JavaFX: Phoenix aus der Asche – Großer Heftschwerpunkt auf über 40 Seiten

S&S

Deutschland €9,80 Österreich €10,80 Schweiz sFr 19,50 Luxemburg €11,15 5.2013

JAVA Mag

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

Java EE 6

Interview mit Arun Gupta ▸10

Android Development Tools

Wo ist der Speicher hin ▸111

JAX 2013 hier im Heft! ▸51

JavaFX 2.0

Vollgas nach holprigem Start ▸36

Open Dolphin

Enterprise JavaFX ▸49

JavaFX Embedded

Mit Raspberry Pi und BeagleBoard ▸52

Gradle Plug-in

Pakete für JavaFX-Apps erstellen ▸73

Mobile JavaFX

Für iOS und Android ▸82

... und vieles mehr!

JavaFX

Alternative für Java User Interfaces auf dem Vormarsch

4 194586 709801 05

Anzeige

JavaFX – jetzt erst recht

Über die letzten Jahre haben wir die UI-Technologie JavaFX mit dem Java Magazin immer begleitet. Ich erinnere mich noch daran, dass das erste Titelthema, das ich nach meinem Start bei Software & Support Media übernommen habe, JavaFX war. Es handelte sich um die Ausgabe 12.2008 (http://bit.ly/13GRBEX) und wir fragten damals, ob wir es hier mit der „Zukunft der GUI-Entwicklung in Java" zu tun haben würden. Schon damals schrieb unser Leitartikelautor Lars Röwekamp: „Bezeichnend ist, dass die größten Skeptiker nicht von den Mitbewerbern, sondern aus dem eigenen Lager – der Welt der Java-Entwickler – zu kommen scheinen."

Die Skeptiker gewannen vorerst die Oberhand in dieser Frage, JavaFX „vegetierte" vor sich hin und Sun unternahm auch nicht genug, um die Technologie adäquat voranzutreiben. Dann kam Oracle und entschied sich (überraschend) *für* JavaFX. Hier waren plötzlich ganz andere Ressourcen verfügbar und so erschien 2011 auch JavaFX 2.0, eine enorme Weiterentwicklung, die viele Kinderkrankheiten hinter sich ließ und mit dem Ursprungsprojekt auch nicht mehr viel zu tun hatte.

Wir holten die UI-Technologie Anfang 2012 erneut auf's Titelblatt: „JavaFX, die Zweite. Desktop Java wachgeküsst" (http://bit.ly/wKGrVL). Die Situation war schon eine andere als noch 2008, aber eigentlich waren wir immer noch „zu früh". Gerade im letzten Jahr scheint JavaFX endgültig angekommen zu sein. Eine sehr lebendige Community, die Eclipse-Community, hat JavaFX akzeptiert und integriert. Das ist ein Vorteil. Die wachsende Embedded-Szene liebt JavaFX und bastelt mit der Technologie fleißig an ihren Raspberry Pis und BeagleBoards – unterstützt werden Embedded Devices, die auf der ARM-Architektur beruhen. Überall entstehen Synergien, JavaFX Plug-ins werden entwickelt, als Beispiel sei in dieser Ausgabe die Gradle-Community genannt. Ja, sogar die Implementierung für Android und iOS wird von Oracle jetzt in Angriff genommen. Der Heftschwerpunkt, den Sie hier in den Händen halten, umfasst über 40 Seiten – das ist ein Zeichen für eine sehr lebendige Community, die sich mitteilen will – und bald auch mitarbeiten kann, denn JavaFX wird in Kürze komplett Open Source verfügbar sein. Jeden Tag werden neue Teile verfügbar gemacht.

Bye-bye Swing

Und einen Punkt darf man natürlich nicht außer Acht lassen: Swing ist deprecated. Das heißt, Oracle wird es weiter unterstützen, aber an der Technologie wird aktiv nicht mehr entwickelt. Es gibt Möglichkeiten, Swing und JavaFX zu kombinieren, offen bleibt aber die Frage, was mit großen Swing-Legacy-Systemen passiert. Eine Migration ist Stand heute noch nicht möglich, Experten hoffen darauf, dass mit JavaFX 8 eine Möglichkeit geschaffen wird.

Haters gonna hate

Klar, die Skeptiker von damals sind geblieben. Ich weiß nicht, ob es dieselben sind wie 2008. Berechtigt ist die Sorge, wie zuverlässig die Technologie heute schon ist. Ich sprach mit einem Autor, der JavaFX schon in Kundenprojekten einsetzt, und er rät dazu, auf JavaFX 8 zu warten, das mit dem JDK 8 dieses Jahr erscheinen wird.

Einige Swing-Jünger haben die Fronten sicherlich gewechselt. Aber natürlich lassen sich Anwendungen auch ganz anders bauen – mit HTML5. Und natürlich gibt es hier Skeptiker, die JavaFX belächeln, denn mit HTML5 läuft ihre Anwendung direkt im Browser, ohne Installation. Das ist ein Argument, das man nicht vernachlässigen darf. Unser Leitartikelautor Björn Müller sagt deshalb auch ganz offen: Nehmen Sie HTML5! Aber wenn es sich bei Ihrer Anwendung um ein größeres, komplexeres Vorhaben handelt, dann empfiehlt er JavaFX. Warum, erklärt er in seinem Artikel ab Seite 36.

Ich sehe das alles nicht negativ und Konkurrenz gibt es IMHO auch nicht. Wir sollten uns freuen, dass wir die Auswahl zwischen zwei (und noch mehr!) Wegen haben, um das zu erreichen, worum es doch in erster Linie geht: dem Kunden das beste Ergebnis zu liefern, das mit heutigen Mitteln möglich ist. Es gibt viel zu entdecken!

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

Claudia Fröhling, Redakteurin

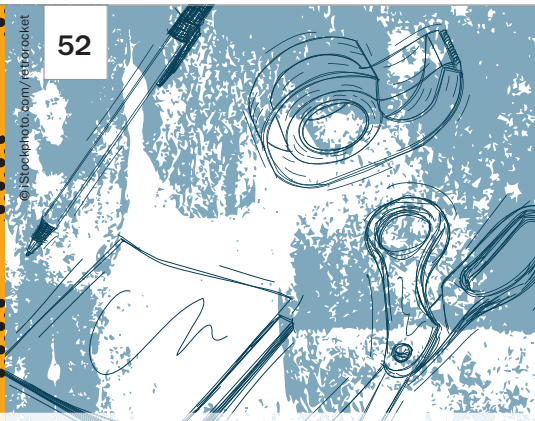
@JavaMagazin

gplus.to/JavaMagazin

JavaFX und Eclipse

In diesem Heft finden Sie einen Artikel zu e(fx)clipse, der JavaFX-Toolunterstützung für Eclipse. Dieser erschien auch im großen JavaFX-Heftschwerpunkt des Eclipse Magazins 2.2013. Alle Informationen finden Sie auf www.eclipse-magazin.de.

JavaFX



JavaFX

Einige UI-Technologien der letzten Jahre sind gekommen – und wieder gegangen. Und ausgerechnet JavaFX, die UI-Technologie, die den holprigsten Start von allen hingelegt hat, hat sich wacker gehalten. Mehr noch: JavaFX ist, man darf es mittlerweile laut sagen, technisch gelungen und erfreut sich – zumindest im Java-Lager – zunehmender Beliebtheit. Das spiegelt sich nicht zuletzt im wachsenden Ökosystem um die mehrfach wiederbelebte Technologie wider: Ob Open Dolphin, e(fx)clipse, das Gradle-JavaFX-Plug-in, auf Embedded Hardware und mobilen Betriebssystemen oder im Zusammenspiel mit JSF – JavaFX ist allgegenwärtig.

JavaFX Embedded

Seit der JavaOne 2012 steht JavaFX auf dem BeagleBoard xM und seit Dezember auch auf dem Raspberry Pi zur Verfügung. Was kann man mit einem Desktop-UI-Framework auf diesen Geräten anstellen? Gerrit Grunwald meint: Der Einsatz von Java und JavaFX auf Embedded Hardware ist sinnvoll und macht obendrein enorm viel Spaß.

Magazin

Java Core

Tutorial

Web

Titelthema

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

Die eigene Play-Webapplikation

Die ersten beiden Teile des Tutorials behandelten Grundlagen und Interna von Play. Im dritten Teil unserer Entdeckungstour werden wir unsere Applikation erweitern: Hinzufügen und Löschen von To-dos (unter Einsatz von Ajax und CoffeeScript), Testen mit Selenium und FluentLenium, Styling mit CSS/LESS und Übergabe in Produktion.

JBoss Forge

Zwar hat sich durch Tools wie Maven oder Gradle in den letzten zehn Jahren sehr viel verbessert beim Set-up neuer Webprojekte. Dennoch sagt unsere innere Entwicklerstimme: „Das muss besser gehen." Es geht auch besser, wie die Arbeit mit dem Produktivitätswerkzeug JBoss Forge zeigt.

Prozessverbesserung mit CMMI

Software bestimmt heutzutage weite Teile unseres Alltags. Die Erwartungen der Kunden zu erfüllen ist das wichtigste Ziel der Softwareentwicklung. Erreichen lässt es sich vor allem aber durch reife Prozesse. Methoden zur Prozessverbesserung in der Softwareentwicklung wie das internationale Reifegradmodell CMMI (Capability Maturity Model Integration) sind dabei wichtige Wegbereiter.

Frisch von JAXenter

News, Kommentare und Interviews von www.jaxenter.de

Der Beginn einer großen Partnerschaft: IBM und OpenStack

Auf der hauseigenen Pulse-Konferenz hat IBM verkündet, künftig seine gesamte Cloud-Architektur auf der Open-Source-Technologie OpenStack aufzubauen. Als ersten Schritt möchte man eine neue private OpenStack Cloud einrichten, anschließend soll das restliche Portfolio folgen. Dieser Schritt kommt wenig überraschend – immerhin war IBM schon im April vergangenen Jahres der OpenStack Foundation als Platinum-Mitglied beigetreten.

Auch zwei Tools zum Management von OpenStack-Instanzen hat IBM auf der in Las Vegas stattfindenden Konferenz vorgestellt. Mithilfe des SmartCloud Orchestrators lassen sich Anwendungen in Public und Private Clouds kontrollieren, die verschiedenen Konfigurationen werden in einem grafischen Interface dargestellt. SmartCloud Monitoring Application Insight bietet die Möglichkeit, Anwendungen in Echtzeit zu überwachen. Und dann wären da noch zwei bislang namenlose Betaprogramme für den Analysebereich.

Den größten Vorteil aus dieser Allianz verspricht IBM sich übrigens aus der freien Natur von OpenStack zu ziehen, denn Open Source sieht man als echten Treiber der Innovation. Schließlich sei es nur so möglich, Unternehmen nicht auf kleinen proprietären Inseln gefangen zu halten.

▶ http://ibm.co/HDKr7w

Eclipse 4.2 SR2 erschienen

Das zweite Service-Release für Eclipse 4.2 ist verfügbar. Die Eclipse-Plattform, zahlreiche Projekte des Eclipse Juno Release Trains sowie die dreizehn vorkonfigurierten Downloadpakete liegen ab sofort in aktualisierten Fassungen vor:

- Eclipse IDE for Java EE Developers
- Eclipse Classic 4.2.2
- Eclipse IDE for Java Developers
- Eclipse IDE for C/C++ Developers
- Eclipse IDE for Java and DSL Developers
- Eclipse for Mobile Developers
- Eclipse Modeling Tools
- Eclipse for RCP and RAP Developers

- Eclipse IDE for Java and Report Developers
- Eclipse for Testers
- Eclipse IDE for Automotive Software Developers (includes Incubating components)
- Eclipse for Scout Developers
- Eclipse for Parallel Application Developers

Vor allem Bugfixing und Performanceverbesserungen zeichnen die Eclipse-Plattform 4.2.2 aus. Wer sich also beim 4.2-Release im letzten Jahr noch über die mäßige Geschwindigkeit beklagt hatte, sollte dem 4.2.2 auf jeden Fall nochmals eine Chance geben. Zum Download geht es unter: http://eclipse.org/downloads.

Remote JMX mit Jolokia – jetzt auch mit Spring-Support

Das Projekt Jolokia bietet neue Wege zum Fernzugang auf JMX MBeans. Der Agent-basierte Zugang unterscheidet sich von den JSR-160-Konnektoren darin, dass er JSON über HTTP für die Kommunikation nutzt.

So enthält Jolokia auch einige neue Features zum JMX Remoting, die von den JSR-160-Konnektoren nicht abgedeckt werden. Bulk Requests ermöglichen JMX Operations mit einem einzigen Remote Server Roundtrip, ein feingranularer Sicherheitsmechanismus kann den JMX-Zugang in spezifischen Vorgängen beschränken. Auch ein JSR-160-Proxy-Modus und Versions-Tracking gehören dazu.

Aktuell ist Version 1.1.0 erschienen. Sie fügt Spring Support hinzu, sodass der Jolokia JVM Agent nun auch in einer Spring-basierten Anwendung hinzugefügt werden kann. Die neue *@JsonMBean*-Annotation erlaubt die automatische Übersetzung willkürlicher Java-Objekte in JSON Strings für JSR-160-basierte Clients wie JConsole. Auch zusätzliche Prozessoptionen für eine optimale Reaktion auf Fehlermeldungen sind im neuen Release enthalten. Jolokia 1.1.0 kann auf der Projektseite heruntergeladen werden.

▶ http://www.jolokia.org/download.html

Browser-IDE Orion 2.0 erschienen

Nach vier Monaten Entwicklungszeit und zwei Meilensteinen ist nun die zweite Major-Version der Browserentwicklungsumgebung Orion erschienen. Erklärtes Hauptziel der neuen Version ist die Verbesserung der Import- und Deployment-Prozesse sowie die Unterstützung der JavaScript-Bibliothek Node.js.

Die zuvor intensive Nutzung der Bibliothek Dojo hat man eingestellt und so die allgemeine Abhängigkeit von Bibliotheken entfernt. Im JS Content Assist wurden weitere Verbesserungen durchgeführt. Außerdem enthält Orion 2.0 einen Prototyp des Features „Projects“, mit dem sich ein SFTP Project Plus Filter einrichten lässt.

Nach Erscheinen von Orion 2.0 arbeiten die Entwickler nun bereits fleißig an Version 3.0 ihrer IDE.

▶ http://planetorion.org/news/2013/03/orion-2-0-release

JRebel 5.2 mit Updates für Apache Camel, Spring, JDeveloper und Plug-ins

Von JRebel 5.1.1 bis 5.1.3 ist nicht allzu viel geschehen. Die drei kleineren Releases haben vor allem bestehende Features verbessert und keine neuen eingeführt. Anders ist es nun bei JRebel 5.2, das eine Reihe neuer Funktionen bringt.

Das neue Release unterstützt das Neuladen von Routen und Endpunkten in Apache Camel, ohne dass ein Neustart der jeweiligen Anwendung vonnöten ist. Auch mit den aus Spring 3 bekannten *@Configuration*-annotierten Klassen kommt JRebel nun zurecht. Wer JRebel in der IDE JDeveloper nutzen möchte, kann das jetzt dank eines Beta-Plug-ins tun.

Den internen Cache kann JRebel 5.2 mit den Template Engines Mustache und Thymeleaf leeren, die Debuggerintegration für Eclipse, IntelliJ IDEA und NetBeans haben die Entwickler vollständig überarbeitet. Mittlerweile sind in JRebel ganze 48 Frameworks als Plug-in verfügbar, neu hinzugekommen sind EclipseLink, TomEE, Log4j und Guice. Alle diese Plug-ins befinden sich mit Version 5.2 nun auf dem neuesten Stand.

Wenn Sie mehr über JRebel 5.2 wissen möchten, sollten Sie sich die entsprechende Dokumentation anschauen. Diese wurde nämlich ebenfalls vollständig überarbeitet.

▶ http://manuals.zeroturnaround.com/jrebel/

JavaScript-Editor von SpringSource: Scripted 0.4 veröffentlicht

Seit Oktober 2012 stellt SpringSource seinen eigenen JavaScript-Editor „Scripted“ zur Verfügung. Bei Scripted handelt es sich eigentlich um einen allgemeinen, browserbasierten Codeeditor. Der initiale Fokus des Projekts liegt aber auf einer guten JavaScript-Programmiererfahrung. Beispielsweise werden Syntax Highlighting für JavaScript, HTML und CSS geboten. Scripted basiert auf der Editorkomponente der Eclipse Orion Web IDE und läuft lokal auf einer Node.js-Instanz. Jetzt ist Version 0.4 von Scripted erschienen.

In Scripted 0.4 wurde das Featureset ausgebaut. Beispielsweise gibt es bessere Tooltips mit jsdoc-Inhalten, ausgebauten Support für Templates und einen ersten Plug-in-Mechanismus zur Erweiterung des Scripted-Verhaltens. Den aktuellen Status des Projekts beschreibt Andy Clement auf dem SpringSource-Blog. Eine Einführung in den Editor vermittelt das untere Video auf der GitHub-Readme-Seite.

▶ http://bit.ly/YM73YE

Links und Downloadempfehlungen zu den Artikeln im Heft

- Sourcecode für Beispiel im Artikel „Memory Leaks“: http://bit.ly/YOJ0Za
- Play-To-do-Beispielapplikation: https://github.com/yanns/play2-todo-list
- ROCA: http://roca-style.org
- Interview mit Stefan Tilkov über ROCA: http://bit.ly/XUBJuL
- Dokuseite zu JavaFX: http://docs.oracle.com/javafx
- Erweiterte JavaFX Controls: http://jfxtras.org
- Open Dolphin: http://opendolphin.org
- Projekt *DolphinJumpStart:* http://github.com/canoo/DolphinJumpStart.git
- Beispielprojekt JavaFX Embedded: http://github.com/HanSolo/pitemp
- Open-Source-Bibliothek Pi4J: http://pi4j.com
- Raspberry-Pi-Beispielprojekt: https://github.com/scheuchzer/raspi-fx-button
- e(fx)clipse: http://www.efxclipse.org/
- Sourcecode zum Projekt aus dem Artikel „Wo bitte geht es zum JBoss-AS-7-Server?“: http://bit.ly/14x6JRt, http://bit.ly/YOJMWa
- Sourcecode zum Artikel „May the Forge be with you!“: http://bit.ly/12wfwb5, http://bit.ly/Xe7OK6
- RabbitEasy: http://github.com/zanox/rabbiteasy
- Beispielprojekt rabbitordering: http://github.com/zanox/rabbitordering
- Android-Security-Tipps: http://bit.ly/VV665V

Neuer Security-Patch verfügbar: Java SE 7 Update 17

Oracle hat ein neues Sicherheitsupdate für Java SE 7 veröffentlicht. Im Java SE 7 Update 17 werden zwei kritische Sicherheitslücken behoben, die erneut das Java-Browser-Plug-in betreffen. Nicht betroffen sind Java-Installationen auf dem Server, Java-Desktopanwendungen oder eingebettete Systeme. In der offiziellen Sicherheitswarnung heißt es: „These vulnerabilities may be remotely exploitable without authentication, i.e., they may be exploited over a network without the need for a username and password. For an exploit to be successful, an unsuspecting user running an affected release in a browser must visit a malicious web page that leverages these vulnerabilities."

Berichten zufolge wurden diese Bugs von Hackern bereits für Attacken genutzt, sodass ein Update dringend empfohlen wird.

▶ http://bit.ly/105Atbb

JUG-Kalender* Neues aus den User Groups

WER?	WAS?	WO?
JUG **Ostfalen**	28.03.2013 – Wicket 6 Bootcamp	www.jug-ostfalen.de
JUG **Augsburg**	28.03.2013 – CDI	www.jug-augsburg.de
JUG **Schweiz**	08.04.2013 – Best Practices & Trends für gute UIs	www.jug.ch
JUG **Darmstadt**	10.04.2013 – Continuous Delivery	www.jug-da.de
JUG **Ostfalen**	11.04.2013 – Return of the Nighthackers	www.jug-ostfalen.de
JUG **Saxony**	18.04.2013 – Acceptance Test-Driven Development	www.jugsaxony.org
JUG **Frankfurt**	24.04.2013 – Spring Data JPA: Repositories done right	www.jugf.de
JUG **Hessen**	25.04.2013 – Stand-up Coding	www.jugh.de
JUG **Ostfalen**	02.05.2013 – Liest du noch (Quellcode) oder programmierst du schon?	www.jug-ostfalen.de
JUG **Saxony**	16.05.2013 – Gradle wird's schon schaukeln	www.jugsaxony.org

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

Autor des Monats

Björn Müller, CaptainCasa GmbH, beschäftigt sich seit 2001 mit Technologiefragen der User-Interface-Entwicklung: zunächst auf Basis von HTML/JavaScript, seit 2007 aber im Rahmen der CaptainCasa-Community auf Basis Java-basierter Frontends.

Wie bist du zur Softwareentwicklung gekommen?

Über einen Apple II in den Achtzigern. Das war auch bislang das einzige, echt von mir genutzte Apple-Gerät – der „Apple-Virus" ist an mir, warum auch immer, ziemlich vorbeigegangen.

Was ist für dich der schönste Aspekt in der Softwareentwicklung?

Dass man als Einzelner oder innerhalb einer kleinen Gruppe Gewaltiges stemmen kann. Und dass Softwareentwickler/innen trotz aller Klischees recht nette Menschen sind!

Was ist für dich ein weniger schöner Aspekt?

Dem Bereich einer gründlichen User-Interface-Architektur wird aus meiner Sicht permanent zu wenig Bedeutung beigemessen. Eigentlich merkwürdig, denn hier jagt ja ein Hype den anderen …

Wie und wann bist du auf Java gestoßen?

Das war ganz früh, als Java aufkam – aus dem TurboPascal-/Delphi-Lager kommend.

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

Ich würde JavaScript verbieten! Nun, zumindest für das verbieten, was man heute damit macht. Selten war die Macht des Faktischen niederschlagender als im Bereich der Web-UI-Programmierung.

Was ist zurzeit dein Lieblingsbuch?

Gerade habe ich alle Krimis von Adler Olsen verschlungen. Und vor der Webseite http://www.alpen-panoramen.de verbringe ich viel zu viel Zeit, in der ich eigentlich arbeiten sollte …

Java EE 6 Pocket Guide

von Arun Gupta

Bücher zu Java EE 6 sind seit mehr als drei Jahren auf dem Markt. Dicke Schinken zum gesamten Themenkomplex, aber auch umfassende Bücher, in welchen Teilbereiche detailliert erläutert werden. Nun hat sich Arun Gupta darangemacht, das gesamte Thema in einem kompakten Band zusammenzufassen. Reicht das? Die typische Antwort eines Consultants: Es kommt darauf an. In diesem Fall auf das, was der Leser erwartet. Geht es darum, Java EE 6 zu erlernen, so ist dies mit dem vorliegenden Buch recht schwer. Der Autor erläutert zwar die wesentlichen Aspekte und ergänzt dies durch entsprechende Codebeispiele. Ohne Vorkenntnisse dürfte das jedoch zu knapp sein. Doch das ist nicht das Ziel des vorliegenden Buchs. Vielmehr geht es darum, dem bereits mit dem Thema vertrauten Entwickler eine knappe Referenz zur Verfügung zu stellen. Hier geht es um das schnelle Nachschlagen im Alltag. Und dafür ist das Buch bestens geeignet. Warum er es geschrieben hat, erläutert der Autor ausführlich in nebenstehendem Interview.

Im ersten Kapital verschafft der Autor dem Leser einen Überblick zu Java EE 6. Hier geht es um die Architektur und die enthaltenen Spezifikationen, die kurz gelistet werden. Arun Gupta weist zehn Schlüsseltechnologien aus, denen er im weiteren Verlauf des Buchs jeweils ein eigenes Kapitel widmet, teilweise mit Querbezügen: ManagedBeans, Servlets, JPA, EJB, CDI, JSF, SOAP und REST, JMS und Bean Validation. Wer mit den meisten dieser Abkürzungen etwas anfangen kann, gehört zur Zielgruppe des Buchs. Jedes Kapitel beginnt mit der Benennung des betreffenden JSRs und dem Verweis auf die komplette Spezifikation. Danach folgt ein kurzer Überblick und dann werden Kon-

Interview mit Arun Gupta

Arun Gupta ist in der Java-EE-Szene als Technologie-Evangelist bekannt. Wenn er als einer der technisch Vorderen nun rund drei Jahre nach Einführung von Java EE 6 ein Buch zu diesem Thema schreibt, so muss dies einen besonderen Anlass haben. Grund genug, ihn selbst darüber zu befragen.

Michael Müller: Hallo Arun. Sie haben gerade Ihren „Java EE 6 Pocket Guide“ veröffentlicht. Java EE 6 wurde bereits vor drei Jahren vorgestellt und seither wurden viele Bücher herausgebracht. Was war Ihre Intention, solch ein Buch gerade jetzt zu schreiben?

Arun Gupta: Auf dem Markt existieren diverse ausgezeichnete Bücher zu Java EE 6 und diese bieten einen besseren Überblick über die gesamte Plattform. Aber in meinen Diskussionen mit Architekten und Entwicklern, die mit Java EE 6 ihre Applikationen erstellen, stellte sich heraus, dass diese die Konzepte zwar gut verstehen, aber typischerweise nach beispielhaften Codeschnipseln suchen, um eine spezifische Aufgabenstellung zu lösen. Dieser Pocket Guide bedient genau die Zielgruppe von Personen, die einen schnellen Überblick benötigen, sowie passende Codefragmente, mit denen sie ihre Arbeit schnell erledigen können.

Alle wichtigen Server sind nun gerade Jave-EE-6-konform und es wird noch eine ganze Weile dauern, bis sie Java-EE-7-konform werden. Enterprise-Entwickler werden Java EE 6 noch einige Jahre nutzen und so bleibt dieses Buch noch aktuell. Mehr noch, die Java-EE-7-Plattform gründet auf Java EE 6. So kann alles auch für die nächste Version genutzt werden.

Müller: Warum sollte ein Leser Ihr Buch kaufen?

Gupta: Dieses Buch ist weder ein Tutorial noch eine detaillierte Erläuterung irgendeiner Spezifikation der Java-EE-6-Plattform. Aber wenn jemand einfach zu verstehende Codebeispiele sucht, welche die Verbesserungen der vorliegenden Version darstellen, dann ist dieses Buch das Richtige.

Das Buch kostet 14,99 US-Dollar (12,00 Euro) in der Druckausgabe und 11,99 US-Dollar (9,49 Euro) als E-Book. Wenn jemand das Buch von http://shop.oreilly.com/product/0636920026464.do erwirbt, erhält er zusätzlich kostenlos elektronische Updates. Die meisten anderen Bücher liegen typischer-

zepte, Annotationen und typische Codebeispiele vorgestellt. Dazu noch ein paar Tabellen und Grafiken und fertig ist ein kleines, aber feines Buch, das sich hervorragend eignet, um im Rahmen der Entwicklung mal eben schnell das ein oder andere Detail nachzulesen.

Der Autor schreibt sachlich, aber nicht kompliziert. Sein Englisch dürfte für die meisten Entwickler kein Problem darstellen. Zusätzlich ist aber eine deutsche Ausgabe in Arbeit bzw. zwischenzeitlich verfügbar.

Auch wenn primär als Nachschlagewerk gedacht, lässt sich das Buch aufgrund seiner Größe auch mal eben schnell von Anfang bis Ende durchlesen. Und so dient es zusätzlich als Auffrischer. Die meisten Leser werden die eine oder andere Nuance entdecken, die sie so noch nicht im Entwickleralltag genutzt haben. Insofern stellt das Buch die ideale Ergänzung zu den bekannten „dicken Schinken" dar. Anders als solche, passe der Pocket Guide, so der Autor, in die Jeanstasche und könne so immer griffbereit sein. Eine durchaus lohnenswerte Anschaffung.

Michael Müller

Arun Gupta

Java EE 6 Pocket Guide

A Quick Reference for Simplified Enterprise Java Development

208 Seiten, 12,00 Euro
O'Reilly, 2012
ISBN 978-1-449-33668-4

weise im Bereich von 30 US-Dollar oder mehr. Dieses Buch ist also erschwinglich. Und es passt in jedermanns Jeanshose, sodass es stets griffbereit ist.

Müller: Java EE 7 wird in Kürze erwartet. Was sind die hauptsächlichen Änderungen? Und kann das vorliegende Buch auch für diese Version genutzt werden?
Gupta: Die meisten Konzepte von Java EE 6 sind unter Java EE 7 gleichlautend verfügbar. Natürlich wird es diverse Erweiterungen geben, wie beispielsweise das Java-API für WebSocket (JSR 356), Java-API für JSON Processing (JS 353), Batch Applications for Java Platform (JSR 352), Concurrency Utilities für Java EE (JSR 236) und Java-Caching-API (JSR 107). Einige der existierenden Spezifikationen, wie JAX-RS 2 und JMS 2, wurden grundlegend überarbeitet, um die Produktivität weiter zu erhöhen. Auch wenn das Buch bestens mit Java EE 7 genutzt werden kann, so sind natürlich die kommenden Veränderungen noch nicht beschrieben. Das Buch wird irgendwann aktualisiert, um die neuen Technologien zu beschreiben.

Müller: Was sind Ihre Pläne bezüglich anderer Bücher, z. B. ein detailliertes Buch zu Java EE 7?
Gupta: Das Schreiben dieses Pocket Guides hat bereits eine Menge Zeit benötigt. Einige exzellente Communitymitglieder haben bereits detaillierte Bücher über Java EE 7 in Arbeit und ich möchte diesen den entsprechenden Erfolg überlassen.

Müller: Arun, vielen Dank für dieses Interview.

Memory Leaks mithilfe von Heap Dumps auffinden

Effective Java: post mortem

Im vorangegangenen Beitrag unserer Reihe über Memory Leaks in Java haben wir uns angesehen, wie man mithilfe von Profilern nach Memory Leaks suchen kann. Aber wie kommt man Memory Leaks auf die Spur, wenn die Anwendung sich infolge eines Speichermangels bereits beendet hat? Oft ist es nur der Heap Dump, der dann übrig bleibt. Wie man bei einer solchen Post-mortem-Analyse vorgeht, diskutiert der folgende Beitrag.

von Klaus Kreft und Angelika Langer

Im Zentrum des vorangegangenen Beitrags stand eine dynamische Memory-Leak-Analyse, bei der wir uns mit einem Profiler an eine laufende Anwendung gehängt und speicherneutrale Use Cases beobachtet haben. Dabei haben wir nach Objekten gesucht, die die Use Cases überleben, obwohl sie eigentlich nur für die Verarbeitung gebraucht wurden und hinterher hätten verschwunden sein sollen.

Dieses Mal wollen wir uns die Post-mortem-Analyse ansehen und wie man sie durchführt, nachdem die Anwendung sich bereits (wegen Speichermangels) beendet hat. Für diese Analyse steht in der Regel wenig Information zur Verfügung. Oft ist es nur der Heap Dump, den die JVM beim Abbruch erzeugt hat.

Wie im letzten Artikel wollen wir wieder das Beispiel aus unserem ersten Artikel der Reihe über Memory Leaks aufgreifen [1]. Darin haben wir ein kurzes, aber fehlerhaftes Programm mit Memory Leak angesehen. Es ging um die Implementierung eines rudimentären Servers auf Basis der mit Java 7 eingeführten *AsynchronousSocketChannels*. Pro Client erfolgte ein Eintrag in einer Map, der aber am Ende der Client-Session nicht wieder gelöscht wurde. Das hat zur Folge, dass die Map stetig anwächst. Bei unserem Server führt dies zunächst

Wie bekomme ich einen Heap Dump?

JVM-Schalter

Man kann die JVM mit diversen Optionen starten, die es später erlauben, Heap Dumps zu ziehen:

- *-XX:+HeapDumpOnOutOfMemoryError:* Es wird automatisch ein Heap Dump erzeugt, wenn die JVM wegen eines *OutOfMemoryError* terminiert.
- *-XX:+HeapDumpOnCtrlBreak:* Erzeugt einen Heap Dump, wenn die entsprechende Tastenkombination an der Konsole eingegeben wird.
- *-agentlib:hprof=heap=dump,format=b,doe=y:* Verwendet den HPROF-Agenten, der auf die JVMTI-Schnittstelle aufsetzt, um einen „dump on exit" (doe=y) zu erzeugen. Das passiert dann immer beim Exit, auch wenn kein *OutOfMemoryError* aufgetreten ist.

Werkzeuge aus dem JDK

Man kann Werkzeuge verwenden, die zum JDK gehören:

- *jmap –dump:format=[a|b],file=<fileName> pid:* Dabei ist *pid* die Java-Prozess-ID, die man mithilfe von *jps* bestimmen kann.
- *JConsole:* Setzt auf die Java Management Beans (JMX beans) auf. In der *HotSpotDiagnostic*-Bean, einer Non-Standard-Bean, die es nur in der HotSpot-JVM von Sun/Oracle gibt, findet man die Operation *dumpHeap()*, mit der ein Heap Dump erzeugt werden kann.

Programmatisch

Man kann das Erzeugen eines Heap Dumps aus der Anwendung heraus anstoßen, beispielsweise über die oben erwähnte *HotSpotDiagnostic* Bean und deren *dumpHeap()*-Methode. Kommerzielle Profiler wie YourKit, JProfiler oder JProbe bieten zusätzlich die Möglichkeit an, Trigger zu setzen, mit denen Heap Dumps (aber auch andere Snapshots) erzeugt werden können. Das geht sowohl aus dem Profiler heraus als auch über eine Programmierschnittstelle.

zu Speicherengpässen mit auffällig langen Stop-the-World-Pausen des Garbage Collectors und am Ende zum Absturz mit *OutOfMemoryError*. Wie findet man nun ein solches Memory Leak, wenn man nichts weiter als einen Heap Dump zur Verfügung hat?

Post-mortem-Memory-Leak-Analyse

Nehmen wir also an, unser Server ist mit der JVM-Option *XX:+HeapDumpOnOutOfMemoryError* gestartet worden; dann wird der Heap Dump im Falle der JVM-Terminierung wegen *OutOfMemoryError* automatisch erzeugt. Man kann Heap Dumps auch anders erzeugen; dazu der Kasten „Wie bekomme ich einen Heap Dump?“

Ein gutes Werkzeug für die Analyse von Heap Dumps ist der kostenlose Memory Analyzer MAT [2]. Der hat gegenüber anderen Werkzeugen den Vorteil, dass er auch sehr große Heap Dumps verarbeiten kann. Das schafft er, weil er den Heap Dump zuallererst einmal indiziert, damit er hinterher in vertretbarer Zeit und mit verkraftbarem Speicherverbrauch eine Analyse auf dem Dump ermöglichen kann.

Dominatoren

MAT analysiert die Referenzbeziehungen zwischen den Objekten auf dem Heap und erzeugt daraus einen so genannten *Dominator*-Baum (engl. *dominator tree*). Darin

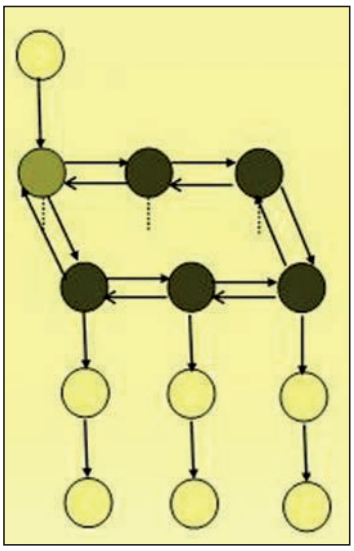


Abb. 1: „Normale“ Referenzbeziehungen

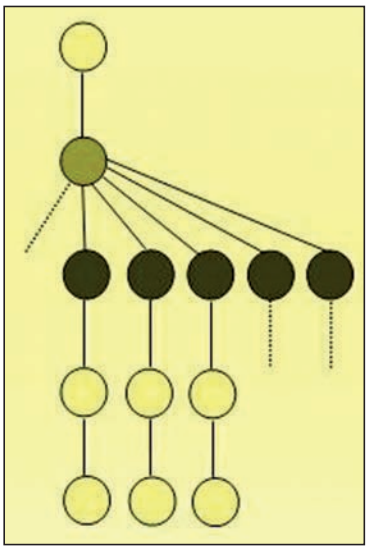


Abb. 2: Dominator-Baum

sind die Verweise zwischen den Objekten so arrangiert, dass die Verweise von einem dominierenden Knoten zu den dominierten Knoten verlaufen. Dabei ist ein dominierender Knoten, d. h. der Dominator, derjenige Knoten, den man im „normalen“ Referenzgrafen passieren muss, um die dominierten Knoten zu erreichen. Das kann man schön am Beispiel einer doppelt verketteten Liste sehen.

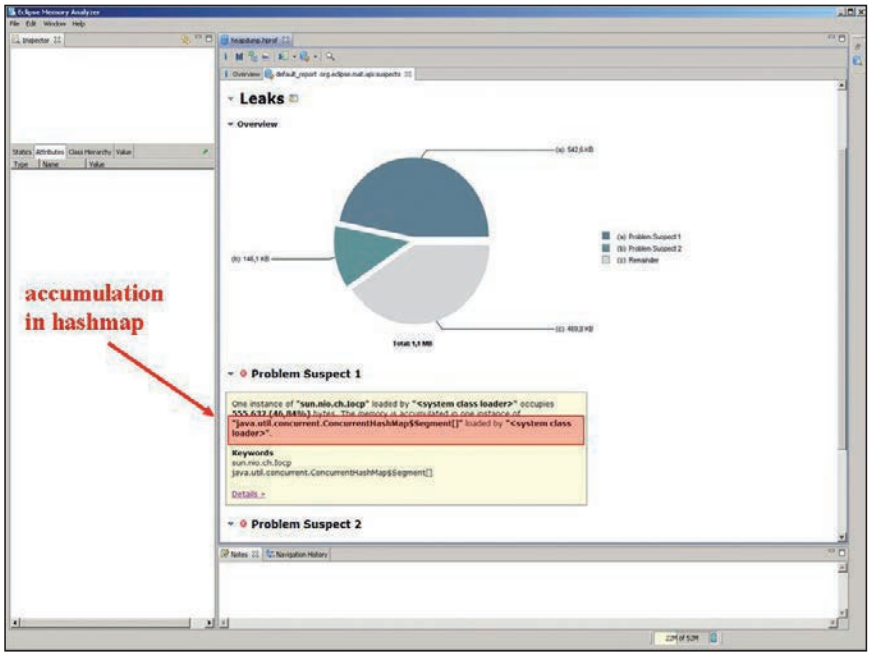


Abb. 3: Der Suspect-Report

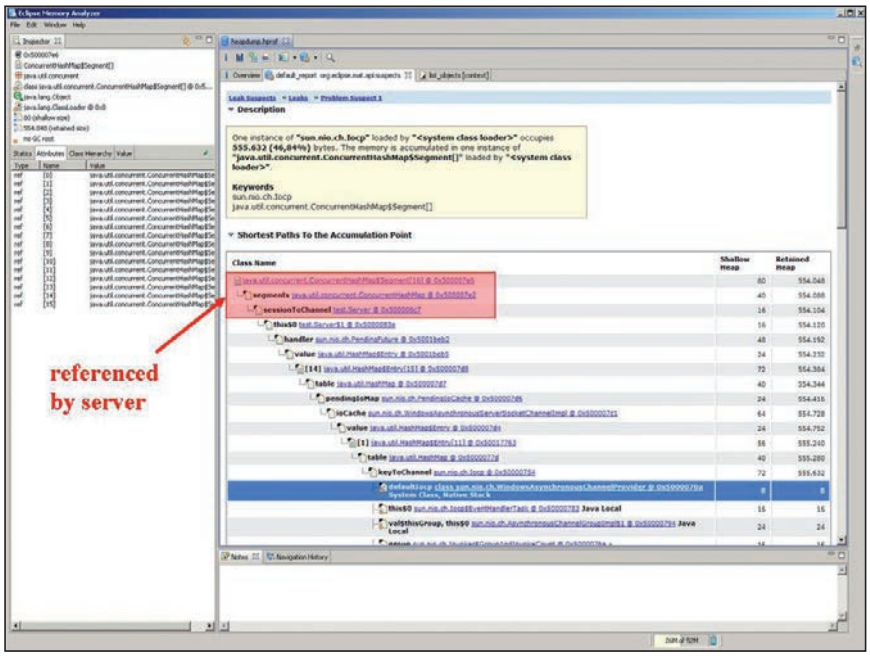


Abb. 4: Was enthält die verdächtige Map und wer referenziert sie?

Die „normalen" Referenzbeziehungen (**Abb. 1**) sind relativ kompliziert, denn von jedem Knoten in der Liste gehen drei weitere Verweise aus: vorwärts zum nächsten Knoten, rückwärts zum vorangegangenen Knoten und ein Verweis auf die Nutzdaten im Knoten. Meistens sind die Knoten einer doppelt verketteten Liste auch noch zyklisch verknüpft (d. h. der letzte Knoten verweist wieder auf den ersten), sodass beim Betrachten der einzelnen Listenknoten und ihrer Verweisbeziehungen nicht ohne Weiteres zu erkennen ist, wer wen am Leben hält.

Im Dominator-Baum (**Abb. 2**) sind die Knoten anders arrangiert. Um von dem Objekt, das auf die Liste verweist (Knoten ganz oben) zu einem der Listenelemente zu gelangen, muss man stets den ersten Knoten in der Liste passieren. Dieser Eintrittsknoten dominiert deshalb alle anderen Listenknoten und steht im Dominator-Baum über den anderen Listenknoten. Wenn man nun für jeden Knoten im Dominator-Baum berechnet, wie viel Speicher an dem Knoten hängt, d. h. an allen Referenzen, die von diesem Knoten ausgehen, dann sieht man sehr gut die Akkumulierungspunkte. Im obigen Beispiel ist es die eine Referenz vom obersten Objekt, die auf den Eintrittsknoten der Liste zeigt, die die gesamte Liste und alle ihre Elemente am Leben erhält.

Der Memory Analyzer MAT baut aus den Referenzen, die er im Heap Dump findet, einen solchen Dominator-Baum für alle Java-Objekte auf, sucht die Akkumulierungspunkte heraus und bietet diese als „Leak Suspects" für die Analyse an. Mit anderen Worten: MAT sucht aus den Objekten auf dem Heap diejenigen heraus, die große Mengen Speicher am Leben erhalten. Für unser Beispiel sieht der Suspect-Report aus wie in **Abbildung 3**.

In unserem Beispiel wird ein HashMap-Segment als Hauptverdächtiger präsentiert. In der Tat, an dem verdächtigen Objekt hängt knapp die Hälfte des verbrauchten Speichers der Anwendung. Man schaut sich dann an, was die verdächtige Map enthält und wer die Map referenziert. Auch diese Information ist im MAT gut aufbereitet (**Abb. 4**).

Man sieht sofort, dass es sich bei der verdächtigen Map um die *ConcurrentHashMap* namens *segments* im Server handelt, die wir auch mittels der dynamischen Analyse im letzten Artikel gefunden hatten.

Noch ein paar Tipps zur Memory-Leak-Analyse

In der Realität ist es mitunter etwas schwieriger, das Memory Leak zu identifizieren. Beispielweise findet man das Leak nicht so leicht, wenn man die Heap Dumps schon relativ früh zieht, d. h. nicht erst beim *OutOfMemoryError*. Dann kann es vorkommen, dass die Map noch vergleichsweise klein ist und ein nicht verdächtiger Dominator identifiziert wird. In unserem Beispiel war die Finalizer-Queue anfangs deutlich größer als unsere Map, die das eigentliche Problem darstellt. Man muss dem Leak also Zeit geben, als Dominator sichtbar zu werden.

Im Übrigen sei noch einmal darauf hingewiesen, dass nicht jedes Programm, das wegen eines *OutOfMemoryErrors* abstürzt, automatisch immer ein Memory Leak haben muss. Manche Anwendungen brauchen einfach mehr Speicher, als ihnen zur Verfügung gestellt wurde. Ob der Absturz durch ein Memory Leak verursacht wurde, kann man herausfinden, indem man der Anwen-

dung mehr Speicher mit *–Xmx<size>* zuteilt. Wenn sie dann immer noch abstürzt, schaut man sich die Dominatoren in den *OutOfMemory*-Heap-Dumps an. Ein Dominator, der bei mehr Speicher noch größer ist als vorher bei weniger Speicher, ist unter Umständen ein Hinweis auf ein Memory Leak.

Ansonsten hilft es, wenn man den Text liest, der im *OutOfMemoryError* enthalten ist. Nicht immer heißt es: *java.lang.OutOfMemoryError: Java heap space*. Texte wie *requested array size exceeds VM limit*, *PermGen space* oder *<reason> <stack trace> (Native method)*. deuten auf andere Ursachen hin, z. B. Fragmentierung, Perm-Bereich zu klein, Probleme in JNI-Teilen der Anwendung o. Ä.

Zusammenfassung

Eine Memory-Leak-Analyse des Heap Dumps wird oft post mortem gemacht, nachdem die Anwendung bereits wegen einem *OutOfMemoryError* abgebrochen wurde. Wir haben für die Heap-Dump-Analyse das frei verfügbare Werkzeug Memory Analyzer MAT verwendet, das auch sehr große Heap Dumps problemlos verarbeiten kann. Alternativ kann man es auch mit einem kostenpflichtigen Profiler wie YourKit, JProfiler oder JProbe machen. In allen Fällen wird man aber feststellen, dass die Werkzeuge nur Hilfestellungen bei der Memory-Leak-Analyse leisten können. Die wirkliche Ursache muss der Entwickler selbst finden; nur er versteht die Programmlogik.

