um das Ausführen von virtuellen Maschinen zu beschleunigen. Hierzu zählen vor allem die Befehlserweiterungen

der Host-CPUs (z. B. VT-x [4] oder AMD-V [5]), um die Virtualisierung von Prozessorbefehlen, Speichermanagement und Interrupts zu beschleunigen. Seit der Version 17 der ADT kann man auch für den Android-Emulator diese Vorteile nutzen, allerdings gibt es einige Restriktionen: Ist das verwendete Host-Betriebssystem Windows oder Mac OS X, benötigt man eine Intel-CPU. Unter Linux werden AMD- und Intel-Produkte unterstützt. Des Weiteren ist unter Windows und Mac OS X die Installation des Intel Hardware Accelerated Execution Managers [6] (kurz HAXM) notwendig. Dieser befindet sich im Android-SDK in dem Verzeichnis *extras/intel/ Hardware_Accelerated_Execution_Manager/*. Unter Linux benötigt man einen Kernel mit KVM-Unterstützung (**Abb. 1**, [7]). Leider funktioniert das Ganze dann auch nur im Zusammenspiel mit x86-AVDs, und diese sind nur für bestimmte Android-Versionen verfügbar. Hat man die Voraussetzungen erfüllt, darf man sich allerdings an einer deutlich gesteigerten Performance des Emulators erfreuen. Eine detaillierte Beschreibung findet man auch auf der Android-Entwickler-Website [8].

Grafikbeschleunigung: Ein weiterer Flaschenhals des Emulators ist die Grafikpipeline. Deutlich wurde dies vor allem, als man mit dem Honeycomb-Release begann, Tablets mit erhöhter Pixelanzahl zu emulieren. Das liegt daran, dass die Grafikausgabe letztendlich in Software berechnet wird. Abhilfe schafft hier die experimentelle OpenGL-Grafikbeschleunigung. Diese Technologie nutzt die OpenGL-Fähigkeiten der Grafikkarte des Hostcomputers, um die OpenGL-ES-Befehle der Android-Grafikpipeline auszuführen. Dies bringt einen deutlichen Performanceschub, kann aber auch zu Abstürzen führen. Außerdem funktioniert diese Technologie lediglich mit AVDs des API-Levels 15 (4.0.3) oder höher, und die oben beschriebene Snapshot-Funktion muss deaktiviert werden.

Um die Grafikbeschleunigung zu aktivieren, setzt man entweder im AVD-Manager die Checkbox „Use Host CPU“ oder das Property „GPU Emulation“ auf „yes“. Alternativ kann man den Emulator auch aus der Kommandozeile mit dem Parameter *-gpu on* starten.

Es gibt also mehrere Ansätze, die Performance des Emulators signifikant zu steigern. Diese schließen sich allerdings zum Teil gegenseitig aus und sind nicht für alle Android-Versionen verfügbar (**Abb. 2**). Hier gilt es also, die richtigen Optimierungen für die jeweilige Situation auszuwählen.

Fazit

Wie gut sind also emulierte Androiden im Vergleich zu ihren realen Freunden? Es gibt hier klare Vor- und Nachteile. Die Emulation bringt Performanceeinschränkungen mit sich. Des Weiteren lässt der Emulator keine Rückschlüsse auf die Geschwindigkeit der App auf echten Geräten zu. Dafür kann er einfach und reproduzierbar bestimmte Zustände simulieren, was ihn für das automatisierte Testen von Apps qualifiziert. Die echte Hardware ersetzen kann er allerdings nicht. Auch die Maßnahmen zur Performanceoptimierung sind noch instabil und teilweise inkompatibel, aber immerhin ein Schritt in die richtige Richtung.

Dominik Helleberg ist bei der inovex GmbH für die Entwicklung von mobilen Applikationen zuständig. Neben diversen Projekten im JME-, Android- und Mobile-Web-Umfeld hat er den JCP und das W3C bei der Definition von Standards für mobile Laufzeitumgebungen unterstützt.