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

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

Links & Literatur

[1] Kreft, Klaus; Langer, Angelika: „Memory Leaks, Teil 1: Ein Beispiel“, in Java Magazin 9.2012, S. 12–18

[2] Memory Analyzer (MAT): http://www.eclipse.org/mat/

[3] Sourcecode für das Memory-Leak-Beispiel aus diesem Artikel: http://www.angelikalanger.com/Articles/EffectiveJava/66.Mem.Analysis/66.Mem.Analysis.zip

[4] Kreft, Klaus; Langer, Angelika: „Memory Leaks, Teil 2: Akkumulation“, in Java Magazin 11.2012, S. 30–33

[5] Kreft, Klaus; Langer, Angelika: „Memory Leaks, Teil 3: Ausgenullt“, in Java Magazin 1.2013, S. 16–21

[6] Kreft, Klaus; Langer, Angelika: „Memory Leaks, Teil 4: Tool Time“, in Java Magazin 3.2013, S. 48–52

Webentwicklung mit dem Play-Framework (Teil 3)

Die eigene Play-Webapplikation

Nachdem wir uns in den vorherigen zwei Teilen unseres Tutorials mit den Grundlagen und Interna von Play vertraut gemacht haben, geht es nun zum letzten Teil. Wir werden unsere Webapplikation vervollständigen und uns mit weiteren wichtigen Themen wie JavaScript-Routen, Styling oder Selenium-Tests beschäftigen. Das Betreiben einer Play-Applikation wird ein weiterer wichtiger Bestandteil sein.

von Yann Simon und Remo Schildmann

In den ersten beiden Teilen haben wir uns mit dem grundlegenden Aufbau einer Play-Applikation beschäftigt. In diesem Teil unserer Entdeckungstour werden wir unsere Applikation erweitern: Hinzufügen und Löschen von To-dos (unter Einsatz von Ajax und CoffeeScript), Testen mit Selenium und FluentLenium, Styling mit CSS/LESS und Übergabe in Produktion. Abschließen wollen wir mit einem kurzen Überblick über den Einsatz von Play im realen Leben und weitere Möglichkeiten mit Play selbst. Fangen wir mit dem Hinzufügen und Löschen von To-dos an.

Aktuell zeigt unsere Applikation lediglich die in der YAML-Datei definierten To-dos an. Wir wollen aber eigene To-dos anlegen und diese bei Bedarf auch als erledigt markieren können, sodass sie aus der Anzeige entfernt werden. Für diese Funktionalitäten erzeugen wir eine eigene *Controller*-Klasse *ToDos.java*. Diese besitzt zwei Methoden, eine zum Hinzufügen und eine zum Löschen von To-dos (Listing 1). Die *delete*-Methode erwartet als Parameter die ID des zu löschenden Datensatzes. Innerhalb dieser Methode behandeln wir nur die zum angemeldeten Nutzer gehörenden To-dos, sodass nur eigene To-dos gelöscht werden können. Auf die Methode werden wir aber weiter unten noch detaillierter

Listing 1

```
package controllers;

import models.ToDo;
import models.User;

import org.apache.commons.lang3.StringUtils;

import play.data.Form;
import play.mvc.Controller;
import play.mvc.Result;
import play.mvc.Security;

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

    public static Result add() {
        String description = Form.form().bindFromRequest().get("description");
        if (StringUtils.isEmpty(description)) {
            flash("error", "Please enter a description");
        } else {
            ToDo toDo = new ToDo();
            toDo.description = description;
            toDo.assignedUser = User.find.byId(request().username());
            toDo.save();
        }
        return redirect(routes.Application.index());
    }

    public static Result delete(Long id) {
        String email = request().username();
        ToDo toDo = ToDo.findTodoByIdAndUserEmail(id, email);
        if (toDo != null) {
            toDo.delete();
        }
        return ok();
    }
}
```

eingehen. Unser *ToDo*-Model muss für die Löschen-Funktionalität noch um eine Methode erweitert werden, die ein To-do eines Benutzers ermittelt. Dazu muss die *Model*-Klasse *ToDo.java* erweitert werden (Listing 2).

Abb. 1: Anmeldebildschirm h2-Browser

Listing 2

```
...
    public static ToDo
            findTodoByIdAndUserEmail(Long id,
                                     String email) {
        return find.where()
                .eq("id", id)
                .eq("assignedUser.email", email)
                .findUnique();
    }
...
```

Hinzufügen von To-dos

Wie Sie sicher schon entdeckt haben, ist diese *Controller*-Klasse generell vor Aufrufen durch unautorisierte Nutzer abgesichert: *@Security.Authenticated(Secured.class)*. Die *add*-Methode fügt ein neues To-do für einen Nutzer hinzu, sofern dieser auf der Webseite eine entsprechende Beschreibung eingegeben hat. Neu ist das Binden der Felder unserer Form. Für diese haben wir kein korrespondierendes Model. Aus diesem Grund greifen wir auf die Felder der Form mit der *Form.form().bindFormRequest()*-Methode zu, welche eine *play.data.DynamicForm* zurückgibt. An dieser können wir die Feldwerte der Form mit *get(<FELDNAME>)* ermitteln. Das ist immer dann sinnvoll, wenn wir kein Model implementieren oder wenn die Form unterschiedliche Felder haben kann. So ermitteln wir die durch den Nutzer eingegebene To-do-Beschreibung. Eine leere Eingabe führt zu einem Fehler, den wir dann auf der Seite anzeigen. Im Fall einer Eingabe erzeugen wir für den angemeldeten Nutzer ein neues To-do und machen einen Redirect auf die *index()*-Methode unseres *Application*-Controllers. Damit wird unser neues To-do in der Liste angezeigt.

Damit die neuen Methoden aufgerufen werden können, müssen wir sie noch in unserer *routes* definieren. Fügen Sie in dieser Datei die zwei Routen hinzu (Listing 3).

Jetzt wollen wir unsere neuen Funktionalitäten natürlich verwenden. Dazu erweitern Sie die *index.scala.html* zunächst um die Funktionalität zum Hinzufügen eines To-dos. Erweitern Sie dazu die View (Listing 4). Wenn Sie unsere Applikation nun starten, sollten Sie neue To-dos eingeben können.

Der eingebaute h2-Browser

Es ist durchaus sinnvoll, die Daten in unserer Datenbank auch außerhalb unserer Anwendung anzeigen zu können. Play bringt einen h2-Browser mit, mit dem wir die aktuell in der In-Memory-Datenbank vorhandenen Daten ansehen und verändern können. Dazu führen Sie vor dem Start der Play-Applikation (mit *run*) in der Play-Konsole *h2-browser* aus. Im Webbrowser öffnet

Listing 3

```
...
# Tasks
POST    /tasks                  controllers.ToDos.add
DELETE  /tasks/:id/delete       controllers.ToDos.delete(id: Long)
...
```

Abb. 2: Anzeige der Daten in der In-Memory-DB

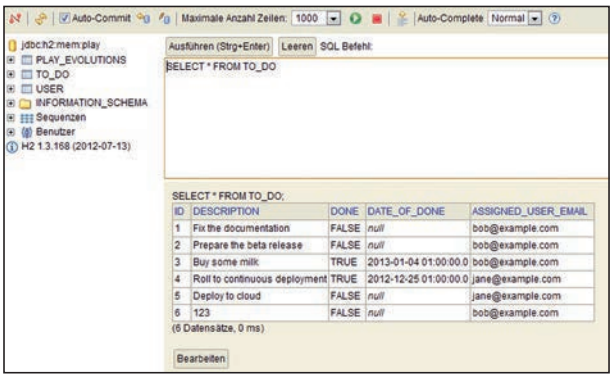
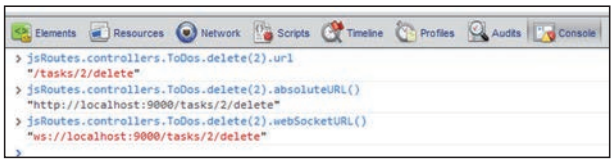


Abb. 3: Testen der JavaScript-Route in der Webkonsole



Listing 4

```
...
</ul>

<div class="separator"></div>

@helper.form(routes.ToDos.add) {
    <h2>Add TODO</h2>

    @if(flash.contains("error")) {
        <p class="error">
            @flash.get("error")
        </p>
    }

    <p>
        <input type="text" name="description" placeholder="description">
    </p>
    <p>
        <button type="submit">Add TODO</button>
    </p>
}
```

Listing 5

```
package controllers;

...
import play.Routes;
...

    public static Result javascriptRoutes() {
        response().setContentType("text/javascript");
        return ok(
            Routes.javascriptRouter("jsRoutes",
                controllers.routes.javascript.ToDos.delete()
            )
        );
    }
...
```

sich ein Anmeldebildschirm, in dem Sie die Konfiguration analog zu der DB-Konfiguration in der *application.conf* eingeben (**Abb. 1**). Wenn Sie nun unsere To-do-Applikation starten und im Browser aufrufen, können Sie anschließend im h2-Browser unsere DB-Daten anzeigen und bearbeiten (**Abb. 2**).

Löschen von To-dos

Natürlich müssen wir To-dos auch als erledigt markieren können. Sie werden in unserem Fall durch das Markieren gelöscht. Die bereits implementierte (*ToDo.java*)- und konfigurierte (*routes*)-Funktionalität zum Löschen wollen wir als REST-Schnittstelle verwenden. Der Nutzer soll auf der Webseite ein To-do mittels Checkbox als erledigt markieren können. Das Selektieren ruft dann per Ajax mit einem *HTTP DELETE* unsere *delete()*-Methode, welche den Status *play.mvc.Results.ok()* zurückgibt, also eine *success*-Meldung (HTTP-Response-Code 200). Die Nutzung von *HTTP DELETE* und des HTTP-Response-Codes entspricht dem *REST*-Paradigma. Auf diese Mel-

Listing 6

```
$ ->
    $('.delete-class').click ->
        $li = $(this).parents('li')
        todoId = $li.data('todo-id')
        jsRoutes.controllers.ToDos.delete(todoId).ajax
            success: ->
                $li.fadeOut
                    complete: ->
                        $li.remove()
            error: (err) ->
                $.error("Error: " + err)
```

Listing 7

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

<!DOCTYPE html>
@minified=@{ if (play.Play.isProd()) ".min" else "" }

<html>
    <head>
        <title>@title</title>
        <link rel="shortcut icon" type="image/png"
                href="@routes.Assets.at("images/favicon.png")"/>
        <script type="text/javascript"
                src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")">
        </script>
        <script type="text/javascript"
                src="@routes.Application.javascriptRoutes()">
        </script>
        <script type="text/javascript" src="@routes.Assets.at("javascripts/
                                                main" + minified + ".js")">
        </script>
    </head>
...
```

dung reagieren wir dann auf unserer Webseite mit dem Entfernen des selektierten To-dos. Dazu wiederum nutzen wir Ajax mit jQuery [1].

Da wir im JavaScript den URL zu unserer *delete()*-Methode nicht fest „verdrahten“ wollen, verwenden wir ein Reverse-Routing. Das gibt uns die Möglichkeit, die Konfiguration später in der *routes* zu ändern, ohne das JavaScript anpassen zu müssen. Erweitern Sie als Erstes den *Applikation*-Controller um eine Methode *javascriptRoutes()* (Listing 5).

Die Methode gibt die Reverse-Route unserer *delete()*-Methode zurück. In unserer *routes* müssen wir diese Methode natürlich noch definieren: *GET /assets/javascripts/routes controllers.Application.javascriptRoutes*. Letztlich müssen wir unserer *main.scala.html* im *<head>*-Tag noch die definierte Route hinzufügen: *<script type="text/javascript" src="@routes.Application.javascriptRoutes()"></script>*. Für die jQuery-Funktionalität muss jQuery noch eingebunden werden. In den *<head>*-Tag kommt noch folgender Eintrag: *<script type="text/javascript" src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")"></script>*. In einer Play-Anwendung ist jQuery Out of the Box verfügbar (Verzeichnis *public/javascripts*).

Im Browser lässt sich in der Konsole nun das Routing kontrollieren. Aktualisieren Sie unsere Applikation im Browser (F5), sodass */assets/javascripts/routes* geladen wird. In der Konsole können wir nun unser Routing testen (**Abb. 3**). Die Autoren verwenden Iron als Webbrowser (Google-Chrome-Klon). Die Konsole lässt sich in Iron mit F12 öffnen.

Jetzt wollen wir noch unsere Funktionalität zum Löschen selbst testen. Geben Sie dazu in der Konsole *jsRoutes.controllers.ToDos.delete(2).ajax({success:alert('deleted todo 2')})* ein. Dieser Aufruf löscht per Ajax-Call unser To-do mit der ID „2“ und gibt im Erfolgsfall eine Alert-Box mit entsprechendem Hinweis aus. Aktualisieren Sie die Webseite, das zweite To-do ist nun tatsächlich gelöscht (im h2-Browser können Sie ebenfalls sehen, dass der Datensatz gelöscht wurde).

Integration des REST-Services mit CoffeeScript und Ajax

Nun wollen wir unseren REST-Service aus unserer Applikation verwenden. Wie weiter oben erwähnt, verwenden wir dafür die Ajax-Funktionalität von jQuery. Den dafür erforderlichen JavaScript-Code wollen wir mit CoffeeScript [2] entwickeln (Sie können in Play auch direkt JavaScript-Code in den Dateien im Verzeichnis *public/javascripts* entwickeln). CoffeeScript ist eine Sprache, die zu JavaScript-Code kompiliert wird. Anlehnung gibt es an Ruby und Python. Der Sourcecode von CoffeeScript ist im Allgemeinen lesbarer. CoffeeScript-Programme sind außerdem meist kürzer als JavaScript-Programme, ohne dabei die Performance zu beeinflussen. Auch bietet CoffeeScript weitere Funktionalität wie bspw. Pattern Matching. Die Kompilierung zu JavaScript übernimmt das Play-Framework für uns. Dazu legen Sie im Package *app.assets.javascripts* eine neue Datei *main.coffee* an, falls diese noch nicht existiert (Listing 6). Unser Script ermittelt beim Selektieren einer Checkbox (CSS-Klasse *delete-class*) die ID (*data-todo-id*) des übergeordneten *<li>*-Elements und ruft mit dieser ID unseren REST-Service. Im Erfolgsfall wird das To-do langsam ausgeblendet und letztlich ganz aus der Liste entfernt.

Listing 8

```
...
  <h1>TODOS</h1>
  <ul>
    @for(todo <- todos) {
      <li data-todo-id="@todo.id">
        <h4>@todo.description
          <input class="delete-class"
            type="checkbox" value="DONE"
            title="mark '@todo.description' as Done"/>
        </h4>
      </li>
    }
  </ul>
...
```

Die CoffeeScript-Datei wird in die Dateien *main.js* und

Listing 9

```
package views;

import static java.util.concurrent.TimeUnit.SECONDS;
import static org.fest.assertions.Assertions.assertThat;
import static org.fluentlenium.core.filter.FilterConstructor.withText;
import static play.test.Helpers.HTMLUNIT;
import static play.test.Helpers.running;
import static play.test.Helpers.testServer;

import java.util.List;

import org.junit.Test;

import play.libs.F;
import play.libs.Yaml;
import play.test.TestBrowser;

import com.avaje.ebean.Ebean;

import controllers.routes;

public class TodoListTest {

  @Test
  public void anUserCanLoginSeeAddAndDeleteToDo() {
    running(testServer(3333), HTMLUNIT,
                                    new F.Callback<TestBrowser>() {
      public void invoke(TestBrowser browser) {
        Ebean.save((List) Yaml.load("test-data.yml"));

        // login page
        browser.goTo("http://localhost:3333" +
            routes.Credential.login().url());
        assertThat(browser.$("h1", withText("Sign in"))).hasSize(1);
        browser.fill("input").with("bob@example.com", "secret");
        browser.click("button");

        // Tasks page
        browser.await().atMost(3,
            SECONDS).until("h1").withText("TODOS").hasSize(1);
        assertThat(browser.find("li")).hasSize(2);

        // add a TODO
        browser.fill("input").with("new TODO");
        browser.click("button");
        assertThat(browser.$("li")).hasSize(3);

        // delete the first TODO
        browser.click(browser.findFirst(".delete-class"));
        browser.await().atMost(5,
            SECONDS).until(".delete-class").hasSize(2);
        assertThat(browser.$("li")).hasSize(2);
      }
    });
  }
}
```

main.min.js in den durch das Framework verwalteten Verzeichnissen übersetzt. Um unseren JavaScript-Code benutzen zu können, müssen wir diesen noch einbinden. Erweitern Sie dazu die *main.scala.html* (Listing 7).

Wir deklarieren ein Attribut *minified*, das je nach Laufzeitumgebung (Entwicklung oder Produktion) die zu verwendende JavaScript-Datei steuert. In Entwicklung steht uns so der formatierte Code für Debug-Zwecke zur Verfügung, in Produktion wird aus Performanzgründen die minimierte Version verwendet.

In unserer *index.scala.html* können wir nun unsere beschriebene Funktionalität umsetzen. Für jedes To-do erzeugen wir eine Checkbox mit der CSS-Klasse *delete-class*. Erweitern Sie bitte die View wie in Listing 8.

Wir können nun To-dos erzeugen, auflisten und löschen. Damit ist unsere Applikation funktional fertig.

Oberflächentests

Wenn wir jetzt ein paar To-dos angelegt und wieder gelöscht haben, ist unsere Anwendung durchaus (manuell) getestet. Das JavaScript ändert sich aber vielleicht in späteren Projektphasen. Ein manuelles Nachtesten clientseitigen Codes ist aufwändig und fehleranfällig. Das Play-Framework bietet auch hier eine sehr gute Unterstützung. Um unsere Applikation in einem Browser zu testen, können wir den Selenium WebDriver [3] verwenden. Play startet den WebDriver und bietet Zugriff mittels des FluentLenium-API [4]. FluentLenium ermöglicht auf einfache Art, Webseiten und die darauf enthaltenen Komponenten zu repräsentieren. Wir können dann mit diesen interagieren und auf erwartete Er-

Listing 10

```
// variables
@main-color: #333333;
@border-color: #DDDDDD;
@error-color: red;
@background-color: white;

// style
body {
  color: @main-color;
  font-family: "Helvetica";
  margin: 0;
  background-color: @background-color;
}
header {
  color: #fff - @main-color;
  background-color: #fff - @background-color;
  display: table;
  width: 100%;
  a { color: #fff - @main-color; }
  dt, dd { display: table-cell; padding: 0 30px; }
  dd { vertical-align: right; }
}
.container { width: 500px; margin-left: 20px; }
.error { color: @error-color; }
.separator { border-top: 1px solid @border-color; }
```

gebnisse prüfen. Erstellen Sie im Package *test.views* eine Testklasse *TodoListTest.java* (Listing 9).

Der Test verwendet unsere Testdaten (YAML). Er wird auf Port 3333 unserer Anwendung gestartet und dann der Reihe nach geprüft, ob eine Anmeldung möglich ist, die initiale Liste zwei Einträge beinhaltet, ein neues To-do angelegt und ein bestehendes gelöscht werden kann. Wir führen also einen vollständigen funktionalen Test unserer Applikation durch, darin eingeschlossen auch die JavaScript-Funktionalitäten.

Styling

Wir wollen unsere Applikation mit etwas CSS optisch verschönern. Wir können in Play eigene CSS-Dateien erzeugen lassen. Dazu schreiben wir LESS-Dateien [5], welche dann zu CSS übersetzt werden. LESS bietet die Möglichkeit, beispielsweise Variablen, Funktionen und Operationen zu definieren und Expression zu verwenden. Auch LESS wird zur Compile-Zeit durch Play geprüft und Syntaxfehler im Browser angezeigt.

Erstellen Sie nun eine Datei *main.less* im Verzeichnis *app.assets.stylesheets* (Listing 10). Vorher sollte die Datei *main.css* im Verzeichnis *public/stylesheets* gelöscht werden. Bauen Sie nun einen Fehler ein, entfernen also beispielsweise die letzte Klammer. Wenn Sie nun die Applikation aktualisieren, bekommen Sie einen Syntaxfehler angezeigt (**Abb. 4**).

Abb. 4: Syntaxfehler in der LESS-Datei

Beheben Sie den Fehler wieder. Um unsere CSS-Datei zu verwenden, müssen wir nun noch die Views *main.scala.html* und *login.scala.html* anpassen. In beiden Views fügen Sie im *<head>*-Tag die Zeile *<link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/main" + minified + ".css")">* hinzu. In der *login*-View müssen wir noch die Variable *minified* definieren: *@minified=@{ if (play.Play.isProd()) ".min" else "" }*. Wenn Sie jetzt unsere Applikation aktualisieren, sollte sie wie in **Abbildung 5** aussehen. Damit haben wir unserer To-do-Liste mit geringem Aufwand ein eigenes Aussehen verpasst.

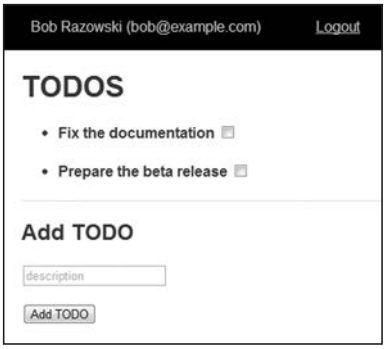


Abb. 5: Applikation mit CSS (LESS)

Produktion

Nachdem wir in wochenlanger mühevoller Teamarbeit eine voll ausgereifte Webapplikation erstellt haben, wollen wir diese final in Produktion bringen. Play unterstützt uns aktuell mit den Shell-Befehlen *play stage* und *play dist*. Für Unix-/Linux-Systeme erzeugt *play stage* im *target*-Verzeichnis ein *staged*-Verzeichnis und ein *start*-Skript. Mit dem Skript können Sie die Applikation in die Produktionsumgebung ausführen. Mit dem Befehl *play dist* wird eine ZIP-Datei erzeugt, die unsere Applikation beinhaltet. Auch hier ist ein *start*-Skript enthalten, das unsere Applikation startet. Erwähnenswert ist noch, dass auf dem Zielsystem kein Play-Framework installiert sein muss, da beide Versionen alle Laufzeitkomponenten mitbringen.

Das Betriebssystem Windows wird dabei nicht unterstützt. Der *stage*-Befehl wird unter Windows nicht korrekt ausgeführt und auch das erzeugte *start*-Skript ist für Unix/Linux. Um unsere Applikation unter Windows im Produktionsmodus zu starten, müssen wir auf dem Zielsystem Play installiert haben. Gestartet wird unsere Applikation mit dem Befehl *play start*. Im Gegensatz zu den oberen Versionen wird die Applikation interaktiv gestartet. Das heißt, dass wir diese erst im Browser aufrufen müssen, um den Startvorgang auszulösen.

Für die Zukunft ist ein Deployment als WAR geplant. Voraussetzung hierfür ist JSR 340 (Servlet 3.1), wegen Plays Asynchronität. Es gibt ein Plug-in [6], das auch jetzt bereits WAR-Files generiert. Offiziell unterstützt wird die Erzeugung eines WAR allerdings nicht. Vielmehr wird das Standard-Deployment empfohlen, das die Performanz der Netty Engine ausnutzt.

Play im realen Leben

Es gibt eine Reihe von Firmen, die das Play-Framework bereits einsetzen. Exemplarisch seien hier genannt: Guardian für seine mobile Website [7], Klout, Gilt, Sears, VMware, Coursera, Egraphs und LinkedIn [8].

An dieser Stelle würden sich die Autoren freuen, wenn Sie Spaß an und mit diesem Tutorial hatten. Wir hoffen, dass wir Ihnen etwas von den Ideen und Konzepten von Play vermitteln konnten und Sie eventuell Lust bekommen haben, selbst einmal Play einzusetzen. Erwähnen möchten wir an dieser Stelle noch, dass Sie unsere vollständige To-do-Applikation unter [10] herunterladen können.

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 eine Vielzahl von Webprojekten, unter anderem mit dem Play-Framework, umgesetzt.

Links & Literatur

[1] http://jquery.com/

[2] http://coffeescript.org/

[3] http://docs.seleniumhq.org/projects/webdriver/

[4] https://github.com/Fluentlenium/FluentLenium

[5] http://lesscss.org/

[6] https://github.com/dlecan/play2-war-plugin

[7] http://m.guardian.co.uk/

[8] http://engineering.linkedin.com/play/play-framework-linkedin

[9] https://github.com/playframework/Play20/wiki/Modules

[10] https://github.com/yanns/play2-todo-list]

Weitere Möglichkeiten mit Play

Es ist aufgrund der Integration von sbt in Play sehr einfach, Play in einen CI-Server zu integrieren. Build-Automatisierung und CI sollten in jedem größeren Projekt gängige Praxis sein. Play erfüllt diese Anforderungen.

Es gibt eine Reihe zusätzlicher Module für Play. Die Anzahl wächst kontinuierlich weiter, da die Community rund um Play sehr aktiv ist. Einen Überblick können Sie sich unter [9] verschaffen.

Da sich wohl sehr viele von uns in den letzten Jahren zum Teil sehr intensiv mit Spring beschäftigt haben und dieses Framework aktuell nicht missen wollen, bietet Play auch hier Möglichkeiten. Die Integration von Spring ist einfach. Ab der Play-Version 2.1 kann Spring sogar für das Bestimmen der *Controller*-Methoden verwendet werden (Dynamic Controller).

Play ist asynchron. Basis in Play ist das Actor-Modell. Damit ist eine bessere Skalierbarkeit einer Play-Applikation gegeben. So kann ein Client in Play einen HTTP Request asynchron verarbeiten und beantworten. Weiter gibt es beispielsweise auch einen asynchronen Treiber für MongoDB. Die Asynchronität ermöglicht auch eine einfache Programmierung, um Daten von einem Server zu einem Browser zu senden (Comet, Server-sent Events, WebSocket).

Das Stichwort Scala ist in unserem Tutorial auch schon hier und da aufgetaucht. Play bietet ein API sowohl für Java als auch für Scala. Wenn Sie Play installieren (entpacken), gibt es ein paar interessante Beispiele im *samples*-Verzeichnis. In diesem gibt es zwei weitere Verzeichnisse: *java* und *scala*. Alle Beispiele sind sowohl in Java, als auch in Scala vorhanden, sodass Sie hier als Java-Entwickler vielleicht einen guten Einstieg in die Scala-Welt bekommen.

Für den Fall, dass unser Projekt einmal komplexer werden sollte, können wir dieses in Play auch in Teilprojekte zerlegen.

Und letztlich spielt das Thema Cloud aktuell eine nicht ganz unwesentliche Rolle im Alltag eines Softwareentwicklers. Da freuen wir uns doch, dass viele Cloud-Provider nativ Play-Applikationen unterstützen (Heroku, CloudBees, Cloud Foundry, OpenShift, Jelastic und weitere).

Was kommt nach der Single-Page-App?

We do it ROCA-Style!

Welche Architektur wählen Sie, wenn Sie den Relaunch einer Ihrer Webanwendungen angehen wollen? Ältere Browser sowie mobile Clients müssen unterstützt werden. Die gesamte Funktionalität muss auch ohne den Einsatz von JavaScript gewährleistet bleiben. Einzelne funktionale Säulen der Anwendung müssen unabhängig voneinander betrieben und entwickelt werden können. Wir haben uns für ROCA-Style entschieden, um eine skalierbare, entkoppelte und den REST-Prinzipien folgende Webanwendung zu entwickeln. ROCA-Style [1] besteht aus einer Sammlung von Grundsätzen, die auf eine hohe Entkopplung von Server und Client abzielen. Unabhängig von Technologien, Werkzeugen und Programmiersprachen kann durch die Befolgung dieser Grundsätze ein Ökosystem von einander ergänzenden Server- und Clientanwendungen geschaffen werden.

von Jacob Fahrenkrug und André von Deetzen

Im Juni 2012 haben wir angefangen, eine neue Webanwendung zu entwickeln und mussten uns entscheiden, welchen Architekturansatz wir wählen. Aufgrund der breiten Zielgruppe der E-Post benötigen wir Unterstützung für viele verschiedene Browser in unterschiedlichen Versionen. Dabei soll die gesamte Funktionalität auch ohne Einsatz von JavaScript zur Verfügung stehen. Einzelne funktionale Säulen der Anwendung (**Abb. 1**) sollen unabhängig voneinander betrieben werden können. Jede Säule bietet Dienste an, die von verschiedenen Clients genutzt werden. Die Clients sind in ihrer Ausprägung sehr unterschiedlich und reichen vom OS-X-Desktop-Client bis zum Customer-Relationship-Management-System. Es werden nicht alle Clients von der hauseigenen IT entwickelt; Partner können eigenständig gegen die APIs ihre Anwendungen entwickeln. Eine weitere Herausforderung ist die Skalierbarkeit auf mehrere Entwicklerteams, die die einzelnen Dienste unabhängig voneinander entwickeln können. Die Dienste werden individuell releast und in der Produktionsumgebung in Betrieb genommen. Eine Webanwendung als Konsument dieser Dienste muss robust gegenüber Schnittstellenerweiterungen sein und tolerant mit Downtimes der anderen fachlichen Säulen umgehen können.

Unsere Alternativen für die Implementierung waren die klassische Webanwendung, basierend auf serverseitigen Sitzungsdaten, und eine Single-Page-App mit clientseitigem Rendering. Beide Ansätze genügten unseren Anforderungen nicht. Serverseitige Sitzungsdaten erhöhen die betriebliche Komplexität, und eine Single-Page-App ist schwer zu implementieren, wenn ältere Browser unterstützt und die volle Funktionalität ohne JavaScript gewährleistet werden soll. Unsere Wahl fiel deshalb auf ROCA-Style.

Einführung ROCA-Style

ROCA ist ein Satz von Empfehlungen, die, unabhängig von Webframeworks oder Programmiersprachen, aus Best Practices der Webarchitektur resultieren. Hinter ROCA stehen Till Schulte-Coerne, Stefan Tilkov, Robert Glaser, Phillip Ghadir und Josh Graham, die eine Sammlung von empfohlenen Konzepten auf einer Website [1] mit der Community teilen.

Die ROCA-Guidelines gliedern sich in mehrere server- und clientseitige Empfehlungen. Auf der Serverseite steht die strikte Einhaltung der REST-Prinzipien [2]. Dabei hat jede Ressource einen eindeutigen URI. Auf serverseitigen Zustand wird verzichtet, denn jeder Request soll in sich alle notwendigen Informationen zur Verarbeitung enthalten. Dazu gehören Aspekte wie z. B. Authentifizierung. Ebenfalls sollen die HTTP-Methoden entsprechend ihrer Definition unter [3] eingesetzt werden. Durch den strikten Einsatz kann auch ein URL jederzeit kopiert, weitergegeben, ein Lesezeichen gesetzt oder die Ressource jederzeit eindeutig angefragt werden. Dadurch, dass die gesamte Geschäftslogik auf dem Server gehalten und nicht teilweise in JavaScript ausgelagert wird, kann eine Webanwendung auch mit der Kommandozeile (Wget, cURL) bedient werden. Andere Repräsentationsformate wie z. B. JSON [4] oder XML [5] sind dabei sogar erwünscht. Die Authentifizierung am Server sollte idealerweise per HTTP Basic Authentication über SSL oder Clientzertifikate geschehen. In der Praxis mit Webanwendungen ist dies jedoch umständlich, da Brow-

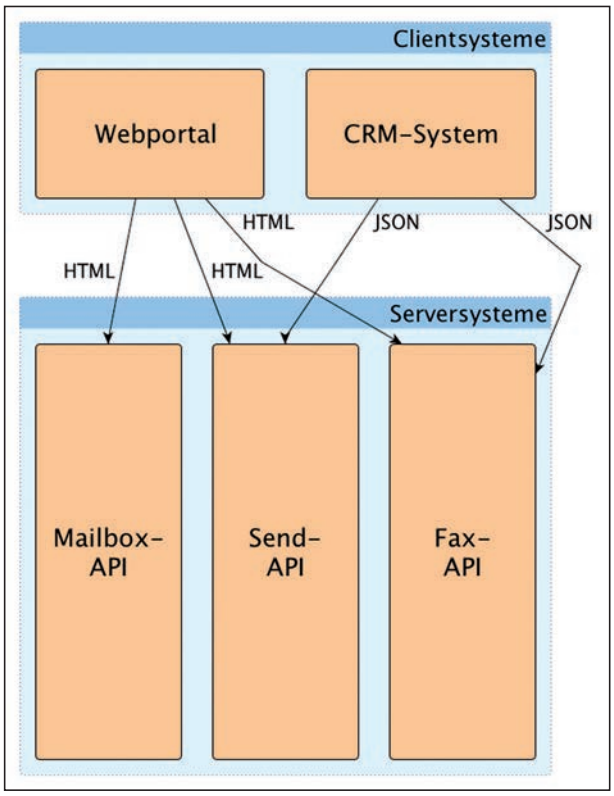


Abb. 1: Fachliche Säulen einer Webanwendung

ser keine Standardfunktionalität für eine Abmeldung vorsehen oder ein Styling des Anmeldedialogs zulassen. Daher sind auch formularbasierte Anmeldedialoge in Verbindung mit Cookies erlaubt. ROCA sieht nur zwei Arten der Benutzung von Cookies vor: für die Authentifizierung und das Tracking von Nutzern.

Auf der Clientseite gibt ROCA die Empfehlung, semantisch korrektes HTML auszuliefern. Für das Styling einer Seite soll das Prinzip von Progressive Enhancement [6] verwendet werden. Das Gleiche gilt für JavaScript, um die Webanwendungen auch für ältere Browser nutzbar zu halten. CSS- und JavaScript-Code sollen unter statischen Domains erreichbar sein. Das erleichtert das Caching durch ein Content Delivery Network (CDN) oder das parallele Laden von statischen Ressourcen im Browser. Dadurch können eine schnellere Seitenladezeit und eine angenehmere Wahrnehmung der User Experience erreicht werden. Ebenfalls zur User Experience gehören funktionierende Browser-Controls (Vor, Zurück, Aktualisieren). Da nur eindeutige URIs und die HTTP-Methoden gemäß ihrer Definition eingesetzt werden, kann die Seite jederzeit gefahrlos neu geladen oder mit den Vor- und Zurück-Buttons navigiert werden. Wenn Ressourcen durch einen JavaScript Call geladen werden, kann über das von HTML5 bereitgestellte History-API dieser Grundsatz auch mit JavaScript-Funktionalität abgedeckt werden.

HATEOAS

HATEOAS steht für „Hypermedia as the Engine of Application State“. Es handelt sich um eine Einschränkung, die die REST-Architektur von den meisten anderen Netzwerkarchitekturen unterscheidet. Dem HATEOAS-Prinzip folgend, interagieren Clients mit einer Webanwendung ausschließlich über Hyperlinks, die dynamisch von der Webanwendung zur Verfügung gestellt werden. Ein REST-Client braucht keinerlei Wissen über die Implementierung des Service. Alles, was notwendig ist, ist ein generelles Verständnis von Hypermedia [9].

ROCA-Style, eine Implementierung

Im ersten Abschnitt haben wir unsere Beweggründe für eine Anwendung im ROCA-Style dargelegt. In diesem Abschnitt möchten wir näher darauf eingehen, mit welchen Technologien und Architekturgrundsätzen wir den ROCA-Style bei uns konkret implementieren. Dabei haben wir die folgenden drei Schwerpunkte identifiziert.

Zustandslosigkeit

Unsere Webanwendung basiert auf einem Java-Stack. Von uns genutzte serverseitige Frameworks sind Spring MVC und Spring Security. Als View-Technologie kommen JavaServer Pages (JSP) zum Einsatz. Ausgeliefert wird die Anwendung als WAR, um in einem Tomcat [7] betrieben zu werden. Durch den Grundsatz der eindeutigen URIs versetzen wir den Nutzer in die Lage, jederzeit Lesezeichen in unserer Anwendung zu setzen oder z. B. einen Link für einen Kontakt per E-Mail an einen Kollegen zu versenden. Die Authentifizierung überprüft dann, ob der Empfänger berechtigt ist, diesen Kontakt einzusehen. Durch die verschlüsselte Ablage von Authentifizierungsinformationen im Cookie können wir ohne serverseitige Sitzungsdaten auskommen, müssen jedoch jeden Request explizit auf Autorisierung prüfen. Durch den Verzicht auf eine Session unterstützt das System bei Bedarf eine dynamische horizontale Skalierung. Auf diese Weise können Lastspitzen abgefangen werden. Auch die Bereitstellung neuer Versionen unter Einsatz eines BlueGreenDeployments [8] ist so möglich.

Statisches CSS und Unobtrusive JavaScript

Ein weiterer Grundsatz von ROCA ist, dass CSS und JavaScript als statische Ressourcen ausgeliefert werden. Mobile Clients sowie Clients mit einer langsamen Internetverbindung können davon durch effizientes Caching profitieren. Obwohl davon ausgegangen werden kann, dass die meisten Browser CSS und JavaScript unterstützen, sind die Unterschiede in den Implementierungen groß. Auch ist es möglich, dass der Nutzer JavaScript deaktiviert hat. ROCA bietet uns mit „Unobtrusive JavaScript“ die Möglichkeit, unsere Anwendung je nach Fähigkeiten der Browser ergonomischer darzustellen. Konkret bedeutet das, dass wir, sobald JavaScript aktiviert ist, mit Ajax-Calls und DOM-Manipulation arbeiten. Dadurch können bspw. Klickpfade verkürzt und mehrere Requests im Hintergrund parallel abgearbeitet werden, um dem Nutzer eine ansprechende und

ergonomische Oberfläche zur Verfügung zu stellen. Für den Fall, dass JavaScript deaktiviert ist, wird dem Benutzer die sonst mit JavaScript implementierte Funktionalität in Form von Links angezeigt. Folgt der Nutzer dem Link, gelangt er zu einer separaten Seite, die die gewünschten Informationen und Funktionalitäten zur Verfügung stellt. Wir konnten unser Ziel erreichen, dem Nutzer trotz deaktivierter JavaScripts den vollständigen Funktionsumfang mit Einbußen in der User Experience zur Verfügung zu stellen. Man kann den Einsatz von Unobtrusive JavaScript folgendermaßen umsetzen:

```
<input type="text" name="email" id="email" />
```

Das HTML repräsentiert ein Eingabefeld für eine E-Mail-Adresse. Durch die Vergabe der ID *email* kann in dem folgenden JavaScript einfach die *onchange*-Methode ergänzt werden:

```
window.onload = function() {
    document.getElementById('email').onchange = verifyEmailAdress;
};
```

Der klassische Einsatz von JavaScript sieht im Vergleich dazu so aus:

```
<input type="text" name="email" onchange="verifyEmailAdress();" />
```

Komposition von Systemen

Die Webanwendung konsumiert verschiedene andere Dienste, um unseren Kunden ein durchgängiges Nutzererlebnis in unserem Produkt zu gewährleisten. Um einen hohen Konfigurationsaufwand zu vermeiden, verwenden wir für alle unsere Dienste einen Hypermediaansatz. Wenn unsere Webanwendung eine andere fachliche Säule integriert und die Links zu diesem System generieren möchte, wird als Erstes das Servicedokument des Diensts angefragt, das alle weiteren Informationen zur Nutzung enthält. Das macht es besonders leicht, die Anfragen auf neu bereitgestellte Systeme umzulenken, ohne dass der Nutzer abgemeldet werden muss oder gar eine Wartungsseite zu sehen bekommt. Die Servicedokumente sind alle unter einem statischen URI erreichbar und exponieren alle Funktionalitäten zur Nutzung des Diensts. Dabei folgen diese dem HATEOAS-Prinzip (siehe Kasten), und der Nutzer kann entweder durch JavaScript oder durch das Verfolgen von Links durch die Anwendungen navigieren. Zurzeit verbindet die Webanwendung ein *Mailbox*-API, das auch von unseren mobilen Clients benutzt wird, und ein *Send*-API, das von externen Partnern benutzt wird. Wir planen, weitere APIs zu entwickeln, wie beispielsweise ein Adressbuch oder einen Faxversand. Diese Funktionalitäten wollen wir dann auch für unsere Nutzer in der Webanwendung zur Verfügung stellen.

Eine Webanwendung aus verschiedenen Diensten zu komponieren, die über Links miteinander lose gekoppelt sind, bietet eine Reihe von Vorteilen. Die Nutzung von Links führt zu einem stabilen und fehlertoleranten System. Denn wenn ein Dienst vorübergehend nicht erreichbar ist, führt der Link vielleicht ins Leere. Alle anderen Funktionen können jedoch trotzdem weiter genutzt werden. Ebenfalls wird der Nutzer nicht ausgeloggt, sondern kann einfach über den Back-Button seines Browsers in den Ursprungszustand zurückgelangen. Die angezeigten Ressourcen sind genauso Cachebar wie die statischen Ressourcen (CSS, JavaScript). Durch den Einsatz von Cache-Headern haben wir ein sehr performantes System geschaffen.

Ausblick

Mittelfristig wollen wir unser gesamtes System in kleine Dienste zerlegen. Dabei ist jeder Dienst einzeln betreibbar und zustandslos. Alle Dienste werden von uns selbst betrieben und durch eine Continous-Delivery-Pipeline, wenn nötig mehrfach täglich, in neuen Versionen in Betrieb genommen. Eine Integration mit Diensten, die wir nicht betreiben, ist zwar möglich, von uns jedoch im Moment nicht gewollt. Dieses Vorgehen erlaubt uns sehr viel Freiheit und Flexibilität, um schnell auf Kundenwünsche sowie den Markt zu reagieren.

Durch die Nutzung von HTTP und REST ist die vorstellbare Anzahl an Clientimplementierungen beliebig. Jeder Dienst liefert, je nach angefragtem „Application Type“, JSON, HTML oder XML und kann in unsere Webanwendung integriert werden. Unsere REST-APIs sind stabil und werden abwärtskompatibel weiterentwickelt, um bestehende Clientimplementierung nicht zu stören. Für die Absicherung unserer Systeme setzen wir OAuth 2.0 [10] ein. So können wir sicherstellen, dass zu jedem Zeitpunkt überprüft werden kann, ob die Aktion in unserem System ausgeführt wird und die Person oder das Clientsystem dazu berechtigt ist. Die Ablage der Authentifizierungsinformationen erfordert keine mehrfache Anmeldung zwischen den Systemen. Ein einheitliches Design stellt dabei das Nutzererlebnis sicher, während sich unsere Nutzer nahtlos über Systemgrenzen hinweg bewegen, ohne davon Kenntnis zu nehmen.

Fazit

Eine Kernkompetenz erfolgreicher Internetunternehmen ist die Erprobung von Ideen und das Eliminieren von Fehlversuchen. Wir sind der Überzeugung, mit ROCA die richtige Designentscheidung getroffen zu haben, um einem modernen Internetunternehmen und seinen Anforderungen gerecht zu werden. Durch ROCA haben wir eine lose Kopplung zwischen unseren Anwendungen geschaffen und können das System bei Bedarf erweitern oder verkleinern. Die aus unserer Sicht wichtigsten Merkmale sind sowohl die Zustandslosigkeit auf der Serverseite und der Einsatz von „Unobtrusive JavaScript“ auf dem Client, als auch die Umsetzung von REST-Prinzipien für unsere Anwendung. Viele Vorteile des Webs können wir nutzen, um unsere Webanwendung stabil, fehlertolerant, performant und für ein breites Spektrum von Nutzern und Clientimplementierungen zur Verfügung zu stellen. Eine solche Anwendung zu entwerfen bedeutet aber auch einen nicht zu unterschätzenden höheren Aufwand an Arbeit und Disziplin bei der Implementierung.

Jacob Fahrenkrug (jacob.fahrenkrug@epost-dev.de) ist Seniorarchitekt bei der Deutsche Post E-Post Development GmbH. Als Teil eines Scrum-Teams treibt er die Entwicklung des E-Post-Ökosystems und dessen Architektur voran.

André von Deetzen (andre.von.deetzen@epost-dev.de) ist Principal Engineer bei der Deutsche Post E-POST Development GmbH. Seine Schwerpunkte sind die agile Softwarentwicklung, Architektur und die Delivery-Pipeline in die Produktion, um Produkte schnell und in guter Qualität ausliefern zu können.

Links & Literatur

[1] http://roca-style.org/

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

[3] http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

[4] http://de.wikipedia.org/wiki/JSON

[5] http://de.wikipedia.org/wiki/XML

[6] http://www.hesketh.com/thought-leadership/our-publications/progressive-enhancement-and-future-web-design

[7] http://tomcat.apache.org/

[8] http://martinfowler.com/bliki/BlueGreenDeployment.html

[9] http://en.wikipedia.org/wiki/HATEOAS

[10] http://oauth.net/2/

[11] http://de.wikipedia.org/wiki/HTTP

[12] http://de.wikipedia.org/wiki/Document_Object_Model

[13] http://www.springsource.org

[14] http://www.innoq.com/blog/st/2012/03/announcing-roca

[15] http://sigs.de/publications/os/2012/Architekturen/hoppe_schulte-coerner_tilkov_OS_Architekturen_12_hgz8.pdf

[16] http://www.heise.de/developer/artikel/Episode-37-ROCA-Resource-oriented-Client-Architecture-1742790.html

[17] Interview mit Stefan Tilkov über ROCA: http://www.youtube.com/watch?v=1bXei7giYCY

[18] http://www.infoq.com/interviews/tilkov-rest-hypermedia

[19] http://alistapart.com/article/understandingprogressiveenhancement

[20] http://en.wikipedia.org/wiki/Unobtrusive_JavaScript

Anzeige

Aufbau eines Frameworks zur Testautomatisierung von Web-Frontends im E-Commerce-Bereich

Eine kleine Dosis Selen

Oft sind die IT-Landschaften, die sich hinter den E-Commerce-Websites großer Anbieter verbergen, von hoher Komplexität geprägt. Sie bestehen neben dem Shop-Frontend aus zahlreichen Spezialanwendungen, wie z. B. CMS, PIM, Artikelsuche, Recommendation Engine, Warenwirtschaft, Lagerhaltung, Zahlungsverkehr etc. Das Testen der einzelnen Komponenten sowie übergreifender Geschäftsprozesse stellt bei jedem Deployment eine große Herausforderung dar. Der folgende Beitrag beschreibt den Einsatz eines Frameworks zur Durchführung von automatisierten Frontend-Tests auf der Basis von Selenium.

von Gregor Schrägle

Was sind die schlimmsten Albträume der Betriebsverantwortlichen von E-Commerce-Websites? Wohl die, in denen die kaufwilligen Kunden eines Unternehmens in dessen Internetshop nicht die Waren des Unternehmens – auf ansprechende Weise präsentiert – zu Gesicht bekommen, sondern lediglich eine einfache Wartungsseite mit dem vagen Hinweis auf ein technisches Problem und der Bitte, den Einkauf doch in Kürze noch einmal zu versuchen.

Das sind die Momente, in denen die verantwortlichen IT-Abteilungen aktiv werden und alle Mitarbeiter mit Hochdruck an der Beseitigung der Probleme arbeiten müssen. Das Ziel ist klar: Der Shop muss so schnell wie möglich wieder in Betrieb gebracht werden. Es dürfen keine Kunden und Bestellungen verloren gehen! Nirgendwo trifft die alte Geschäftsregel mehr zu als beim E-Commerce: *Time is money!*

Die verantwortlichen IT-Leiter unternehmen heute bereits im Vorfeld eine Reihe von Maßnahmen, um die Anzahl und Dauer der Downtimes ihrer E-Commerce-Plattformen so gering wie möglich zu halten – egal, ob es sich um vorgesehene Downtimes handelt, wie z. B. für regelmäßige Wartungsmaßnahmen, oder um ungeplante, die durch schwere Fehler verursacht werden. Die wohl wichtigste Präventivmaßnahme ist die Durchführung von umfassenden Tests der einzelnen Softwareanwendungen und deren Zusammenspiel innerhalb der IT-Landschaft. Je mehr verschiedene Anwendungen dabei im Spiel sind, desto größer werden die Komplexität, aussagekräftige Testszenarien zu definieren, und der Aufwand, diese Tests durchzuführen.

Im Folgenden werde ich den Aufbau eines Frameworks vorstellen, das es ermöglicht, die webbasierten Frontends der Anwendungen einer E-Commerce-Landschaft automatisiert zu testen. Dieses Framework wurde für einen großen internationalen Kunden im Bereich Handel realisiert und für seine E-Commerce-Anwendungslandschaft eingesetzt, die aus ca. 25 Applikationen besteht.

Die Ziele

Zunächst wurden in Abstimmung mit den verschiedenen Teams für Entwicklung, Integrationstest, Rollout und Betrieb die wichtigsten Ziele vereinbart, die durch den Einsatz des Testautomatisierungsframeworks erreicht werden sollten:

- Die Reduzierung des Aufwands zur Testdefinition
- Die Reduzierung des manuellen Aufwands bei der Testdurchführung
- Die Verkürzung der benötigten Zeit zur Durchführung aller Tests
- Nutzbarkeit sowohl bei erstmaligen Deployments der Landschaft als auch für Regressionstests

Daraus wurden für das Testframework einige Minimalanforderungen abgeleitet:

- Eine einheitliche Definition und Durchführung der Tests
- Eine möglichst große Testabdeckung (im Bezug auf die 25 Einzelanwendungen)
- Die automatisierte Durchführbarkeit der Tests
- Eine einfache Wiederholbarkeit von Tests

- Bereitstellung einer automatischen Auswertung der Testergebnisse und integrierte Eskalationsmöglichkeiten

Nach der Zieldefinition wurden einige Werkzeuge zur Durchführung von GUI-Tests [1] verglichen und das Selenium-Framework [2] für die Umsetzung ausgewählt. Die entscheidenden Gründe für die Wahl von Selenium waren dessen Verfügbarkeit als Open-Source-Software, die breite Abdeckung von Web-Frontend-Technologien und seine aktive Benutzergemeinde.

Die Basiskomponenten des Testframeworks

Das Selenium-Framework besteht im Wesentlichen aus drei Komponenten (Kasten: „Die Selenium Suite“): Als erste kommt das Browser-Plug-in *Selenium IDE* zum Einsatz, mit dem Benutzerinteraktionen auf einer Website – so genannte „Klick-Strecken“ – aufgezeichnet und später wieder abgespielt werden können. Anschließend kann mit einem Generator aus den aufgezeichneten Klick-Strecken für verschiedene Programmiersprachen Quelltext generiert werden. Wir haben eine Konvertierung in Java vorgenommen. Diese Java-Quelltexte wurden dann als JUnit-Tests mit dem *Selenium WebDriver* ausgeführt. Eine Ergänzung stellt das *Selenium Grid* dar, mit dem das Abarbeiten von Testfällen parallelisiert werden kann, indem man sie zur Ausführung auf verschiedene Knoten im Grid verteilt.

Die Umsetzung des Testframeworks

Die Nutzung eines Testautomatisierungsframeworks beginnt immer mit dem erstmaligen Aufzeichnen von Testfällen. Dazu werden für jede zu testende Webanwendung mit der *Selenium IDE* einzelne Klick-Strecken aufgezeichnet, in der Regel eine Klick-Strecke pro Testfall.

Ein typischer Testfall „Einkauf“ für ein Shop-Frontend könnte so aussehen: Der Benutzer führt im Webshop eine Produktsuche durch, browst durch den Kategoriebaum, wählt mehrere Produkte zum Vergleich aus, legt zwei Produkte in den Warenkorb und führt anschließend den Bestellprozess durch, inklusive der Eingabe synthetischer Zahlungsdaten.

Je nachdem, um welche Applikation es sich handelt, gibt es natürlich sehr unterschiedliche Testszenarien, die im Vorfeld funktional genau definiert und zusammen mit den zu erwartenden Ergebnissen spezifiziert werden müssen.

Die technische Basis für die Aufzeichnung von Benutzerinteraktionen mit dem Browser sind dabei immer zwei Dinge: erstens die Identifizierung von Elementen des GUI bzw. die Lokalisierung der Elemente innerhalb der Elementhierarchie einer Seite. Und zweitens die Ak-

tion, die der Benutzer mit dem Element ausführt, beispielsweise einen Klick, eine Auswahl, eine Texteingabe.

Für die Identifizierung bzw. Lokalisierung von Elementen auf einer Webseite benutzt Selenium die so genannten *Element Locators*. Dabei handelt es sich um Konfigurationseinstellungen in der *Selenium IDE*, die spezifizieren, auf welche Weise die Identifizierung bzw. Lokalisierung von GUI-Elementen durchgeführt werden soll. Im Wesentlichen stehen dazu folgende *Locators* zur Verfügung:

- **ID:** Identifikation eines GUI-Elements anhand seiner ID
- **NAME:** Identifikation anhand des Namens
- **DOM:** Lokalisierung eines GUI-Elements innerhalb des HTML-Objektbaums, z. B. *dom=document.forms['myForm'].myDropdown*
- **XPATH:** Lokalisierung eines Elements mithilfe eines XPath-Ausdrucks, z. B. *xpath=//table[@id='table1']//tr[4]/td[2]*
- **LINK:** Spezifikation des Ankerelements eines Links durch den Linktext, z. B. *link="our specials"*
- **CSS:** Lokalisierung eines GUI-Elements durch CSS-Selektoren, z. B. *css=a[href="#id3"]*

Die Selenium Suite

Selenium ist ein Open-Source-Framework zur automatisierten Steuerung von Webbrowsern. Durch seine vielfältigen Funktionen eignet es sich sehr gut zur Umsetzung eines Testframeworks für Webanwendungen. Die bereitgestellten Funktionalitäten sind flexibel und bieten umfangreiche Möglichkeiten, Elemente innerhalb der grafischen Benutzeroberflächen von Webanwendungen zu lokalisieren sowie die erwarteten Ergebnisse für Testfälle mit dem echten Verhalten der Anwendung zu vergleichen. Ein wichtiges Feature von Selenium ist, dass einmal aufgezeichnete Testfälle mit einer Reihe von Webbrowsern abgespielt werden können: Firefox, Internet Explorer, Chrome etc.

Neben den Werkzeugen zum Aufzeichnen und Abspielen von Browserinteraktionen umfasst Selenium auch eine domänenspezifische Sprache, *Selenese,* die zur Spezifikation von Tests in verschiedenen Programmiersprachen benutzt werden kann, z. B. in Java, C#, Groovy, Perl, PHP, Python und Ruby.

Die wichtigsten Komponenten von Selenium

Die Selenium Suite besteht aus folgenden Hauptkomponenten:

- *Selenium IDE* ist ein Firefox-Plug-in zum Aufzeichnen und Abspielen von Browserinteraktionen (Testfällen). Sie kann auch benutzt werden, um aus aufgezeichneten Testfällen Quellcode zu generieren, der mit *Selenium WebDriver* abgespielt werden kann.
- *Selenium WebDriver* ist eine Komponente zur Steuerung von Webbrowsern über deren native Schnittstelle. Sie kann entweder lokal auf der eigenen Maschine oder auch remote genutzt werden.
- *Selenium Grid* stellt eine Ergänzung zu *Selenium WebDriver* dar, um die Ausführung von Testfällen parallel auf mehrere Server verteilen zu können. Das ist besonders für umfangreiche Testsuiten interessant bzw. zum parallelen Testen mit verschiedenen Browsern.

Darüber hinaus besteht die Möglichkeit, eigene Implementierungen für *Element Locators* zu erstellen. Je nachdem, welche technische Implementierung einer Website zugrunde liegt, sind die einen oder anderen *Locators* geeigneter. Die Festlegung der *Locator*-Präferenzen und ihrer Reihenfolge sollte auf jeden Fall frühzeitig erfolgen, bevor mit dem Aufzeichnen der einzelnen Klick-Strecken für die verschiedenen Testfälle begonnen wird. Sonst kann es vorkommen, dass die verwendeten *Locators* immer wieder einmal angepasst oder ausgetauscht werden müssen.

Werden keine Präferenzen angegeben, verwendet Selenium standardmäßig die drei *Element Locators* DOM, XPATH und ID.

Nach dem Aufzeichnen und Abspeichern einer Klick-Strecke kann diese über die *Selenium IDE* jederzeit wieder abgespielt werden. Dabei laufen dann die einzelnen Benutzerinteraktionen mit der Website wie von Geisterhand gesteuert in einem eigenen Browserfenster ab. Zusätzlich können Klick-Strecken aber auch als Quelltext für verschiedene Programmiersprachen exportiert werden. In unserem Fall wurde aus der *Selenium IDE* heraus pro Klick-Strecke ein JUnit-Testfall erzeugt. Für die Codegenerierung gibt es dabei eine Reihe von Konfigurationsparametern, von denen wir nur einige einfache benutzt haben, z. B. zur Festlegung von Namenskonventionen für die zu generierenden Java-Klassen und -Pakete. Für eine erste Strukturierung der ganzen Testfälle bietet sich an, die Klassennamen der JUnit-Tests aus den Namen der aufgezeichneten Klick-Strecken und die Paketnamen anhand der Namen der getesteten Webanwendungen erzeugen zu lassen. Darüber hinaus bestehen aber noch zusätzliche weitreichende Möglichkeiten zur Anpassung und Erweiterung der Codegenerierung.

Zur Ablaufsteuerung der JUnit-Testfälle wurde ein Ant-Skript [3] verwendet. Dieses Skript definierte dedizierte Ant *Targets*, die sich an den oben erwähnten Namenskonventionen orientierten und über die sowohl einzelne Testfälle, Gruppen von Testfällen, als auch alle Testfälle sequenziell ausgeführt werden konnten. Innerhalb der Targets wurde auf die Ant *Tasks <JUnit>* [4] und *<JUnitReport>* [5] zurückgegriffen. Diese spezifizieren einerseits die Ausführungsparameter der einzelnen JUnit-Testfälle, andererseits die Struktur der Ergebnisdarstellung. Die *JUnitReports* wurden nach jedem Testlauf als HTML-Dateien abgespeichert und lieferten auf oberster Ebene eine Zusammenfassung der Ergebnisse (Anzahl ausgeführter Tests, erfolgreiche Testfälle, fehlgeschlagene Testfälle, Laufzeit der Tests etc.), auf der zweiten Ebene eine Zusammenfassung der Testfälle pro Webanwendung und schließlich auf unterster Ebene eine detaillierte Ansicht jedes einzelnen Testfalls inklusive detaillierter technischer Fehlermeldungen bei fehlgeschlagenen Testfällen.

Für die Ausführung der JUnit-Testfälle mit dem Ant-Skript wurde ein dedizierter Testserver bereitgestellt. Auf ihm wurden unterschiedliche Testläufe so eingeplant, dass bei jedem neuen Deployment der E-Com-

merce-Umgebung der komplette Testumfang ausgeführt wurde und während des normalen Betriebs zusätzliche Testläufe regelmäßig die Verfügbarkeit der einzelnen Anwendungen überprüften.

Zusammenfassung der Ergebnisse – Fazit

Die Selenium Suite stellt umfangreiche Möglichkeiten zur automatisierten Browsersteuerung zur Verfügung. Mithilfe der Selenium-Hauptkomponenten *Selenium IDE*, *Selenium WebDriver* und *Selenium Grid* lässt sich sehr gut ein Framework zur Testautomatisierung von Webanwendungen aufbauen. Sowohl das Erlernen der Selenium-Konzepte als auch die Handhabung der einzelnen Werkzeuge zum Aufzeichnen und Abspielen von Testfällen oder zur Codegenerierung fallen einem versierten Benutzer erfreulich leicht.

Neben der Basistechnologie zur Browsersteuerung gibt es allerdings noch ein paar wichtige Punkte, die für die vollständige und praxistaugliche Umsetzung eines Frameworks zur Testautomatisierung notwendig sind und von Selenium nicht mitgeliefert werden: Dazu zählen einerseits Vorgaben zur logischen Strukturierung der Testfälle während der Testentwicklung sowie Komponenten zum Scheduling, Reporting und zur Steuerung während der Testdurchführung. Doch das sind eher Kleinigkeiten, um die das Selenium-Framework unter Zuhilfenahme von JUnit und Ant schnell ergänzt werden kann.

Darüber hinaus fehlen aber noch zwei wesentliche Bestandteile, die nicht ohne Weiteres hinzugefügt werden können: Zum einen ist das eine Datenbank zum Parametrieren der ganzen Testautomatisierung, angefangen bei Umgebungsparametern bis hin zur Hinterlegung von fachlichen Daten für erwartete Ergebnisse von Testfällen. Zum anderen fehlt ein durchgängiger Mechanismus, der es auf einfache Weise erlaubt, fachliche Fehler zu definieren und bestimmten Ereignissen während des Ablaufs eines Testfalls zuzuordnen. Das wäre sehr wichtig, um während eines Testlaufs neben technischen Fehlern, welche die Browsersteuerung von Selenium wirft – z. B. ein angegebenes Element kann auf einer Website nicht gefunden werden –, auch fachliche Fehler erkennen zu können, etwa, ob eine angezeigte Bestellnummer sinnvoll ist.

Als Hauptaufwandstreiber für die Umsetzung des Testframeworks haben sich einerseits die technische Bereitstellung der oben beschriebenen Zusatzkomponenten herausgestellt und andererseits das erstmalige Anlegen der ganzen Testfälle. Dabei entfällt der übliche Aufwand auf die ursprüngliche fachliche Spezifikation von Testfällen; zusätzlicher Aufwand aber entsteht für das technische Aufzeichnen der Tests mit der *Selenium IDE*. Aus organisatorischer Sicht hat es sich als sehr sinnvoll erwiesen, die Aufzeichnung der Testfälle mit der *Selenium IDE* jeweils von einem Anwendungsexperten (Rolle: „fachlicher Analyst“) zusammen mit einem IT-Spezialisten durchführen zu lassen, der Selenium und das Testframework genau kennt (Rolle: „technischer Experte“).

Ein wichtiger Aspekt kommt noch hinzu: Nach dem ersten Anlegen von Testfällen muss ein gewisser Aufwand für Wartungsarbeiten einkalkuliert werden. Die getesteten Webseiten können sich sehr oft inhaltlich, manchmal aber auch strukturell ändern. Das bedeutet, unter Umständen müssen die aufgezeichneten Testfälle den Änderungen angepasst werden. Hier wiederum kann sich eine am Anfang gut durchdachte Auswahl der *Element Locators* positiv auswirken.

Zusammenfassend lässt sich sagen, dass sich die kluge Umsetzung eines Frameworks zur Testautomatisierung auf jeden Fall lohnt, sowohl im Hinblick auf die Einsparung manueller Testaufwände als auch durch die deutliche Verkürzung der Ausführungszeiten für alle Tests. Gerade im E-Commerce-Bereich, wo oft eine große Anzahl von Anwendungen mit webbasierten Benutzeroberflächen innerhalb der IT-Landschaft existiert, wirken sich diese Faktoren sehr positiv aus. Und noch ein kleiner Tipp zum Schluss: Sowohl für das Aufzeichnen von Testfällen mit der *Selenium IDE* als auch bei der Fehlersuche erweisen sich *Firebug* [6] und *XPath Checker* [7] als sehr hilfreiche Werkzeuge!

Gregor Schrägle ist freiberuflicher IT-Architekt und Berater für Enterprise-Integration. Er unterstützt Unternehmen verschiedenster Branchen bei der Technologieauswahl und der Umsetzung von IT-Transformationsprojekten. Seine Spezialgebiete umfassen Integrationsarchitekturen, SOA, EAI, Datenintegration, BPM und CEP. Mehr finden Sie auf www.schraegle-consulting.de.

Links & Literatur

[1] Übersicht wichtiger GUI-Testframeworks: http://en.wikipedia.org/wiki/List_of_GUI_testing_tools

[2] Selenium-Website: www.seleniumhq.org

[3] Apache Ant: http://ant.apache.org/

[4] Ant JUnit Task: http://ant.apache.org/manual/Tasks/junit.html

[5] Ant JUnitReport Task: http://ant.apache.org/manual/Tasks/junitreport.html

[6] Firebug: https://getfirebug.com/

[7] XPath Checker: https://addons.mozilla.org/en-US/firefox/addon/xpath-checker/

[8] Selenium-Onlinedokumentation: http://docs.seleniumhq.org/docs/

[9] Übersicht zum Thema Testautomatisierung: http://en.wikipedia.org/wiki/Test_automation

[10] Martin Fowler, Continuous Integration: http://www.martinfowler.com/articles/continuousIntegration.html

[11] Martin Fowler, Test Pyramid: http://martinfowler.com/bliki/TestPyramid.html

[12] JUnit: http://junit.sourceforge.net/

Anzeige

Anzeige

JavaFX erfreut sich zunehmender Beliebtheit

Vollgas nach holprigem Start

Einige UI-Technologien der letzten Jahre sind gekommen – und wieder gegangen. Und ausgerechnet JavaFX, die UI-Technologie, die den holprigsten Start von allen hingelegt hat, hat sich wacker gehalten. Mehr noch: JavaFX ist, man darf es mittlerweile laut sagen, technisch gelungen und erfreut sich – zumindest im Java-Lager – zunehmender Beliebtheit.

von Björn Müller

JaxSpeaker

Einführungen in JavaFX gibt es genug. Oracle selbst bietet hierzu einiges an Informationen an [1]. In diesem Artikel geht es deswegen nicht um Feinheiten des Ersteinstiegs, sondern um eine Standortbestimmung der grundsätzlicheren Art:

- Was sind die Kernbestandteile von JavaFX?
- Wie gestaltet sich die Arbeit mit JavaFX-Komponenten?
- Wie werden JavaFX-basierte Anwendungen ausgeliefert?
- Welche Erfahrungen gibt es?
- Was sind die Vor- und Nachteile gegenüber HTML5? Welches sind die bevorzugten Einsatzgebiete?
- Wo fehlt's noch?

Eventuell stecken Sie gerade in einer Phase, in der Sie sich mit der Wahl einer UI-Technologie für ein größeres Vorhaben beschäftigen. Und Sie stellen fest, dass es immer schwieriger wird, in der heutigen UI-Welt eine Strategie zu finden, die für Ihr Vorhaben eine langfristige, stabile Grundlage bildet. Dieser Artikel wird Ihnen diese Strategie nicht geben, er wird Ihnen aber aufzeigen, wo und wie JavaFX in Ihrer UI-Strategie eine Rolle spielen kann – und vielleicht auch sollte.

Kernbestandteile von JavaFX

Abbildung 1 zeigt die Schichtung der JavaFX-Architektur [2]. Fangen wir mal unten an: Die unterste Ebene, die Basis von allem, ist eine normale Java Virtual Machine. Irgendwo läuft also alles im bewährten Muster ab, dass Programme in Form von Bytecode abgearbeitet werden. In diesem Artikel wollen wir einfach mal davon ausgehen, dass Java-Programme diesen Bytecode erzeugen – wohl wissend, dass es mittlerweile auch genügend andere Sprachen (Groovy und Co.) gibt, die ebenfalls Java-Bytecode erzeugen.

Gehen wir in der Architektur weiter nach oben. Hier findet sich die Klassenwelt der JavaFX-Komponenten mit all ihren APIs – sprich hier findet sich das, was man als UI-Entwickler im täglichen Umgang mit JavaFX sieht: die Grundklassen, aus denen man seine Dialoge zusammensteckt. Hier gibt es zunächst auch keine Überraschungen: Es gibt grafische Komponenten (wie Feld, Button ...), es gibt Container-Komponenten (wie horizontale Boxen, vertikale Boxen ...) und es gibt ein Zusammenfügen der Komponenten in einen Komponentenbaum, der bei JavaFX „Scene Graph" genannt wird.

Das Schöne an diesem Scene Graph ist, dass einige Eigenschaften und Operationen auf einen Knoten angewendet werden können, die dann automatisch

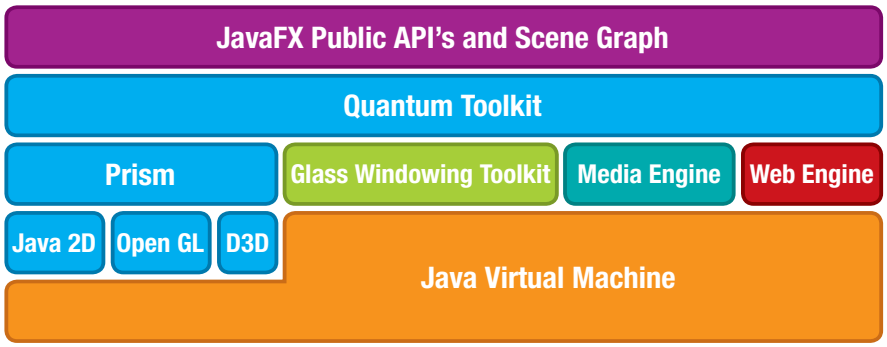


Abb. 1: JavaFX-Architektur

Mehr zum Thema

Wollen Sie sich noch intensiver mit dem Thema JavaFX beschäftigen? Dann sollten Sie sich den JavaFX Special Day auf der JAX 2013 genauer ansehen. Einen ganzen Tag lang beschäftigen sich Experten wie Björn Müller, Gerrit Grunwald und Terrence Barr mit dem Thema. Alle Infos finden Sie auf www.jax.de.

Abb. 2: JavaFX Ensemble – hier lernt man die Komponenten kennen

auch für alle Unterknoten gelten. Beispielsweise gibt es verschiedene Arten perspektivischer Transformationen: Dreht man einen Container-Knoten im Scene Graph, so dreht sich dessen gesamter Inhalt wie selbstverständlich mit. Alle Controls bleiben dabei funktional komplett aktiv. Selbiges gilt dann auch für Animationen.

Zwischen der obersten Schicht der Java-Klassen und der untersten Schicht der Virtual Machine liegt die „Magic", die dafür sorgt, dass am Ende auch wirklich etwas Sichtbares auf dem Bildschirm herauskommt. Die beinhalteten Komponenten sind aus Entwicklersicht eigentlich uninteressant, da man mit ihnen nicht in direkten Kontakt gerät. Trotzdem tauchen hier Begriffe auf, die man zumindest einordnen sollte.

„Prism" ist das Modul, das für die eigentliche Ausführung des Renderings zuständig ist. Es leitet Rendering-Befehle an die jeweils passende physikalische Ausführungsebene weiter. Bei einem zeitgemäßen Clientsystem mit Grafikkarte werden entsprechende Methoden der DirectX-Schnittstelle (MS Windows) oder der OpenGL-Schnittstelle (Mac, Linux, Embedded Devices) aufgerufen. Animationen und Transformationen geschehen somit in der Regel nicht in irgendeinem Java-Software-Layer, sondern werden direkt über die Grafikkarte verarbeitet. Die Performance von Animationen ist dementsprechend gut. Nun denn: wenn keine Grafikkarte ermittelt wird, dann wird doch das gute alte Java 2D bemüht. Aber dann heißt es auch: besonders schnell wird das Rendering nicht.

Das „Glass Windowing Toolkit" spricht mit dem Betriebssystem über Dinge wie Fenster und Dialoge, in denen ja schließlich ein Scene Graph eingebettet werden muss. Die „Media Engine" – drei Mal dürfen Sie raten – stellt Audio- und Videokomponenten zum Abspielen von MP3-, WAV-, MPEG4-Inhalten zur Verfügung. Und die „Web Engine" stellt die Browserkomponente („WebView") zur Verfügung – übrigens auf Basis des Open-Source-WebKit-Browsers. Sowohl Media als auch Web Engine zeigen, dass die Zeiten vorbei sind, als Java wie unter Swing noch versuchte, einen eigenen Java-basierten Browser zu bauen. Solche hochkomplexen Komponenten werden nun nativ als definierter Bestandteil der JavaFX-Umgebung zur Verfügung gestellt.

Fehlt noch das „Quantum Toolkit" – nichts anderes als eine einheitliche Schnittstelle oberhalb der aufgezählten nativen Bestandteile, die dafür sorgt, dass die in Java implementierten Komponenten einheitlich mit der darunter befindlichen nativen Welt reden können.

JavaFX in den beschriebenen Bestandteilen wird zurzeit auf den gängigen Desktopbetriebssystemen Windows, Linux und Mac OS zur Verfügung gestellt. Des Weiteren gibt es eine Betaversion einer Unterstützung von Embedded Devices, die auf der ARM-Architektur beruhen (siehe die Artikel von Gerrit Grunwald und Thomas Scheuchzer ab Seite 52). Heiß diskutiert und nun endlich mal wieder mit einer offiziellen Aussage von Oracle selbst versehen ist das Thema „JavaFX auf den mobilen Betriebssystemen iOS und Android" (siehe den Artikel von Wolfgang Weigend auf Seite 82). Jeder kennt die Berichte von Prototypen, die bereits zur JavaOne 2011 (!) gezeigt wurden – danach aber kehrte eine fast schon qualvolle Ruhe ein. Nun endlich hat Oracle angekündigt, die Sourcen seiner Entwicklungsarbeiten in zügigem Tempo der Open-Java-Community zu übergeben. Dies ist ein erster Schritt in die richtige Richtung, da die Geschwindigkeit einer Portierung dann zumindest aus der Community heraus selbst gesteuert werden kann. Eine konkrete Roadmap, wann (und ob) welche Bestandteile von JavaFX zur Verfügung stehen könnten, gibt es also leider noch nicht, aber die Hoffnung auf eine ebensolche ist zumindest gestiegen.

Der Umgang mit JavaFX-Komponenten

Dem Java-Entwickler offenbart sich JavaFX als große Komponentenbibliothek mit folgenden Bereichen:

- Fensterkomponenten wie Stage, Window
- Eingabekomponenten wie Feld, Label, Button, Checkbox etc.
- Container-/Layoutkomponenten wie Pane, HBox, VBox, ScrollPane, SplitPane etc.

- Formenkomponenten wie Rectangle, Circle etc.
- Tabellen-/Tree-Komponenten

Der Umfang der Grundkomponenten, die fester Bestandteil von JavaFX sind, ist „in Ordnung" – aber auch nicht mehr. Java-Entwickler, die von Swing kommen, finden zunächst ihre gewohnten Grundkomponenten wieder. Darüber hinaus gibt es als echten Fortschritt die bereits angesprochenen Browser- sowie die Media-Player-Komponenten. Auch im Bereich von Grafik und „Charting" gibt es nun Standardkomponenten, die recht flexibel Kuchen-, Balken- und weitere Diagramme erzeugen.

Eine Reihe von Komponenten, die man aber immer wieder braucht (z. B. eine Kalenderkomponente) kommen nicht von der Stange. Es gibt hier aber eine Reihe von Projekten, die aktiv sind – zum Beispiel „JFXtras" [3].

Eine konkrete Oberfläche entsteht dadurch, dass Instanzen der Komponentenklassen erzeugt werden und hierarchisch im Scene Graph eines Dialogs (Stage) angeordnet werden. Das wiederum geschieht z. B. durch ein normales Java-Programm und gestaltet sich unspektakulär einfach. Für Freunde deskriptiver Dialogbeschreibungen gibt es auch die Möglichkeit der Definition einer sog. FXML-Datei, in der die Komponentenfolge und -attributierung enthalten ist, und die zur Laufzeit interpretiert wird.

Das Aussehen einer Komponente wird über eine Style-Definition festgelegt. Diese stammt zum einen aus einer oder mehreren CSS-Dateien (Cascading Style Sheets), kann zum anderen aber zur Laufzeit durch das Programm erweitert oder überschrieben werden. Das Anpassen des Grundstylings einer Anwendung an konkrete Vorgaben wird somit über das Schreiben einer eigenen CSS-Datei ermöglicht. Nebenbei: das standardmäßig mitgelieferte Style Sheet *caspian.css* ist durchaus optisch gelungen – für leidgeprüfte Java-Swing-Entwickler ist das eine ganz neue Erfahrung, dass eine mit Standard-Java zusammengebaute Oberfläche keinen allzu großen Anlass zum Schämen bietet.

Die Style-Sheet-Definitionen sind angelehnt an HTML Style Sheets, dies gilt auch für Vererbung und Hintereinanderschaltung.

Die Style-Sheet-Definitionen selbst sind angelehnt an HTML Style Sheets, dies gilt auch für Dinge wie Vererbung bzw. Hintereinanderschaltung (Kaskadieren) mehrerer Style Sheets. Die Zuordnung, welche Style-Sheet-Datei(en) konkret verwendet werden soll, geschieht normalerweise durch eine dynamische Zuweisung auf oberster Fensterebene – kann aber auch, wenn erforderlich, auf jedem Knotenpunkt des Scene Graphs definiert werden. Mehr Flexibilität kann man, denke ich, hier gar nicht bereitstellen.

Auch beim Thema Eventverwaltung merkt man, dass sich JavaFX verglichen mit Java Swing weiterentwickelt hat. Events werden in zwei Phasen abgearbeitet: der Filter- und der Verarbeitungsphase. In der Filterphase kann ein Event Listener entscheiden, dass ein Event gar nicht erst zu Verarbeitung kommt, sprich weggefiltert wird. In der Verarbeitungsphase kann man dann auf gewöhnliche Art und Weise seine Event Listener einbauen. In beiden Phasen werden die Events den Scene Graph hinauf verarbeitet: Ein Event endet also nicht an dem Knoten, an dem es entsteht, sondern wandert so lange seine Elternknoten hinauf, bis entweder ein Knoten dieses Hinaufwandern abbricht („consume" des Events) oder der oberste Knoten erreicht ist. Dies vereinfacht die Behandlung von Events ganz enorm.

Es wird zwischen „echten" Events (z. B. User dragged mit der Maus) und den daraus folgenden Änderungen

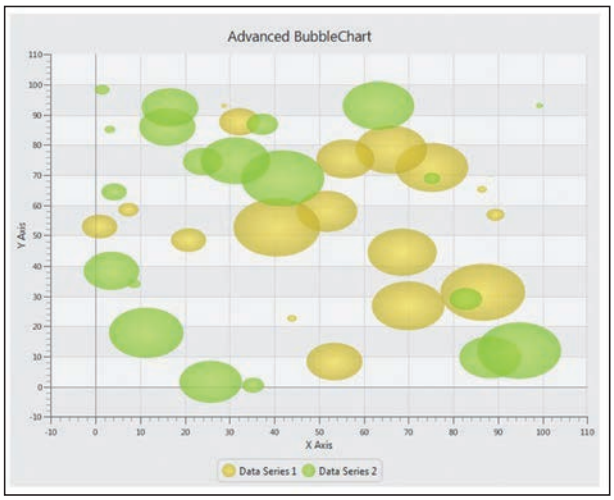


Abb. 3: Auch Grafikkomponenten gehören bei JavaFX zur Standardbibliothek

(z. B. Fenstergröße wird geändert) unterschieden. Die Verarbeitung von echten Events geschieht über oben angeführte Event Listener, die Verarbeitung von Folgeaktionen über entsprechende Properties: die betreffenden Eigenschaften einer Komponente werden nicht als einfache Werte (double) zur Verfügung gestellt, sondern als Property-Objekte (DoubleProperty). An diese Property-Objekte kann man dann in entsprechenden Change Listenern die eigene Verarbeitung hängen. All dies macht Sinn und bringt Ordnung: Wenn man eine Verarbeitung an die Änderung der Fenstergröße koppeln will, so tut man dies, indem man entsprechende Listener an die „width" und „height" Property koppelt – unabhängig davon, über welche originären echten Events die Größenänderung nun herbeigeführt wurde.

Perspektivische Transformationen und Animationen sind ein weiteres Herzstück der JavaFX-Klassenbibliothek – schließlich kommt der Ursprung von JavaFX ja aus einer Zeit, in der man interaktive Inhalte im Web in Konkurrenz zu Adobe Flashplayer und Microsoft Silverlight ermöglichen wollte. Eine Animation beispielsweise ist nichts anderes als ein Objekt, in dem man angibt, welche Zustände zu welchen Zeitpunkten durchlaufen werden sollen. Dieses Objekt wird dann auf einen bestimmten Knotenpunkt/-bereich des Scene Graphs angewendet – mehr ist nicht zu tun. Eigendefinierte Animationen, die nicht in ein bestimmtes Standardmuster (Drehung, Skalierung, ...) passen, sind sehr einfach über Property-Animationen durchzuführen. Kurz: Animationen sind so einfach zu gestalten, dass man nun zum ersten Mal in einer Java-Umgebung überhaupt ernsthaft darüber nachdenkt, wo man sie sinnvoll einsetzen kann.

Fehlt noch die Betrachtung der Erweiterbarkeit der Komponentenwelt. Neue Komponenten entstehen entweder über Ableitung oder durch Aggregation bestehender Komponenten. Da die zugrunde liegende Sprache Java ist, gestaltet sich das naturgemäß einfach. Dies gilt auch für die Schaffung eigener neuer Container-/Layoutkomponenten, da das Layouting-Konzept einfach und klar ist: eine Container-Komponente hat in der *layoutChildren()*-Methode die Aufgabe, ihren Inhalt anzuordnen, und sie muss über *minWidth/Height()*- und *preWidth/Height()*-Methoden ihre Größe kundtun. 60 Mal pro Sekunde wird ein so genannter „Pulse" abgearbeitet, in dem veränderte Bereiche des Scene Graphs neu berechnet und gerendert werden. Dieses asynchrone Konzept verhindert automatisch und verlässlich das Mehrfachlayouten gleicher Bereiche.

Auslieferung JavaFX-basierter Anwendungen

Auch hier tut sich einiges im Java-Lager! Doch beginnen wir mit dem Altbekannten. Ein JavaFX-Programm kann standardmäßig in den drei von Java bekannten Formen ausgeliefert werden: als Applet, per Web Start oder als eigenständiges Java-Programm (Application). Alle drei erfordern eine vorherige Installation einer Java-Runtime-Umgebung (JRE). Ab Java 7 ist JavaFX der JRE beigepackt – ab Java 8 ist es fester Bestandteil der Java-Umgebung.

Die explizite Installation einer JRE erfreut sich nicht gerade großer Beliebtheit. Der Benutzer hat das Gefühl, sein System einem mittelgroßen Update unterziehen zu müssen, um überhaupt ein Java-Programm zum Laufen zu bringen. Unterstützt wird dieses Gefühl durch eine lange Laufzeit der JRE-Installation und durch die Erforderlichkeit, das Administratorpasswort im Laufe der Installation einzugeben. Im betrieblichem Umfeld bedeutet Letzteres, dass alle Java-Installationen über die Systemadministration gemanagt werden müssen – was den negativen Gesamteindruck noch verstärkt.

Deswegen ist es eine gute Nachricht, dass eine neue Art der Installation nun von Java-/Oracle-Seite aus nicht nur erlaubt, sondern auch durch entsprechendes Tooling gefördert wird: das Bundling. Hierbei wird per ANT-Skript das von Ihnen erstellte Java-Programm, die benötigten Bibliotheken und die plattformspezifische native Runtime (also die JRE) zu einem Bundle zusammengefügt. Aus z. B. Windows-Sicht entsteht dann entweder ein *setup.exe* oder ein *setup.msi* Installer. Die Installation ist dann fast schon „App-like". Man lädt sich den Installer über einen Link aus dem Internet, führt ihn aus – und das eigentliche Programm steht auf dem Clientsystem zur Verfügung. Die Installation geschieht automatisch in das Userverzeichnis hinein, man wird also nicht einmal nach einem Installationsort gefragt. Und es gibt auch kein Administratorpasswort, das man eingeben muss.

Nun, anhand der Größe des Installer-Programms merkt man schon noch, dass irgendwo Java im Inneren steckt: unter 30 MB läuft da nichts. Aber: Für sehr viele Szenarien ist die „App-like"-Installation eine wesentlich einfachere und günstigere Form der Softwareverteilung – sie entbindet den Benutzer von dem schwergewichtigen Schritt der expliziten Installation einer Java Runtime.

Auch für den Softwarelieferanten ist das Bundling vorteilhaft: Über das Bundle wird die Laufzeitumgebung

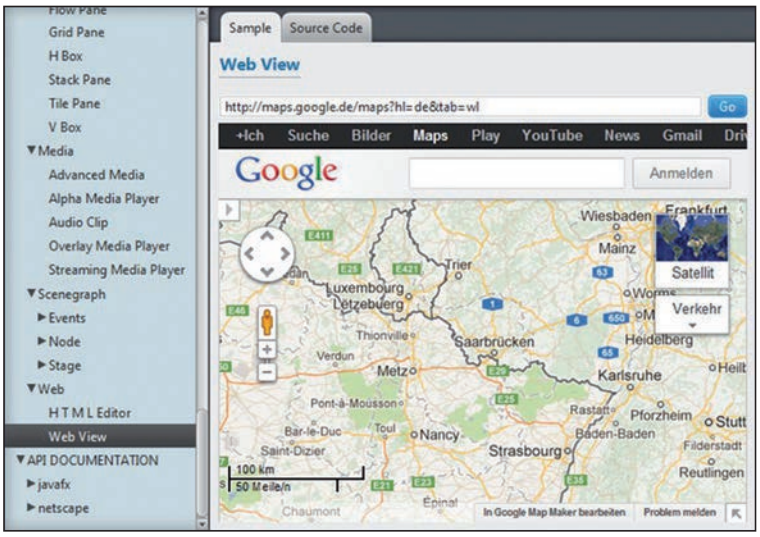


Abb. 4: Dank eingebautem WebKit-Browser: einfaches Einbauen von Webinhalten

für ein Java-Programm mitgegeben, der Lieferant kann also genau bestimmen, dass seine Software mit genau der Version 2.x.y beim Endanwender ausgeführt wird und ist somit unabhängig von dem Java-Installationsstand beim Kunden.

Erfahrungen mit JavaFX

JavaFX ist zurzeit auf Versionsstand 2.2 – kann und sollte man darauf einsteigen? Oder kann man sich mit der Ausrede „erst einmal auf 3.0 warten" wieder gemütlich zurücklehnen? Die folgenden Aussagen beruhen auf einer eigenen intensiven Nutzung von JavaFX 2.2 über die letzten neun Monate – vornehmlich innerhalb der Windows-Welt, aber mit Deployment nach Linux und Mac OS.

Zunächst ein Blick auf das Thema Stabilität: hier gibt's nur eine Note zu verteilen, und die heißt „erstaunlich gut". Es gibt keine grundsätzlichen Probleme, über die wir massiv gestolpert sind oder die wir nicht durch einfache Workarounds umgehen konnten. Neben der Grundstabilität der technischen Umgebung selbst trägt hier auch die organisatorische JavaFX-Infrastruktur bei: Es ist erstaunlich und für Java-Entwickler eine ganz neue Erfahrung, dass man auf Fragen im Oracle Technology Network (OTN) wirklich in der Regel sehr schnell Antworten erhält. Selbiges gilt für Bug-Reports, die man im zugehörigen JIRA-System einpflegt und bei deren weiteren Verarbeitung das Gefühl zurückgegeben wird, dass man ernst genommen wird.

Die Entwicklung mit JavaFX gestaltet sich recht einfach – hier zahlt sich Oracles Entscheidung aus, ab Release 2.0 wieder ganz normales Java als Grundsprache für JavaFX zu verwenden. Alle Tools, die man liebgewonnen hat – Debugger, Profiler, Stackanalyzer –, stehen somit ganz normal zur Verfügung und ermöglichen von Anfang an einen professionellen Umgang. Die präferierte Umgebung ist zwar, wen wundert es, NetBeans – eine Entwicklung in Eclipse ist aber ebenso problemlos möglich (siehe Artikel von Marc Teufel auf Seite 66).

Die Performance von JavaFX-Anwendungen auf modernen Rechnern ist gut und stellt keine Probleme für den Entwicklungsprozess dar. Im Vergleich mit Swing-Anwendungen konnten wir keine spürbaren Unterschiede feststellen, wenn es um das Zusammenbauen großer Seiten mit komplexen Controls geht. Wenn es um Transitionen und Animationen geht, so braucht man überhaupt nicht zu vergleichen – alles, was man hier unter Swing zur Verfügung stellen konnte, war eine Krücke gegenüber dem flüssigen Ablauf von Animationen in JavaFX. Fehlt dem Clientrechner die Hardware-Grafikunterstützung, dann sieht es leider anders aus: dann ist JavaFX recht zäh. Messungen z. B. auf Industrieterminals (das sind die robusten, staubisolierten Touch-PCs für Produktionsumgebungen, die immer dem aktuellen Hardwarestand um ein paar Jahre hinterherhinken) ergaben, dass Swing hier doch noch eine deutliche Ecke schneller ist.

Anbindung an die eigentliche Anwendungslogik

JavaFX ist eine Java-basierte Umgebung zum einfachen und effizienten Entwickeln moderner Oberflächen – nicht mehr und nicht weniger! Irgendwo ist ja eine Oberfläche in der Regel kein Selbstzweck, sondern dient dazu, eine darunter liegende Logik (z. B. die Geschäftslogik einer Anwendung) zu bedienen. Frameworkmenschen werden somit gleich die Frage stellen: Was bietet JavaFX hier an Konzepten und damit an Bibliotheken? Gibt es zum Beispiel Binding-Konzepte, wie Objekte aus den Tiefen der logischen Schichten direkt in die Oberfläche eingebaut werden können und Datenänderungen in der Oberfläche direkt die logische Validierung der Objekte anstoßen?

Die Antwort ist recht einfach: JavaFX bietet hier nichts. Oder positiver formuliert: JavaFX macht hier keine Vorgaben, sondern erlaubt eine offene Einbindung in entsprechende Denkkonzepte. Und das ist auch gut so, zumindest aus meiner Sicht: Die Art und Weise der Anbindung einer Oberfläche an bestehende Anwendungslogik sollte unterhalb der UI-Ebene auf einer eigenen Architekturebene geschehen. Andersherum gesagt: Die UI-Ebene sollte und darf nicht irgendwelche Vorgaben auf die Strukturierung der Anwendungslogik machen!

Ein Frontend-Entwickler muss sich somit ein paar prinzipielle Fragen stellen, bevor er/sie loslegt – insbesondere, wenn es um Anwendungen geht, in denen die eigentliche Verarbeitung und Datenhaltung in einem (oder mehreren) zentralen Servern durchgeführt wird:

- Wie viel Logik soll auf dem Client laufen?
- Welche Art von Logik soll auf dem Client laufen?
- Wie kommuniziert der Client mit einem hinten liegenden Server?

Die Überlegungen und daraus resultierende Architekturen sind nicht JavaFX-spezifisch – JavaFX ist aber eine Umgebung, die durch ihre Offenheit praktisch für den Einsatz in allen Architekturen geeignet ist:

- Falls im Client explizit ein Teil der Anwendungslogik laufen soll, hat man das typische Szenario: Der Frontend-Entwickler bindet seine vorne laufende Logik direkt in die explizit entwickelten Dialoge ein und kommuniziert über irgendwelche Web-Service-Verfahren mit dem hinten liegenden Server. Hierfür eignet sich natürlich gerade im Vergleich zu HTML5-Szenarien JavaFX hochgradig: die zugrunde liegende Programmierung in Java mit ihren einfachen, sauberen Strukturprinzipien ermöglicht doch eine wesentlich effektivere Entwicklung als z. B. unter JavaScript.
- Im Falle der Verwendung von generischeren Clientkonzepten („Thin Client"-Konzepte) geschieht die eigentliche Interaktionsverarbeitung auf dem Server – der Client wird von diesem nur noch mit generischen Maskenbeschreibungen beschickt, die dieser ausrendert. Solche Konzepte eignen sich in der Regel für größere, komplexere Anwendungen, erfordern aber ein gewisses Grundframework. Auch hierfür ist JavaFX sehr gut geeignet. Und es gibt auch erste Frameworks, die genau in diese Richtung zielen [4], [5].

Positionierung insbesondere mit/gegenüber HTML5

Wie ist die Reaktion, wenn Sie in Ihrem Entwicklungsumfeld vorschlagen, in einem neuen Vorhaben JavaFX einzusetzen? Mit einer großen Wahrscheinlichkeit werden Sie mit der Aussage konfrontiert: „Ist ja ganz nett, aber warum nehmen Sie nicht HTML5?" Und das ist noch freundlich formuliert: Oft wird seitens irgendeines Managements die Losung „HTML5 everywhere!" ausgegeben, Punkt aus. Es ist dann Ihr Job, entweder die Segel zu streichen und mit HTML5 und seinen Frameworks zu kämpfen – oder mit guten Argumenten zu kontern.

JavaFX beginnt dann attraktiv zu werden, wenn es sich bei Ihrer Anwendung um ein größeres, komplexeres Vorhaben handelt.

Zunächst gilt: Kämpfen Sie nicht mit Windmühlen! HTML5 hat unbestreitbar einige technische Nachteile (JavaScript als Programmier-/Ausführungsbasis, Plattforminkompatibilitäten, immer noch: Performance) – hat aber ein gewichtiges, zentrales Argument auf seiner Seite: es läuft direkt im Browser – „zero installation"! Wenn es also um eine überschaubar geringe Anzahl von Oberflächen geht, in denen noch dazu die Komplexität der Oberflächen begrenzt ist und die von den Benutzern nur sporadisch genutzt werden: machen Sie's über HTML5! Ähnliches gilt, wenn Sie eine Anwendung bauen, die sich komplett an den anonymen Benutzer wendet (etwa einen Onlineshop): Nehmen Sie HTML5, und planen Sie bei Ihrem Management genug Ressourcen zur Entwicklungs- und Wartungszeit ein, um damit verbundenen Problemen begegnen zu können. „Run everywhere" zusammen mit „zero installation" hat seinen Preis.

JavaFX beginnt dann attraktiv zu werden, wenn es sich bei Ihrer Anwendung um ein größeres, komplexeres Vorhaben handelt, dessen Benutzer in der Regel bekannt sind (z. B. Mitarbeiter einer Firma) und bei denen es sich vornehmlich um Desktopbenutzer handelt (leider steht JavaFX ja auf z. B. Android-/iOS-Systemen noch

nicht zur Verfügung). Und von solchen Anwendungen gibt es nicht wenige, insbesondere im Umfeld von Geschäftsanwendungen. Die Proargumente, die Sie dann bringen können, sind:

- JavaFX basiert auf Java – und bringt dabei eine wesentlich höhere Entwicklungseffizienz als eine wie auch immer geartete JavaScript-Entwicklung mit. Dies gilt insbesondere, wenn es sich um clientlastige Architekturen handelt, in denen ein Teil der Logik direkt im Client ausgeführt werden soll.
- JavaFX-Anwendungen sind keine „zero installation"-Anwendungen. Aber – ich spreche jetzt von der Auslieferung in Bundles – sie sind sehr einfach, „App-like" verteilbar. Hier muss man generell ausholen und daran erinnern, dass es zurzeit zwei Auslieferungsparadigmen gibt, die beide Sinn machen und beide in Massen funktionieren: das eine ist die HTML-geprägte „zero installation" – das andere ist die iOS-/Android-geprägte „App-Installation". Mit JavaFX sind Sie definitiv im „App-Lager". Die Verteilung ist einfach, erfordert kein Systemupdate – und ganz wichtig: die Anwendung läuft in einer Umgebung ab, die weitestgehend durch Sie definiert und nicht durch die Browserwahl des Benutzers von außen bestimmt ist. Dies sind unschätzbare Vorteile, wenn es darum geht, mit einem kleinen Team Software zu bauen, die wirklich auch auf den dafür vorgesehenen Endgeräten stabil und mit geringem Serviceaufwand laufen soll.
- JavaFX hat zwar eine leidvolle Geschichte, aber auch eine lange Geschichte. Die Firmen dahinter (zunächst Sun, dann Oracle) haben – teilweise wider Erwarten – ein großes Commitment gezeigt und nie an der strategischen Bedeutung von JavaFX gezweifelt. Der Schritt, JavaFX in die Open-Java-Community zu übergeben, ist genau der richtige und passiert zum richtigen Zeitpunkt. Open Source wird hier nicht als Vehikel missbraucht, einem sinkenden Schiff nochmal einen neuen Anstrich zu geben, sondern es wird gebraucht, um eine heranwachsende Technologie auf eine breitere Basis zu stellen. Vergleichen Sie diese Frameworkzyklen im Java-Lager (Swing gibt es schon seit über fünfzehn Jahren und wird auch noch weiter unterstützt) mit den Framework-/Hype-Zyklen in der Web-/HTML5-Welt: praktisch alle Jahre kommen und gehen dort die Hypes und die damit verbundenen Frameworks. Vergleichen Sie den Lebenszyklus Ihrer Anwendung mit dem Lebenszyklus von HTML-Frameworks: Anwendungen im betriebswirtschaftlichen Umfeld haben gut und gerne Lebensdauern, die über fünfzehn Jahre weit hinausgehen. Gilt dieser Lebenszyklus auch für das Kern-UI-Framework Ihrer Anwendung?

Sie sehen: Es gibt durchaus Diskussionsbedarf, wenn es um die Richtigkeit des Slogans „HTML5 everywhere" geht. Ganz klar: das Szenario muss passen!

Wunschliste ...

Keine Bange: Es kommt nun keine lange Liste nach dem Motto „an diesem Control bitte noch diese Möglichkeit". Derlei kleine Wünsche sollte man nicht in Artikeln wie diesem formulieren, sondern man sollte direkt per OTN mit der JavaFX-Entwicklung in Kommunikation treten. Also doch keine Wunschliste?

Nun denn, einen großen Wunsch habe ich natürlich – und dieser wird von vielen JavaFX-Entwicklern geteilt: der Wunsch, JavaFX zügig von den Desktopbetriebssystemen auszudehnen auf die gängigen mobilen Betriebssysteme – Android und iOS in vorderster Linie. Wie bereits erwähnt: Die Open-Java-Community wird sich nun diesem Thema stellen dürfen und müssen.

Fazit

JavaFX ist technologisch gelungen und ein mehr als würdiger Nachfolger von Swing. Die Entwicklung von JavaFX macht Spaß, da die Struktur des Systems wohl durchdacht ist und sich das System in der Regel so verhält, wie man es erwartet. Die Stabilität ist absolut in Ordnung.

Wird es sich langfristig durchsetzen? Ich glaube: ja! Es gibt genug Anwendungsarten, für die JavaFX sehr gut geeignet ist. Verbunden mit der „App-artigen-Installation" sehe ich sehr großes Potenzial. Und nicht zu vergessen: Alleine die Anzahl von Swing-Entwicklern, die weiterhin in Java Oberflächen bauen wollen, ist hoch genug, um eine signifikante Anzahl an Anwendungen entstehen zu lassen.

Wird es mal ein Megahype? Mit etwas Sarkasmus zum Abschluss sage ich: hoffentlich nicht! Hypes verbinde ich immer mit Webframeworks – und da gibt es eher zu viele Hypes als zu wenige.

Björn Müller, CaptainCasa GmbH, beschäftigt sich seit 2001 mit Technologiefragen der User-Interface-Entwicklung: zunächst auf Basis von HTML/JavaScript, seit 2007 aber im Rahmen der CaptainCasa-Community auf Basis Java-basierter Frontends.

Links & Literatur

[1] Dokuseite zu JavaFX: http://docs.oracle.com/javafx/

[2] Architekturbeschreibung: http://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm

[3] Erweiterte JavaFX Controls: http://jfxtras.org/

[4] Rich Client Framework auf Basis von JavaFX und JSF: http://www.CaptainCasa.com

[5] Generisches Kopplungsmodell zwischen Client- und Serververarbeitung: http://opendolphin.org

Anzeige

Anzeige

Ein Open Source verfügbarer Ansatz

Enterprise JavaFX

Dieses Heft macht Lust auf JavaFX. Wie aber soll man vorgehen, wenn man eine echte Enterprise-Applikation mit JavaFX bauen oder gar eine bestehende auf JavaFX migrieren möchte? Hier bietet Open Dolphin [1] einen architektonischen Ansatz, der den Wechsel zwischen UI-Toolkits erleichtert und einen Investitionsschutz für die Fachlogik garantiert.

von Dierk König

JAX Speaker

Open Dolphin wurde in der JavaOne Strategy Keynote als Integrationsbeispiel von JavaFX und Java Enterprise vorgestellt. Als Demo diente eine Applikation (**Abb. 1**), mit der man die Container in einem Containerhafen überwachen und steuern kann. Ein Video dieser Demo findet sich unter [2].

Diese Demo ist auf zwei Arten typisch für JavaFX-Enterprise-Applikationen:

- Erstens möchte man von den reichhaltigen Visualisierungmöglichkeiten von JavaFX profitieren.
- Zweitens hat der Anbieter dieser Software seit vielen Jahren ein serverseitiges Programmiermodell in Betrieb und das möchte er erhalten.

Sei freundlich!

Natürlich braucht nicht jede Geschäftsanwendung eine ausgefeilte 3-D-Erlebniswelt für ihre Daten – auch wenn das häufig sinnvoller ist, als man vermuten würde. Zumindest jedoch erwarten die Anwender heute ein Benutzungserlebnis wie sie es von Smartphones und Tablets her kennen, mit sinnvoller Eingabeunterstützung, direkter Manipulation, weichen Übergängen und sofortiger Anzeige der Auswirkungen auf das Ergebnis. Professionelle Desktopanwendungen müssen genauso anwenderfreundlich sein!

Abbildung 2 demonstriert eine auf den ersten Blick konventionelle Anwendung für die Zusammenstellung von finanziellen Portfolios. Sie ist im Lieferumfang von Open Dolphin enthalten. Schon beim Anschauen des Videos [3] fallen aber die Besonderheiten auf:

- sofortige, konsistente Aktualisierung aller Anzeigen und Ergebnisse
- weiches Ein- und Ausblenden
- animierte Übergänge und Materialisierung
- trotz Serveranbindung keine Wartezeiten

Diese Effekte werden durch eine Kombination von Open Dolphin und JavaFX bewirkt. Open Dolphin stellt das Presentation Model bereit und bestimmt damit, *was* angezeigt werden soll. JavaFX bindet sich an das Presentation Model und übernimmt das *Wie* der Darstellung, also die Visualisierung des Zustands und der Übergänge.

Die Konsistenz der Darstellung und die sofortige Aktualisierung ist eine Eigenschaft von Open Dolphin. Die Views kennen einander nicht mehr. Es gibt auch keinen „Presenter“ mehr, der alle Views kennen muss. Die Views binden sich an ihr Presentation Model. Das wars.

Migration durch Ignoranz

Nun ist es aber Open Dolphin vollständig egal, ob sich ein JavaFX oder Swing oder SWT oder Eclipse RCP oder NetBeans oder sonst eine Java-basierte View auf

Abb. 1: JavaFX-Demo – Containerhafen

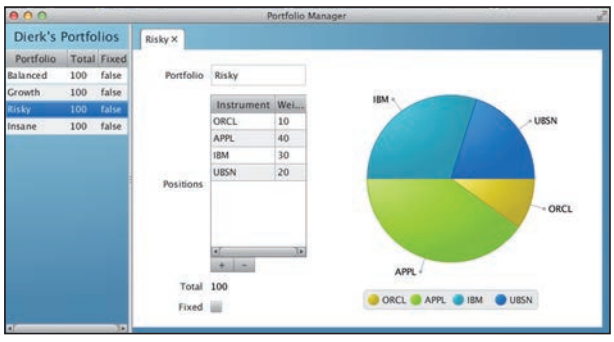


Abb. 2: Portfoliodemo

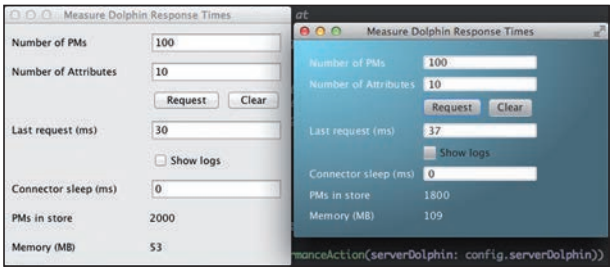


Abb. 3: Performancedemo

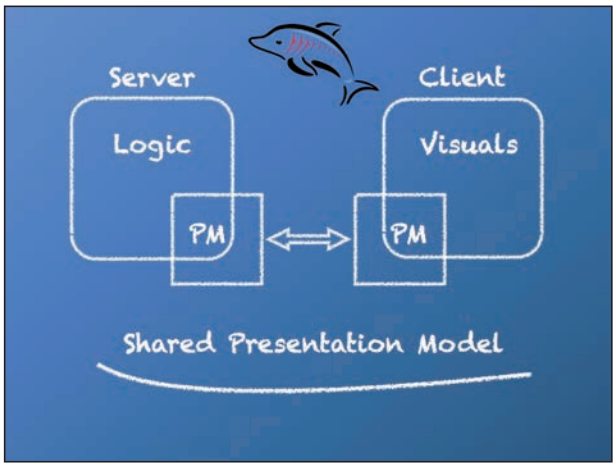


Abb. 4: Open-Dolphin-Architektur

die Presentation Models als Listener registriert. Sie werden alle gleichermaßen freundlich bedient.

Das ist der Trick für die Austauschbarkeit des UI-Toolkits: zuerst die Visualisierung abkapseln und dann ersetzen. Auf diese Art ermöglicht es Open Dolphin, sich heute schon auf eine zukünftige Migration zu JavaFX vorzubereiten, oder sofort mit JavaFX anzufangen, aber sich für eventuelle Änderungen abzusichern. Die Open-Dolphin-Performancedemo (**Abb. 3**) zeigt, wie die gleiche Logik mit Swing und mit JavaFX dargestellt werden kann. Auch hierzu gibt es ein passendes Video [4].

Bereit sein und parallel arbeiten

Die „eventuellen Änderungen" kommen meist schneller als man denkt. Es sind nicht nur technologische Neuerungen, neue Toolkits oder deren Versionen. Manchmal gibt es einfach neue Optionen. Die Containerapplikation zum Beispiel haben wir zuerst vollständig in 2-D gebaut. Dann gab es plötzlich die Möglichkeit für 3-D und wir brauchten lediglich die Visualisierung erweitern. Die Modi sind sogar zur Laufzeit umschaltbar.

Listing 1