Links & Literatur

[1] http://wiki.qemu.org/Main_Page

[2] http://developer.android.com/tools/devices/emulator.html#console

[3] http://www.wireshark.org/

[4] http://www.intel.com/content/www/us/en/virtualization/intel-virtualization-transforms-it.html

[5] http://sites.amd.com/us/business/it-solutions/virtualization/Pages/client-side-virtualization.aspx

[6] http://software.intel.com/en-us/articles/intel-hardware-accelerated-execution-manager

[7] http://www.linux-kvm.org

[8] https://developer.android.com/tools/devices/emulator.html#acceleration

Tipp: Reale Gerätegrößen simulieren

Die verschiedensten Android-Geräte tummeln sich am freien Markt mit unterschiedlichsten Auflösungen, Pixeldichten und Bildschirmgrößen. Um sich einen Eindruck davon zu verschaffen, wie das GUI einer App auf verschiedensten Geräten aussieht (speziell die Größe und Anordnung der Elemente), ist es hilfreich, die reale, physische Größe eines Geräts zu simulieren. Diese nicht ganz triviale Aufgabe erledigt der Emulator, indem er einen Skalierungsfaktor berechnet, und zwar auf Basis der Daten über das zu simulierende Gerät sowie des verwendeten Monitors des Entwicklungsrechners. Stellt man alles korrekt ein, bekommt man einen guten Eindruck von der realen Größe des GUI auf dem Zielgerät (**Abb. 3**).

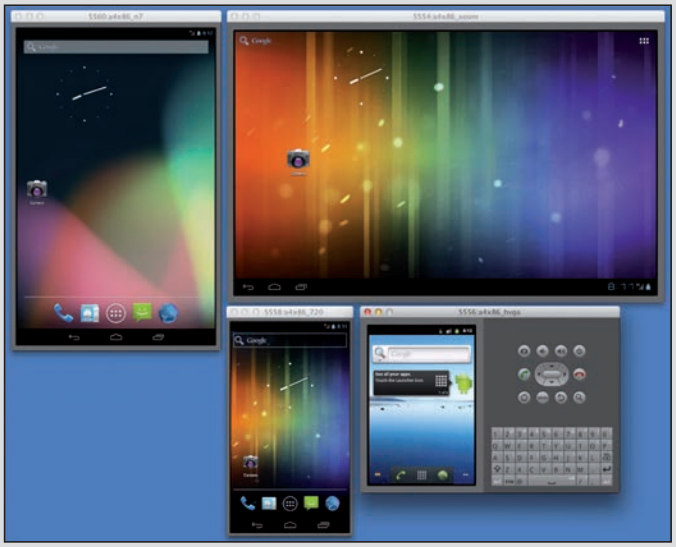


Abb. 3: Emuliert: Nexus 7, Motorola Xoom, Galaxy Nexus und Sony Ericsson Xperia mini

ORMLite, greenDAO und stORM

O/R Mapping in Android

Aus der Enterprise-Welt ist man es gar nicht mehr gewohnt, das Datenbank-Objekt-Mapping von Hand zu schreiben. So manch ein Java-Entwickler reibt sich daher verwundert die Augen, wenn er in Android die Ergebnisse einer Datenbankabfrage Spalte für Spalte auslesen und per Hand in ein Objekt mappen muss. Aber auch Android-Entwicklern, die vorher keinen Kontakt mit Object-Relational-Mapping-Frameworks aus der Java-Welt hatten, ist dieser Boilerplate-Code schnell lästig. Daher stellt sich zwangsläufig die Frage, ob es nicht auch im Android-Umfeld Möglichkeiten gibt, diesen Code nicht immer wieder manuell schreiben zu müssen.

von Lars Röwekamp und Arne Limburg 

Kein Entwickler mag es, denselben Code wieder und wieder programmieren zu müssen. Noch schlimmer ist es, wenn es sich zwar um gleich strukturierten, aber dennoch nicht völlig identischen Code handelt, da dann weder das Copy-Paste-(Anti-)Pattern hilft noch der Code geeignet auslagert werden kann. Eben vor diesem Dilemma steht jeder Android-Entwickler, der Daten aus einer relationalen Datenbank laden und in einem Objekt zurückliefern möchte. Die Daten müssen Spalte für Spalte ausgelesen und Setter für Setter gesetzt werden. Dieser Code kann verständlicherweise nicht so leicht ausgelagert werden, weil die zu befüllenden Objekte und sich damit die aufzurufenden Setter von Fall zu Fall unterscheiden. Zusätzlich muss beim Speichern des Objekts derselbe Weg rückwärts gegangen werden. Im Enterprise-Umfeld hat sich mit dem JPA-Standard längst ein API etabliert, das dem Entwickler diese lästige Arbeit abnimmt. Hierbei wird über Annotationen, die zur Laufzeit ausgewertet werden, analysiert, welche Tabellenspalten auf welche Objektattribute zu mappen sind (und vice versa).