```
serverDolphin.action("PrintText", new NamedCommandHandler() {
  public void handleCommand(NamedCommand namedCommand,
                            List<Command> commands) {
    Object text = serverDolphin.getAt("input").getAt("text").getValue();
    System.out.println("server text field contains: " + text);
  }
});
```

Nun unterscheiden sich die UI-Toolkits nicht nur in ihren Widgets, sondern auch im Threading Model. Auch dafür schafft Open Dolphin einen wichtigen Mehrwert. Alle visuellen Komponenten werden garantiert immer im UI Thread gehandhabt. Alle Controller Actions werden immer asynchron ausgeführt. Das ist möglich, weil sie ausschließlich auf Presentation Models operieren. Man kann die Nebenläufigkeit so gut wie nicht mehr falsch machen. Der UI Thread wird nie blockiert, genauso wenig wie die Steuerungslogik.

Logische und physische Verteilung

Soweit betrafen die Strukturen und Verfahren von Open Dolphin nur den Client. Enterprise-Applikationen laufen aber typischerweise auf dem Server. Deshalb bietet Open Dolphin die Möglichkeit, sämtliche Logik auf dem Server zu installieren. Die Presentation Models werden dann automatisch zwischen Client und Server synchronisiert. **Abbildung 4** zeigt eine Skizze dieser Architektur.

Open Dolphin hat eine logische MVC-Trennung, wobei das M im Sinne von Presentation Model zu verstehen ist. View und Controller sind immer logisch und durch Threads getrennt. Typischerweise – aber nicht zwingend – werden sie auch physisch verteilt. Dann liegen die Views auf dem Client und die Controller und ihre Actions auf dem Server. Die Fachlogik liegt also dort, wo sie zentral verwaltet und gepflegt werden kann und die beste Verbindung zu den Unternehmensdatenquellen hat. Der zentrale Server ist auch der natürliche Ort, über den sich die Clients falls notwendig gegenseitig synchronisieren können. Die TrainControlDemo [5] und die ManyEventsDemo [6] machen davon ausgiebig Gebrauch.

Die Demos beeindrucken auch durch die Schnelligkeit der Datenübertragung. Das liegt an dem schlauen Einsatz der Nebenläufigkeit und den kleinen Übertragungspaketen. Zwischen Client und Server werden ausschließlich Commands gesandt – in kleinen Paketen und meist ohne dass der Benutzer es überhaupt bemerkt. Der Transportmechanismus zwischen Client und Server ist austauschbar.

Doing is believing

Open Dolphin selbst kommt mit vielen Demos, an denen man die verschiedenen Eigenschaften und Funktionen ausprobieren kann. Den besten Einstieg in Open Dolphin findet man aber mit einem einfachen Beispiel. Zu diesem Zweck gibt es auf GitHub [7] das Projekt *DolphinJumpStart*. Es kommt mit einem Maven Build und JavaFX Views, die in Java geschrieben sind (sonst machen wir uns gerne das Leben einfacher und nehmen GroovyFX).

Nehmen wir an, unsere View hätte folgendes Textfeld:

```
TextField field = new TextField()
```

Nun möchten wir den Inhalt dieses Textfelds an ein Open Dolphin Presentation Model binden. Dafür brau-

chen wir ersteinmal ein Model. Es soll unter der ID "*input*" auffindbar sein und ein Attribut mit dem Namen "*text*" haben:

```
PresentationModel input =
  clientDolphin.presentationModel("input", new ClientAttribute("text"));
```

Jetzt können wir die "text" Property des Textfelds an das Attribut binden:

```
JFXBinder.bind("text").of(field).to("text").of(input);
```

Der Inhalt des Textfelds wird nun bei jeder Änderung automatisch mit dem Presentation Model synchronisiert und der Server asynchron benachrichtigt.

Sagen wir, wir wollten den vom Benutzer eingegebenen Text auf dem Server verwenden, z. B. ausdrucken. Dann würden wir eine Action registrieren, wie in Listing 1 dargestellt. Anstoßen kann man die Serveraktion vom Client aus per

```
clientDolphin.send("PrintText");
```

Wie es Euch gefällt

Das war natürlich nur die allereinfachste Anwendung von Open Dolphin, um das Prinzip zu zeigen. Selbst wenn man sich nur das Binding anschaut, verbergen sich dort vielerlei Perlen – wie das Binden an den Dirty State oder an weitere Metainformationen.

Am wichtigstem beim Binding ist aber die Stabilität. Gerade bei Master-Detail-Views (**Abb. 5**) hat man häufig das Problem, dass man bei Selektionsänderungen die Detail-View neu binden muss, was alle Arten von Schwierigkeiten nach sich ziehen kann. Bei Open Dolphin ist das nicht notwendig. Das Binding bleibt immer stabil. Ein Video zur Push-Demo findet sich unter [8].

Mach mit!

Einen guten Überblick über die Eigenschaften von Open Dolphin gibt der Architekturteil der Dokumentation. Die Dokumentation selbst wird mit der 1.0-Version von Open Dolphin fertiggestellt, die für Ende April angekündigt ist. Bis dahin ist auch das API finalisiert.

Zu Open Dolphin gibt es viele Ressourcen im Netz, vor allem Videos über die Demos und die technischen Präsentationen. Open Dolphin ist Open Source mit Apache-2-Lizenz und liegt auf GitHub. Wir freuen uns über Fragen und sonstige Beiträge!

Die ersten größeren Applikationen mit Open Dolphin sind im Bau und zum Teil schon in Betrieb. Zusätzlich gibt es einige interessante Prototypen und Proof-of-Concept-Implementierungen. Die Early Adopters sind eine beständige Quelle neuer Anregungen. Sie haben zum Beispiel herausgefunden, dass sich Open Dolphin perfekt für das funktionale Testen der Applikation eignet. Man legt einfach die erwarteten Presentation Models an, sendet ein Command und überprüft, ob die Applikations- und Präsentationslogik die Presentation Models wie erwartet verändert hat. Um einen der Entwickler zu zitieren: „Das ist UI Testing ohne UI!“

Wer zu JavaFX umsteigen will, seine Investitionen absichern oder sonst Java-Desktop-Applikationen Enterprise-ready machen möchte, der sollte sich Open Dolphin genauer ansehen. Mit JavaFX für iOS und Android ergeben sich für Open Dolphin zukünftig auch noch völlig neue Einsatzgebiete im Mobile Computing.

Abb. 5: Push-Demo

Dierk König ist Fellow bei der Canoo Engineering AG, Basel. Er betreibt die Open-Source-Projekte Canoo WebTest und Dolphin und ist Committer in den Projekten Groovy, Grails und GPars. Er publiziert und spricht auf internationalen Konferenzen zu Themen moderner Softwareentwicklung. Er ist Autor des Buchs „Groovy in Action“.

@mittie. **@OpenDolphin**

http://open-dolphin.org

git@github.com:canoo/open-dolphin.git

http://www.youtube.com/user/dierkkoenig

Links & Literatur

[1] http://open-dolphin.org/download/guide/index.html

[2] http://www.youtube.com/watch?v=AS26gZrYNy8

[3] http://www.youtube.com/watch?v=W-LCvTa5MQQ

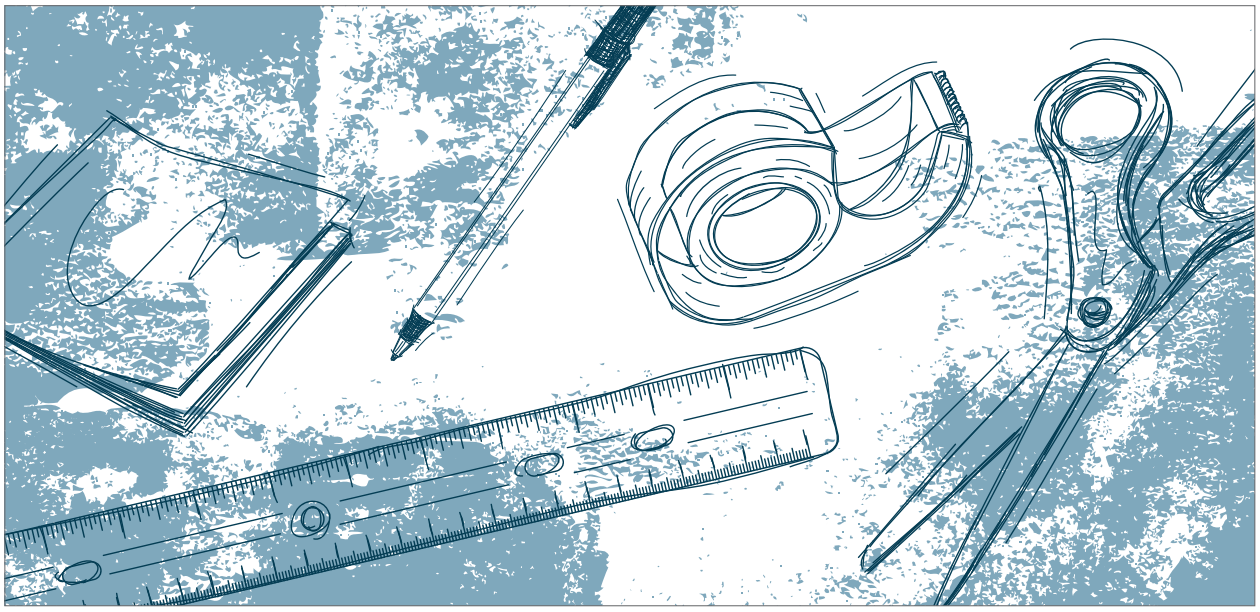
[4] http://www.youtube.com/watch?v=OoOaCOvMFW8

[5] http://www.youtube.com/watch?v=T4IrmafN39k

[6] http://www.youtube.com/watch?v=bMKcpM4znJI

[7] git@github.com:canoo/DolphinJumpStart.git

[8] http://people.canoo.com/mittie/dolphin.mov



JavaFX auf dem Raspberry Pi und BeagleBoard xM

Just for the hack of it

Seit der JavaOne 2012 steht JavaFX auf dem BeagleBoard xM und seit Dezember auch auf dem Raspberry Pi zur Verfügung. Da stellt sich die Frage: Was kann man mit einem Desktop-UI-Framework auf diesen Geräten anstellen?

von Gerrit Grunwald

Oktober 2012, JavaOne, Technical Keynote: Richard Bair und Jasper Potts stehen auf der Bühne, und alle warten auf die neuesten Errungenschaften des Desktopteams. Doch was dann kommt, hat irgendwie niemand erwartet. Die beiden präsentieren einen Informationskiosk, wie man ihn von Messen und Ähnlichem kennt, auf dem JavaFX läuft. OK, so weit, so gut. Das wirklich Interessante daran ist aber, dass JavaFX dort nicht auf einem PC läuft, sondern auf einem Embedded Device (in diesem Fall einem PandaBoard). Dies sind Geräte, die meistens lediglich aus einer kleinen Platine bestehen, an die bestenfalls noch ein Touchdisplay angeschlossen ist. Die Leistungsfähigkeit dieser Geräte ist ebenfalls sehr eingeschränkt, was aber im Regelfall kein Problem darstellt, da dort Applikationen zum Einsatz kommen, die maßgeschneidert für ein bestimmtes Problem sind. Aber zunächst einmal ein kurzer Abriss über JavaFX Embedded.

JavaFX Embedded

JavaFX wird seit JDK 7 Update 6 zusammen mit dem JDK ausgeliefert und ist im Prinzip nichts anderes als eine Grafikbibliothek (*jfxrt.jar*), die zu den Projekten hinzugelinkt wird (ähnlich wie Swing in seiner Frühzeit). Die Hardware auf den bereits erwähnten Geräten basiert oft auf der ARM-Architektur (Advanced RISC Machines), was dazu führte, dass Oracle das JDK inklusive JavaFX auf diese Plattform portierte. Im Prinzip handelt es sich

bei JavaFX für ARM um ein Subset von JavaFX für den Desktop. In der ARM-Version fehlen folgende Dinge:

- Swing/SWT
- Systemmenü
- Drag and Drop
- Web-View
- Media (z. B. AudioClip)

Allerdings gibt es, wie auch in der Desktopvariante, hardwarebeschleunigte Grafik (soweit ein unterstützter Grafikprozessor vorhanden ist). Hierbei kommt OpenGL zum Einsatz, und es wird direkt in den Framebuffer geschrieben. Das bedeutet, man schreibt die Anwendungen nicht, damit sie unter X11 laufen – obwohl das teilweise auch funktioniert. Die typischen Einsatzgebiete von JavaFX auf Embedded Hardware könnten damit zum Beispiel die folgenden sein:

- Heimautomatisierung
- Media Center
- Medizinische Geräte
- Informationskiosk
- Schule/Ausbildung

Die Software

Interessanterweise hat Oracle nicht nur eine JDK-7-Version mit JavaFX 2 auf die ARM-Plattform portiert, sondern auch eine JDK-8-Version mit JavaFX 8. Dies ist insbesondere deshalb erwähnenswert, da es in JavaFX 8 noch einmal Änderungen geben wird, die nicht abwärtskompatibel sind. Somit kann man also schon jetzt zukunftssicher auch für Embedded Hardware entwickeln. Sobald man sich in die Embedded-Welt begibt, wird man auf einmal mit Begriffen konfrontiert, die einem als Desktop- oder Serverentwickler meistens unbekannt sind. Dazu zählen zum Beispiel auch *Softfloat* und *Hardfloat*. Softfloat bedeutet in diesem Fall, dass Fließkomma-Operationen in Software stattfinden, wohingegen Hardfloat bedeutet, dass eben diese Fließkomma-Operationen auf spezieller Hardware berechnet

Embedded Experience Day auf der JAX

Besuchen Sie auch Gerrit Grunwalds Session „JavaFX auf Embedded Hardware“ auf der diesjährigen JAX! Oder eine der anderen Sessions, die im Rahmen des „Embedded Experience Day“ stattfinden. Dieser brandneue Special Day beleuchtet unterschiedliche Facetten eingebetteter Softwareentwicklung innerhalb, aber auch außerhalb der Java-Welt. Gadgets, Demos und Hands-on-Entwicklung dürfen bei dieser „Embedded Experience“ natürlich nicht fehlen, und so wird es neben den technischen Sessions auch eine Embedded-Werkstatt geben. Alle Sessions des Special Days finden Sie auf der Website: http://jax.de.

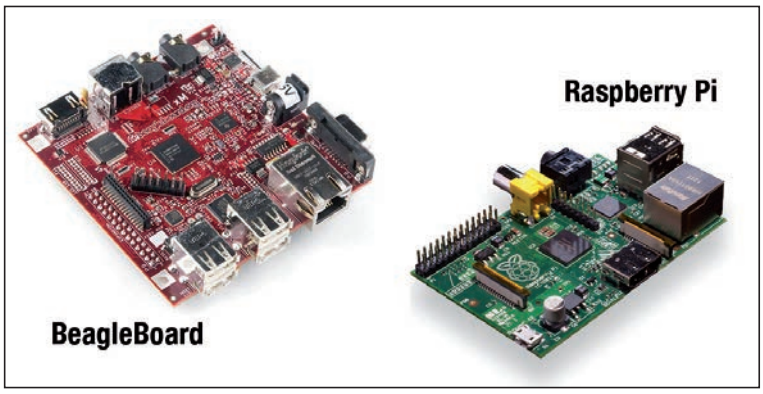


Abb. 1: BeagleBoard xM vs. Raspberry Pi

werden. Die Berechnung in Hardware ist im Regelfall schneller (teilweise um mehr als 20 Prozent). Wichtig ist die Unterstützung von Soft- bzw. Hardfloat schon für die Wahl des Betriebssystems: Für den Raspberry Pi gibt es eine Linux-Distribution mit Hardfloat-Unterstützung, wohingegen für das BeagleBoard xM eine Linux-Distribution mit Softfloat-Unterstützung zum Einsatz kommt.

Die Hardware

Zurzeit gibt es die folgenden zwei Plattformen, auf denen JavaFX ARM unterstützt wird:

- BeagleBoard xM
- Raspberry Pi

Dabei läuft auf dem BeagleBoard xM das JDK 7 [1] mit Softfloat-Unterstützung und auf dem Raspberry Pi JDK 8 [2] mit Hardfloat-Unterstützung. Damit man eine bessere Vorstellung davon bekommt, über welche Hardware wir hier sprechen, gibt Tabelle 1 einen kleinen Überblick.

BeagleBoard xM	Raspberry Pi
ARM A8	ARM V6
1 GHz	700 MHz (übertaktbar)
512 MB Ram	512 MB Ram (Model B)
Power VR SGX ser. 5	Broadcom VideoCore IV
4 USB Ports	2 USB Ports
Ethernet RJ45	Ethernet RJ45
HDMI out	HDMI out, composite out
Audio in/out	Audio out
I²C, JTAG, SPI	GPIO, I²C, UART, SPI
ca. 130 Euro	ca. 35 Euro

Tabelle 1: Hardwareübersicht

Klasse	Mindestgeschwindigkeit
2	2 MB/sec
4	4 MB/sec
6	6 MB/sec
10	10 MB/sec

Tabelle 2: SD-Karten-Übersicht

Obwohl der Raspberry Pi dem BeagleBoard xM unterlegen sein sollte, schlägt er sich wacker, und durch die Möglichkeit, den Pi auf bis zu 1 GHz zu übertakten, kann man ihn als gleichwertig ansehen. Zudem scheint der Broadcom VideoCore IV auf dem Raspberry Pi etwas leistungsfähiger zu sein als der Power VR des BeagleBoard xM, da er selbst mit einer Auflösung von 1 920 x 1 200 Pixeln noch zurechtkommt. Beim BeagleBoard xM indes ist bei 1 280 x 1 024 Pixeln Schluss. Herkömmliche Festplatten hat man auf diesen Boards nicht zur Verfügung. Stattdessen verwendet man handelsübliche microSD- und SD-Cards. Dabei sollte man darauf achten, dass man auf jeden Fall eine Karte der Geschwindigkeitsklasse 10 einsetzt. Tabelle 2 zeigt die Übertragungsgeschwindigkeiten der einzelnen Klassen an, die garantiert sind.

Durch den Einsatz von SD-Karten hat man zwar nicht gerade die schnellste IO-Performance, aber eine große Flexibilität, was einem Wechsel des Betriebssystems oder Einsatzzwecks entgegenkommt. Bei der Jfokus 2013 in Stockholm im Februar hatte ich beispielsweise das Problem, dass mein Raspberry Pi nicht lief. Glücklicherweise hatte ein Zuhörer aus dem Publikum auch einen Raspberry Pi dabei. Wir stöpselten nur kurz die Stromversorgung an, steckten meine SD-Karte in seinen Raspberry Pi, und ich konnte meine Demo fortführen.

Möchte man sich nun einen Raspberry Pi oder ein BeagleBoard xM zulegen, so findet man recht detaillierte Anweisungen zur Einrichtung dieser Systeme auf den Websites von Oracle [1], [2]. Wobei die Installation von Java auf der Embedded Hardware das geringste Problem ist, da man hier lediglich das Archiv von der Oracle-Seite herunterladen und auf dem Gerät entpacken muss. Man sollte jedoch nicht ganz unbewandert im Umgang mit Linux sein, da man sich oft auf der Konsole wiederfindet.

Der Workflow

Möchte man nun etwas in JavaFX für seinen Raspberry Pi oder sein BeagleBoard xM entwickeln, so ist dies denkbar einfach: Man verwendet einfach seine Lieblings-IDE oder seinen Lieblingseditor auf dem Desktop und erstellt ein Java-Programm. Aus diesem erzeugt man dann eine *.jar*-Datei, die man nun lediglich auf den Raspberry Pi bzw. das BeagleBoard xM kopieren muss (z. B. mittels des Shell-*scp*-Kommandos). Dort lässt sich dann das Ganze auf der Konsole starten. Es sei darauf hingewiesen, dass es Benutzer von Linux und Mac betriebssystembedingt (weil Unix-basiert) einfacher haben, da ja auf den Embedded-Geräten ebenfalls ein Linux läuft. Klingt einfach und ist es auch. Aber man darf nicht vergessen: Obwohl wir hier mit Java arbei-

ten, gilt nicht „Write once, run anywhere“. Das hat folgende Gründe: Zum einen ist die Hardware in ihrer Leistungsfähigkeit sehr beschränkt (wie bereits aus Tabelle 1 erkenntlich ist), und das betrifft nicht nur die CPU, sondern auch entscheidend den zur Verfügung stehenden Arbeitsspeicher und die Leistungsfähigkeit der GPU. Wir reden hier von Single-Core CPUs, 512 MB RAM und einer Single-Core GPU. Dem gegenüber stehen auf einem Desktoprechner Multi-Core CPUs, RAM im Gigabytebereich und Multi-Core GPUs zur Verfügung.

Dazu kommt nicht selten die Beschränkung in der nutzbaren Bildschirmfläche. Man kann zwar den Raspberry Pi auch an einen HD-Fernseher anschließen, aber das ist nicht gerade der Standard. Somit muss man davon ausgehen, dass man im Regelfall mit einem relativ kleinen Touchscreen zu tun hat. Das alles bedeutet, dass die Entwicklung von Software für Embedded Hardware eher der Entwicklung für mobile Endgeräte (mit ihren Touch-Controls und reduzierter Bildschirmgröße) gleicht als der Entwicklung für Desktopsysteme. Allerdings lässt sich jede Menge Code wiederverwenden, vorausgesetzt, man sorgt für eine saubere Trennung von Logik und User Interface. In diesem Fall braucht man in der Regel lediglich einen neuen UI-Layer zu programmieren.

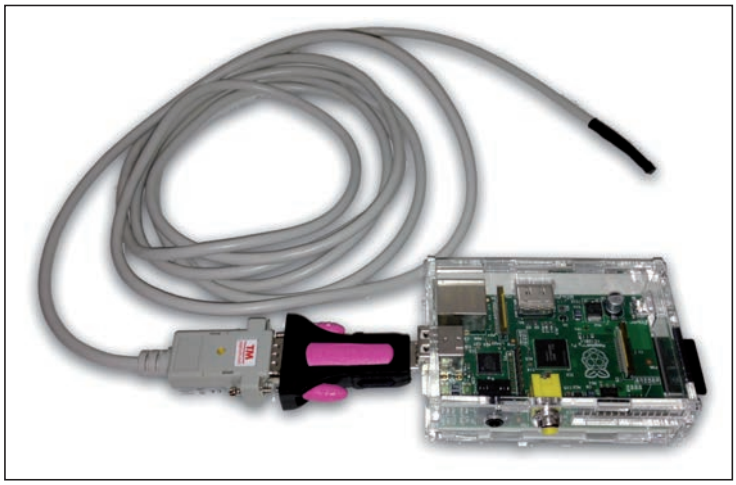


Abb. 2: Raspberry Pi zur Temperaturüberwachung

Temperaturüberwachung als Beispiel

Ein klassisches Einsatzgebiet für Embedded Hardware ist die Temperaturüberwachung. Zur Realisierung eines solchen Projekts benötigt man lediglich die Embedded Hardware (einen Raspberry Pi in diesem Fall) inklusive Netzteil, einen Sensor, den man von Java aus ansteu-

Die Ansteuerung von Hardware ist unter Java nicht so einfach, wie man sich das wünschen würde.

ern kann und eine Netzwerkverbindung. Das Ganze ist für 80 bis 100 Euro zu haben und somit auch für ein einfaches Hobbyprojekt durchaus erschwinglich. Die Idee ist es, den Raspberry Pi kontinuierlich (Intervall von zehn Sekunden) die Temperatur messen zu lassen und bei Überschreiten eines Grenzwerts eine Nachricht auf der Konsole auszugeben. Das ist natürlich nicht besonders spektakulär und nutzt kaum Eigenschaften von JavaFX aus. Es sollte jedoch trotzdem reichen, um einen Eindruck zu erlangen. Jetzt gibt es immer wieder Leute, die sofort aufschreien und behaupten, ein Raspberry Pi sei für solch einen Einsatzzweck der totale „Overkill", das könne man doch auch mit ganz einfacher Elektronik realisieren. Natürlich kann man das. Aber im Regelfall ist der Durchschnittsprogrammierer nicht daran interessiert, einen Lötkolben in die Hand zu nehmen und Schaltpläne zu studieren. Aus diesem Grund werde ich hier eine Plug-and-Play-Lösung vorstellen, die ganz ohne den Einsatz von Lötkolben auskommt.

Listing 1

```
public class SerialComm {
  public static final String     PI_PORT     = "/dev/ttyUSB0";
  private static final int       TIME_OUT    = 5000;
  private static final int       DATA_RATE   = 9600;
  private static final Pattern   PATTERN     = Pattern.compile("-?[\\d\\.]+");
  private static final Matcher   MATCHER     = PATTERN.matcher("");
  private DoubleProperty         celsius;
  private CommPort               commPort;
  private InputStream            inputStream;
  private BufferedReader         portReader;

  public SerialComm() {
    celsius = new SimpleDoubleProperty(0);
    connect(PI_PORT);
  }

  private void connect(final String PORT_NAME) {...}

  private void readSerial() {
    try {
      if (inputStream.available() > 0) {
        MATCHER.reset(portReader.readLine());
        while(MATCHER.find()) {
          celsius.set(Double.parseDouble(MATCHER.group()));
        }
      }
    } catch (IOException exception) {}
  }

  public ReadOnlyDoubleProperty celsiusProperty() { return celsius; }
}
```

Die Ansteuerung von Hardware ist unter Java leider nicht ganz so einfach, wie man sich das wünschen würde. Somit muss man erst einmal einen Temperatursensor auftreiben, der sich ansteuern lässt. Allerdings gibt es mittlerweile auch eine sehr schöne Open-Source-Bibliothek namens Pi4J [3], die die Ansteuerung der Raspberry Pi Hardware Ports (GPIO, I²C, RS232, SPI) wesentlich vereinfacht.

Ich habe mich im Rahmen dieses Beispiels für das TM-RS232-Thermometer von Papouch [4] entschieden, da es zum einen relativ günstig ist (um die 20 Euro) und zum anderen über eine RS232-Anbindung verfügt. Ein weiterer Vorteil dieses Sensors ist seine Funktionsweise: Sobald er ein DTR-Signal via RS232 erhalten hat, misst er automatisch die Temperatur in einem Intervall von zehn Sekunden und gibt den Messwert im Klartext als String via RS232 aus. Somit muss man lediglich dafür sorgen, dass man die RS232-Schnittstelle auslesen kann (in diesem Fall verwende ich *RXTX.jar* dafür). Damit man diesen Sensor mit der seriellen Schnittstelle nutzen kann, benötigt man nun noch einen RS232-auf-USB-Umsetzer, der einen so genannten FTDI-Chip verwendet. Dieser wird auf Linux unterstützt. Solch einen Adapter erhält man auch für ca. 20 Euro. Hat man nun noch einen Raspberry Pi und eine Stromversorgung – im Notfall reichen ein Y-USB-Kabel und ein aktiver USB-Hub dazu aus –, ist die Hardwareausstattung komplett.

Damit das Ganze softwareseitig auch läuft, benötigen wir noch die Unterstützung auf der Seite des Raspberry Pi. Dies lässt sich durch folgendes Kommando bewerkstelligen:

```
sudo apt-get install librxtx-java
```

Nachdem diese Hürde genommen ist, benötigen wir noch die Datei *RXTXcomm.jar*, die man unter [5] herunterladen kann, bevor wir loslegen können. Das Projekt wird aus zwei einfachen Klassen bestehen, einer *Main*- und einer *SerialComm*-Klasse. Listing 1 zeigt die *SerialComm*-Klasse.

Die Funktionsweise ist folgendermaßen: Zunächst wird eine Verbindung zum Serial Port hergestellt, und dann wird ein *SerialPortEventListener* an diesen seriellen Port gehängt. Jedes Mal, wenn der Sensor jetzt seine Temperatur auf den seriellen Port schreibt, wird die Methode *readSerial()* aufgerufen, die mittels eines *BufferedReaders* eine Zeile einliest und aus dieser Zeile dann mithilfe eines regulären Ausdrucks den aktuellen Temperaturwert extrahiert. Dieser Wert wird dann in die *celsius-DoubleProperty* geschrieben. JavaFX hat also nicht nur grafische Nettigkeiten zu bieten, sondern auch andere nützliche Dinge wie z. B. Properties. An eben die-

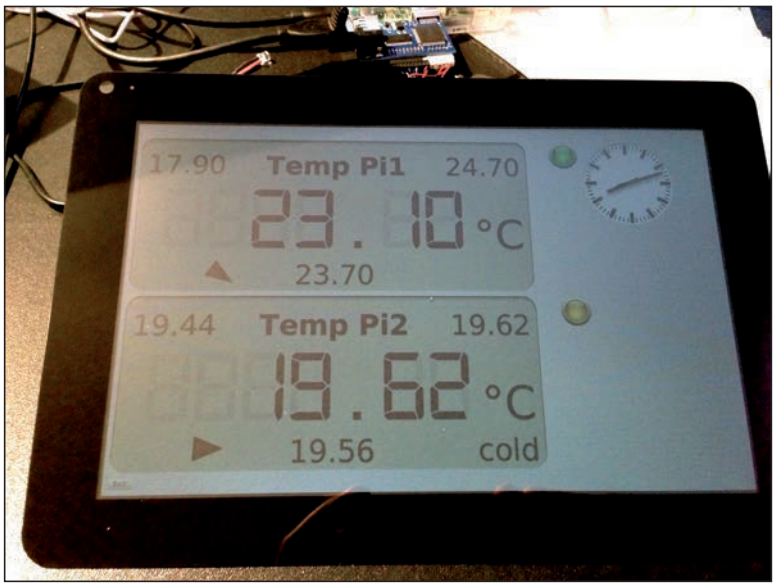


Abb. 3: JavaFX-Anwendung auf Raspberry Pi

se Properties kann man sich nun mit einem so genannten *InvalidationListener* oder *ChangeListener* hängen, um über Änderungen informiert zu werden. Gut, das hätte man auch alles mit einem guten alten *PropertyChangeListener* realisieren können, aber warum sollte man mehr Code schreiben als nötig?

Die zweite Klasse, die noch zum Einsatz kommt, ist die *Main*-Klasse mit dem Inhalt aus Listing 2.

Listing 2