Will man in Android mit der mitgelieferten SQLite-Datenbank interagieren, steht man allerdings vor einem Problem. Die Standard-JPA-Implementierungen kommen schon aufgrund verschiedener Abhängigkeiten, die sich nicht nach Android portieren lassen, nicht in Frage. Außerdem disqualifizieren sie sich bereits aufgrund ihres Memory-Footprints für den Einsatz auf mobilen Geräten.

Und es geht doch ...

Für den Einsatz von O/R-Mapping in Android müssen also andere Ansätze her. Und tatsächlich gibt es in Android mehrere Frameworks, die O/R-Mapping-Funktionalität anbieten. Einen ähnlichen Ansatz wie JPA bietet ORMLite [1], ein O/R-Mapping-Framework speziell für die SQLite-Datenbank. Neben eigenen Annotationen ist es ORMLite zusätzlich möglich, direkt die von JPA bekannten Annotationen zu verwenden. Dabei wird sowohl das normale Mapping zwischen Spalten und Objektattributen als auch das Mapping von Objektbeziehungen unterstützt. Das aus JPA bekannte Feature des dynamischen Nachladens von Beziehungen, das so genannte *Lazy Loading*, wird von ORMLite allerdings nicht unterstützt.

Generierung

Das scheinbare Highlight von ORMLite, nämlich die Möglichkeit, das Mapping komplett über Annotationen zu regeln, ist bei näherer Betrachtung als durchaus kritisch zu bewerten, bedeutet es doch, dass ORMLite zur Laufzeit per Reflection die Daten aus dem Objektmodell auslesen und auch wieder hineinschreiben muss. Die Verwendung von Reflection ist bekanntermaßen nicht gerade als performant zu bezeichnen. Performance wiederum ist aber auf mobilen Geräten einer der entscheidenden Faktoren.

Unter Berücksichtigung dieser speziellen „mobilen" Anforderungen gehen andere O/R Mapper für Android bewusst einen anderen Weg: Das Auslesen und Befüllen der Objekte erfolgt zur Laufzeit nicht über Reflection, sondern über Code, der vorher automatisch generiert wurde.

greenDao [2] z. B. nimmt dem Entwickler die komplette Implementierung des Domänenmodells ab und generiert dieses stattdessen automatisch zum Entwicklungszeitpunkt. Die Modellierung des Domänenmodells und das damit verbundene Object-Relational-Mapping erfolgen dabei nicht über Annotations oder XML, sondern direkt im Java-Code, und zwar in einem separaten Projekt, das neben der eigentlichen Applikation angelegt werden muss und aus dem dann die Java-Objekte inklusive Zugriffslogik generiert werden:

```
...
Schema schema = new Schema(1, "de...");
Entity person = schema.addEntity("Person");
person.addIdProperty();
person.addStringProperty("firstName");
person.addStringProperty("lastName");
person.addIntProperty("age");
```

Das klingt umständlich, ist aber sinnvoll. Durch diesen Ansatz wird sowohl ein Mapping von Objektbeziehungen als sogar auch das dynamische Nachladen von ihnen (das Lazy Loading) ohne Performanceeinbußen möglich.

Wem die Konfiguration von greenDao zu umständlich erscheint, dem bleibt noch die Möglichkeit, auf das recht junge Projekt stORM [3] zu setzen. Auch dieses Projekt basiert auf dem Ansatz der Generierung. Im Gegensatz zu greenDAO benötigt es allerdings keine umständliche Sourcecode-basierte Konfiguration, sondern funktioniert, wie ORMLite, über Annotationen. Aus diesen wird dann direkt über einen Java-Annotation-Prozessor [4] der passende O/R-Zugriffscode generiert.