```
public class Main {
  private static final double THRESHOLD = 24.0;
  private SerialComm sensor;

  public Main() {
    sensor = new SerialComm();
    initSensor();
  }

  private void initSensor() {
    sensor.celsiusProperty().addListener(new ChangeListener<Number>() {
      @Override public void changed(ObservableValue<? extends Number> ov,
                                    Number oldValue, Number newValue) {
        if (newValue.doubleValue() > THRESHOLD) {
          System.out.println("Temperature (" + newValue.toString() +
                             ") exceeds threshold of " + THRESHOLD + " °C !");
        }
      }
    });
  }

  public static void main(String[] args) {
    Main app = new Main();
  }
}
```

Diese Klasse ist sehr einfach und macht eigentlich nichts anderes als die *SerialComm*-Klasse zu instanziieren und einen *ChangeListener* an die *celsiusProperty* der *SerialComm*-Klasse zu hängen. In diesem Fall verwenden wir einen *ChangeListener*, da dieser nur dann getriggert wird, wenn sich der Wert wirklich geändert hat. Soll heißen: Solange der Messwert konstant bleibt, wird auch die *changed()*-Methode des *ChangeListeners* nicht aufgerufen. Möchte man über jede Änderung des Werts informiert werden, auch wenn der aktuelle Wert der gleiche ist wie der vorherige, so lässt sich dies mit einem InvalidationListener bewerkstelligen. Der Code dieses Beispiels steht auf GitHub und kann unter [6] heruntergeladen werden. Natürlich kann man die Ausgabe auf der Konsole nur dann sehen, wenn man den Raspberry Pi an einen Fernseher oder Monitor anschließt, was nicht wirklich die Möglichkeiten von JavaFX auf Embedded Hardware aufzeigt. Damit man sieht, dass man aber auch durchaus schicke grafische Ausgaben mit einem Raspberry Pi und JavaFX realisieren kann, sieht man in **Abbildung 3** eine Anwendung, die ich ebenfalls zur Temperaturüberwachung geschrieben habe. Diese Anwendung läuft auch auf einem Raspberry Pi und zeigt mehrere JavaFX Controls auf einem angeschlossenen 10-Zoll-LCD-Touchscreen.

Fazit

Der Einsatz von Java und JavaFX auf Embedded Hardware wie dem Raspberry Pi oder dem BeagleBoard xM ist durchaus sinnvoll. Aber was meines Erachtens noch viel wichtiger ist: Es macht enorm viel Spaß. Man fühlt sich ein wenig zurückversetzt in die Tage, als Computer noch mit Kassettenrekordern als Speichermedium und Fernsehern als Monitor betrieben wurden. In diesem Sinne: Keep coding!

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

Links & Literatur

[1] jdk7.java.net/fxarmpreview

[2] jdk8.java.net/fxarmpreview

[3] pi4j.com

[4] papouch.com

[5] rxtx.qbang.org/wiki/index.php/Download

[6] github.com/HanSolo/pitemp

JavaFX mit selbst entwickelter Hardware ergänzen

Himbeerkuchen mit Kaffee

Schon einmal darüber nachgedacht, selber ein Hardware-Gadget zu bauen? Sei es für eine Bastelei oder eine innovative Idee für ein Kundenprojekt: Manchmal wäre es schon etwas Feines, die eigene Software mit ein paar Handgriffen an ungewöhnliche Hardware jenseits von USB koppeln zu können.

von Thomas Scheuchzer

„Arduino!“ werden jetzt viele rufen. Ja, die Arduino-Plattform [1] ist seit vielen Jahren eine Lösung für solche Herausforderungen. Leider muss man als Java-Entwickler für Arduino zuerst eine neue Programmiersprache erlernen. Und wie verpasse ich meinem Arduino ein nettes GUI?

Seit letztem Jahr macht der Minicomputer Raspberry Pi dem Arduino Konkurrenz. Für gerade einmal 35 US-Dollar erhält man einen kompletten 700-MHz-PC inklusive 512 MB RAM, Netzwerk, USB und HDMI. Neben dieser Standardausstattung werden noch einige „General Purpose Input/Output“-(GPIO-)Pins bereitgestellt. Diese Pins lassen sich, genau wie bei einem Arduino, beliebig für IO programmieren. Die für Raspberry Pi zur Verfügung stehenden Betriebssysteme sind meist **nix*-basierend. Somit lässt sich mit Package-Managern Software installieren, wie auf einem normalen Desktop auch. Damit nicht genug: Oracle hat den Raspberry Pi zur Referenzplattform [2] von Java 8 (inkl. JavaFX) für ARM-Prozessoren erkoren! Ich finde, das alles ist Grund genug, sich auch als Java-Entwickler einmal einen Schritt näher an die Hardware zu wagen und so den eigenen Horizont zu erweitern.

Der Raspberry-FX-Button

Im Rahmen dieses Artikels wollen wir nun zusammen ein Hello World Gadget erstellen. Zu diesem Zweck binden wir einen externen Taster an einen Full-HD-Bildschirm. Dazwischen packen wir einen Raspberry Pi, der den Status des Tasters ausliest und in einer JavaFX-Applikation darstellt. Wieso einen Full-HD-Bildschirm? Weil es so mehr Spaß und Eindruck macht! Da wir jetzt ein Stück Hardware entwickeln, benötigen wir einige Bauteile. Einige davon habe ich in meinen alten Arduino-Bausätzen gefunden:

- 1 Steckplatine (Breadboard)
- 1 Widerstand 10 kOhm (Braun-Schwarz-Orange-Gold)
- 1 Push-Button
- ein paar Verbindungskabel

Zudem sollten wir natürlich einen Raspberry Pi (Model B) sowie einen Bildschirm, eine kabelgebundene USB-Maus und -Tastatur sowie ein Netzwerkkabel bzw. einen USB-WiFi-Adapter griffbereit haben. Funktastaturen bzw. -mäuse bereiten JavaFX nach meinen Erfahrungen Probleme. Mit den erwähnten Bauteilen stecken wir uns die in **Abbildung 1** gezeigte Schaltung zusammen.

Gleich vorweg: Ich bin kein ausgebildeter Elektrotechniker, und ziemlich sicher gibt es bessere und sicherere Schaltungen. Aber da die aufgezeigte Schaltung simpel ist und trotzdem funktioniert, begnügen wir uns damit. Grundsätzlich platzieren wir unseren Taster

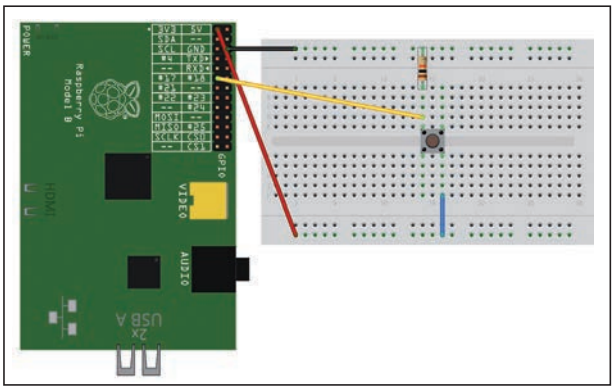


Abb. 1: Der Steckplan für unser Gadget

Listing 1

```
<dependency>
  <groupId>com.pi4j</groupId>
  <artifactId>pi4j-core</artifactId>
  <version>0.0.5-SNAPSHOT</version>
</dependency>
<dependency>
  <groupId>com.oracle</groupId>
  <artifactId>javafx-runtime</artifactId>
  <version>2.0</version>
  <scope>system</scope>
  <systemPath>${java.home}/lib/jfxrt.jar</systemPath>
</dependency>
```

Listing 2

```
<repositories>
  <repository>
    <id>oss-snapshots-repo</id>
    <name>Sonatype OSS Maven Repository</name>
    <url>https://oss.sonatype.org/content/groups/public</url>
    <snapshots>
      <enabled>true</enabled>
      <updatePolicy>always</updatePolicy>
    </snapshots>
  </repository>
</repositories>
```

zwischen der 3,3-V-Stromleitung und der Masse. Nach dem Taster führt eine Leitung zu einem Pin, an dem wir den Zustand ablesen können. Es ist unbedingt darauf zu achten, dass die 3,3-V-Leitung verwendet wird. Eine höhere Spannung kann den Raspberry Pi zerstören!

Kaffee zum Kuchen

Der Raspberry Pi wird ohne Betriebssystem ausgeliefert. Im Netz finden sich bereits verschiedene Imagedateien, die dem Raspberry Pi über eine SD-Karte Leben einhauchen. Wie das funktioniert, wird im Folgenden beschrieben. Sämtliche Befehle in diesem Artikel beziehen sich auf das Betriebssystem Ubuntu.

Für unser Vorhaben verwenden wir Raspbian, einen Debian-Klon speziell für den Raspberry Pi. Nach dem Download von [3] entpacken wir die Zip-Datei und erhalten unser Betriebssystem als *IMG*-Datei. Mittels des Programms ImageWriter schreiben wir das Image auf die SD-Karte. Bei Bedarf kann ImageWriter mit *sudo apt-get install usb-imagewriter* installiert werden.

Nun noch Bildschirm, Netzwerkkabel, Maus und Tastatur an den Raspberry Pi anschließen und Letzteren über ein USB-Kabel mit Strom versorgen. Beim ersten Start landen wir automatisch im *Raspi-config*. Hier kann das System konfiguriert werden. Über *sudo raspi-config* können wir später wieder hierher zurückkehren. Für unser Beispiel müssen wir in *Raspi-config* die in Tabelle 1 gezeigten Anpassungen vornehmen.

Die anderen Menüpunkte sind für unsere aktuellen Bedürfnisse nicht relevant. Eine kleine Warnung für Tüftler: Wenn im *overclock*-Menü eine Warnung erscheint, dass Benutzer den Totalverlust von SD-Karten gemeldet hätten, so ist diese Warnung durchaus ernst zu nehmen. War wirklich schade um meine brandneue Class-10-32-GB-Karte. Ich arbeite nun nur noch mit billigen 4-GB-Karten.

Nach Verlassen von *Raspi-config* und einem allfälligen Reboot befinden wir uns entweder am Log-in-Prompt oder auf dem Desktop. In diesem Schritt wollen wir nun die IP-Adresse in Erfahrung bringen, damit wir uns anschließend von Remote mit dem Raspberry Pi verbinden können. Benutzer des WiFi-Adapters sollten nun einen Internetzugang mittels *WiFi Config* herstellen und sich die zugewiesene IP merken. Wer am Netzwerkkabel hängt und den Desktop nicht gestartet hat, loggt sich mit dem User *pi* und Passwort *raspberry* ein und bekommt seine IP mittels *ifconfig* angezeigt.

Ab diesem Zeitpunkt können wir uns zurück in unsere gewohnte Entwicklungsumgebung begeben und uns fortan über *scp* auf den Raspberry Pi einloggen.

```
> ssh-copy-id pi@192.168.1.100
> ssh pi@192.168.1.100
```

Raspbian kennt Java 8 noch nicht. Darum installieren wir die aktuelle Previewversion nach dem Einloggen über *ssh* mit den Befehlen:

```
> wget http://www.java.net/download/JavaFXarm/jdk-8-ea-b36e-linux-arm-
                                                    hflt-29_nov_2012.tar.gz
> tar xvzf ~/jdk-8-ea-b36e-linux-arm-hflt-29_nov_2012.tar.gz
> sudo mkdir -p -v /opt/java
> sudo mv -v ~/jdk1.8.0 /opt/java
> sudo update-alternatives --install "/usr/bin/java" "java"
                                                "/opt/java/jdk1.8.0/bin/java" 1
> sudo update-alternatives --set java /opt/java/jdk1.8.0/bin/java
> java -version
```

Menüpunkt	Änderung
memory_split	Oracle empfiehlt, der Grafikkarte 128 MB zur Verfügung zu stellen, also setzen wir den Wert auf 128.
ssh	Um später von unserer Entwicklungsmaschine auf den Raspberry Pi zuzugreifen, aktivieren wir den SSH-Server.
boot_behaviour	Wer einen USB-WiFi-Adapter verwendet, sollte hier einmalig den Desktop aktivieren. Mit dem Tool WiFiConfig lässt sich die Wireless-Verbindung bequem einrichten. Für JavaFX muss der Desktop nicht gestartet werden, da Java direkt in den Framebuffer schreibt. Der Boot-Vorgang wird durch die Deaktivierung des Desktops verkürzt.

Tabelle 1: Menüpunkte, die beim ersten Start des Betriebssystems bearbeitet werden sollten

Java 8 ist jetzt aktiv. Der letzte Befehl sollte *1.8.0-ea* als Teil der Ausgabe liefern.

Um die zusätzlichen GPIO-Pins des Raspberry Pis aus Java heraus ansprechen zu können, werden wir Pi4J verwenden. Dieses Projekt ist sehr jung und noch nicht ganz ausgereift. Leider funktioniert die aktuelle Version 0.0.4 nicht zusammen mit der aktuellen Raspbian-Version. Der 0.0.5-SNAPSHOT Release hingegen schon. Diesen können wir mit folgenden Befehlen nachinstallieren:

```
> wget http://pi4j.googlecode.com/files/pi4j-0.0.5-SNAPSHOT.deb
> sudo dpkg -i pi4j-0.0.5-SNAPSHOT.deb
```

Das wars vorerst für den Raspberry Pi. Wir wenden uns von ihm ab und entwickeln unseren Code komplett auf unserer Entwicklungsmaschine. Der Raspberry Pi ist für Eclipse und Co. einfach zu schwach. Auch fördert dieses Vorgehen eine saubere Trennung von GUI und der Hardware, und auch Build-Server kommen bei Bedarf nicht zu kurz.

GPIOs und Java

Wir erstellen ein neues Maven-Projekt und fügen die Abhängigkeiten zu Pi4J sowie JavaFX hinzu (Listing 1).

Da wir mit der Snapshot-Version von Pi4J arbeiten, müssen wir auch das Snapshot Repository bekannt machen (Listing 2).

Die JavaFX-Abhängigkeit kann nicht aus einem Maven Repository geladen werden. Darum verweisen wir auf unsere bestehende Java-Installation. Für unser Programm genügt übrigens eine aktuelle Java-7-Installation, falls es Java 8 noch nicht auf Ihre Maschine geschafft hat. Danach bereiten wir unser Projekt mit *mvn eclipse:eclipse* für Eclipse vor.

Gemäß dem Schaltplan haben wir unseren Taster über das gelbe Kabel an den Pin #17 angeschlossen. Für die Kommunikation mit Pin #17 verwenden wir Pi4J. Leider verwendet diese Bibliothek eine andere Nummerierung. Gemäß [4] entspricht der verwendete Pin dem Pi4J Pin 0.

Unsere Applikation soll den Status des Tasters ausgeben. Dazu teilen wir den Code in die Hardwareanbindung und die Ausgabe auf. Da wir Pi4J mangels GPIO-Pins auf der Entwicklungsmaschine nicht testen können, empfiehlt es sich, hier ein Java-Interface (z. B. *HwController*) zu erstellen, damit wir zur Entwicklungszeit die GPIO-Pins einfach simulieren können. Um unseren Beispielcode kurz zu halten, verzichten wir aber jetzt bewusst darauf. Der Status des Tasters wird innerhalb der Applikation in einem Model gehalten. Für unser extrem einfaches Model begnügen wir uns mit der Klasse *BooleanProperty* aus JavaFX.

Das Schöne an Pi4J ist, dass es unter anderem auch Listeners anbietet. Anstelle eines Pollings auf unseren Pin werden wir über Statusveränderungen informiert. Die Interaktion mit den Hardware-Interrupts übernimmt Pi4J. Wir Softwareentwickler fühlen uns also sofort wieder heimisch, und die Hardware ist schon beinahe vergessen. Die Implementierung der Hardwarekommunikation gestaltet sich wie in Listing 3.

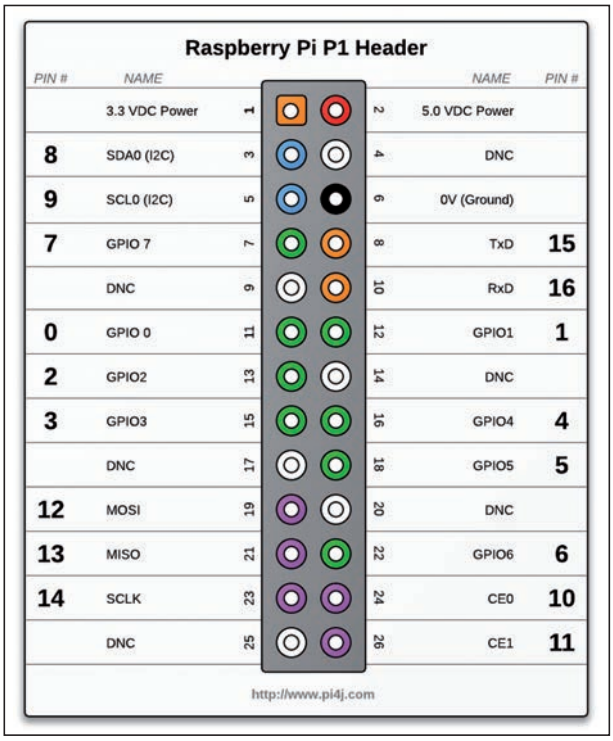


Abb. 2: Die Nummerierung der GPIO-Pins (Quelle: pi4j.com)

Unser *Pi4JController* initialisiert im Konstruktor die benötigten GPIO Pins. Im Speziellen wird der Pin 00 als digitaler Input-Pin aufgeschaltet. Anschließend registrieren wir auf dem Pin einen Listener, der bei einer Statusänderung unser Model aktualisiert. *PinState.HIGH* bedeutet, dass Strom fließt bzw. der Taster gedrückt wurde.

Listing 3

```
public class Pi4JController {
  private BooleanProperty model;
  private GpioController gpio;
  public Pi4JController() {
    this.gpio = GpioFactory.getInstance();
    GpioPinDigitalInput input = gpio.provisionDigitalInputPin(RaspiPin.GPIO_00,
                                                              PinPullResistance.PULL_DOWN);
    input.addListener(new GpioPinListenerDigital() {
      @Override
      public void handleGpioPinDigitalStateChangeEvent(
          GpioPinDigitalStateChangeEvent event) {
        model.set(PinState.HIGH.equals(event.getState()));
      }
    });
  }
  public void setModel(BooleanProperty model) {
    this.model = model;
  }
}
```

Damit ist der hardwarenahe Teil bereits abgeschlossen. Mit einem einfachen Programm wie dem in Listing 4 könnte nun der Controller bereits angesprochen werden.

Wie schon erwähnt, können wir dieses Programm nur auf dem Raspberry Pi ausprobieren. Da dies so simpel und nicht besonders beeindruckend ist, machen wir uns direkt an den JavaFX-Code.

Anders als mit Swing besteht in JavaFX die Möglichkeit, das GUI über XML zu definieren und mittels CSS zu stylen. Davon machen wir nun gleich Gebrauch, indem wir die Datei *Gui.css* im *main/resource*-Verzeichnis erstellen. Die Ressourcendateien sollten sich im gleichen Unterverzeichnis oder Package befinden wie anschließend der JavaFX-Code, allerdings im Ressourcenverzeichnis und nicht im Source-Verzeichnis.

Die Klassen *.On* und *.Off* in Listing 5 spiegeln den Status des Tasters wider. Wir ändern einfach das Hintergrundbild bei einem Statuswechsel. Die beiden Bilder werden im gleichen Verzeichnis wie *Gui.css* platziert. Das GUI selbst wird wie in Listing 6 in der Datei *Gui.fxml* definiert.

Listing 4

```
public class CliApp {
  public static void main(String[] args) throws InterruptedException {
  BooleanProperty model = new SimpleBooleanProperty();
    Pi4JController hwController = new Pi4JController();
    hwController.setModel(model);
    model.addListener(new ChangeListener<Boolean>() {
      @Override
      public void changed(ObservableValue<? extends Boolean> observable,
                                                Boolean oldValue, Boolean newValue) {
        System.out.println("Der Taster ist " + (newValue ? "an" : "aus"));
      }
    });
    while(true) {
      Thread.sleep(2000);
    }
  }
}
```

Listing 5

```
#Background {
  -fx-background-color: #18171d;
}
.Off {
  -fx-background-color: #18171d;
  -fx-background-image: url('panic_off.png');
  -fx-background-size: stretch;
}
.On {
  -fx-background-color: #18171d;
  -fx-background-image: url('panic_on.png');
  -fx-background-size: stretch;
}
```

In der GUI-Definition wird ein Button der Größe 400 x 400 Pixel in der Mitte des Bildschirms platziert. Als Startwert setzen wir die CSS-Klasse *Off*. Durch unsere *Style*-Klasse wird der Button nur als Bild und nicht als klassischer Button erscheinen. Mit *onMousePressed* und *onMouseReleased* registrieren wir noch zwei Events, damit wir die Statusänderung auch über die Maus auslösen können. Die beiden Events referenzieren Methoden in einem Controller, den wir gleich implementieren werden. Zum Schluss wird unter *stylesheets* unsere CSS-Definition bekanntgemacht.

Jetzt muss noch das Model mit dem JavaFX-Button verknüpft werden. Dies geschieht in einer *Controller*-Klasse, die in Listing 7 zu sehen ist. Im *GuiController* registrieren wir in der Initialisierungsmethode einen Listener, der die CSS-Klasse des Buttons entsprechend dem Modellzustand setzt. Durch die Annotation *@FXML* füllt uns JavaFX bereits die Referenz zum Button ab. Dabei muss die Member-Variable gleich heißen wie die *fx:id* in *Gui.fxml*.

Am Ende der Klasse finden wir ebenfalls die beiden *Event*-Methoden, die wir in *Gui.fxml* definiert haben. Der erwähnte Listener für die CSS-Updates gestaltet sich auch sehr einfach und lässt sich, wie Listing 8 zeigt, in wenigen Zeilen implementieren.

In der *changed*-Methode modifizieren wir einfach die Liste mit *Style*-Klassen unseres Buttons. Dabei ist zu

Listing 6

```
<?xml version="1.0" encoding="UTF-8"?>
  <?import java.net.*?>
  <?import javafx.scene.layout.*?>
  <?import javafx.scene.control.Button?>
  <AnchorPane id="Background" maxHeight="-Infinity"
          maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
    prefHeight="480" prefWidth="640" xmlns:fx="http://javafx.com/fxml">
  <children>
    <VBox spacing="20" alignment="CENTER" AnchorPane.
       bottomAnchor="213.0" AnchorPane.leftAnchor="168.0" AnchorPane.
                    rightAnchor="168.0" AnchorPane.topAnchor="214.0">
      <children>
        <HBox alignment="CENTER" minHeight="400.0">
          <children>
            <Button fx:id="button" prefWidth="400.0" prefHeight="400.0"
                                                              styleClass="Off"
              onMousePressed="#onMousePressed"
              onMouseReleased="#onMouseReleased"/>
          </children>
        </HBox>
      </children>
    </VBox>
  </children>
  <stylesheets>
    <URL value="@Gui.css" />
  </stylesheets>
</AnchorPane>
```

beachten, dass die Änderungen im JavaFX Application Thread durchgeführt werden. Dies wird durch *runLater* in der *Platform*-Klasse sichergestellt. Beim ersten Wechsel wird unser Raspberry Pi etwas Zeit benötigen, um unser Bild zu laden. Würde dies nicht im Application Thread vorsichgehen, so könnte ein sehr kurzer Klick auf unseren Hardware-Taster unser GUI durcheinander bringen, da sich die On- und Off-Events überholen könnten. Solche Effekte werden erst auf dem langsameren Endgerät richtig gut sichtbar.

Listing 7

```
public class GuiController implements Initializable {
  private BooleanProperty model;
  @FXML
  private Button button;
  public GuiController(BooleanProperty model) {
    this.model = model;
  }
  @Override
  public void initialize(URL url, ResourceBundle resourceBundle) {
  model.addListener(new OnOffCssChangeListener(button));
  }
  public void onMousePressed(MouseEvent event) {
    model.set(true);
  }
  public void onMouseReleased(MouseEvent event) {
    model.set(false);
  }
}
```

Listing 8

```
public class OnOffCssChangeListener implements ChangeListener<Boolean> {
  private static final String ON_CLASS = "On";
  private static final String OFF_CLASS = "Off";
  private Node node;
  public OnOffCssChangeListener(Node node) {
    this.node = node;
  }
  @Override
  public void changed(final ObservableValue<? extends Boolean>
              observable, final Boolean oldValue, final Boolean newValue) {
    Platform.runLater(new Runnable() {
      @Override
      public void run() {
      ObservableList<String> styleClass = node.getStyleClass();
        styleClass.remove(ON_CLASS);
        styleClass.remove(OFF_CLASS);
        styleClass.add(newValue ? ON_CLASS : OFF_CLASS);
      }
    });
  }
}
```

Unser GUI ist somit fertig und testbereit. Das GUI ohne Hardware kann über das Programm in Listing 9 auf dem Entwicklungsrechner gestartet werden.

Mausklicks auf das Zentrum des Bildschirms werden nun unser Bild gemäß CSS verändern. Im obigen Programm wird die Applikation in den Vollbildmodus geschaltet, weil JavaFX auf dem Raspberry Pi momentan nur so gestartet werden kann. Nun vereinigen wir das GUI mit der Hardware. Wir übernehmen den Code aus *GuiOnlyApp* und erstellen die richtige Applikation. Nach der *model*-Variable erstellen wir den Hardware-Controller:

```
public class App extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    BooleanProperty model = new SimpleBooleanProperty();
    Pi4JController hwController = new Pi4JController();
    hwController.setModel(model);
    ...
```

Packt man die Erstellung des Hardware-Controllers in eine Factory-Methode, so lassen sich später auch andere Implementierungen instanziieren, z. B. über ein System-Property. Damit lässt sich für Tests auch ein GPIO-Simulator an unser Programm hängen.

Deployment

Nun ist der große Auftritt für unseren Raspberry Pi gekommen. Wir kompilieren den Programmcode und transferieren ihn mittels *scp* in das *Home*-Verzeichnis des Pi-Benutzers auf unserem Raspberry:

```
> mvn clean install
> scp target/raspi-fx-button-1.0-SNAPSHOT.jar pi@192.168.1.100:/home/pi
```

Vermutlich sind wir über SSH noch mit dem Raspberry Pi verbunden. Im *Home*-Verzeichnis auf dem Raspberry

Listing 9

```
public class GuiOnlyApp extends Application {
  @Override
  public void start(Stage stage) throws Exception {
    BooleanProperty model = new SimpleBooleanProperty();
    FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource("Gui.fxml"));
    fxmlLoader.setController(new GuiController(model));
    Parent root = (Parent) fxmlLoader.load();
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setFullScreen(true);
  stage.show();
  }
  public static void main(String[] args) {
    launch(args);
  }
}
```

Abb. 3 und 4: Das JavaFX GUI in Aktion

Pi starten wir nun unsere Applikation. Dazu benötigen wir im Klassenpfad sowohl die JavaFX Runtime als auch die Pi4J-Bibliothek mit ihren Native Bindings:

```
> sudo java -Djavafx.platform=eglfb \
  -cp ./'*':/opt/java/jdk1.8.0/jre/lib/jfxrt.jar:/opt/pi4j/lib/'*' \
  de.javamagazin.raspifxbutton.App
```

Es dauert eine Weile, bis unser GUI erscheint. Nach dem Laden läuft das Programm aber flott. Ein Druck auf den Taster verändert nun die Ausgabe. Wie bereits erwähnt, wird das in CSS konfigurierte Bild erst bei der ersten Verwendung geladen, beim ersten Knopfdruck dauert das Update also einen Moment.

Zugriff auf die GPIO-Pins ist nur mit Root-Rechten möglich, weshalb wir den Java-Prozess mit *sudo* starten. Der Parameter *javafx.platform* ist unbedingt erforderlich, damit Java direkt in den Framebuffer schreibt. Fehlt er, wird sich Java mit einem *UnsatisfiedLinkError* melden. Bei der Verwendung einer veralteten Pi4J-Version wird ebenfalls ein Fehler auftreten, da die nativen Bindings nicht gefunden werden. Startet man die Applikation über die SSH-Konsole, kann die Applikation einfach über CTRL + C beendet werden. Direkt auf dem Raspberry Pi ist dies nicht möglich, wenn man ohne Desktop startet. Um noch etwas mehr Power aus dem Raspberry Pi zu holen, empfiehlt Oracle [5], in der Datei */boot/config.txt* die beiden Zeilen *framebuffer_width* und *framebuffer_height* zu aktivieren.

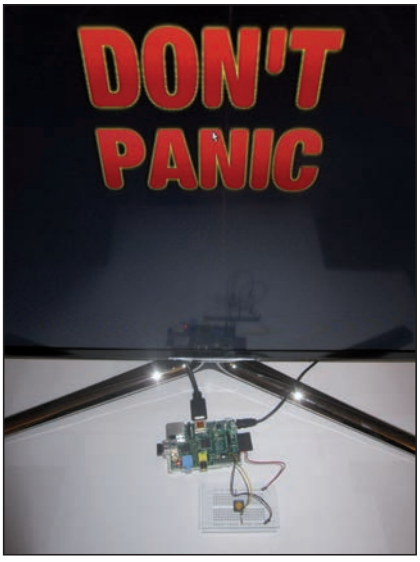


Abb. 5: Das komplette Set-up in Betrieb

Somit ist unser Hello World Gadget fertig! Der komplette Sourcecode ist unter [6] abrufbar.

JDK 8 für ARM

Das verwendete JDK ist noch kein vollkommener Release oder Release Candidate. Es handelt sich um eine Previewversion. Aus diesem Grund sind einige Features noch nicht implementiert. Einige fehlende sind unter [5] beschrieben, andere muss man selbst herausfinden. So werden Effekte wie *BoxBlur* noch nicht unterstützt. Auch CSS hält immer wieder Überraschungen für den Tüftler parat. Wie gesehen, ist auch Pi4J noch in der Entwicklung. Auch hier stolpert man über Fehler wie *ConcurrentModificationExceptions*, wenn man Listeners wieder löschen will.

Fazit

Mit diesem Beispiel wollte ich hardwareunerfahrenen Java-Entwicklern zeigen, wie einfach sich mit der Kombination aus Raspberry Pi, JavaFX und ein paar Elektronikbausteinen einfache Hardwareprototypen mit wenig Aufwand effektvoll umsetzen lassen. Auch wenn wir hier nur einen externen Taster abfragen – mit einer schönen Verpackung und einem 50-Zoll-Bildschirm wird unser Werk sehr schnell zum Hingucker.

Oracle zeigt mit der Ernennung des Raspberry Pi zur Referenzplattform, wohin die JavaFX-Reise gehen soll: hin zu kleinen Embedded-artigen Geräten mit vollem Funktionsumfang. Dazu gehören Waschmaschinen genauso wie Kioskapplikationen. Da lohnt es sich als Java-Entwickler auf jeden Fall, einen Blick über den Web-/JEE-Tellerrand zu werfen.

Thomas Scheuchzer ist Senior Software Engineer bei Zühlke Engineering AG, Schweiz, und beschäftigt sich seit über zehn Jahren mit dem Java-Universum, von JEE-Lösungen für Banken bis hin zu unkonventionellen Lösungsvorschlägen im Kuhstall.

thomas.scheuchzer@zuehlke.com

Links & Literatur

[1] http://arduino.cc/

[2] https://blogs.oracle.com/jtc/entry/a_raspberry_pi_javafx_electronic

[3] http://downloads.raspberrypi.org/images/raspbian/2012-12-16-wheezy-raspbian/2012-12-16-wheezy-raspbian.zip

[4] http://pi4j.com/usage.html

[5] http://jdk8.java.net/fxarmpreview/javafx-arm-developer-preview.html

[6] https://github.com/scheuchzer/raspi-fx-button

Quellcode zum Download

e(fx)clipse: JavaFX-Tooling für Eclipse

Eclipse meets JavaFX

Es tut sich was in der Eclipse-UI-Welt: Dank der reichhaltigen Möglichkeiten von e(fx)clipse zieht JavaFX mit Pauken und Trompeten in die bisher klar von SWT dominierte Eclipse-Welt ein – und erobert dabei die Herzen der Entwickler im Sturm.

von Marc Teufel

Vor noch gar nicht langer Zeit war die Welt der Entwickler von Java-basierten Rich Clients noch in Ordnung: Das GUI wurde entweder mit Swing oder SWT entwickelt. Fertig. Doch die Zeiten ändern sich. Insbesondere HTML5 brachte diese schöne heile Welt der Java-UI-Technologien durcheinander, denn früh stand fest: Swing und SWT können es mit der Konkurrenz nicht aufnehmen. Oracle reagierte darauf mit der Veröffentlichung von JavaFX 2 und positionierte es als Nachfolger von Swing. Diese neue Technologie hat bis auf den Namen nahezu nichts mehr mit dem Vorgänger JavaFX gemeinsam. Mit JavaFX 2 scheint es Oracle jedoch tatsächlich gelungen zu sein, eine neue Java-UI-Technologie auf dem Markt zu platzieren, die nicht nur einen großen Teil der Entwicklergemeinde fasziniert und mitreißt, sondern tatsächlich auch das Zeug hat, es mit der Konkurrenz aufzunehmen.

JavaFX 2 überzeugt nicht nur durch sein schickes und modernes Aussehen, sondern punktet vor allem mit seinem durchdachten Gesamtkonzept. Das fängt bei elementaren Dingen wie seiner Architektur an – also damit, wie man eine Anwendung mit JavaFX 2 aufbaut, geht weiter über das detaillierte und umfangreiche CSS Styling bis hin zu der Möglichkeit, mithilfe von FXML deklarative UIs bauen zu können. Während die NetBeans IDE schon bald mit einer ersten Unterstützung von JavaFX 2 punkten konnte, begann man bei Eclipse erst im Sommer letzten Jahres aktiv mit der Entwicklung eines entsprechenden Toolings. Das maßgeblich von der Firma BestSolution.at angetriebene Projekt e(fx)clipse [1] bringt dabei nicht einfach nur umfangreiches Tooling zum Entwickeln von JavaFX 2 nach Eclipse. Es geht noch einen Schritt weiter, indem es JavaFX selbst erweitert (Stichwort FXGraph) und außerdem JavaFX 2 in Eclipse-Technologien integriert. e(fx)clipse bringt Unterstützung für folgende Teilbereiche von JavaFX mit:

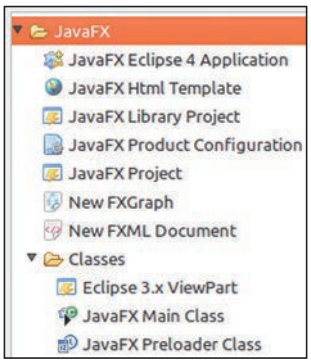


Abb. 1: e(fx)clipse stellt eine Vielzahl von Wizards für Java FX 2 bereit

- Erweiterungen in JDT/PDE
- FXML-Editor
- Live Preview
- CSS-Editor
- FXGraph
- Eclipse 4 Renderer für JavaFX 2

Erweiterungen in JDT/PDE

Wie man in **Abbildung 1** sehen kann, hält e(fx)clipse eine Vielzahl von Wizards bereit. Das Spektrum reicht dabei vom Erstellen einfacher JavaFX-Anwendungen über OSGi-Applikationen bis hin zu Eclipse-4-Anwendungen mit JavaFX-Unterstützung. Als Grundlage stellt e(fx)clipse in diesem Zusammenhang für Java-Projekte zunächst einmal einen eigenen Classpath-Container bereit, der auf alle benötigten Libraries von JavaFX 2 verweist. Erst mit dieser JDT-Erweiterung gelingt die Integration von JavaFX in den berühmten Eclipse-Code-Editor problemlos. JavaFX-Anwendungen lassen sich damit grundsätzlich genauso programmieren, wie es mit Swing oder SWT auch möglich ist. Die JDT-Erweiterungen gehen natürlich noch weiter. Zusätzlich zum JavaFX-Classpath-Container und einem Wizard zum Anlegen von JavaFX-Projekten soll an dieser Stelle unbedingt auch auf die hervorragende Integration des (auf Ant basierenden) Build-Prozesses von JavaFX-Anwendungen in JDT hingewiesen werden. Grundsätzlich bringt JavaFX schon Ant-Tasks zum Bauen, Signieren und Deployen von JavaFX-Anwendungen mit. e(fx)clipse setzt darauf auf und führt ein neues Konfigurationsformat mit der Dateiendung *.fxbuild* mitsamt

Editor ein, mit dem der Build- und der Deployment-Prozess sehr komfortabel verwaltet werden können. **Abbildung 2** zeigt den besagten Konfigurationseditor. Auf Grundlage solcher *.fxbuild*-Konfigurationen ist der Entwickler später in der Lage, quasi auf Knopfdruck Ant-Files zu generieren, die ihrerseits die mit JavaFX gelieferten Ant-Tasks verwenden. Das *.fxbuild*-Format ist somit also ein (im Kern auf XML basierendes) besseres Properties-File – die Magie kommt jedoch von JavaFX selbst! Die Erweiterungen betreffen nicht nur JDT; auch PDE hat einen ähnlichen Classpath-Container – den so genannten PDEClasspath-Container – spendiert bekommen. Erst so lassen sich Equinox-Anwendungen und Plug-in-Projekte mit JavaFX-Unterstützung in der Eclipse-IDE programmieren.

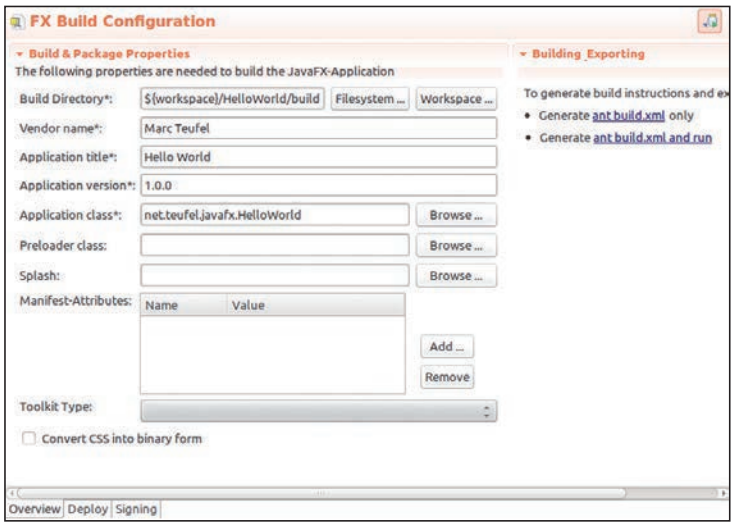


Abb. 2: e(fx)clipse erleichtert das Bauen, Signieren und Ausliefern von JavaFX-Anwendungen durch Editoren, mit denen sich Ant-Files erzeugen und ausführen lassen

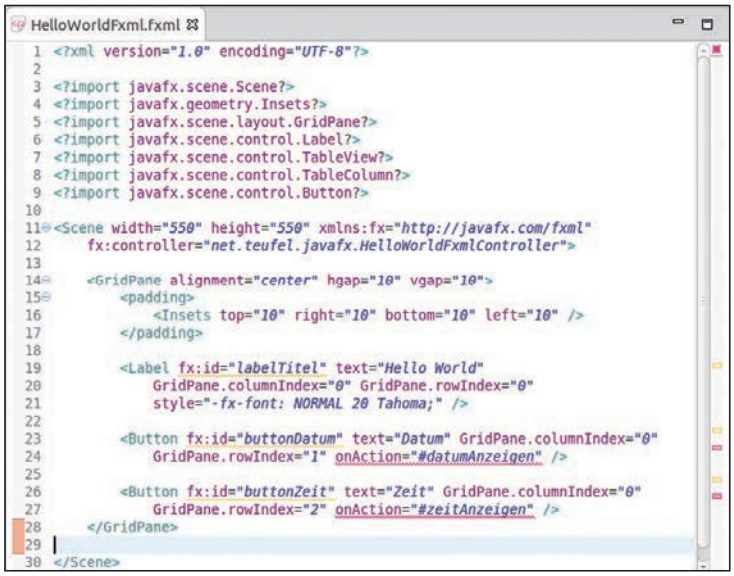


Abb. 3: e(fx)clipse hat einen gut in JDT integrierten Editor für FXML-Dokumente an Bord

Deklarative Oberflächen mit FXML

JavaFX 2 ist zweifelsohne eine moderne UI-Technologie. Zu Zeiten von Swing und SWT wurden die Oberflächen ausschließlich mit Code aufgebaut. Egal, ob man das UI per Hand programmierte oder einen GUI-Editor wie WindowBuilder bediente – am Ende stand immer Sourcecode, der das UI definierte. Moderne Oberflächentechnologien wie beispielsweise WPF (Windows Presentation Foundation) aus dem Microsoft-Universum dagegen verfolgen einen ganz anderen Ansatz: Hier werden UIs in einer XML-Syntax definiert und mit einer Controller-Klasse verbunden, die dann die Methoden für das Event Handling enthält. Die Definition der Oberfläche und die Ereignisbehandlung sind also strikt getrennt. Mit FXML kommt diese Idee nun auch endlich im Rahmen von JavaFX in Java an. **Abbildung 3** zeigt eine typische FXML-Datei im FXML-Editor, den e(fx)clipse ebenfalls mitbringt. Zu Beginn des abgebildeten FXML stehen einige Importanweisungen für alle im Code verwendeten JavaFX-Elemente. Es wird ein Szenegraph vom Typ *Scene* aufgebaut. Die beiden Attribute deklarieren dabei den JavaFX-Namensraum *fx*, und über das Attribut *fx:controller* erfolgt die Verknüpfung mit der zugehörigen Java-Controller-Klasse. Die Java-Klasse ist hier mit ihrem vollqualifizierenden Klassennamen anzugeben. Im weiteren Verlauf werden zusätzliche Controls in den Szenegraphen aufgenommen und entsprechend konfiguriert. Wichtig ist an dieser Stelle, auf das Attribut *fx:id* hinzuweisen: Über diese ID lassen sich später im zugehörigen Java-Controller auf die einzelnen Controls programmatisch zugreifen. Neben dem eigentlichen FXML-Editor gibt es zum Anlegen neuer FXML-Definitionen natürlich auch einen eigenen Wizard. Nach Abschluss dieses Wizards wird das soeben erstellte Dokument automatisch im FXML-Editor geöffnet. Dieser Editor kennt alle Elemente, die Bestandteil eines FXML-Dokuments werden können, und bietet hierfür eine umfangreiche Codevervollständigung an. Diese Codevervollständigung hilft nicht nur beim Schreiben von FXML-Tags und Attributen, sondern fügt auch bei der ersten Verwendung eines bisher noch nicht ins Dokument importierten Controls den entsprechenden Importbefehl automatisch ein. Zusätzlich ist der FXML-Editor tief mit JDT verwurzelt. So werden beim Verknüpfen von FXML mit dem Java-Controller über das Attribut *fx:controller* auch gleich alle infrage kommenden Java-Klassen samt Java Doc zur Auswahl angeboten. Andersherum stellt der FXML-Editor fest, wenn ein Verweis auf eine zuvor

JavaFX und Eclipse

Wollen Sie noch mehr über die Symbiose zwischen JavaFX und der Eclipse-Welt erfahren? Das Eclipse Magazin 2.2013 behandelt dieses Thema ausführlich in einem großen Heftschwerpunkt. Mehr Informationen finden Sie auf www.eclipse-magazin.de im Ausgabenarchiv. Direkter Link: http://bit.ly/XjHTR7

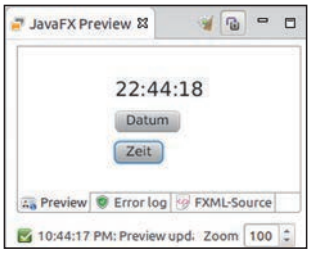


Abb. 4: Die Live Preview in Aktion

angegebene Java-Klasse nicht aufgelöst werden kann, zum Beispiel, wenn die referenzierte Java-Klasse unter dem eingetragenen Namen nicht verfügbar ist. **Abbildung 3** unterstreicht auch, wie weit die Integration mit JDT tatsächlich geht: Alle Zeilen im XML-Listing, die direkt auf entsprechende Stellen in der zugehörigen Java-Controller-Klasse verweisen, aber dort noch nicht verfügbar sind, sind farblich markiert. Selbstverständlich bietet e(fx)clipse an dieser Stelle Quick Fixes an, mit denen sich die fehlenden Attribute und Methoden in der Controller-Klasse automatisch erzeugen lassen. Listing 1 zeigt exemplarisch, wie der zugehörige Controller für das FXML-Dokument aus **Abbildung 3** implementiert werden kann. Über die Annotation *@FXML* sind alle Bestandteile der Klasse markiert, die in Verbindung mit dem FXML-Dokument stehen. Dabei müssen die Namen der Felder mit den entsprechenden IDs, die im Dokument definiert sind, übereinstimmen. Erst so wird es möglich, dass das im Controller verwendete Label alle Eigenschaften von dem im FXML-Dokument definierten Label übernehmen kann. Das Gleiche gilt für die Ereignisbehandlung. Eine Methode, die ein Ereignis verarbeiten soll, ist ebenso mit der Annotation *@FXML* zu markieren, und der Methodenname muss identisch sein mit der Angabe im FXML-Dokument.

Listing 1

```
package net.teufel.javafx;

import java.text.SimpleDateFormat;
import java.util.Date;
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.control.Button;

public class HelloWorldFxmlController {

  @FXML Label labelTitel;
  @FXML Button buttonDatum;
  @FXML Button buttonZeit;

  @FXML
  public void datumAnzeigen() {
    labelTitel.setText(new SimpleDateFormat("dd.MM.yyyy").format(new Date()));
  }

  @FXML
  public void zeitAnzeigen() {
    labelTitel.setText(new SimpleDateFormat("HH:mm:ss").format(new Date()));
  }
}
```

Live Preview

Eine weitere sehr hilfreiche View, die e(fx)clipse anbietet, ist die so genannte „JavaFX Preview“ (**Abb. 4**). Wird diese View aktiviert, zeigt sie jeweils eine Vorschau des UIs an, das man gerade im FXML-Editor bearbeitet. Mit jedem Speichervorgang aktualisiert sich auch die Preview-Ansicht. Bei etwaigen Fehlern kann natürlich keine Vorschau angezeigt werden. Stattdessen hat man dann aber die Möglichkeit, in den Reiter Error log zu wechseln, wo ein entsprechender Strack Trace angezeigt wird. Aus diesem kann man meist sehr leicht herauslesen, was beim Erzeugen der Vorschau schief gelaufen ist. Sehr schön ist in der JavaFX Preview auch die Möglichkeit, einen eventuell schon im FXML definierten Controller aktiv zu schalten (dies erfolgt mit einem der beiden Toolbar-Schalter oben). Wird der Controller hinzugeschaltet, sieht man nicht nur die Anwendung in der Vorschau, man kann auch mit dieser interagieren. So wird die Ereignisverarbeitung eines Knopfs ausgeführt, wenn man diesen in der Vorschau drückt. Auf diese Weise kann man sich das ständige Neustarten der Anwendung zum Testen bestimmter Änderungen sparen, weil man direkt in der Vorschau viel ausprobieren kann.

e(fx)clipse bringt keinen GUI Builder für JavaFX-Oberflächen in der Art mit, wie ihn beispielsweise WindowBuilder anbietet. Es beschränkt sich aktuell ausschließlich auf die Unterstützung von JavaFX auf Code-, FXML- und DSL-Ebene und bietet darüber hinaus die soeben beschriebene Vorschauansicht an. Ein „Zusammenklicken“ von JavaFX-Anwendungen ist mit e(fx)clipse dagegen nicht möglich. Leider bietet auch der WindowBuilder aktuell keine Unterstützung für JavaFX an, und zum gegenwärtigen Zeitpunkt ist auch keine aktive Entwicklung in diese Richtung zu erkennen. Wenn sich JavaFX jedoch weiter in der Geschwindigkeit verbreitet, wie es aktuell der Fall ist und das Interesse an dieser neuen UI-Technologie weiter wächst, dann wird sicherlich auch der WindowBuilder schon bald mit einer entsprechenden Erweiterung aufwarten können.

CSS

Die Idee, Oberfläche mithilfe von CSS zu gestalten, ist nicht neu. In Eclipse 4 sind bereits ähnliche Möglichkeiten für SWT enthalten. Das Problem hier ist nur, dass man mit CSS zwar viel gestalten kann, es aber doch technische Grenzen gibt. So ist es mit dem CSS-Styling-Feature aus Eclipse 4 nicht möglich, eine pixelgenaue Umsetzung eines Designs etwa bei Menüs oder Buttons zu erreichen. In JavaFX 2 gibt es ebenso die Möglichkeit, die Gestaltung mit CSS vorzunehmen. Im Gegensatz zu seinem Eclipse-4-Pendant sind die Möglichkeiten hier aber viel reichhaltiger, es gibt nahezu keine Einschränkungen. Die Abbildung eines pixelperfekten Designs sollte mit dem JavaFX-Styling-Feature problemlos machbar sein. In JavaFX gibt es gleich mehrere Möglichkeiten, CSS-Code zu hinterlegen. So bietet jedes

Control zunächst einmal mit der Methode *setStyle()* die Möglichkeit, CSS direkt zu setzen:

```
labelTitel.setStyle("-fx-font-size: 12px;
-fx-font-weight: bold;
-fx-text-fill: #333333;
-fx-effect: dropshadow( gaussian , rgba(255,255,255,0.5) , 0,0,0,1 ); ");
```

Ferner gibt es die Möglichkeit, CSS-Styling über das *Style*-Attribut, wie man auch in **Abbildung 3** am Beispiel des Labels sehen kann, direkt in ein FXML-Dokument zu schreiben. Ist die zuvor beschriebene Vorschauansicht aktiviert, werden auch CSS-Veränderungen im *Style*-Attribut erkannt und angezeigt. Beim Schreiben von CSS-Attributen im FXML-Editor gibt es leider keine direkte Unterstützung.

Die letzte und wahrscheinlich am weitesten verbreitete Möglichkeit, CSS-Regeln mit der Oberfläche zusammenzubringen, ist es, diese in einer separaten Datei unterzubringen und dann beim Start der Anwendung einzulesen und zu aktivieren. Listing 2 zeigt einen solchen Anwendungsstarter, der sowohl FXML- als auch CSS-Datei in einer Scene zusammenbringt.

Zum Editieren solcher CSS-Dateien bringt e(fx)clipse einen leistungsfähigen CSS-Editor mit (**Abb. 5**). Weil dieser alle Attribute kennt, die verwendet werden können, wird das Schreiben von CSS-Regeln für JavaFX-Anwendungen zum Kinderspiel. Ist die CSS-Datei – wie in Listing 2 gezeigt – in die Anwendung eingebunden, kann die Preview zunächst noch keine Vorschauansicht anzeigen, weil der Eclipse IDE die Verknüpfung noch nicht bekannt ist. Um dies zu bewerkstelligen, ist folgende Zeile zusätzlich ins FXML-Dokument aufzunehmen:

```
<?scenebuilder-stylesheet design.css?>
```

FXGraph

Bei FXGraph handelt es sich nicht um ein Feature, das der JavaFX-Standard mitbringt. Vielmehr hat man es hier mit einer Erweiterung für JavaFX zu tun, die man im Rahmen von e(fx)clipse erhält. FXGraph ist im Kern eine eigene, kleine DSL zum einfachen Schreiben von JavaFX-Oberflächen. Die Syntax erinnert dabei an einen Mix aus Java und Groovy und ist sehr leicht verständlich. In FXGraph geschriebene Oberflächen werden zur Entwicklungszeit von e(fx)clipse ausgewertet; nach jedem Speichervorgang wird aus dem FXGraphen ein FXML-Dokument generiert beziehungsweise aktualisiert. Darauf aufbauend wird umfangreiche Codevervollständigung angeboten. Dabei wird nicht nur die FXGraph-eigene Syntax berücksichtigt, sondern auch auf JDT zurückgegriffen. Ist beispielsweise ein Controller eingebunden, zeigt die Codevervollständigung des FXGraph-Editors beim Zuweisen einer Methode, etwa beim *onAction*-Attribut eines Buttons, alle Methoden aus dem Controller an, die zur Verfügung stehen. Auch die Live Preview ist in der Lage, den FXGraphen in der Vorschauansicht anzuzeigen. Auf einfache Weise können hier externe CSS-Dateien und sonstige Ressourcen in den FXGraphen eingebunden werden. Das ist insbesondere beim CSS-Styling von großem Wert, weil so die CSS-Regeln extern gehalten werden können, die Live Preview diese aber trotzdem erkennen und darstellen kann. Dies erleichtert das Testen von CSS-Styling erheblich. Listing 3 zeigt, wie der FXGraph für das FXML-Dokument, wie in **Abbildung 3** dargestellt, geschrieben werden könnte. Die DSL lässt sich sehr leicht lesen. Auf den ersten Blick ist erkennbar, welcher externe Controller verwendet wird und in welcher Datei sich das CSS-Styling befindet. Zum Aufbau der Oberfläche verwendet FXGraph eine hierarchische Struktur: Die Scene enthält das GridPane, auf dem GridPane liegen dann Label und die beiden Buttons. Die Eigenschaf-

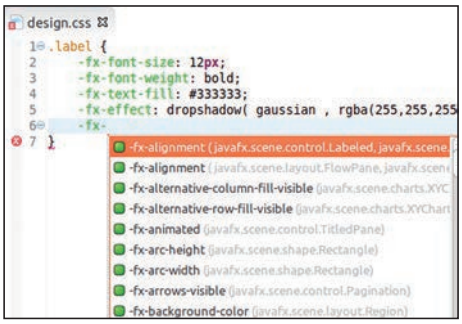


Abb. 5: Der CSS-Editor zeigt alle verfügbaren CSS-Attribute an

Listing 2

```
package net.teufel.javafx;

import java.io.IOException;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class HelloWorldFxml extends Application {

  @Override
  public void start(Stage primaryStage) {
    primaryStage.setTitle("FXML Hello World Application");
    try {
      Scene scene =
        (Scene)FXMLLoader.load(getClass().getResource("HelloWorldFxml.fxml"));
      scene.getStylesheets().add(getClass()
          .getResource("design.css").toExternalForm());
      primaryStage.setScene(scene);
    } catch (IOException e) {
      e.printStackTrace();
    }
    primaryStage.show();
  }

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

ten und Attribute werden durch Kommata getrennt, der FXGraph-Editor unterstützt beim Schreiben an nahezu jeder Stelle mit Codevervollständigung.

Mittlerweile sind FXML- und FXGraph-Editor von der Funktionalität her gleichauf, sodass es letztendlich reine Geschmackssache ist, ob man JavaFX-Oberflächen direkt in FXML schreibt oder den Weg über einen FXGraphen bevorzugt. In jedem Fall ist die FXGraph-DSL im Vergleich zu FXML sehr viel einfacher zu lesen und schreiben, sie wirkt wie ein Fluent Interface. Im Moment ist e(fx)clipse nur in der Lage, aus einem FXGraphen eine FXML-Darstellung zu generieren. In zukünftigen Versionen soll hier ein vollständiger Zyklus angeboten werden, sodass aus einem FXML-Dokument auch ein FXGraph generiert werden kann. Dies ist dann wichtig, wenn man zwischendurch Veränderungen mit dem SceneBuilder (der nur auf Basis von FXML arbeitet) durführen will, danach aber wieder im FXGraphen arbeiten möchte. Da sich die FXGraph-Sprache von Xbase ableitet, wird es in Zukunft vielleicht auch möglich sein, einfache Expressions in FXGraph zu schreiben.

Listing 3

```
package net.teufel.javafx

import javafx.scene.Scene
import javafx.scene.layout.GridPane
import net.teufel.javafx.HelloWorldFxGraphController
import javafx.geometry.Insets
import javafx.scene.control.Label
import javafx.scene.control.Button

component HelloWorldFXGraph controlledby HelloWorldFxGraphController
                                                    styledwith "design.css" {
 Scene {
  GridPane {
   alignment : "CENTER",
   hgap : 1, vgap : 1,
   padding : Insets {
    top : 10, right : 10, left : 10, bottom : 10
   },
   Label id labelTitel {
    text : "Hello World",
    static columnIndex : 0, static rowIndex : 0
   },
   Button id buttonDatum {
    text : "Datum",
    static columnIndex : 0, static rowIndex : 1,
    onAction : controllermethod datumAnzeigen
   },
   Button id buttonZeit {
    text : "Zeit",
    static columnIndex : 0, static rowIndex : 2,
    onAction : controllermethod zeitAnzeigen
   }
  }
 }
}
```

Fazit

Mit e(fx)clipse gibt es nun auch vernünftiges JavaFX Tooling für Eclipse. Für nahezu alle Bereiche bietet das Werkzeug dabei Unterstützung an: Wizards erleichtern das Anlegen von JavaFX-Projekten oder einzelner Artefakte. e(fx)clipse integriert JavaFX in JDT, sodass die Entwicklung von JavaFX-Anwendungen genauso angenehm möglich wird wie das Schreiben von Swing- oder SWT-Anwendungen. Die Verbindung des integrierten FXML-Editors mit der JavaFX-Live-Preview ist eine hervorragende Kombination, wenn es darum geht, deklarative Oberflächen zu bauen. Auch das Schreiben von CSS geht mit dem CSS-Editor leicht von der Hand. Zu den eher fortgeschrittenen Möglichkeiten gehört sicherlich FXGraph, das eine DSL zum Schreiben von JavaFX-Oberflächen anbietet, an dessen Ende jedoch immer FXML steht. Die Vorteile von FXGraph sind dabei der gute, tief mit JDT verwurzelte Editor und die leicht lesbare und erlernbare Syntax. Außerdem ist e(fx)clipse jederzeit in der Lage, einen FXGraphen zu parsen und in der Live Preview darzustellen. Schlussendlich bekommen die Entwickler im Rahmen von e(fx)clipse einen speziellen Renderer für die Eclipse 4 Application Platform geliefert, mit dem es möglich wird, Eclipse-4-Anwendungen auf Basis von JavaFX zu schreiben. Zum Erstellen solcher Eclipse-4-Anwendungen hält e(fx)clipse einen eigenen Wizard bereit, mit dem sich Beispielanwendungen und Grundgerüste generieren lassen. Die meisten Aspekte rund um das Werkzeug sind auf seiner Webseite [1] sehr gut dokumentiert. Auch finden sich dort viele interessante Tutorials. Obwohl e(fx)clipse aktuell noch nicht in einer 1.x-Version vorliegt, hat es schon heute einen erstaunlichen Reifegrad erreicht.

Marc Teufel arbeitet als Projektleiter und Softwarearchitekt bei der hama GmbH & Co und ist dort für die Durchführung von Softwareprojekten im Bereich internationale Logistik zuständig. Er ist Autor zahlreicher Fachartikel im Web-Services- und Eclipse-Umfeld. Er hat drei Bücher zu Web Services veröffentlicht. Sein aktuelles Buch „Eclipse 4 – Rich Clients mit dem Eclipse 4.2 SDK“ ist kürzlich bei entwickler.press erschienen.

Links & Literatur

[1] e(fx)clipse: http://www.efxclipse.org/

[2] JavaFX CSS-Styling Tutorial: http://docs.oracle.com/javafx/2/get_started/css.htm

[3] FXGraph: http://efxclipse.org/trac/wiki/FXGraph

Mit dem Gradle-JavaFX-Plug-in Pakete für JavaFX-Anwendungen erstellen

Verpackungskünstler

Als Oracle JavaFX 2.2. veröffentlichte, waren Tools zur Paketierung von JavaFX-Anwendungen in nativen Installationsprogrammen und ausführbaren Dateien an Bord. Wie bei jeder Cross-Platform-Lösung gibt es aber auch hier eine ganze Reihe an Knöpfen zu drücken und Hebel in Gang zu setzen. Bei der Erstellung eines nativen JavaFX Bundles können geringe Abweichungen vielerorts drastische Auswirkungen auf Funktionalität und Verlässlichkeit haben. Es empfiehlt sich daher, diesen Prozess zu automatisieren.

von Danno Ferrin

Oracle hat eine Reihe von Ant Tasks bereitgestellt, um den Deployment-Prozess zu erleichtern. Allerdings weist Ant erste Altersspuren auf. Viele Organisationen sind daher zu neueren Build-Systemen übergegangen, die „Convention-over-Configuration"-Patterns und eine weniger geschwätzige Konfiguration unterstützen. Das Gradle-JavaFX-Plug-in, das Open Source unter [1] zur Verfügung steht, ist eines dieser Tools, die die JavaFX-Paketierung vereinfachen, wenn man das Build-System Gradle verwendet.

Die vielen Komponenten eines JavaFX-Deployments

Eine JavaFX-Anwendung zu schreiben und zu paketieren ist nicht ganz das Gleiche wie eine gute alte Swing-Anwendung für einen Kunden zu paketieren. Schrieb man Swing-Anwendungen, so war man bei der Zusammenstellung der erforderlichen Distributionskette ganz auf die eigenen Möglichkeiten angewiesen. In den Anfängen der Swing-Entwicklung war das zeitgemäß. Es gab nur sehr wenige Best Practices und keine vorab von Sun definierten Prozesse, um die letzte Meile zum Nutzer zu meistern – Applets ausgenommen. Zusätzlich sind einige Aspekte bei paketiertem Deployment zu beachten, die in JavaFX einzigartig sind.

Die erste nennenswerte Besonderheit: Der JAR-Erstellungs- und Signierungsprozess ist in JavaFX individuell anpassbar. Der Hauptgrund dafür ist, dass JavaFX einen anderen Startmechanismus bei Anwendungen bevorzugt als die alte *main*-plus-*args*-Methode. Diese neue *Application*-Klasse ist ein Überbleibsel aus JSR 296. Sie erlaubt es dem Framework, Threading-Probleme und andere frameworkspezifischen Aufgaben, um die sich der Entwickler normalerweise nicht kümmern müssen sollte, zu verwalten. Wie erwähnt, bietet JavaFX zusätzlich zur Erstellung der JAR-Datei eine neue Methode, diese zu signieren: Statt dies für jede Klassendatei und -ressource separat zu erledigen, signiert JavaFX das gesamte JAR als ein BLOB. Dadurch, dass nur eine Signatur verifiziert werden muss, gehen die Validierung und Verifizierung viel schneller vonstatten. Zur Erstellung und Verifizierung der Zertifikate werden dieselben Signierschlüssel und Infrastrukturen verwendet.

Die zweite wichtige Besonderheit ist, dass JavaFX Paketierungswerkzeuge bereitstellt, die das Parsing und Handling beim CSS Styling beschleunigen. Diese kompilierten Style Sheets unterscheiden sich von den reinen Textversionen durch die *bss*-Extension. Das „b" steht für „binary": Die Custom-JAR-Anwendung sowie ein Standalone-*css2bss*-Tool kompilieren die CSS-Datei ins binäre Format.

Schließlich schnappt sich der JavaFX-Packager alle der zahlreichen JAR-Dateien und -ressourcen, die von der Anwendung verwendet werden, und erstellt eine plattformspezifische Datei, um die Anwendung zu verteilen, zu installieren und zu starten. Komplettiert werden diese Pakete durch die Cross-Platform-Menüintegration, Dock- und Tray-Icons, Menüleistenintegration und einfach anklickbare Symbole. Einiges, was die Integration betrifft, hat allerdings einen Preis: Strebt man eine Integration von hoher Qualität an, muss man oft zusätzliche Dateien wie etwa plattformspezifische Iconformate beisteuern.

Convention over Configuration

Diese Liste an Schritten, die zur Erstellung einer JavaFX-Anwendung notwendig sind, mag einen zunächst einschüchtern. Sie fragen sich vielleicht, wie groß die Gradle-Datei sein wird, die Sie bauen müssen. Lassen Sie uns dazu einen Blick auf eine Build-Datei für eines der klassischen JavaFX-Beispiele werfen: *brickbreaker*.

```
apply from: 'http://dl.bintray.com/content/shemnon/javafx-gradle/
                                                    javafx.plugin'
```

Das ist noch nicht einmal eine abgekürzte Build-Datei – das ist die gesamte Build-Datei für die Brickbreaker-Beispiel-App! Sie können sich unter [2] sogar selbst davon überzeugen. Und dahinter steckt kein Zaubertrick, bei dem ich die tatsächliche Build-Datei anderswo verstecke. Das gleiche Plug-in-Skript kommt in jedem Gradle-Build zum Einsatz, bei dem Sie eine JavaFX-Applikation entwickeln. Der Trick ist nur: Wenn man sich an das Standarddateilayout und Namenskonventionen hält, wird das Plug-in für alle anderen Teile des Build-Prozesses alle erforderlichen Schlüsse ziehen.

> Dahinter steckt kein Zaubertrick, bei dem ich die Build-Datei anderswo versteckt habe.

Um welche Dateikonventionen handelt es sich konkret? Die Konventionen für das Standard-Maven-Layout dienen als Fundament. Die Verzeichnislayouts *src/main/java* und *src/main/resources* stellen die richtigen Speicherorte für den Sourcecode und andere Ressourcen wie z. B. CSS- und Bilddateien bereit. Aufgaben, die von Sourcecode und andere Sourcen abhängen, sind standardmäßig so konfiguriert, dass sie in diesen Verzeichnissen nachsehen. Ein Beispiel wäre die Aufgabe, die CSS zu einer Binärdatei für JavaFX kompiliert – ändert man die Konfiguration nicht, werden alle CSS-Dateien in der *src/main/resources*-Datei zu BSS-Dateien kompiliert. Die ursprünglichen CSS-Dateien werden außerdem im JAR abgespeichert, sodass der Code, der die binäre Form nicht nutzt, trotzdem die CSS-Dateien verwenden kann.

Die Dateikonventionen (Tabelle 1) erstrecken sich auch auf Bereiche, die von den Standardkonventionen von Maven üblicherweise nicht abgedeckt werden. Derzeit werden zwei Gebiete ins Visier genommen: Erstens gibt es einige Installer-spezifische Anpassungen, die sich direkt mit den Installer-Skripten vornehmen lassen. Sie werden in *src/deploy/package/<platform>* gespeichert, wo jede Plattform ihr eigenes Unterverzeichnis hat. Die zweite Dateiengruppe, die von der Konvention erfasst wird, ist die der Start- und Installer-Icons. Man kann diese Icons zwar plattformspezifisch im jeweiligen Verzeichnis abspeichern; man kann sie aber auch als PNG-Dateien unter *src/deploy/package* ablegen. Das Plug-in wird sie dann in plattformspezifische Binaries konvertieren – unter Windows und Mac OS X sogar in unterschiedlichen Auflösungen.

Konfiguration je nach Bedarf

Es gibt in der eigenen Applikation immer einige Aspekte, die nicht durch Dateilayoutkonventionen geregelt sind. Hierfür gibt es Konfigurationsmöglichkeiten, die sich in der *build.gradle*-Datei selbst befinden. Das Gradle-JavaFX-Plug-in setzt zwar alles dran, vernünftige Standards bereitzustellen, aber in manchen Fällen gibt es eben keine universellen Standards. Gradle nutzt in der Build-Datei das Konzept von Erweiterungen, um diese Konfigurationswerte festzuhalten. JavaFX ist nicht das einzige Plug-in, das diese Erweiterungen zur Verfügung stellt – fast alle Plug-ins verwenden es in irgendeiner Form. Die Java-Web-App- und Maven-Plug-ins nutzen den Extension-Mechanismus jeweils, um Dinge zu konfigurieren, die schlicht nicht im Dateisystem konfiguriert werden können. Einige der Konfigurationen enthalten Informationen über die zu verwendende Sprachversion, den Deployment-Deskriptor und diverse Metadaten zu den generierten Maven-*pom*s, z. B. Lizenzierung und Versionsverwaltung. Wie aus Listing 1 hervorgeht, ist die Erweiterung im Wesentlichen ein Codeblock mit etlichen Variablenbelegungen.

Listing 1 ist frei erfunden und auf kein bestimmtes Beispiel bezogen. Nichtsdestotrotz zeigt es jeden Knopf, an dem man drehen und jeden Regler, den man verschieben kann. Auch wichtig zu erwähnen ist, dass keiner dieser Werte eine absolute Voraussetzung darstellt, weil irgendeine Art Standardeinstellung zur Verfügung steht.

Der erste Abschnitt enthält allgemeine Informationen zur Anwendung. Der *appName* ist der Name, der dem Nutzer angezeigt wird, während die *appID* betriebssystemintern verwendet wird.

Der nächste Abschnitt listet die Systemeigenschaften, Umgebungsvariablen und Kommandozeilenargumente

Verzeichnis	Dateien	Zweck
src/main/java	Standard-Java-Layout	Quelldateien der Anwendung
src/main/java	*<project name>/Main.java*	Standard-*main*-Datei/Applikation
src/main/resources	*Non-java files*	Anwendungsressourcen (Bilder, CSS etc.)
src/deploy/package	*shortcut*.png*	Start- und Dock-Icons
src/deploy/package	*volume*.png*	Mac-OS-X-Lautstärke-Icon
src/deploy/package	*setup*.png*	Windows-Installer-Eck-Icon
src/deploy/package/linux	*	Installer-spezifische Skripts für Linux
src/deploy/package/macosx	*	Installer-spezifische Skripts für Mac
src/deploy/package/windows	*	Installer-spezifische Skripts für Windows

Tabelle 1: Konventionen für Gradle-JavaFX-Plug-in-Dateien

der Anwendung auf. Der JavaFX Packager nimmt sich diese Werte und verwendet sie den Paketierungsangaben entsprechend. Sie stehen dann der JavaFX-Anwendung ganz wie erwartet zur Verfügung.

Der dritte Codeabschnitt bezieht sich rein auf die Metadaten. Diese Werte werden nicht zur Laufzeit verwendet, sondern von den plattformspezifischen Paketen. Die Windows Installer verwenden z. B. den Publisher im Applet für das Control Panel, das für das Hinzufügen und Entfernen von Programmen zuständig ist. Für Linux-Pakete ist die Art der Lizenz relevant. Die drei Booleschen Werte am Ende geben an, wo im System das Paket installiert wird, obwohl nicht alle Paketierer auf diese Werte achten. Der Mac-OS-X-DMG-Packager z. B. kreiert keine Dock Shortcuts.

Als Nächstes beschreiben wir die Icons. Diese können sowohl per Dateikonvention als auch per Konfiguration definiert werden. Warum sollte man allerdings ein Icon in der Build-Datei definieren wollen, wenn es doch eine passende Dateikonvention gibt? Der Hauptgrund ist der, dass man Bilddateien möglicherweise mehrfach für unterschiedliche Arten von Icons verwenden möchte. Um die Bilder für die Verwendung in einer bestimmten Art von Icon bereitzustellen, weist man einen String-Wert oder eine Liste an String-Werten dem jeweiligen Icontyp zu. Diese String-Werte sind URLs. Wenn es sich dabei um relative URLs handelt, werden sie aus dem *src/deploy/package*-Verzeichnis aufgelöst. Stellt man eine Liste an Bilddateien für ein Icon zusammen, so werden im Fall eines Installer-Icons mit unterschiedlichen Auflösungen alle Bilder mit der passenden Größe verwendet.

Der letzte Codeabschnitt bezieht sich auf das JNLP und Applet Packaging. Das Meiste davon durchläuft einfach den JavaFX Packager, um die Erstellung von JNLP-Dateien zu ermöglichen. Dazu gehört ein alternatives Layout des Iconformats, das weitere, für eine JNLP-Datei erforderliche Felder enthält. Die *releaseKey*-, *debugKey*- und *signingMode*-Konfigurationen werden für die Signierung des Applets im Schritt *custom-signjar* verwendet. Obwohl die Konfiguration es erlaubt, die Signierschlüsselpasswörter in der Build-Datei abzuspei-

Listing 1

```
javafx {
    appID 'SampleApp'
    appName 'Sample Application'
    mainClass 'com.example.sample.Main'

    jvmArgs = ['-XX:+AggressiveOpts', '-XX:CompileThreshold=1']
    systemProperties = [ 'prism.disableRegionCaching':'true' ]
    arguments = ['-l', '--fast']

    embedLauncher = false

    // deploy/info attributes
    category = 'Demos'
    copyright = 'Copyright (c) 2013 Acme'
    description = 'This is a sample configuration, it is not real.'
    licenseType = 'Apache 2.0'
    vendor = 'Acme'
    installSystemWide = true
    menu = true
    shortcut = true

    // app icons
    icons {
        shortcut = ['shortcut-16.png', 'shortcut-32.png', 'shortcut-128.png',
                'shortcut-256.png', 'shortcut-16@2x.png', 'shortcut-32@2x.png',
                                                    'shortcut-128@2x.png']
        volume = 'javafx-icon.png'
        setup = 'javafx-icon.png'
    }

    // applet and webstart stuff
    debugKey {
        alias = 'debugKey'
        //keyPass = 'password' // provide via command line
        keyStore = file('~/keys/debug.jks')
        //storePass = 'password'  // provide via command line
    }
    releaseKey {
        alias = 'production'
        //keyPass = 'password' // provide via command line
        keyStore = file('/Volumes/ProdThumbDrive/production.jks')
        //storePass = 'password'  // provide via command line
    }
    signingMode 'release'

    width = 800
    height = 600
    embedJNLP = false
    codebase = 'http://example.com/bogus/JNLP/Codebase'

    // arbitrary jnlp icons
    icon {
        href = 'src/main/resources/javafx-icon.png'
        kind = 'splash'
        width = 128
        height = 128
    }
    icon {
        href = 'shortcut-32@2x.png'
        kind = 'selected'
        width = 16
        height = 16
        scale = 1
    }
}
```

Das Plug-in unterstützt in jedem Fall JavaFX auf mobilen Plattformen wie Android und iOS.

chern, ist man damit generell schlecht beraten. Es gibt eine Vielzahl von Möglichkeiten, diese Information an das Build-System weiterzugeben, wobei sich die Wahl der Methode nach den eigenen Bedürfnissen und den Sicherheitsanforderungen der jeweiligen Organisation richten wird. Trotzdem: Legen Sie Ihre Signierschlüssel auf keinen Fall in der Versionskontrolle ab. Was den Debug-Schlüssel angeht, so wird das Gradle-JavaFX-Plug-in einen selbstsignierten Schlüssel für Sie erzeugen, falls er nicht spezifiziert wird. Sie müssen jedoch Ihren eigenen Release-Schlüssel bereitstellen.

Ein Blick in die Zukunft

Das Gradle-JavaFX-Plug-in soll langfristig die einfachste Methode zur Generierung von Paketen für JavaFX-Anwendungen werden – sowohl für kommerzielle als auch für betriebsinterne Projekte – und nicht zuletzt für solche, die der persönlichen Unterhaltung dienen.

Wie bei vielen Open-Source-Committern ist es wichtig, eine Liste an erwünschten Patches bereitzustellen, die die aktuellen Committer selbst nicht zur Verfügung stellen können oder wollen. Wenn es nach mir geht, steht auf dieser Liste alles, was die JNLP- und Applet-Paketierung betrifft. Da keine der Anwendungen, an denen ich arbeite, wirklich eine Web-Start-Kompatibilität voraussetzt, steht mir auch kein entsprechendes Testbed zur Verfügung, mit dem sich feststellen ließe, ob ein Feature funktioniert oder nicht. Es wäre wünschenswert, wenn der Code für diese Teile von jemandem käme, der ein persönliches Interesse an Web-Start-Deployment hat. Entgegenkommen würden mir folglich auch Patches und Beispielcode für die Verwendung des JavaFX Preloaders und die Integration mit JVMs mit aktiviertem Security Manager.

Welche nicht aus Patches resultierenden Features darf man von zukünftigen Versionen erwarten? Da wären erst einmal bestehende Features, die unterstützt werden müssen. Der aktuelle Packager-Code von Oracle unterstützt Lizenztexte und „Click-Through"-Lizenzen. Das Problem ist, dass jede Plattform eigene Anforderungen hat: Linux hat nichts gegen Text, während Windows sich weigert, von RTF abzulassen. Ich würde es dem Entwickler gern ermöglichen, einfache Textdateien zu erstellen, die das Plug-in in andere Formate konvertiert.

Das zweite Feature auf der Roadmap ist ein Mechanismus zur Handhabung plattformspezifischer Werte. Lizenzdateiformate sind erst der Anfang: Windows Installer benötigen GUIDs (Globally Unique Identifier) für den App-Identifier, während Mac Bundles eines Formats bedürfen, das mehr wie ein Java-Package-Name aussieht. Die Mac- und Linux-Kategorien gehen ebenfalls aus einer begrenzten Anzahl von Werten hervor, und im Idealfall wird man vom Build informiert, wenn man davon abweicht. Um dieses Feature zu unterstützen, soll auch ein Mechanismus hinzugefügt werden, der die JRE, die mit den Installer-Paketen gebündelt werden soll, spezifiziert. Das ist beispielsweise dann sinnvoll, wenn man eine Raspberry-Pi-Distribution in einer Ubuntu VM baut.

Am allerwichtigsten ist aber, dass das Gradle-JavaFX-Plug-in in jedem Fall JavaFX auf mobilen Plattformen wie Android oder iOS unterstützt. Idealerweise werden die unangenehmen Details bereits vom JavaFX Packager übernommen; alles Übrige sollte aber von diesem Plug-in geregelt werden. Ich gehe davon aus, dass Features wie das Final Packaging für das Hinzufügen zum App Store und zu Google Play außerhalb der Verantwortung des Core Packagers liegen, aber es gibt zahlreiche Schritte, die unternommen werden können, bevor die Dateien auf die Store-Server hochgeladen werden.

Zusammenfassung

Das Gradle-JavaFX-Plug-in ermöglicht eine hochwertige Integration der obskuren Einzelkomponenten der neuen JavaFX-Paketierungswerzkeuge. Es hält sich einerseits an aktuelle Best Practices in Sachen Dateikonventionen, stellt andererseits aber auch Konfigurationsmöglichkeiten zur Verfügung, die für ein hochwertiges Deployment-Paket unabdingbar sind. Ob Sie eine kleine Spielzeug-App mit Freunden teilen möchten oder eine Cross-Platform-Anwendung auf dem Mac, unter Windows oder Linux starten – das JavaFX-Plug-in geht Ihnen dabei zur Hand.

Aus dem Englischen von der Redaktion des Java Magazins.

Danno Ferrin ist Software Engineer bei Fluke Networks. Dort macht er eine seit Langem bestehende Anwendung für JavaFX und den mobilen Einsatz fit. Mit Java arbeitet er seit dem Beta-Release 1995 und hat seitdem nur einen Softwarejob gehabt, in dem er nicht täglich eine JVM verwendete. Er hat zu verschiedenen Open-Source-Projekten beigetragen, darunter Apache Tomcat, Apache Ant und Groovy. Danno ist außerdem Initiator des Griffon-Frameworks. Er singt Tenor, hat früher Klarinette und Klavier gespielt und gewann als Debattierter mehrere Preise für den Rhetorikclub seiner High School. Seine Lieblingsfarben sind Grün und Orange.

Links & Literatur

[1] https://bitbucket.org/shemnon/javafx-gradle

[2] https://bitbucket.org/shemnon/javafx-gradle/src/009e6d4d2487/samples/brickbreaker/build.gradle?at=release_0.2.0

Chancen und Möglichkeiten

JavaFX vor JSF

Mit JSF bringt man sogleich Webframeworks in Verbindung. Aber: JSF wurde unabhängig von einem konkreten Rendering-Modell definiert und ist ein allgemeines Framework für die serverseitige Interaktionsverarbeitung. Warum also nicht einen JavaFX-basierten Client ansteuern?

von Björn Müller

JavaFX ist eine UI-Umgebung – dem Entwickler offenbart sich JavaFX als Klassenbibliothek mit grafischen Komponenten, Transformationen, Animationen und vielem mehr. Es ist nun also Aufgabe einer Entwicklung, in JavaFX umgesetzte Oberflächen in irgendeiner Weise an eine Anwendungslogik anzuschließen. Diese Anwendungslogik liegt in der Regel auf einem entfernten Server. JavaFX macht zum Glück keine Vorgaben, wie dies nun zu geschehen hat, sondern bietet hier eine Offenheit an, die Architektur zu wählen, die für ein bestimmtes Anwendungsvorhaben passt.

Client-Centric UI

Bei kleineren, überschaubaren Vorhaben ist der Ruf nach einem gesonderten Framework neben JavaFX nicht allzu laut. Hier wählt man in der Regel eine **„clientzentrische" Architektur**, sprich man baut sein Clientprogramm aus JavaFX und Java zusammen. Hierbei überlegt man sich: Wann werden Daten vom Server geholt und zurückgesendet? Wie geschieht die Kommunikation zum Server, welche Art von Web Service wird genutzt? Werden Daten lokal im Client gepuffert? Welche logischen Prüfungen führe ich bereits im Client aus?

Heraus kommt ein explizit entwickeltes Clientprogramm, welches das UI zur Verfügung stellt, das ein gewisses Maß an lokaler Logik enthält und das – hoffentlich in optimierter Weise – mit dem Server kommuniziert.

Und was ist nun mit JSF? JSF passt in solche Ansätze in keiner Weise hinein – und ist auch nicht dafür vorgesehen. Die Interaktionsverarbeitung findet komplett im Client statt. Die Einfachheit der clientzentrischen Architektur bringt aber auch Nachteile mit sich, die sich dann äußern, wenn ein Vorhaben eine gewisse Größe hat. Wir sprechen hier beispielsweise von Anwendungen mit einer hohen Anzahl an Dialogen, mit einer komplexen, regelgesteuerten serverseitigen Anwendungslogik und mit hohen Anforderungen an die Effizienz der Anwendungsentwicklung. Diese Nachteile sind:

- Der Client wird mit der Größe der Anwendung selbst immer größer. Größe an sich ist nicht dramatisch, solange der Client nur einmal geladen wird. In der Regel muss der Client aber aufgrund von Bugfixes und/oder Erweiterungen recht häufig immer wieder zum Benutzer verteilt werden.
- Die Anzahl der Schnittstellen (Web Services) zwischen Client und Server wird mit zunehmender Anwendungsgröße immer größer. Damit einhergehend muss die Effizienz der Aufrufe permanent überwacht werden – wo können Daten in einem Schwung gelesen werden, statt sie in mehreren Einzelaufrufen zu lesen? Schließlich geht ja jeder Aufruf zum Server über das Netzwerk. Auch der Verwaltungsaufwand der Schnittstellen (Security) steigt.
- Und schließlich: Die Effizienz der Anwendungsentwicklung ist dadurch geprägt, dass es die „Frontend-UI-Entwickler" und die „Backend-Serverentwickler" gibt. Die „UI-Entwickler" brauchen ständig neue, feinere Schnittstellen in die Logik hinein, um das UI möglichst interaktiv zu gestalten. Die „Serverentwickler" sind bestrebt, die Anzahl der Schnittstellen so klein wie möglich zu halten – alles in allem also ein breites Feld für Reibungsverluste in der Anwendungsentwicklung.

Server-Centric UI

Nun kommt also die Stunde der **„serverzentrischen" Architektur**. In dieser Architektur gibt es einen generischen UI-Client. Dieser erhält vom Server eine Lay-

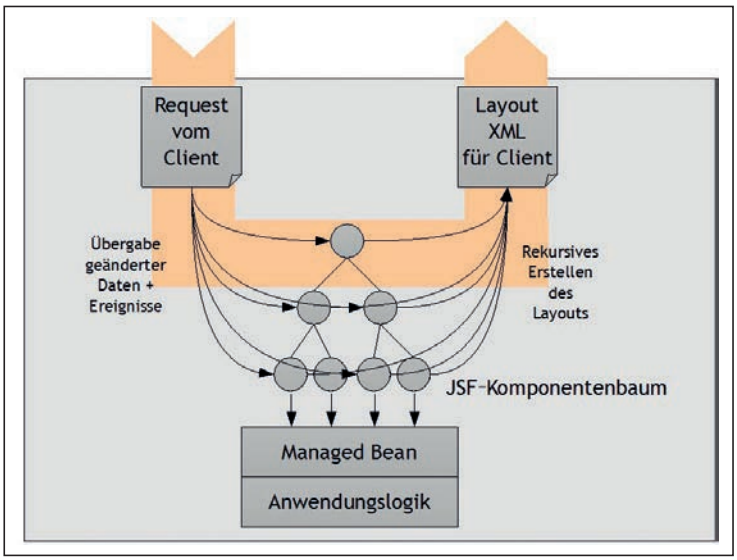


Abb. 1: Strukturbild der JSF-Server-Verarbeitung

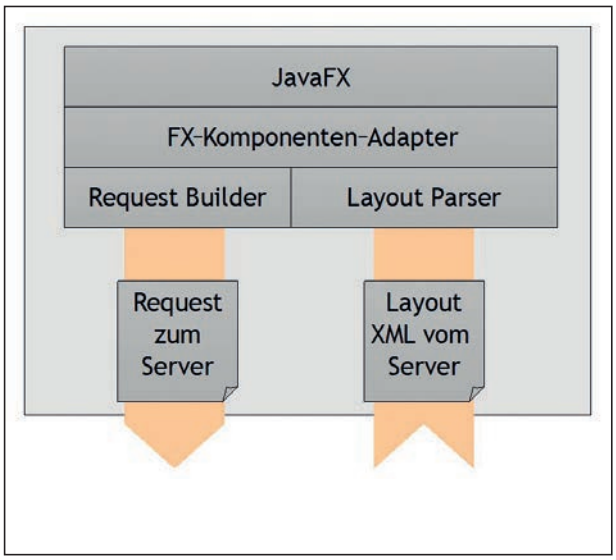


Abb. 2: Strukturbild des generischen Rendering-Clients

outbeschreibung (z. B. eine Art XML-Formular) und wandelt diese dynamisch in eine Benutzeroberfläche um. Der Benutzer arbeitet nun mit dieser Oberfläche, z. B. indem Felder ausgefüllt oder Selektionen getroffen werden. Bei bestimmten Ereignissen, z. B. wenn ein Button gedrückt wird, wird nun die Oberfläche mit dem Server synchronisiert – alle geänderten Daten werden in einem Schwung zum Server gesendet.

Am Server werden nun die geänderten Daten in die Anwendung eingearbeitet, es wird eine Verarbeitung des Ereignisses angestoßen. Die Logik am Server entscheidet beispielsweise, dasselbe Layout noch einmal zum Client zu schicken, weil gewisse Eingaben noch fehlen oder ungültig sind – oder der Server entscheidet, ein neues oder geändertes Layout zum Client zu schicken. Der Client erhält also wieder ein Layout, stellt dieses als Oberfläche dar usw.

Man sieht: Im serverzentrischen Modell spielt sich die eigentliche Interaktionsverarbeitung im Server ab, hier werden Daten der Oberfläche in die Anwendungslogik eingeschleust, und hier werden Entscheidungen getroffen, welches Layout als Nächstes dem Benutzer präsentiert wird. Der generische Client ist nichts anderes als eine aus Sicht der Anwendungssemantik „dumme" Rendering Engine. Welche Vorteile bietet nun dieser Architekturansatz:

- Es gibt keine zwei Stellen der Anwendungsentwicklung, sondern nur eine: alle Anwendungsentwicklungen finden auf Serverseite statt. Zwischen UI-Entwicklung und Serverentwicklung gibt es keine aufwändige Netzwerkgrenze.
- Es gibt nur eine Schnittstelle zwischen Client und Server – der Austausch von Layoutinformationen. Aus Administrationsgesichtspunkten (wieder: Security) ist dies optimal. Diese Schnittstelle wird automatisch geblockt aufgerufen – Änderungen der Daten im Client kommen in einem Schwung zum Server, das neue/geänderte Layout kommt in einem Schwung wieder zurück.
- Der Client ist von seinem Umfang her stabil – und wächst nicht mit wachsender Anwendung. Fehlerkorrekturen und/oder Erweiterungen in der Anwendungsentwicklung betreffen nicht den Client sondern rein die Serverseite. Die Verteilung des Clients muss also deutlich seltener angestoßen werden – nur dann, wenn im generischen Client selbst etwas faul ist.

Server-Centric UI braucht ein serverseitiges Framework!

Und nun kommt JavaServer Faces ins Spiel. Mit JSF gibt es einen bewährten Standard, der genau das macht, was im Rahmen einer serverzentrischen Architektur erfordert wird. JSF ermöglicht die serverseitige Entwicklung von Dialogen und deren Anbindung an die Anwendung. Aus diesen Dialogen entsteht zur Laufzeit die Layoutanweisung für den Client – und in diese Dialoge werden Datenänderungen des Clients eingearbeitet, wenn dieser sich zurückmeldet.

Auch wenn es weh tut: Am besten nähert man sich JSF mit der klassischen HTML-Browser-Denke:

- In JSF wird auf dem Server pro Session ein Komponentenbaum gehalten – also hierarchisierte Objekte, von denen jedes einem grafischen Control entspricht. Zum Beispiel ein Bereich (Pane), darin mehrere Zeilen (Rows), jede dieser Zeilen mit Controls gefüllt (Button, Label, Field, ...). Dieser Komponentenbaum entsteht in der Regel aus einer deskriptiven XML-Beschreibung, kann aber auch rein programmatisch erstellt werden.
- Der Baum wird rekursiv abgegangen, um daraus eine Layoutanweisung für den Client zu erzeugen. Im HTML-Fall erzeugt jedes Objekt im Baum sein „kleines Stückchen HTML" – durch Konkatenation

der kleinen Stückchen während der Baum abgegangen wird, entsteht die gesamte HTML-Seite, die dann letztlich an den Browser geschickt wird.

- Nun, der HTML-Browser stellt diese Seite dar und meldet sich irgendwann mit einem Post Request an den Server zurück. In diesem Request sind die geänderten Inhalte sowie Informationen über das auslösende Ereignis enthalten – diese werden im Server auf die entsprechenden Komponentenobjekte im Baum verteilt und von diesen an die Anwendung übergeben. Die Anwendung kann nun in der Verarbeitungsphase des Ereignisses beliebig den Komponentenbaum verändern.
- Nach der Verarbeitung wird wiederum das neue Layout in rekursiver Weise erstellt und zum Browser geschickt – und das Spiel geht wieder von vorne los.

Wie schon in der Einleitung gesagt: JSF ist ein offenes Framework, das von vornherein von der Beschaffenheit eines Clients und von der Syntax des Layouts, das an einen Client geschickt wird, abstrahiert wurde.

Sprich: Man kann jederzeit in JSF eigene Komponenten bauen, die beim rekursiven Abwandern zur Erstellung des Layouts kein HTML, sondern ein eigenes Format erzeugen – beispielsweise ein XML-Layout, das dann auf die Fähigkeiten eines in JavaFX programmierten generischen Clients zugeschnitten ist.

Die **Abbildungen 1** und **2** zeigen die Struktur einer solchen Verarbeitung. Die Aufgabe des Clients (**Abb. 1**) ist es, das vom Server kommende XML-Layout zu parsen und in entsprechende JavaFX-Komponenten umzusetzen. Hierbei ist es sinnvoll, die FX-Komponenten mit einer einfachen Adapter-Hüllung zu versehen – der Adapter ist das Bindeglied zwischen der XML-Beschreibung einer Komponente und dem konkreten JavaFX Control: die XML-Attributierung wird überführt in das entsprechende Setzen von Properties im Control.

Abb. 3: UI auf Clientseite

Konkretes Beispiel

Zur beispielhaften Verdeutlichung der Architektur und des Programmiermodells wird das frei zu beziehende Framework der CaptainCasa-Community herangezogen, das genau auf dem beschriebenen JSF-Ansatz basiert.

Das UI, das mithilfe von JavaFX für den Benutzer am Client aufgebaut wird, ist in **Abbildung 3** gezeigt. Hierzu wird auf der Serverseite im Rahmen von JSF eine deskriptive Beschreibung gehalten, aus der der JSF-Komponentenbaum aufgebaut wird. Die Beschreibung ist in Listing 1 zu sehen. Die XML-Beschreibung der Seite wird in der Regel über einen WYSIWYG-Editor aufgebaut. In ihr werden zunächst die verwendeten JSF-Komponentenbibliotheken aufgeführt, danach erfolgt die eigentliche Schachtelung der verschiedenen Controls.

In der Seite werden über Expressions Properties und Methoden einer so genannten ManagedBean referenziert, einer einfachen Java-Klasse, die die eigentliche Verarbeitung der Seite übernimmt (Listing 2). Die Klasse ist also quasi der logische „Counterpart“ der Seite.