Fazit

Auch in der mobilen Anwendungsentwicklung unter Android muss man nicht zwangsläufig auf automatisches O/R Mapping verzichten. Unterschiedliche Frameworks bieten die verschiedensten Ansätze. Wer sich von seinem O/R-Mapping-Framework nicht die Entitäten generieren lassen möchte, ist mit ORMLite und stORM gut bedient. ORMLite ist dabei von diesen beiden Alternativen weiter entwickelt und unterstützt im Gegensatz zu stORM auch das Mapping von Beziehungen. Gleichzeitig bringt es aber den Nachteil der Reflection-Zugriffe zur Laufzeit und die damit verbundenen Performanceeinbußen mit sich.

Wen aber generierte Entitäten und ein Quellcode-basierter Konfigurationsmechanismus nicht stören, der erhält mit greenDAO ein vollwertiges O/R-Mapping-Framework inkl. Objektbeziehungen und Lazy Loading.

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://ormlite.com/

[2] http://greendao-orm.com/

[3] https://code.google.com/p/storm-gen/

[4] http://docs.oracle.com/javase/1.5.0/docs/guide/apt/GettingStarted.html

Vorschau auf die Ausgabe 5.2013

JavaFX 2.0: Jetzt erst recht!

Mit JavaFX lassen sich Rich Internet Applications für verschiedene Plattformen erstellen. Spätestens seit Version 2.0 kann man die Technologie nicht mehr als Spielerei abtun. Höchste Zeit, JavaFX genauer unter die Lupe zu nehmen. Wir stellen den Status quo der Technologie vor und beleuchten die Embedded-Seite, denn gerade JavaFX und Raspberry Pi bilden ein hervorragendes Team. Außerdem schauen wir uns weitere Details des wachsenden Ökosystems wie den Gradle-Support, die Verbindung zu JSF und die Eclipse-Welt genauer an.

Aus redaktionellen Gründen können sich Themen kurzfristig ändern.

Die nächste Ausgabe erscheint am 3. April 2013

Querschau

eclipse MAGAZIN — Ausgabe 2.2013 | www.eclipse-magazin.de

- **Perfekte Ehe: JavaFX 2.x und die Eclipse 4.x Application Platform**
- **Am Anfang war das Wort: Acceptance Test-driven Development**
- **OrientDB: NoSQL-DBMS für den Embedded-Einsatz in RCP-Anwendungen**

entwickler magazin — Ausgabe 2.2013 | www.entwickler-magazin.de

- **Mac Security: Ein Satz mit X**
- **Schokolade für den Mac: Dependency Management in Xcode mit CocoaPods**
- **Die Effizienzmaschine: Xtext 2.3**

MOBILE TECHNOLOGY — Ausgabe 1.2013 | www.mobiletechmag.de

- **iOS6: Win-Win für User und Entwickler**
- **Android Push: Auch ohne Googles C2DM**
- **Performance: Die Wahrheit über Web-Apps**

Inserenten

Verlag:
Software & Support Media GmbH

Anschrift der Redaktion:
Java Magazin
Software & Support Media GmbH
Darmstädter Landstraße 108
D-60598 Frankfurt am Main
Tel. +49 (0) 69 630089-0
Fax. +49 (0) 69 630089-89
redaktion@javamagazin.de
www.javamagazin.de

Chefredakteur: Sebastian Meyen
Redaktion: Claudia Fröhling, Corinna Kern, Diana Kupfer
Chefin vom Dienst/Leitung Schlussredaktion: Nicole Bechtel
Schlussredaktion: Jennifer Diener, Frauke Pesch, Lisa Pychlau
Leitung Grafik & Produktion: Jens Mainz
Layout, Titel: Tobias Dorn, Flora Feher, Dominique Kalbassi, Laura Keßler, Nadja Kesser, Maria Rudi, Petra Rüth, Franziska Sponer

Autoren dieser Ausgabe:
Jeanfrançois Arcand, Andreas Bauer, Christian Gross, Martin Dilger, Walid El Sayed Aly, Sven Haiges, Dominik Helleberg, Lukas Holzamer, Jendrik Johannes, Christine König, Arne Limburg, Bernhard Löwenstein, Christian Meder, Michael Müller, Florian Potschka, Nils Preusker, Lars Röwekamp, Chris Rupp, Remo Schildmann, Michael Schnell, Dirk Schüpferling, Yann Simon, Kai Spichale, Mario Tonelli, Dino Tsoumakis, Eberhard Wolff, Tobias Zeck

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