Das Beispiel zeigt, um nicht zu komplex zu werden, die Verarbeitung einer Seite. Diese Seite kann nun als Einheit in anderen Seiten verwendet werden, oder beispielsweise in modalen oder nicht modalen Dialogen

geöffnet werden. Somit kommt man Schritt für Schritt in eine modular aufgebaute Oberfläche, deren Interaktionsverarbeitung „hinten“ im Server abläuft und deren eigentliche grafische Umsetzung mit JavaFX „vorne“ im Client stattfindet.

Wenn schon, denn schon!

Nun hat man also eine Umgebung, in der im Server ein JSF-Komponentenbaum verwaltet wird, der sich in eine Formularbeschreibung für den Client materialisiert. JSF bietet nun genügend Schnittstellen, um hier einige wichtige Optimierungen vorzunehmen, wenn es um die rekursive Erzeugung der Formularbeschreibung geht.

Es macht hochgradig Sinn, eine „Delta-Verarbeitung“ innerhalb der serverseitigen Erzeugung der Layoutbeschreibung einzubauen. Sprich: Wenn beim Durchgehen eines Komponentenbaums festgestellt wird, dass sich eigentlich im Vergleich zum letzten Durchgehen recht wenig geändert hat, dann sollten nur die Änderungen zum Client gesendet werden – und nicht noch einmal (wie bei HTML ...) alles. Natürlich muss der Rendering-Client auf solch eine Delta-Verarbeitung eingestellt sein – und profitiert seinerseits bzgl. Rendering-Geschwindigkeit enorm, da nur noch die Teile im bestehenden User Interface geändert werden müssen, die auch wirklich eine Änderung erfahren haben.

Umgekehrt macht es Sinn, sich gründlich zu fragen, wann der generische JavaFX-Client eine Synchronisation mit dem Server vornimmt. Wird der Server beispielsweise bei jedem Tastendruck in einem Feld benachrichtigt (nun ja, etwas viele Roundtrips), wird er beim Verlassen des Felds benachrichtigt (immer noch viele) oder führen Feldänderungen zu gar keinem Update – die Daten werden erst synchronisiert, wenn z. B. ein expliziter Button gedrückt wird. Hier hilft nur eines: die Controls (in diesem Fall das Feld-Control) müssen entsprechend parametrierbar sein, sodass der Anwendungsentwickler dediziert das Verhalten steuern kann. Für die meisten

Listing 1

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<%@taglib prefix="t" uri="/WEB-INF/eclnt"%>



<f:view>
 <h:form>
  <f:subview id="ztest_fxonjsfg_sv">
   <t:rowtitlebar id="g_1" text="JavaFX in front of JSF" />
   <t:rowbodypane id="g_2">
    <t:row id="g_3">
     <t:tabbedpane id="g_4" width="100%">
      <t:tabbedpanetab id="g_5" padding="20" rowdistance="5"
       text="First Tab">
       <t:row id="g_6">
        <t:label id="g_7" text="Your Name" width="100" />
        <t:field id="g_8" text="#{DemoHelloWorld.name}" width="200" />
       </t:row>
       <t:row id="g_9">
        <t:coldistance id="g_10" width="100" />
        <t:button id="g_11" actionListener="#{DemoHelloWorld.onHello}"
                            text="Hello" />
       </t:row>
       <t:rowdistance id="g_12" height="20" />
       <t:row id="g_13">
        <t:label id="g_14" text="Result" width="100" />
        <t:field id="g_15" enabled="false"
                           text="#{DemoHelloWorld.output}"
                           width="100%" />
       </t:row>
      </t:tabbedpanetab>
      <t:tabbedpanetab id="g_16" text="Second Tab" />
     </t:tabbedpane>
    </t:row>
   </t:rowbodypane>
   <t:pageaddons id="g_pa" />
  </f:subview>
 </h:form>
</f:view>
```

Listing 2

```
package demo;

import javax.faces.event.ActionEvent;

public class DemoHelloWorld
{
   String m_name;
   String m_output;

   public void setName(String value) { m_name = value; }
   public String getName() { return m_name; }

   public String getOutput() { return m_output; }

   public void onHello(ActionEvent ae)
   {
      if (m_name == null)
         m_output = "No name set.";
      else
         m_output = "Hello World, "+m_name+"!";
   }

}
```

Felder wird es genügen, wenn sie nicht sofort synchronisiert werden, für bestimmte Felder ist eine sofortige Synchronisation nach Änderung wichtig, beispielsweise um sie einer Konsistenzprüfung am Server zu unterziehen.

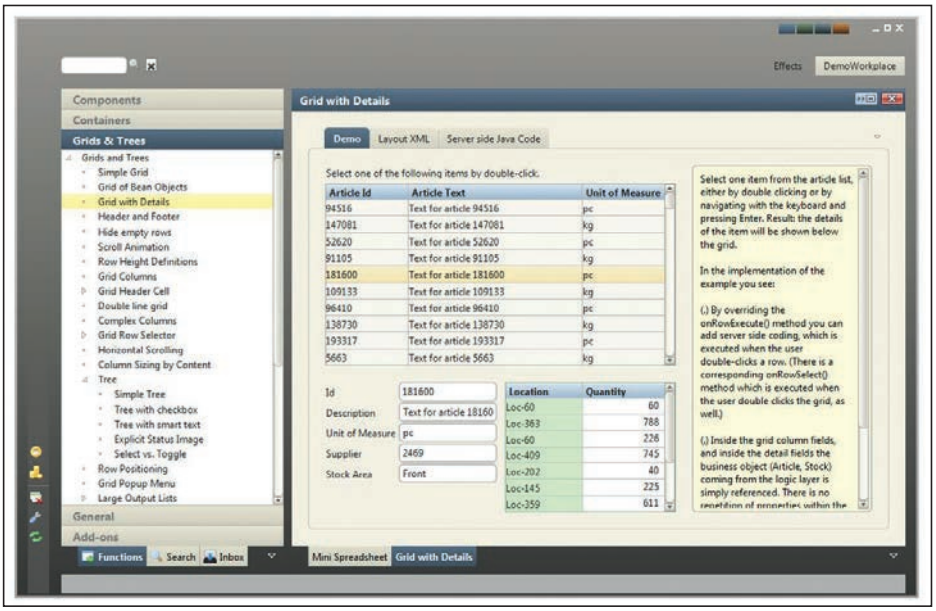


Abb. 4: Beispiel CaptainCasa Enterprise Client: DemoWorkplace

Aspekte der Nutzung von JSF

Fast so ähnlich wie JavaFX hat auch JSF nicht gerade einen Glanzstart hingelegt – obwohl dieser mittlerweile aber fast ein Jahrzehnt zurückliegt. „JSF ist komplex" ist ein Standardspruch, typischerweise von Entwicklern geäußert, die von HTML-nahen und leichtgewichtigeren Frameworks kommen.

Unsere Erfahrung: JSF hat eine gewisse Komplexität, die zu hoch ist, um sie direkt an die Anwendungsentwicklung zu übergeben. Die Nutzung von JSF muss durch eine Konzeption und eine Toolwelt anwendungsgerecht aufbereitet werden. Anders gesagt: Der JSF-Kuchen ist sehr groß und man muss sich die Stückchen herausnehmen, die im konkreten Szenario Sinn machen.

Ein Beispiel: Die Seitenfolgesteuerung von JSF passt gut für typische Webszenarien, ich befinde mich in einer Seite, die Logik leitet mich zur nächsten. In Rich-Client-Szenarien hat man in der Regel wesentlich feiner gesteuerte Navigationsszenarien, in denen die Modularisierung von Seiten eine wesentlich stärkere Rolle spielt. Eine Stammdatenmaske muss beispielsweise an verschiedenen Stellen des User Interfaces flexibel eingebaut werden können. Sprich: Hier macht es keinen Sinn, sich mit der JSF-Navigation auseinanderzusetzen, bzw. diese krampfhaft in die eigene Verarbeitung zu übernehmen.

Wenn man aber JSF in anwendungsgerechter Paketierung verwendet, dann erweist es sich als ein sehr effizientes Framework, das viele Fragestellungen der serverbasierten UI-Entwicklung (Erweiterbarkeit mit eigenen Komponenten, Fragen einer Session-Persistierung ...) aus dem Standard heraus abdeckt und das zudem einen großes Maß an Produktionsreife besitzt.

Und: JSF ist ein Java-Standard. Wer es also zur Bedienung eines JavaFX-Clients nutzt, für den ist die Nutzung von JSF für einen „klassischen" HTML-Client nicht allzu fern!

CaptainCasa Enterprise Client

CaptainCasa Enterprise Client ist eine Rich-Client-Infrastruktur für Geschäftsanwendungen, die genau auf dem beschriebenen Architekturansatz basiert. Die Entwicklung wird durch eine offene Corporate-Community getrieben. Bestandteile sind einerseits der generische JavaFX-Client, andererseits eine JSF-Komponentenbibliothek und diverse Tools (Layouteditor), die JSF anwendungsgerecht aufbereiten (**Abb. 4**). CaptainCasa kann in vollem Umfang frei genutzt werden.

CaptainCasa hat bislang einen Swing-basierten Client als generischen Rendering-Client genutzt und hat diesen nun im Zuge der JavaFX 2.*-Verfügbarkeit um einen JSF-basierten Client erweitert. Gerade hier zeigt sich ein großer Vorzug der beschriebenen Architektur: Der Umstieg von einer Rendering-Technologie (Swing) auf eine andere Technologie (JavaFX) gelingt durch eine technische Umstellung des generischen Clients. Die Anwendung auf der Serverseite ist davon technologisch unbeeinflusst, der Wechsel der UI-Technologie findet quasi „vor der Anwendung", nicht „in der Anwendung" statt.

Fazit

JSF als Standard für serverseitige Interaktionsverarbeitung macht Sinn – wenn man eine serverzentrische UI-Architektur wählt. Hierbei handelt es sich nicht um eine Spezialnutzung von JSF, sondern um eine in der JSF-Architektur vorgesehene und „willkommene" Nutzung.

Björn Müller, CaptainCasa GmbH, beschäftigt sich seit 2001 mit Technologiefragen der User-Interface-Entwicklung: zunächst auf Basis von HTML/JavaScript, seit 2007 aber im Rahmen der CaptainCasa-Community auf Basis Java-basierter Frontends.

iOS- und Android-Implementierungen

JavaFX goes Open Source

Seit der Vorstellung von JavaFX 2 auf der JavaOne 2011 hat die strategische Technologie für Clientwebanwendungen von Oracle eine stetig steigende Zahl von Entwicklern begeistern können. Im Herbst letzten Jahres wurden erste namhafte JavaFX-Anwendungsbeispiele vorgestellt, dicht gefolgt von weiteren neuen Geschäftsanwendungen, die künftig mit JavaFX 8 realisiert werden können. Nun geht die bislang proprietäre Technologie JavaFX vollständig in Open Source auf.

von Wolfgang Weigend

Die einzelnen JavaFX-Projekte Glass Windowing Toolkit, Image I/O und Hauptbestandteile der Prism Rendering Engine werden durch 138 034 Zeilen Code repräsentiert, die sich jetzt Open Source wiederfinden. Die restlichen Codezeilen von Prism, als auch die Web- und Mediaquellen folgen nach. Damit liegt der Quellcode nahezu vollständig als Open Source vor, bis auf javafx-font, der durch direkte Aufrufe zum Betriebssystem im Open-Source-Code ersetzt wird. Dies steht bereits auf der Aufgabenliste vom Projekt OpenJFX.

iOS- und Android-Implementierungen werden Open Source

Den Interessierten ist nicht verborgen geblieben, dass gezielte Umfragen zum Bedarf von JavaFX mit Unterstützung für Tablets und mobile Betriebssysteme zum Jahreswechsel stattgefunden haben. Die Auswertung der Befragung hat ergeben, dass sich die Mehrheit der Entwickler an einem iOS/Android Port beteiligen möchte, sei es durch Bug-Reports oder mittels direkter Codebeiträge. Dies nahm das Java-Produktmanagement zum Anlass, die notwendigen Voraussetzungen dafür zu schaffen, dass der JavaFX-Code als Open Source verfügbar ist und die Java-Community daran arbeiten kann. Eine wesentliche Maßnahme besteht darin, den Build- und Test-Set-up so zu verbessern, dass sich das Erzeugen und Testen von JavaFX Fixes für die Entwickler stark vereinfacht. Die Priorisierung eines vereinfachten Build- und Testsystems führt dazu, dass das Projektteam wesentlich schneller Codebeiträge annehmen kann. Die ersten Bestandteile für iOS liegen vor und die restlichen Komponenten für iOS und Android kommen zum selben Zeitpunkt wie die restliche Kodierung von Prism, da hierbei zeitliche Abhängigkeiten bestehen. Beide Portierungen basieren auf einer bisher unveröffentlichten Version von Java SE Embedded für iOS/Android.

Lizenzierung mit eigenem Applikations-Co-Bundle

Die Lizenzierung von Apple iOS im App Store erlaubt keine GPL-Lizenzierung für Applikationen. OpenJFX und OpenJDK sind jeweils über GPLv2 mit Classpath Exception lizenziert. Bei Verwendung von OpenJDK und OpenJFX, ohne die Binär-Stubs mit unterschiedlicher Lizenzierung, besteht die Möglichkeit in der Kombination von OpenJDK und OpenJFX eine eigene Applikation, unter eigener Lizenz, als einzelnes Applikations-Co-Bundle herauszubringen. Die gleiche Möglichkeit besteht mit den offiziellen Versionen von JavaFX und Java SE. Im Unterschied dazu befinden sich derzeit iOS und Android nicht auf der offiziellen Release-Roadmap für JavaFX, sodass in der Zwischenzeit der einzige Weg zur binären Verfügbarkeit von JavaFX auf iOS über OpenJDK und OpenJFX führt. Damit sind die Möglichkeiten beim Erschließen neuer Ports mit Open Source größer und es beschreibt auch das hohe Enga-

gement, den Open-Source-Prozess von JavaFX mit maximaler Priorität voranzutreiben. Mit dem Aufbau von OpenJFX ohne Binär-Stubs kann die Entwicklergemeinschaft keiner daran hindern, das OpenJFX mit dem iOS Port und OpenJDK zu verwenden, darauf kommerzielle Applikationen zu entwickeln und diese im Apple iOS App Store anzubieten (**Abb. 1**).

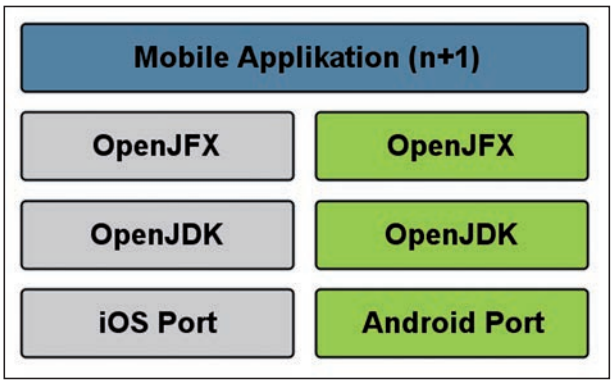


Abb. 1: Mobile Applikation mit OpenJFX und OpenJDK

Ausblick und Fazit

Die große Zustimmung der Entwickler, den entscheidenden Anteil zur Portierung von JavaFX auf mobile Endgeräte mit iOS und Android zu erbringen, gepaart mit dem positiven Feedback der Kundenumfrage, lässt schlussfolgern, dass die Entwicklergemeinschaft in der Lage ist, mit OpenJDK, OpenJFX, iOS und Android, eigenständige Open-Source-Beiträge zur Laufzeitumgebung von JavaFX auf den mobilen Betriebssystemen iOS und Android zu leisten.

Oracle legt den Quellcode der JavaFX-Bibliotheken mit den Arbeiten an den Prototypen für iOS und Android als Open Source offen. Die darunter liegende Virtual Machine und die Java-Core-Bibliotheken-Ports für iOS und Android werden nicht als Open Source zur Verfügung gestellt. Dies ist mit dem geschützten Java Port für ARM-Prozessoren vergleichbar.

Für Entwickler ist es deshalb notwendig, die offenen JavaFX-Bibliotheken für iOS oder Android zu testen, indem sie ein Subset vom OpenJDK verwenden und die VM auf iOS/Android portieren. Eine weitere Möglichkeit besteht darin, eine Third-Party-VM-Implementierung für ARM-basierte Prozessoren zu benutzen. Damit können sich die Entwickler auf die JavaFX-UI-Bibliotheken-Interfaces mit Blick auf iOS und Android konzentrieren, ohne sich um tiefgreifende Abhängigkeiten einer Implementierung von ARM-basierten CPUs mit iOS und Android kümmern zu müssen.

Die einheitliche Java-Entwicklungsplattform mit den Entwicklungsumgebungen Eclipse (e(fx)clipse), IntelliJ IDEA, NetBeans und anderen in Kombination mit JavaFX, ermöglicht die übergreifende Erstellung von Anwendungen für den Smartphonemarkt, in dem Android und iOS insgesamt 87,8 Prozent Marktanteil im gesamten Jahr 2012 belegten (siehe Studie der Agentur Strategy Analytics, Tabelle 1). Folglich lohnt es sich, dass die Entwicklergemeinschaft den Open-Source-Prozess von JavaFX nutzt und die Portierung für iOS und Android in die eigenen Hände nimmt.

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

Links & Literatur

[1] http://fxexperience.com/2013/02/february-open-source-update/

[2] http://tomsondev.bestsolution.at/2013/02/13/thoughts-on-javafx-for-android-and-ios/

[3] http://www.oracle.com/technetwork/java/javafx/documentation/index.html

[4] http://jdk7.java.net/fxarmpreview/index.html

[5] http://openjdk.java.net/projects/openjfx/

Global Smartphone OS Shipments (Stückzahlen in Millionen)	4. Quartal 2012	Gesamtjahr 2012
Android	152,1	479,0
Apple iOS	47,8	135,8
Andere	17,1	85,3
Summe	217,0	700,1

Global Smartphone Marktanteil %	4. Quartal 2012	Gesamtjahr 2012
Android	70,1	68,4
Apple iOS	22,0	19,4
Andere	7,9	12,2
Summe	100 %	100 %

Tabelle 1: Global Smartphone Operating System Shipments and Market Share Q4 2012 (Quelle: Agentur Strategy Analytics)

Ein Duke erobert die Welt

Java everywhere

Java hat in den letzten Jahren besonders im Serverbereich weite Verbreitung gefunden. Doch auch im Embedded-Bereich nimmt die Zahl der mittels Java programmierbaren Geräte stetig zu. Interessanterweise kommt es eigentlich auch aus dieser Ecke. Ist das einstige Motto „Java everywhere“ also bereits Realität geworden?

von Bernhard Löwenstein

Java hat sich im letzten Jahrzehnt zu einer der beliebtesten, wenn nicht sogar zur populärsten Programmiersprache gemausert [1]. Es wäre aber falsch, Java auf eine Sprache zu reduzieren. Vielmehr ist es heute eine Plattform, auf der Anwendungen und Komponenten ablaufen können. Diese wurden in einer Sprache geschrieben, aus der mithilfe eines entsprechenden Compilers Java-Bytecode gewonnen werden kann. Zentrale Bedeutung kommt dabei der JVM (Java Virtual Machine) zu. Sie abstrahiert die Unterschiede zwischen den verschiedenen Betriebssystemen und führt somit eine plattformunabhängige Abstraktionsschicht ein. Der Programmierer kann sich nun ganz auf die eigentliche Anwendungsentwicklung konzentrieren. Um plattformspezifische Details muss er sich im Gegensatz zu früher nicht mehr kümmern. Gerade in Verbindung mit eingebetteten Geräten, deren Zahl in den letzten Jahren dank der immer leistungsfähigeren und günstigeren Elektronikteile stark zunahm, macht dies Java beziehungsweise eine darauf basierende Alternative zu einer äußerst attraktiven Plattform.

Werfen wir einen Blick auf die Entwicklungsgeschichte von Java, so stellen wir erstaunt fest, dass Java ursprünglich eigentlich für den Embedded-Bereich konzipiert wurde. So verfolgte das Entwicklerteam um James Gosling anfänglich nicht das Ziel, bloß eine weitere Programmiersprache auf den Markt zu bringen: Das Projekt „Green“, aus dem später „Oak“ und in weiterer Folge „Java“ wurde, sollte eine Technologie zu Tage bringen, mit der verschiedenste Endgeräte einfach programmiert werden konnten. Ein weiteres zentrales Anliegen war, den Datenaustausch zwischen unterschiedlichen Geräten möglichst einfach zu gestalten. Technisch realisiert werden sollte dies durch die Bereitstellung einer einfach portablen Betriebssystemumgebung inklusive virtueller CPU (Central Processing Unit), auf deren Basis die Programme entwickelt werden und ablaufen sollten. Anstatt wie bisher mit C++ direkt gegen die Hardware zu programmieren, sollte die Entwicklung zukünftig in einer neuen Programmiersprache erfolgen, die auf dieser Abstraktionsebene aufsetzt. Die Programmiersprache sollte von der Syntax her an C++ angelehnt sein, jedoch um höhere Konzepte wie die automatische Speicherverwaltung ergänzt werden. Von der Set-Top-Box über das Smartphone bis hin zur Kaffeemaschine sollten so alle möglichen Geräte programmiert und gesteuert werden können. Diese Idee floppte – oder kennen Sie eine auf Java-Basis arbeitende Kaffeemaschine? Selbst in Wien, dem weltweiten Zentrum der Kaffeehauskultur [2], ist mir keine solche Maschine bekannt. Sehr wohl aber setzte sich die Grundidee durch, dass Applikationen und Komponenten nicht mehr für die jeweilige Plattform, sondern in den plattformneutralen Java-Bytecode übersetzt werden sollten. Statt auf Betriebssystemebene wurde die Java-mäßige Virtualisierungsschicht aber eine Stufe höher etabliert. Die Jahre vergingen, und die Idee eines universalen Betriebssystems für unterschiedlichste Endgeräte schien endgültig der Vergangenheit anzugehören. Doch wie so oft im Leben, wenn keiner mehr damit rechnet, kam es schließlich doch noch zum Comeback. Google brachte Android auf den Markt. Dieses legt seitdem einen wahren Erfolgslauf hin, den vor Jahrzehnten noch keiner für möglich gehalten hatte. Wer die Vision des kalifornischen Internetgiganten in Bezug auf Android genauer unter die Lupe nimmt, kann interessanterweise sehr starke Ähnlichkeit mit Javas Vision erster Stunde feststellen. Google verzichtete darauf, eine eigene Programmierplattform bereitzustellen und setzte von Haus aus auf einen Java-Dialekt. Vom Siegeszug Androids profitiert somit auch Java – und eventuell erleben wir nun doch noch, dass Kaffeemaschinen auf den Markt kommen, die unter Android laufen und in Java programmierbar sind.

Auch eine weitere Idee, die die Entwickler der Java-Plattform hatten, scheiterte – und setzte sich dann Jahre später doch noch durch: Java sollte seinerzeit als „DOS of the Internet“ etabliert werden. Mit genau diesen Schlagworten wurde es nämlich 1995 auf der Sun World offiziell vorgestellt. Die Applikationen sollten hierbei in Form von downloadbaren Applets bereitgestellt und von den in den Browsern integrierten Java-Laufzeitumgebungen interpretiert werden. Interessanterweise war es der damalige Branchenprimus Netscape, der durch Integra-

tion eines Java-Interpreters in seinen Navigator 2 für den Durchbruch von Java sorgte. Die in der Sandbox ablaufenden Applets waren anfangs auch der große Hit, doch bald schon standen die gleichen Kunden wieder auf der Matte und wünschten die Migration zu einer Java-Applikation. Jahre später sollte in den meisten Fällen dann eine RIA (Rich Internet Application) daraus werden. Netscapes Schiff ist als Teil von AOL (America Online) mittlerweile untergegangen, und auch über Applets redet heute keiner mehr. Doch die Idee dahinter überlebte und feierte Jahre später mit den Mobile-Apps eine beeindruckende Wiedergeburt. Diese mischen seitdem nicht nur ordentlich den mobilen Markt auf, sie werden zukünftig wohl noch eine viel größere Bedeutung einnehmen. In einer Welt, die sich dank der zunehmenden Vernetzung und immer kostengünstigeren Hardwarekomponenten immer mehr in Richtung Internet der Dinge entwickelt, ist es heute ein Leichtes, eine Smartphone-App zur Steuerung von Endgeräten zu implementieren. Darüber ließe sich dann sicherlich auch unsere vorhin erwähnte Kaffeemaschine kontrollieren. Andererseits: Das bisschen Bewegung vom Arbeitsplatz zur Kaffeemaschine schadet uns Entwicklern doch wirklich nicht…

Neben den Android-basierten Devices finden sich mittlerweile aber auch viele weitere Geräte, auf denen eine spezielle, auf die eingebettete Umgebung zugeschnittene Java-Laufzeitumgebung läuft, die die Ausführung von Java-Bytecode ermöglicht. Die Java-Hersteller zollten dem Trend der immer leistungsfähigeren Endgeräte Tribut, indem sie der Java-Community neben der bisherigen Java ME (Java Platform, Micro Edition) mit der Java SE Embedded (Java Standard Edition Embedded) [3] eine zweite Plattform spendierten. Diese basiert auf der Standardausgabe und implementiert vollständig die zugehörige Spezifikation, weist aber signifikante Speicher- und Laufzeitoptimierungen für Embedded Devices auf. Die meisten neuen Geräte werden eine Implementierung dieser Plattform an Bord haben, die Mikroausgabe wird auf lange Sicht wohl das Dinosaurierschicksal erleiden.

Nachdem in Zukunft nicht mehr die klassischen Rechner, sondern vorrangig eingebettete Geräten vorhanden sein werden, wird das Java Magazin ab sofort stärker auf diesen Trend eingehen und in den nächsten Ausgaben unter der Rubrik „Embedded“ über verschiedene interessante Projekte berichten. Dabei werden Ihnen unterschiedliche Hardwareplattformen wie Arduino [4] und Raspberry Pi [5], verschiedene Robotiksysteme wie NAO (**Abb. 1**, [6]) und LEGO MINDSTORMS (**Abb. 2**, [7]), Spielzeugeisenbahnen, Flugdrohnen und noch vieles mehr begegnen – alles natürlich in Java programmierbar. Herzlich einladen möchte ich Sie ebenfalls zum Embedded Experience Day [8], s. auch Kasten auf S. 53 (Artikel Grunwald), auf die JAX 2013 in Mainz. Dort gibt es nicht nur spannende Vorträge zu diesem Themenbereich zu hören – in der zugehörigen Werkstatt können Sie einige Technologien auch gleich selbst ausprobieren beziehungsweise in Java programmieren.

Abb. 1: Der hochentwickelte NAO-Roboter begeistert nicht nur Kinder und Erwachsene, er lässt sich auch mittels Java programmieren

Abb. 2: Dank des Open-Source-Projekts leJOS sind LEGO-MINDSTORMS-Roboter ebenfalls in Java programmierbar

„Java everywhere“ war das Motto der JavaOne-Konferenz im Jahre 2003 [9]. Zehn Jahre später kann diese Vision als in die Realität umgesetzt angesehen werden. Am Desktop, auf den Servern und in eingebetteten Geräten läuft heute in vielen Fällen Java – und ich finde, das ist auch gut so!

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

Links & Literatur

[1] http://www.jaxenter.de/news/066494

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

[3] http://www.oracle.com/technetwork/java/embedded/overview/getstarted/index.html

[4] http://www.arduino.cc

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

[6] http://www.aldebaran-robotics.com/en/

[7] http://de.wikipedia.org/wiki/Lego_Mindstorms

[8] http://jax.de/2013/sessions/?tid=2880

[9] http://www.javaworld.com/javaworld/jw-08-2003/jw-0822-wireless.html

Verbindungsmöglichkeiten einer Standalone-Anwendung als EJB-Client

Wo bitte geht es zum JBoss-AS-7-Server?

NEUE SERIE

Ein JBoss AS 7 wird anders gestartet, ist schneller und wird anders konfiguriert. Das lässt die Vermutung aufkommen, dass auch der Client auf andere Art und Weise eine Verbindung aufbauen muss. Dieser Beitrag beleuchtet die verschiedenen Verbindungsmöglichkeiten und deren Unterschiede zu früheren Versionen.

von Wolf-Dieter Fink

Nachdem der Server installiert ist und die Anwendung erfolgreich gestartet wurde, soll noch ein Client die vorhandene Geschäftslogik der EJB aufrufen. Der bestehende Code früherer Versionen funktioniert nicht mehr, ohne dass Änderungen vorgenommen werden.

Ein einfacher EJB-Client

Anders als in der Konfiguration einer Applikation, die mit einem JBoss 5 kommuniziert, ist eine zusätzliche Datei im Klassenpfad zu erstellen. In der *jboss-ejb-client.properties* (Listing 1) werden die Parameter für die Serververbindung festgelegt.

Es können mehrere Server in der Liste *remote.connections* mit Kommata separiert angegeben werden. Die hier angegebenen Namen haben nur für die Konfiguration innerhalb der *jboss-ejb-client.properties* eine Bedeutung und verweisen auf die verschiedenen Properties mit dem Präfix *remote.connection.<name>*. Der entsprechende Java-Code kann Listing 2 entnommen werden.

Der verwendete Identifier für den Lookup ist jetzt vereinheitlicht und setzt sich aus dem Namen der Anwendung, dem Modul, dem Bean-Namen und dem Interface zusammen. Ist eine *StatefulSessionBean* referenziert, muss zusätzlich noch *?stateful* angefügt werden. Weitere Informationen und Beispiele sind in der AS-7-Dokumentation unter [1] zu finden. In Listing 2 wird damit die im *app.ear* bereitgestellte Anwendung referenziert, die das Modul *ejb.jar* mit der Bean enthält. Ein Verändern der JNDI-Namen durch Mapping-Einträge im Deployment Descriptor wird im AS 7 zurzeit nicht unterstützt.

Die Property für den *InitialContext* kann auch in der Datei *jndi.properties* angegeben werden. Mit dem *InitialContext* wird festgelegt, dass der Client die JBoss-EJB-Client-Library und damit die *jboss-ejb-client.properties* verwenden soll. Wird der Client auf dem gleichen Rechner gestartet, auf dem der Server läuft, funktioniert der Aufruf. Befindet sich die Client-JVM auf einem anderen System, so bricht der Methodenaufruf mit der Fehlermeldung *EJBCLIENT000025: No EJB receiver available for handling [appName:<app>, modulName:<ejb>, distinctName:]* ab.

Warum geht denn der Methodenaufruf nicht?

Befindet sich der Client auf dem gleichen System, so erkennt der Server dies und authentifiziert die Verbindung. In 7.1.1 ist das nicht änderbar, mit neueren Versionen kann das Verhalten mit dem Element *authentication local* der entsprechenden *ApplicationRealm* beeinflusst werden. Wird das Element gelöscht, so verhält sich der Aufruf immer gleich. Im Client kann die Authentifizierung durch Setzen der Property *remote.connection.default.connect.options.org.xnio.Options.SASL_DISALLOWED_MECHANISMS=JBOSS-LOCAL-USER* erzwungen werden.

Wie kommt es aber überhaupt dazu, dass der Methodenaufruf fehlschlägt und nicht der *lookup()*? Die Verbindung wurde optimiert, um weniger und effektiver zwischen Client und Server zu kommunizieren. Der Aufruf von *lookup()* erzeugt im Falle einer Stateless Session Bean nur einen Proxy anhand des angegebenen Interfaces, es findet keine Kommunikation und kein Laden der Klassen vom Server statt. Daraus folgt auch, dass

Artikelserie

Teil 1: Verbindungsmöglichkeiten und Unterschiede zu früheren Versionen

Teil 2: Komplexe Szenarien und NodeSelectoren

Konfigurationen wie Interceptoren und Load Balancing auf der Seite des Clients erfolgen und nicht mehr vom Server durch Konfiguration bereitgestellt werden. Bei einer Stateful Session Bean wird durch den *lookup()* eine Instanz der Stateful Session Bean erzeugt, und der Proxy erhält eine Affinität zum entsprechenden Server. Durch diese Änderungen kommt es jetzt zu einer verzögerten Fehlermeldung, sofern die EJB nicht verfügbar ist. Kam es früher bereits beim Aufruf von *lookup(name)* zu einer *NamingException*, wird diese jetzt erst geworfen, wenn eine Methode an dem so erhaltenen Proxy aufgerufen wird. Der *lookup* wirft nur dann eine *NamingException*, wenn das in dem angegebenen Namen spezifizierte Interface nicht instanziiert werden kann.

In dem Beispiel fehlt noch die Authentifizierung, die in AS 7 standardmäßig aktiviert ist. Auf der Serverseite kann der Benutzer mit dem Skript *bin/add-user.[sh bat]* hinzugefügt werden. In der Datei *jboss-ejb-client.properties* aus Listing 1 müssen die Properties *remote.connection.username* und *remote.connection.password* mit dem entsprechenden Benutzernamen und Passwort hinzugefügt werden. Danach lässt sich die EJB aufrufen.

Jetzt ein weiterer Server

Fügen wir jetzt einen zweiten Server mit der gleichen Anwendung hinzu. Es ist dabei egal, ob die zwei Server im Standalone-Modus gestartet werden oder ob wir eine Domain verwenden. Die entsprechenden Serverkonfigurationen würden den Rahmen des Artikels sprengen. Wichtig ist nur, dass die Server mit dem *standalone.xml* oder dem *default*-Profil gestartet werden, der zweite Server mit einem *PortOffset* von 100. Das kann z. B. für den Standalone-Modus mit dem Kommando *bin/standalone.sh -Djboss.socket.binding.port-offset=100* erfolgen.

Fügen wir jetzt den Server der Konfiguration auf dem Client hinzu. Der Liste der Connections wird der Server *two* hinzugefügt, die Properties werden entsprechend eingetragen.

Wird der Client jetzt gestartet, wird einer der beiden eingetragenen Server aufgerufen. Auf jeden Fall wird bei mehrmaligem Aufrufen einer Methode am gleichen Proxy zufällig einer der beiden Server verwendet, also ein Load Balancing erfolgen. Das liegt daran, dass entgegen des Verhaltens bei früheren JBoss-Versionen der Proxy nicht mehr dem Server zugeordnet wird, der initial verwendet wurde, sondern die EJB-Client-Library erst zum Zeitpunkt des Methodenaufrufs anhand des Identifiers einen Server auswählt. Die Selektion der Server kann mittels einer Klasse, die *org.jboss.ejb.client.DeploymentNodeSelector* implementiert, beeinflusst werden. Die Klasse muss auf dem Client verfügbar sein und durch den voll qualifizierten Namen mit der Property *deployment.node.selector* in der Konfiguration für den gesamten Client aktiviert werden.

Kann ein EJB auf zwei Servern unterschieden werden?

Da die EJB am Identifier erkannt werden, der sich aus dem Namen des Anwendungsarchivs, des *ejb*-Moduls, der Bean und dem Interface zusammensetzt, sind für den Client beide Beans als gleichwertig anzusehen. Eine Unterscheidung kann durch Ändern des Archivdateinamens erfolgen.

Weiterhin besteht die Möglichkeit, einen frei wählbaren, eindeutigen Distinct-Namen zu verwenden. Der Server oder die Anwendung muss mit diesem Distinct-Namen markiert werden.

In dem EE-Subsystem kann ein Default-Distinct-Name für den gesamten Server vergeben werden (Listing 4). Eine einzelne Applikation kann durch einen Eintrag im Deployment Descriptor *jboss-ejb3.xml* mit dem Namen markiert werden (Listing 5).

So markierte EJBs können damit auf dem Client durch die Angabe des Distinct-Namens beim Lookup unterschieden werden. Vielleicht ist Ihnen der doppelte „/“ in

Listing 1

```
remote.connections=one
remote.connection.one.host=localhost
remote.connection.one.port=4447
remote.connection.one.connect.options.org.xnio.Options.SASL_POLICY_
                                                    NOANONYMOUS=true
remote.connection.one.connect.options.org.xnio.Options.SASL_DISALLOWED_
                                                    MECHANISMS=JBOSS-LOCAL-USER
```

Listing 2

```
Properties p = new Properties()
p.put(Context.URL_PKG_PREFIXES,"org.jboss.ejb.client.naming");
InitialContext ic = new InitialContext(p);
MyBean bean = ic.lookup("ejb:app/ejb//SessionBean!org.jboss.example.Session");
bean.executeMethod();
```

Listing 3

```
remote.connections=one,two
remote.connection.one.*  # aus der Konfiguration in Listing 1
remote.connection.two.host=localhost
remote.connection.two.port=4547
remote.connection.two.connect.options.org.xnio.Options.SASL_POLICY_
                                                    NOANONYMOUS=true
remote.connection.two.connect.options.org.xnio.Options.SASL_DISALLOWED_
                                                    MECHANISMS=JBOSS-LOCAL-USER
remote.connection.two.username=user
remote.connection.two.password=password
```

Listing 4

```
<subsystem xmlns="urn:jboss:domain:ejb3:1.3">
  ...
  <default-distinct-name value="FirstServer"/>
</subsystem>
```

Listing 2 aufgefallen? Zwischen diesen beiden wird der *distinct name* angegeben:

```
ejb:app/ejb/distinct/SessionBean!org.jboss.example.Session
```

Für eine so markierte Bean muss dieser beim Lookup immer angegeben werden, andernfalls wird die EJB nicht gefunden. Ist kein Distinct-Name vorhanden, kann anstatt des doppelten auch ein einfacher Schrägstrich verwendet werden.

Listing 5

```
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
 xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns=:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
 http://www.jboss.org/j2ee/schema/jboss-ejb3-2.0.xsd
 http://java.sun.com/xml/ns/javaee
 http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
 version="3.1" impl-version="2.0">
  <enterprise-bean/>
  <jboss:distinct-name>FirstApplication</jboss:distinct-name>
</jboss:ejb-jar>
```

Listing 6

```
remote.clusters=ejb
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=true
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_DISALLOWED_
                                                    MECHANISMS=JBOSS-LOCAL-USER
remote.cluster.ejb.username=user
remote.cluster.ejb.password=password
```

Listing 7

```
<subsystem xmlns="urn:jboss:domain:ejb3:1.3">
...
  <caches>
    <cache name="clustered" passivation-store-ref="infinispan"
                                                    aliases="StatefulTreeCache"/>
  </caches>
...
  <passivation-stores>
    <cluster-passivation-store name="infinispan" cache-container="ejb"/>
  </passivation-stores>
</subsystem>
<subsystem xmlns="urn:jboss:domain:infinispan:1.3>
  ...
  <cache-container name="ejb" aliases="sfsb sfsb-cache" default-cache="repl"
                                    module="org.jboss.as.clustering.ejb3.infinispan">
    <transport lock-timeout="60000"/>
    <replicated-cache name="repl" mode="ASYNC" batching="true">
      <eviction strategy="LRU" max-entries="10000"/>
      <file-store/>
    </replicated-cache>
  ...
```

Jetzt im Cluster

Werden die Server im Cluster betrieben, also der Server zum Beispiel mit *-c standalone-ha.xml* gestartet oder in der Domain das Profil *ha* verwendet, profitieren EJBs, die als *@Clustered* markiert sind. Es werden Topologieinformationen übertragen, und Stateful Beans replizieren ihren Status für einen Failover. Der Client benötigt somit nicht jeden Server in der Konfiguration. Kann zu einem der Server, die in der Clientkonfiguration angegeben sind, eine Verbindung aufgebaut werden, so wird die EJB-Methode dort aufgerufen und gleichzeitig asynchron eine Liste der dem Cluster zugeordneten Server übertragen. Es wird versucht, zu diesen Servern im Hintergrund eine Verbindung aufzubauen, im Moment allerdings nur bis zu einer Grenze von fünfzig Verbindungen. Für diese Verbindungen ist eine gesonderte Konfiguration notwendig. Die Parameter werden auch in der *jboss-ejb-client.properties* konfiguriert (Listing 6); die *connect.options* und verwendeten Login-Daten müssen dann selbstverständlich für die initiale Verbindung und auf allen Servern im Cluster gleich konfiguriert sein.

Sobald bei der initialen Verbindung eine Cluster-Konfiguration geliefert wurde, erhält der Proxy eine Affinität zu diesem Cluster, und weitere Aufrufe werden von Letzterem verarbeitet. Die Verteilung auf die einzelnen verfügbaren Nodes erfolgt nicht mehr durch den *DeploymentNodeSelector*, sondern durch eine Implementierung des *ClusterNodeSelector*s. Dieser kann in der Konfiguration mit *remote.cluster.<cluster-name>.clusternode.selector* für jedes Cluster spezifisch angegeben werden.

Wird keine Cluster-Konfiguration angegeben, so wird der Client eine Fehlermeldung protokollieren. Allerdings ist es weiterhin möglich, die EJBs aufzurufen. Das liegt daran, dass EJB über den initialen Server weiterhin erreichbar ist. Allerdings sind die Features eines Clusters wie Discovery, Load Balancing und Failover nicht aktiv. Es existiert bereits ein Feature Request [2], um diese Konfiguration möglichst durch „Convention over Configuration“ zu ersetzen. Der in den Properties angegebene Cluster-Name *ejb* referenziert die entsprechende Serverkonfiguration, siehe Listing 7 mit einem Auszug aus der *standalone-ha.xml*.

Im Subsystem *ejb3* wird der Cache mit dem Namen *clustered* für den *SessionBean*-Container konfiguriert. Das Attribut *passivation-store-ref* verweist auf den Eintrag des *cluster-passivation-store* im Element *passivation-stores*. Das Attribut *cache-container* legt hier den zu verwendenden Namen des Clusters fest. In der Standardkonfiguration ist dieser *ejb*. Der entsprechende Cache wird im Subsystem von Infinispan konfiguriert. Dieser Cluster-Name wird auch für den Aufruf von Stateless Session Beans benutzt, allerdings wird hier nur ein Load Balancing stattfinden.

Kommunikation mittels des Remote-Naming-Projekts

Für einfache Clients kann auch das Remote-Naming-Projekt verwendet werden. Dieses setzt auf der EJB-Client-Library auf und kann nur für Standalone-Clients verwendet werden. Die Parameter werden über die JNDI Properties angegeben, es ist allerdings nur ein eingeschränkter Funktionsumfang implementiert. Aus diesem Grund ist die Verwendung der EJB-Client-Implementierung in den meisten Fällen vorzuziehen. Es fehlt die Unterstützung von erweiterten Properties, Clustern (Load Balancing, Failover, Auto-detect, Distinct-Name) und Differenzierung mittels eines Distinct-Namens. Weiterhin ist es nicht möglich, so eine Verbindung innerhalb einer JEE-Anwendung von einem AS-7-Server zu einem anderen herzustellen.

Listing 8 zeigt ein einfaches Codebeispiel für einen EJB-Aufruf. Durch die Angabe von *jboss.naming.client.ejb.context* wird im Hintergrund die EJB-Client-Library verwendet. Der Identifier für die EJB muss dann nicht mehr das Präfix *ejb:* enthalten. Beans, die in einem Server mit Default-Distinct-Namen oder einer so markierten Anwendung bereitgestellt werden, können hiermit nicht mehr aufgerufen werden. Der im Beispiel verwendete *PROVIDER_URL*, mit der Angabe mehrerer Verbindungen, ist erst in Versionen nach 7.1.1 verfügbar.

Weitere Informationen können in der Dokumentation des AS 7 unter [3] nachgelesen werden.

Es geht auch ohne jboss-ejb-client.properties

Mit etwas zusätzlichem Code kann der Client auch ohne eine *Properties*-Datei auskommen und die EJB-Client-Library benutzen. Es steht dann der volle Funktionsumfang zur Verfügung. Hierfür muss ein *EJBClientContextSelector* registriert werden, der mit den entsprechenden Daten für die Verbindung versorgt wird. Im Codebeispiel von Listing 9 habe ich auf die komplette Angabe der Properties verzichtet, es können die gleichen Properties verwendet werden wie in der *jboss-ejb-client.properties*.

Wie in Listing 9 zu sehen, wird der Selektor an dem statischen *EJBClientContext* registriert. Damit ist klar, dass nicht mehrere Selektoren gleichzeitig für den Aufruf von EJBs registriert sein können.

Etwas technischer Hintergrund

Frühere Versionen von JBoss haben das JNP-Projekt verwendet. In Clientanwendungen wurde daher ein *PROVIDER_URL* mit dem Präfix *jnp://* verwendet. Seit JBoss AS 7 (oder EAP 6) wird das JNP-Projekt weder auf der Server- noch auf der Clientseite verwendet. Es wird entweder das Remote Naming oder das EJB-

Remoting unterstützt eine feingranulare Security, das war bei JNP nicht der Fall.

Client-Projekt verwendet. Für die Verbindung wird JBoss Remoting verwendet. Hintergrund ist, dass Remoting eine feingranulare Security unterstützt, was beim JNP-Projekt nicht der Fall war.

Entgegen des von früheren Versionen bekannten Verhaltens wird der Client-Stub (EJB Proxy) nicht mehr mit dem Aufruf von *lookup()* beim Server angefordert und heruntergeladen. Das erspart eine Extraverbindung, da die benötigte Klasse von der Client-Library erzeugt wird, und führt dazu, dass Fehlermeldungen unter Umständen verzögert werden und der Proxy keinerlei Informationen mehr über den Server enthält. Es ist z. B. nicht mehr möglich, Interceptoren auf der Seite des Clients im Server zu konfigurieren. Der Selektor selbst hält nur noch eine physikalische Verbindung zu jedem Server und wickelt alle Kommunikation über diese ab. Die maximale Anzahl der gleichzeitigen EJB-Aufrufe kann über die Property *remote.connection.<name>.channel.options.org.jboss.remoting3.RemotingOptions.MAX_OUTBOUND_MESSAGES* begrenzt werden, der Default ist *80*. Weitere Aufrufe werden blockiert, bis wieder ein Slot frei wird. Das Schließen der Connection erfolgt in diesem Fall durch das Abräumen des Selektors duch den *Garbage Collector*. Das kann unter Umständen zu Problemen führen, wenn der Client den Selektor laufend überschreibt, um Verbindungs- oder Benutzerinformationen zu verändern und die veralteten Selektoren nicht schnell genug vom GC verworfen werden.

Fazit und Ausblick

Für Standalone-Anwendungen müssen der entsprechende Code für die Verbindung und die Lookups geändert werden. Multi-Threaded Clients, die EJBs mit gleichem Identifier auf unterschiedlichen Servern aufrufen oder die Benutzerinformationen zur Laufzeit ändern, müssen entsprechend angepasst werden, da der *EJBClientContext* statisch für die gesamte JVM gehalten wird und nicht wie früher an den *InitialContext* und den Proxy gebunden ist. Für solche Szenarien besteht die Möglichkeit, eigene Selektoren zu implementieren. Weiterhin gibt es bereits einen Feature Request [4], um solche komplexen Szenarien auch dynamisch mithilfe von Properties am *InitialContext* abzubilden. Diese komplexen Szenarien und *NodeSelectoren* sollen in einem weiteren Artikel genauer betrachtet werden.

Verschiedene, lauffähige Beispiele werden als Quickstarts [5] im JBoss-Developer-Framework bereitgestellt. Es ist möglich, dass bis zum Erscheinen dieses Artikels die entsprechenden Quickstarts *ejb-multi-server* und *ejb-clients* noch nicht als finale Versionen auf der Seite verlinkt sind. In dem Fall kann der Code unter [6] und [7] eingesehen werden. Um die Beispiele auf dem eigenen System auszuprobieren, können die Projekte mit Git und Maven erstellt werden.

Listing 8

```
Properties p = new Properties();
p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,"org.jboss.naming.remote.client.
                                                    InitialContextFactory");
p.put(javax.naming.Context.PROVIDER_URL,"remote://localhost:4547,remote://
                                                    localhost:5147");
p.put("jboss.naming.client.ejb.context", true);
p.put(javax.naming.Context.SECURITY_PRINCIPAL, this.user);
p.put(javax.naming.Context.SECURITY_CREDENTIALS, this.password);
InitialContext c = new InitialContext(p);
MyRemote bean = c.lookup("app/modul/MyBean!org.MyRemote");
```

Listing 9

```
Properties p = new Properties();
p.put("remote.connections", "default");
p.put("remote.connection.default.port", String.valueOf(4647));
p.put("remote.connection.default.host", "localhost");
// create the client configuration based on the given properties
EJBClientConfiguration cc = new PropertiesBasedEJBClientConfiguration(p);
// create and set the selector
ContextSelector<EJBClientContext> selector = new ConfigBasedEJBClientContextSelector(cc);
EJBClientContext.setSelector(selector);
```

Wolf-Dieter Fink ist Senior System Manintenance Engineer bei Red Hat und im Bereich Global Support für JBoss tätig. Er beschäftigt sich seit über zehn Jahren mit der Architektur und Implementierung diverser hochverfügbarer Cluster-Anwendungen im B2B- und Mobil-Billing-Umfeld basierend auf JBoss AS. Ein besonderer Schwerpunkt liegt hier auf Clustering und EJB.

Links & Literatur

[1] https://docs.jboss.org/author/display/AS72/EJB+invocations+from+a+remote+client+using+JNDI

[2] https://issues.jboss.org/browse/EJBCLIENT-47

[3] https://docs.jboss.org/author/display/AS72/Remote+EJB+invocations+via+JNDI+-+EJB+client+API+or+remote-naming+project

[4] https://issues.jboss.org/browse/EJBCLIENT-34

[5] http://www.jboss.org/developer/quickstarts.html

[6] https://github.com/wfink/jboss-as-quickstart/tree/ejb-clients/ejb-clients

[7] https://github.com/wfink/jboss-as-quickstart/tree/ejb-multi-server/ejb-multi-server

NEUE SERIE

Das Productivity-Tool JBoss Forge unter der Lupe

May the Forge be with you!

Kennen Sie das? Sie sind gerade dabei, die Projektkonfiguration für Ihre gefühlt tausendste Java-Webanwendung zu erstellen. Sie nehmen sich Ihr letztes erfolgreiches Projekt-Set-up als Vorlage. Und dann versuchen Sie, durch beherztes Copy and Paste das neue Projekt zu erstellen. Zwar hat sich durch Tools wie Maven oder Gradle in den letzten zehn Jahren sehr viel verbessert beim Set-up neuer Projekte. Dennoch sagt unsere innere Entwicklerstimme: „Das muss besser gehen." Es geht auch besser, wie die Arbeit mit dem Produktivitätswerkzeug JBoss Forge zeigt.

von Sandro Sonntag und Christian Brandenstein

Das Werkzeug JBoss Forge verspricht Rapid Application Development für Java EE. Das bedeutet, dass man aufgrund von Konventionen große Teile der Software generieren kann und dabei nicht das Wie, sondern vielmehr das Was in den Vordergrund gestellt wird. Anders als bei anderen RAD-Ansätzen handelt es sich um ein Entwicklertool, das über eine Shell bedient wird. Davon sollte man sich jedoch keineswegs abschrecken lassen, bietet die Forge Shell doch sehr komfortable Features wie Coloring, Tab-Completion, aber auch die Möglichkeit zu skripten und somit Aufgaben automatisiert auszuführen. Wer dennoch „Grün auf Schwarz" für nicht mehr zeitgemäß hält, für den wird im Rahmen der JBoss-Tools eine Forge Shell angeboten, die in einer Eclipse View dargestellt wird.

Neu ist der Ansatz einer RAD Shell nicht, gab es doch mit Ruby on Rails oder Spring Roo bereits ähnliche Vertreter dieser Toolgattung für Ruby und Spring.

Artikelserie: JBoss Forge

Teil 1: Einführung
Teil 2: Plug-ins

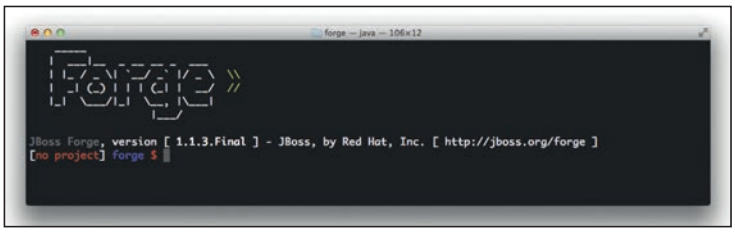


Abb. 1: JBoss Forge Shell

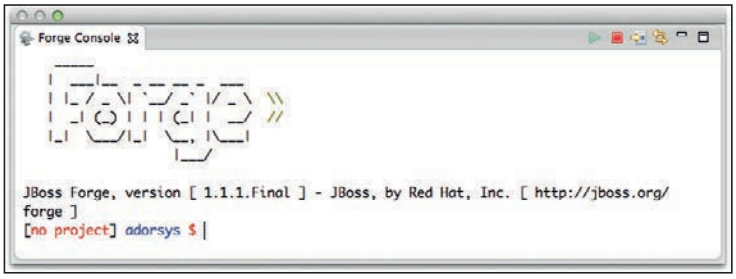


Abb. 2: JBoss-Forge-Eclipse-Plug-in

Anders als Spring Roo hat sich JBoss Forge den Java-EE-Technologien verschrieben und ist derzeit in diesem Umfeld alternativlos.

Vom Namensbestandteil JBoss sollte man sich nicht täuschen lassen. Forge ist völlig unabhängig vom JBoss Application Server und unterstützt die Generierung von Java-EE-Code für jeglichen JEE-Container. Fairerweise muss man jedoch einräumen, dass der JBoss Application Server zurzeit am besten unterstützt wird. Dies zeigt sich in erster Linie beim Deployment auf einen Entwicklungsserver. Dazu später mehr.

Forge wie auch Roo sind, vereinfacht gesehen, Codegeneratoren, die lästigen Boilerplate-Code mittels Shell-Kommandos generieren können. Anders als beim bekannten Maven-Archetype-Ansatz handelt es sich nicht um eine „One-Shot"-Generierung, sondern es ist möglich, das Projekt schrittweise durch die entsprechenden Forge-Befehle zum gewünschten Ergebnis hin zu führen. Möchte man also eine Webapplikation erstellen, die ein JSF User Interface und REST-APIs bereitstellt, würde hier der Archetype-Ansatz schnell an seine Grenzen geraten, da Archetypes nicht kombinierbar sind.

Installation

Wie bei jeder Software üblich, wird zunächst die Forge-Distribution heruntergeladen [1], entpackt, und die benötigten Betriebssystempfade (hier Windows/Mac OS X) werden gesetzt:

```
export FORGE_HOME=~/forge/
export PATH=$PATH:$FORGE_HOME/bin
```

Da Forge vollständig in Java implementiert ist, kann es prinzipiell auf jedem Betriebssystem installiert werden, für das das JDK Java 6 verfügbar ist.

Nachdem das erledigt wurde, kann, wie in **Abbildung 1** zu sehen, die Forge Shell durch Aufruf des gleichnamigen Kommandozeilenbefehls *forge* gestartet werden. Windows-Usern sei empfohlen, das JBoss-Tools-Forge-Eclipse-Plug-in [2] zu verwenden (**Abb. 2**), da Forge auf Windows-PCs keine Tab-Completion unterstützt und die Benutzbarkeit dadurch erheblich eingeschränkt wird. Die Forge-Tools können einfach in Eclipse installiert werden, indem man die JBoss-Tools-Update-Site [2] wie üblich im Eclipse-Installationsdialog einträgt. Man findet sie dann im Bereich „JBoss Web and Java EE Development".

Nachdem nun die Installationshürde genommen ist, empfiehlt es sich, ein Gefühl für die Shell zu entwickeln. Das wichtigste Kommando ist am Anfang sicherlich *list-command* oder alternativ der Tab Key. Beide geben eine Liste der verfügbaren Kommandos aus. Detailliertere Auskunft zu den Befehlen bekommt man mit *help {pluginname} {befehl}*. Wie wir bereits durch *list-command* gesehen haben, bringt Forge bereits die typischen Shell-Kommandos mit, sodass man nicht ständig zwischen einer Betriebssystem-Shell und der Forge Shell wechseln muss. Man kann beispielsweise mit *cd* Verzeichnisse wechseln, mit *mkdir* welche anlegen, mit *cat* Dateien ausgeben.

Projekte mit Forge generieren

Ein häufiger Use Case, der uns ohne unser Helferlein Forge schon mal eine Stunde beschäftigen kann, ist beispielsweise ein JAX-RS-Projekt mit EJB und JPA-Persistenz. Zum Einstieg werden wir so ein Projekt jetzt mit Forge erzeugen. Nachdem wir uns mit *cd* in das entsprechende Verzeichnis (idealerweise den Workspace der IDE) bewegt haben, können wir mit *new-project* ein Projekt anlegen (Listing 1). Forge generiert daraufhin ein Java-Maven-Projekt mit der typischen *pom.xml* und dem gängigen Verzeichnislayout. Nach Eingabe des Befehls werden wir nach dem Projektverzeichnis gefragt, was wir entweder mit Y und ENTER oder nur ENTER bestätigen. Y ist in diesem Fall die Voreinstellung. Die ist übrigens – wie bei Shell-Eingaben üblich – in Großbuchstaben dargestellt.

Wie wir jetzt schon gesehen haben, scheint Forge eine enge Freundschaft mit Maven zu pflegen. Und das ist in der Tat so: Forge-Projekte sind eng mit Maven verzahnt, was für Maven-User ein großer Vorteil ist. So können mit den Befehlen *build* und *test* Projekte gebaut und getestet werden. Und wem das nicht reicht, der hat mit dem *mvn*-Befehl vollständigen Zugriff auf Maven. Forge-Befehle können auch auf Mavens Project Object Model zugreifen und somit Dependencies managen, Plug-ins konfigurieren oder wissen, in welchem Verzeichnis sich die Tests befinden. Die Kontrolle über das Dependency Management wird schon in unserem nächsten Schritt benötigt.

Konfiguration von EJB, JAX-RS und JPA

Um EJBs und JAX-RS einsetzen zu können, benötigen wir die entsprechenden APIs als Dependency. Das erledigen wir mit *ejb setup* und wählen daraufhin das entsprechende Java-EE-API aus oder übernehmen den

Default (0) – so wie bei *rest setup*, das uns berechtigterweise auffordert, unser Projekt zu einem Webprojekt zu konvertieren und den entsprechenden Wurzelpfad für unsere RESTful-Ressourcen zu wählen.

Vorerst letztes Puzzleteil ist die Konfiguration von JPA. Das geschieht – Überraschung! – mit *persistence setup*, wenngleich hier einige Argumente mehr benötigt werden. In diesem Beispiel (Listing 2) soll Hibernate verwendet werden und ein JBoss AS 7 als Application Server dienen.

Wie erwartet, passt Forge daraufhin wieder die *pom.xml* an und erstellt eine *persistence.xml*-Datei. Die aktuelle Persistence-Konfiguration kann übrigens jederzeit mit dem Befehl *persistence* ausgegeben werden.

Entitäten und RESTful-API generieren

Jetzt fehlen uns nur noch eine JPA-Entität und das zugehörige RESTful-API. Wir generieren uns zunächst die Entität *Person* (Listing 3) und fügen die Felder *name* und *age* hinzu. Anschließend eine Entitätadresse mit den Attributen *street* , *zip* und *city*. Unsere Person sollte mehrere bekannte Adressen haben, was bedeutet, dass wir eine One-to-many-Relation auf *Adresse* brauchen. Bei genauerem Hinsehen stellt sich schnell die Frage, worauf sich das Kommando *field* bezieht. Die Antwort ist einfach: Ähnlich wie bei ei-

Listing 1

```
new-project --named myfirstwebapp --topLevelPackage javamag.forge.myfirstproject
? Use [/Users/sso/Documents/dev/forge/myfirstwebapp] as project directory? [Y/n]Y
***SUCCESS*** Created project [myfirstwebapp] in new working directory [/Users/sso/
                                                    Documents/dev/forge/myfirstwebapp]
Wrote /Users/sso/Documents/dev/forge/myfirstwebapp
Wrote /Users/sso/Documents/dev/forge/myfirstwebapp/pom.xml
Wrote /Users/sso/Documents/dev/forge/myfirstwebapp/src/main/java
Wrote /Users/sso/Documents/dev/forge/myfirstwebapp/src/test/java
Wrote /Users/sso/Documents/dev/forge/myfirstwebapp/src/main/resources
Wrote /Users/sso/Documents/dev/forge/myfirstwebapp/src/test/resources
Wrote /Users/sso/Documents/dev/forge/myfirstwebapp/src/main/java/javamag/forge/
                                                                    myfirstproject
```

Listing 2

```
persistence setup --provider HIBERNATE --container JBOSS_AS7 --database HSQLDB
***INFO*** Setting transaction-type="JTA"
***INFO*** Using example data source [java:jboss/datasources/ExampleDS]
 ? Do you want to install a JPA 2 metamodel generator? [y/N]
 ? The JPA provider [HIBERNATE], also supplies extended APIs. Install these as well? [y/N]
***SUCCESS*** Persistence (JPA) is installed.
Wrote /Users/sso/Documents/dev/forge/myfirstwebapp/src/main/resources/META-INF/
                                                                    persistence.xml
```

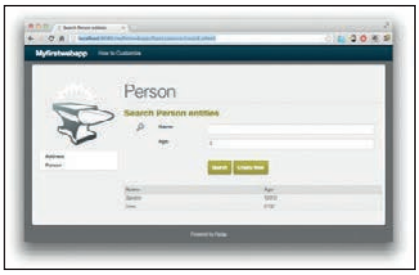


Abb. 3: Generiertes CRUD User Interface

nem Dateisystem kann man in Forge mit dem *cd*-Befehl auch auf Symbole wie Klassen, Methoden und Felder navigieren. Befehle wie *entity* oder *field* machen davon Gebrauch, indem sie im Falle von *entity* auf die erstellte Java Source navigieren und ein Attribut anlegen. Um das Ergebnis in Augenschein zu nehmen, können wir ein *ls -all* auf einer Entität aufrufen und sehen, dass uns Forge eine vollständige Entität samt Getter, Setter, ID und *hashCode-/equals*-Methoden generiert hat.

Zu guter Letzt generieren wir noch mit dem Befehl *rest endpoint-from-entity --contentType application/json* ein RESTful-API. Soweit ist unsere Anwendung vorerst komplett, sodass wir sie nun deployen können.

Installation von Plug-ins

Das Deployment auf dem JBoss Application Server erfordert die Installation eines zusätzlichen Plug-ins, da es kein Bestandteil der Forge-Distribution ist. Hierfür greift Forge auf sein Plug-in-Repository zurück. In diesem werden alle Plug-ins in einem Katalog-YAML-Repository verwaltet, das neben Metadaten wie Name, Identifier und Such-Tags auch die Adresse zum entsprechenden Code-Repository enthält. Bei Installation wird das referenzierte Plug-in-Git-Repository ausgecheckt, gebaut und in die laufende Forge-Installation deployt. Das macht es Plug-in-Entwicklern sehr einfach, neue Plug-ins bereitzustellen.

Wollen wir auf die Suche nach einem JBoss-AS-7-Plug-in gehen, geben wir *forge find-plugin as7* ein und finden das *jboss-as-7*-Plug-in. Bei der Suche können auch Wildcards verwendet werden. Da wir nun den genauen Identifier kennen, ist es ein Leichtes, mit *forge install-plugin jboss-as-7* das AS-7-Plug-in zu installieren. Möchten Sie ein Plug-in nutzen, das noch nicht in den Katalog aufgenommen wurde, können Sie dies auch mit *forge git-plugin {git url}* aus einem beliebigen Git Repository installieren.

Nachdem wir die Installation des Plug-ins durchgeführt haben, rufen wir *as7 setup* auf. Je nachdem, ob bereits eine JBoss-Installation auf dem PC existiert, kann entweder die bestehende benutzt werden oder von Forge ein JBoss installiert werden. Durch das Plug-in sind wir jetzt in der Lage, mit dem Forge-Befehl *as7 deploy* unsere Web-App zu deployen. Doch zuvor müssen wir noch den Application Server starten. Das ist zwar auch aus dem Plug-in heraus möglich, empfohlen wird es aber nicht, weil der Server ansonsten die Forge Shell blockiert. Besser ist es also, diesen in einer eigenen Shell zu starten. Wenn alles geklappt hat, kann nun per Webbrowser ein *GET* auf http://localhost:8080/myfirstwebapp/rest/persons durchgeführt werden, und ein leeres JSON-Array von Personen wird angezeigt. Unser REST-Service scheint zu funktionieren.

Gerüstbau von User Interfaces

Jetzt wird's bunt. Wir kommen zum Generieren eines User Interfaces – auch *Scaffolding* genannt. Das vor allem für Prototyping interessante Feature erzeugt uns aus Entitäten CRUD-Masken, die dann typischerweise weiter verfeinert werden. Klar ist, dass in den wenigsten Fällen ein Auftraggeber von diesem generierten Ergebnis begeistert sein wird. Dennoch ist es sinnvoll, diese als Basis für die Weiterentwicklung zu nutzen und soweit anzupassen, dass sie dem gewünschten Ergebnis entsprechen. Forge generiert in der Version 1.2.0 ausschließlich UIs für das JSF-Framework. Sollten Sie nach einer Lösung für das GWT-Framework suchen, dann werfen Sie doch mal einen Blick auf das GWT-Forge-Plug-in [3].

Wir bleiben bei JSF, aktivieren das Scaffolding und erzeugen CRUD-Masken für alle Entitäten unseres Projekts:

```
scaffold setup --scaffoldType faces;
scaffold from-entity javamag.forge.myfirstproject.model.*;
```

Der Parameter *--scaffoldType* zeigt, dass zukünftig weitere Frameworks vorgesehen sind. Nach Build und Deploy finden wir unter dem URL http://localhost:8080/myfirstwebapp/ ein CRUD User Interface (**Abb. 3**), mit dem man Personen und Adressen pflegen kann.

Was jetzt noch fehlt, ist die Eingabevalidierung. Wie wir wissen, hält Java EE hierfür die Bean Validation bereit, mit der sich deklarativ einfache Feldprüfungen realisieren lassen. Mit dem Befehl *validation setup*

Listing 3

```
[myfirstwebapp] myfirstwebapp $ entity --named Person
Created @Entity [javamag.forge.myfirstproject.model.Person]
Picked up type <JavaResource>: javamag.forge.myfirstproject.model.Person
Wrote /Users/sso/Documents/dev/forge/myfirstwebapp/src/main/java/javamag/forge/
                                                    myfirstproject/model/Person.java
[myfirstwebapp] Person.java $ field string --named name
[myfirstwebapp] Person.java $ field int --named age

[myfirstwebapp] Person.java $ entity --named Adress
Created @Entity [javamag.forge.myfirstproject.model.Adress]
Picked up type <JavaResource>: javamag.forge.myfirstproject.model.Adress
Wrote /Users/sso/Documents/dev/forge/myfirstwebapp/src/main/java/javamag/forge/
                                                    myfirstproject/model/Adress.java
[myfirstwebapp] Adress.java $ field string --named street
[myfirstwebapp] Adress.java $ field string --named zip
[myfirstwebapp] Adress.java $ field string --named city
[myfirstwebapp] Address.java $ cd ../Person.java
[myfirstwebapp] Person.java $ field oneToMany --named adressen --fieldType javamag.
                                                    forge.myfirstproject.model.Address
```

können wir die erforderlichen Dependencies sowie eine *validation.xml* generieren lassen. Weiter geht's mit dem Hinzufügen von einigen exemplarischen Validierungsregeln. Diese werden mit dem Befehl *constraint* an die selektierte Entität gehängt und ohne weiteres Zutun von der JSF-Anwendung geprüft. Die folgenden Befehle konfigurieren die *Size* und *NotNull* Constraints:

```
cd ~
cd src/main/java/javamag/forge/myfirstproject/model/Person.java
constraint NotNull --onProperty name
constraint Size --onProperty name --min 3 --max 25
```

Positiver Nebeneffekt des Ganzen ist, dass diese Constraints auch von Hibernate für die DDL-Generierung genutzt werden. Das löst ganz nebenbei das Problem, dass der *field*-Befehl keine *Constraint*-Argumente unterstützt.

Scripting

Wie es sich für eine ordentliche Shell gehört, können mit Forge auch Skripte automatisch ausgeführt werden. Das ist besonders dann nützlich, wenn sich Projekte immer wieder ähneln oder man – wie im Fall dieses Artikels – die Mächtigkeit des Tools zeigen möchte. Im Folgenden werden die wichtigsten Konzepte der Forge Shell erläutert. Für einen tieferen Einblick muss man sich noch etwas gedulden. Im Augenblick ist die Dokumentation für diesen Aspekt der Forge Shell noch etwas dürftig. Im Anhang [4] finden Sie Links zu Beispielskripten, die zur Einarbeitung sehr hilfreich sind.

Forge Script ist eine nicht typisierte, prozedurale Skriptsprache, die mit sehr wenigen Sprachkonstrukten auskommt und bei Bedarf auf die Mächtigkeit der an Java angelehnten Sprache MVEL zurückgreifen kann. Starten wir mit einem einfachen Skript und erstellen uns die Datei *projectsetup.fsh*. Alle Forge-Befehle können mit „;“ abgeschlossen werden. So kann beispielsweise mit *build; as7 deploy;* mit einer Anweisung gebaut und deployt werden.

Zur Erläuterung der Skriptsprache werden wir uns ein Skript erstellen, das uns die zeitaufwändige Aufgabe zum Erstellen eines modularisierten Java-EE-Projekts abnimmt. Sehen wir uns Listing 4 an. Unser Skript zeigt uns als Erstes, wie Kommentare eingeleitet werden. Diese beginnen mit dem @-Operator, gefolgt von einem Java-typischen Kommentar. In der Tat handelt es sich um einen Java-Kommentar, denn der @-Operator hat die Bedeutung, den folgenden Quelltext als Java-ähnlichen MVEL-2.0-Code zu interpretieren. MVEL ist eine Scripting-Sprache, die sich stark an Java orientiert.

Die folgende *echo*-Anweisung zeigt uns die Anwendung einer MVEL Expression. Expressions werden durch den *$*-Operator angeführt und evaluieren die nachfolgende Zeichenkette. So kann man beispielsweise mit *$System.getProperty("os.name")* das Betriebssystem auslesen. Das Ergebnis des Ausdrucks wird von der Shell als Befehl evaluiert. *$("he"+"lp")* würde beispielsweise die Hilfe aufrufen und *echo $1+2, 3* ausgeben. Die Anlage des Projekts haben wir in die Funktion *createComplexProject* ausgegliedert.

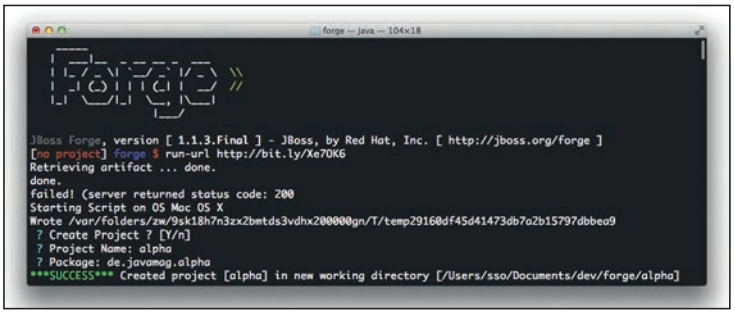


Abb. 4: Gestartetes Forge-Skipt zum Erstellen eines Java-EE-Projekts

Unser Skript soll vor der Anlage den Benutzer fragen: „Create Project?“. Dies zeigt auch gleich, wie man Benutzereingaben anfordern kann. *SHELL* ist eine vordefinierte Forge-Variable, hinter der sich die Funktionalität der Java-Schnittstelle *org.jboss.forge.shell.Shell* verbirgt. Nach erfolgreicher *if*-Bedingung wird der Benutzer noch nach dem Package sowie dem Projektnamen gefragt. Ist das erledigt, wird mit dem Operator @ die Methode *createComplexProject* ausgeführt. Da wir alle Eingaben bereits vom Benutzer angefordert haben und alle Kommandos ohne Nachfrage ausgeführt werden sollen, setzen wir die Variable *ACCEPT_DEFAULTS* auf „true“. Zwangsläufig drängt sich an dieser Stelle die Frage auf: Wie finde ich heraus, welche Variablen vordefiniert sind? Die Antwort ist: Dokumentation ist eher rar, jedoch kann man durch Aufruf des *set*-Befehls alle definierten Variablen ausgeben lassen.

Kommen wir zum Anlegen unseres Multi-Modul-Projekts, das aus einem EJB-Projekt und einer JSF-Web-App besteht und diese in einem *EAR* bündelt. Multiprojekte erstellt man, indem *new-project* in einem Maven-Projekt aufgerufen wird. Forge übernimmt dann für uns bereits die Verkettung von Parent- und Child-Modul. Die Befehle *new-project*, *ejb* und *faces* haben wir bereits kennengelernt – sie sind selbsterklärend. Was noch fehlt, sind die Abhängigkeiten zwischen den Modulen. So sollte das Frontend Zugriff auf die EJBs haben und das *EAR* ein Assembly aus EJB und *WAR* sein. Sehr schön ist, dass wir die Maven Dependencies zwischen den erzeugten Modulen einfach mit *project add-dependency* setzen können.

Dies war der letzte Schritt unseres ersten Skripts, und es ist Zeit für einen Test. Hierzu müssen wir nur noch das Skript durch *run* oder *run-url* starten. Wir entscheiden uns für *run-url http://bit.ly/Xe7OK6*, das das bereits vorbereitete Listing 4 enthält, und sollten die Applikation nun generiert und kompiliert bekommen (**Abb. 4**). Außerdem besteht die Möglichkeit, dass Forge automatisch beim Starten ein Skript ausführt

(*forge -e "run file.fsh"*). Dies macht es beispielsweise möglich, die Forge-Generierung als Teil eines Shell-Skipts zu verwenden.

Fazit

Zum Generieren von Maven-basierten Java-EE-Applikationen bekommt man mit JBoss Forge ein sehr mächtiges Produktivitätswerkzeug an die Hand. Durch Forge können Java-Entwickler wieder zum Feld der sehr produktiven Rails-Entwickler aufschließen und dennoch die Vorteile der Java-Plattform nutzen.

Vieles der Java-EE-Spezifikation wird bereits gut unterstützt. Allerdings soll nicht verschwiegen werden, dass (noch) nicht alle Features genutzt werden können und dies manuelle Nacharbeit erfordert, besonders bei JPA. Der Ansatz verfolgt jedoch auch nicht das Ziel, vollständige Applikationen zu generieren. Eher soll das Grundgerüst erzeugt werden, um die Verfeinerungen dann auszuimplementieren.

Forge Script indes kommt weniger ausgereift daher, was die fehlende Dokumentation auch nicht gerade verbessert. Der Mix aus zwei Sprachen, Forge und MVEL, führt häufig zu Parser-Problemen, bei denen man vor der Frage steht: „Wie bring ich's ihm bei?" Besonders die Fehlersuche bei Syntaxfehlern ist mangels Zeilenangabe kein Vergnügen. Für die meisten Fälle ist es jedoch ausreichend, und mit MVEL kann auch schon sehr viel erreicht werden.

Die wahre Mächtigkeit von Forge liegt in der Erweiterbarkeit und dem Plug-in-System, das es dem Entwickler sehr einfach macht, Plug-ins auf Basis von CDI zu entwickeln. Mehr darüber, wie man Forge um eigene Befehle erweitert, lesen Sie im nächsten Java Magazin.

Listing 4

```
@/** Forge Script zum Erstellen eines Multi-Module-EAR-Projekts mit EJB und WAR Teil*/;

echo  "Starting Script on OS" $System.getProperty("os.name");

def createComplexProject(projectName, package) {
  set ACCEPT_DEFAULTS true;
  new-project --named $projectName --topLevelPackage $package --type pom;

  new-project --named $projectName.concat(".service") --topLevelPackage $package
                                                    .concat(".service") --type jar;
  maven set-groupid $package;maven set-artifactid $projectName.concat(".service");
  ejb setup;
  cd ..;

  new-project --named $projectName.concat(".gui") --topLevelPackage $package
                                                  .concat(".gui") --type war;
  maven set-groupid $package; maven set-artifactid $projectName.concat(".gui");
  project add-dependency $package.concat(":").concat(projectName)
                                              .concat(".service:1.0.0-SNAPSHOT");
  faces setup;
  cd ..;

  @/* Leider kennt Forge aktuell keinen Typ ear, dies muss nachtraeglich im pom xml
                                                                  geaendert werden */;
  new-project --named $projectName.concat(".ear") --topLevelPackage $package
                                                  .concat(".ear") --type pom;
  maven set-groupid $package;
  maven set-artifactid $projectName.concat(".ear");
  project add-dependency $package.concat(":").concat(projectName)
                                              .concat(".service:1.0.0-SNAPSHOT");
  project add-dependency $package.concat(":").concat(projectName)
                                              .concat(".gui:1.0.0-SNAPSHOT");
  cd ..;

  build;

  set ACCEPT_DEFAULTS false;
};

if ( SHELL.promptBoolean("Create Project ?") ) {
  @projectName = SHELL.prompt("Project Name:");
  @packageName = SHELL.prompt("Package:");
  @createComplexProject(projectName, packageName);
};
```

Sandro Sonntag beschäftigt sich als Technical Lead bei der adorsys GmbH & Co. KG seit vielen Jahren mit leichtgewichtigen Architekturen auf Basis von Spring und Java EE. In den letzten Jahren lag dabei sein besonderes Interesse auf hochskalierbaren RESTful-„Thin-Server"-Architekturen, NoSQL DBs sowie Webclients mit HTML5 und JavaScript.

Christian Brandenstein ist Germanist und arbeitet seit vielen Jahren als Softwareentwickler. Er ist Mitarbeiter der ersten Stunde der adorsys GmbH & Co. KG. Sein besonderes Interesse gilt dem Test-driven Development und CDI.

Links & Literatur

[1] http://forge.jboss.org/

[2] http://download.jboss.org/jbosstools/updates/stable/indigo/

[3] http://forge-gwtplugin.github.com/

[4] https://github.com/forge/core/tree/master/showcase

[5] https://raw.github.com/xandrox/javamag-forge-samples/master/scaffolding.fsh

[6] https://raw.github.com/xandrox/javamag-forge-samples/master/projectsetup.fsh

Kolumne: EnterpriseTales

von Lars Röwekamp und Arne Limburg

Was du später kannst besorgen: Lazy Loading in JPA 2.1

In der letzten Kolumne haben wir einen Blick über den Java-EE-Standard-Tellerrand hinaus geworfen und das alternative Query-API „QueryDSL" vorgestellt. Heute kehren wir wieder zurück in den JPA-Standard. Dort gibt es seit dem 01.03.2013 den „Proposed Final Draft" der Version 2.1. Und dieser bringt ein interessantes neues Feature im Bereich *Lazy Loading* mit sich. Bei diesem neuen JPA-Feature geht es darum, Daten erst zu einem späteren Zeitpunkt zu „besorgen", nämlich dann, wenn sie tatsächlich gebraucht werden. Dass dieses Feature auch seine Tücken hat und welche Neuerung JPA 2.1 hier bringt, wollen wir im Folgenden zeigen.

Jeder erfahrene Java-EE-Entwickler kennt das Problem: In der klassischen Java-EE-Schichtenarchitektur besteht die Serviceschicht aus EJBs. In diese wird via *@PersistenceContext* ein EntityManager injiziert. Die EJB-Schicht stellt gleichzeitig die Transaktionsgrenze dar, was bedeutet, dass der EntityManager standardmäßig nach Verlassen der EJB-Schicht geschlossen wird. Nach dem Schließen funktioniert das Nachladen von Beziehungen, das so genannte Lazy Loading, aber nicht mehr. Dies führt zu hässlichen Exceptions, wenn man z. B. in der Facelets-Seite über eine Collection einer Entity iterieren will (Listing 1 und 2), die nicht bereits in der EJB-Schicht geladen wurde.

Wie kann aber sichergestellt werden, dass die benötigte Beziehung bereits in der EJB-Schicht geladen wurde? Der JPA-Standard 2.0 bietet hier zwei Möglichkeiten: Eager Fetching im Mapping oder Fetching über die Query. Eine dritte Möglichkeit, nämlich, dass innerhalb der Transaktionsgrenze „manuell" auf die benötigten Attribute zugegriffen wird, um ein Laden zu forcieren, wird hier nicht näher betrachtet.

Eager Fetching im Mapping

JPA bietet die Möglichkeit, im Mapping anzugeben, ob eine Beziehung gleich mitgeladen werden soll, wenn die eigentliche Entität geladen wird. Dies ist möglich, indem man im Mapping den *FetchType.EAGER* angibt. Dieser ist bei To-one-Beziehungen (also *@ManyToOne* und *@OneToOne*) der Default. Bei To-many-Beziehungen (also *@OneToMany* und *@ManyToMany*) dagegen muss er explizit angegeben werden. In Listing 1 würde dies bedeuten, dass dank expliziter Angabe des FetchTyps beim Laden der Person automatisch auch die Adressen mitgeladen würden.

Die Angabe von *FetchType.EAGER* im Mapping hat allerdings auch Nachteile. Zum einen sollte diese Variante auf keinen Fall an jeder Beziehung angegeben werden. Das würde zwar das Lazy Loading inkl. der damit einhergehenden, oben beschriebenen Probleme komplett ausschalten. Es würde aber auch dazu führen, dass man beim Laden einer Entität potenziell den gesamten Inhalt der Datenbank in den Speicher lädt, nämlich genau dann, wenn alle Entitäten miteinander verbunden sind. Dies ist sowohl vom Speicherbedarf als auch von der Menge der Datenbankoperationen her eine mittlere bis schwerwiegende Katastrophe.

Aber selbst, wenn man mit der Angabe von *FetchType.EAGER* gezielt umgeht und es nur an solchen Beziehungen angibt, wo ein gemeinsames Laden Sinn

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

macht, hat die Angabe den Nachteil, dass sich immer Use Cases finden lassen (und sei es in ferner Zukunft), in denen man die Objekte nicht gemeinsam laden will. Einen einmal eingeführten *FetchType.EAGER* wird man in der Praxis nur schwer wieder los, da sich die Auswirkungen auf die anderen Programmteile nicht absehen lassen (wohl dem, der dann ausreichend Unit Tests geschrieben hat).

FetchType.EAGER hat einen weiteren unerwünschten Nebeneffekt: Im JPA-Standard ist nicht spezifiziert mit wie vielen *SELECTs* die Abhängigkeiten geladen werden sollen. Man hat hier also Anzahl und Umfang der Datenbankoperationen nicht im Griff, was zu einem weiteren „berüchtigten" JPA-Problem führen kann: den n+1 *SELECTs*. Beim Iterieren über eine Liste kann es dadurch vorkommen, dass zunächst ein *SELECT* ausgeführt wird, um die Liste (lazy) zu laden und dann n weitere, um die Entitäten, die *EAGER* an den Listeneinträgen hängen, nachzuladen. n entspricht dabei der Größe der Liste und kann damit potenziell recht groß werden. Einige JPA-Anbieter erkennen das n+1-Problem mittlerweile und führen dann nur ein weiteres *SELECT* für die anhängenden Entitäten aus. Diese Optimierung ist aber längst nicht bei allen Anbietern umgesetzt.

Listing 1

```
@Entity
public class Person {
   ...
   @OneToMany(fetch = FetchType.EAGER)
   private List<Address> addresses;
   ...
}
```

Listing 2

```
@Entity
@NamedEntityGraph(
   name = "Person.addresses",
   attributeNodes = @NamedAttributeNode("addresses")
)
public class Person {
   ...
   @OneToMany(fetch = FetchType.LAZY) // default fetch type
   private List<Address> addresses;
   ...
}
```

Listing 3

```
EntityManager entityManager  = ...
EntityGraph entityGraph = entityManager.getEntityGraph("Person.addresses");

Map<String, Object> hints = new HashMap<String, Object>();
hints.put("javax.persistence.fetchgraph", entityGraph);
Person person = entityManager.find(Person.class, id, hints);

TypedQuery<Person> query
   = entityManager.createNamedQuery("Person.findAll", Person.class);
query.setHint("javax.persistence.loadgraph", entityGraph);
List<Person> persons = query.getResultList();
```

Listing 4

```
EntityManager entityManager  = ...
EntityGraph entityGraph = entityManager.createEntityGraph(Person.class);
entityGraph.addAttributeNodes("addresses");

Map<String, Object> hints = new HashMap<String, Object>();
hints.put("javax.persistence.fetchgraph", entityGraph);
Person person = entityManager.find(Person.class, id, hints);

EntityManagerFactory factory = entityManager.getEntityManagerFactory();
factory.addNamedEntityGraph("Person.addresses", entityGraph);
```

Fetching über die Query

Eine weitere Möglichkeit zu beeinflussen, welche Teile meines Entity-Graphen geladen werden sollen, bieten die Abfragesprachen von JPA. Sowohl in der JPQL (Java Persistence Query Language) als auch in dem Criteria-API ist es möglich, explizit anzugeben, welche Daten direkt mitgeladen werden sollen. Hiermit kann man sogar direkt steuern, wie viele Abfragen zum Laden abgesetzt werden.

Ein Nachteil des Fetchings über die Query ist, dass man Use-Case-bezogene Queries schreiben muss. Man kann sich vom EntityManager z. B. nicht direkt die Person mit der ID 5 geben lassen, sondern muss, je nach Use Case eine andere Query absetzen, um den korrekten Entity-Graphen mitzuladen. Das ist ärgerlich, weil auf diese Weise eine ganze Reihe Use-Case-spezifischer Abfragen entstehen kann, die eigentlich dasselbe tun, nämlich eine Entity mit einer bestimmten ID zu laden.

Beim Laden einer Entity zu einer konkreten ID mag sich der Ärger noch in Grenzen halten, aber spätestens, wenn man komplexe Abfragen (z. B. Suchabfragen) mit einer komplizierten *WHERE* Clause und mehreren Parametern benötigt, möchte man diese im Quellcode nicht kopieren müssen, nur um Use-Case-bezogen verschiedene Daten mitzuladen.

Entity Graphs

Genau hier setzt das neue Feature von JPA 2.1 ein, die Entity Graphs. Die Idee ist es, die Angabe der Fetch/Load Graphs von der Abfrage zu trennen, um sich in der Abfrage auf die Logik zu konzentrieren und bei der Angabe der Fetch/Load Graphs auf die Spezifikation, welche Daten geladen werden sollen.

Dabei wird es mehrere Möglichkeiten geben, einen solchen Entity Graph zu erstellen. Die erste Möglichkeit

ist, den Entity Graph via Annotation an der Entity selbst anzugeben (Listing 2) und ihn dann später über seinen Namen zu referenzieren, z. B. in einer *find*-Operation oder in einer Query (Listing 3). Hierbei wird zwischen den Properties *javax.persistence.fetchgraph* und *javax.persistence.loadgraph* unterschieden. Der Unterschied beider Properties liegt dabei in der Behandlung von Attributen, die nicht im jeweiligen Entity-Graphen angegeben sind: Während beim Fetch-Graphen alle nicht angegebenen Attribute automatisch lazy sind, behalten sie beim Load-Graphen ihr im Mapping konfiguriertes Fetch-Verhalten und werden, falls sie als *EAGER* gemappt sind, zusätzlich zu den im Load-Graphen angegebenen Attributen geladen.

Die zweite Möglichkeit, einen Entity-Graphen zu erstellen, ist, dieses dynamisch im Code zu tun. Hierfür bietet JPA 2.1 ein eigenes API. Über *EntityManager.createEntityGraph* kann ein neuer Entity Graph erzeugt werden. Dieser kann dann im Nachgang durch das Hinzufügen von Attributen und Sub-Graphen mit Leben gefüllt werden (Listing 4). Ein so erzeugter Entity-Graph kann sogar in der *EntityManagerFactory* über *addNamedEntityGraph* registriert werden und muss so im weiteren Programmverlauf nicht immer wieder neu erzeugt werden.

Fazit

Lazy Loading ist nach wie vor ein immer wiederkehrendes Problem für Entwickler, die JPA zum Mapping von Objektmodell und relationaler Datenbank verwenden. Mit der Version 2.1 wird eine neue Möglichkeit eingeführt, Use-Case-bezogen zu definieren, welche Entitäten mit einer Datenbankabfrage in den Speicher geladen werden sollen. Dieses Feature der Entity-Graphen wurde hier näher vorgestellt. Der große Vorteil von Entity-Graphen ist es, dass sie es ermöglichen, Lazy Loading pro Use Case zu spezifizieren und das ggf. schon in der Serviceschicht. Auf diese Weise wird eine Use-Case-unabhängige Datenzugriffsschicht ermöglicht.

Neben dem Standardisieren von DB-Schemagenerationon sind die Entity-Graphen sicher das große neue Feature in JPA 2.1. Mit ihnen wird ein weiteres Feature, das bisher nur in verschiedenen Persistenzanbietern proprietär vorhanden war, in den Standard aufgenommen.

Links & Literatur

[1] http://jcp.org/en/jsr/detail?id=338

[2] http://wiki.eclipse.org/EclipseLink/Examples/JPA/AttributeGroup

[3] http://openjpa.apache.org/builds/2.2.1/apache-openjpa/docs/ref_guide_fetch.html

RabbitMQ mit CDI integrieren

Wie der Hase läuft

RabbitMQ erfreut sich am Markt der leichtgewichtigen Message Broker zunehmend wachsender Beliebtheit. Die Kombination aus Simplizität und Robustheit machen den Broker besonders in kleinen und mittleren Unternehmen zu einer ernstzunehmenden Alternative zu etablierten Messaging-Plattformen. Dieser Artikel, eine erweiterte und aktualisierte Version des in Ausgabe 3.2013 erschienenen Beitrags „RabbitMQ und CDI“, zeigt, wie man mithilfe von CDI eine elegante Einbindung von RabbitMQ implementiert – insbesondere in Java-EE-Umgebungen.

von Christian Bick

RabbitMQ [1] ist eine Open-Source-Message-Broker-Software, die das Advanced Message Queuing Protocol (AMQP) implementiert. Mithilfe dieser Middleware ist das Zusammenspiel von Komponenten einer Event-driven Architecture über verschiedene Plattformen hinweg deutlich einfacher geworden: Zusammen mit einer aktiven Community hat die Firma VMware Clients für die meisten gängigen Programmiersprachen hervorgebracht. RabbitMQ präsentiert sich damit insbesondere in einer heterogenen, Java-dominierten Systemlandschaft als interessante Option gegenüber dem klassischen JMS-Ansatz.

Der bereits angesprochene Java-Client von VMware ist sehr robust und unterstützt das vollständige Featureset von RabbitMQ. Dessen ungeachtet ist die Integration in Java EE suboptimal, was u. a. daran liegt, dass sich deutlich zu viele technische Details im fachlichen Teil des Codes befinden und – im Gegensatz zu JMS – kein standardisierter Weg der Broker-Konfiguration existiert. Diese Nachteile der RabbitMQ-Middleware können aber behoben werden.

Der naive Ansatz, JMS mit RabbitMQ zu verbinden, führt schnell zu unerfreulichen Kompromissen. Besonders heikel wird es bei dem Versuch, die verschiedenen Philosophien hinter Transaktionen zu integrieren: Während bei AMQP und somit auch bei RabbitMQ Transaktionen am Broker enden, können sie bei JMS mehrere Clients überspannen. Unter solchen Voraussetzungen

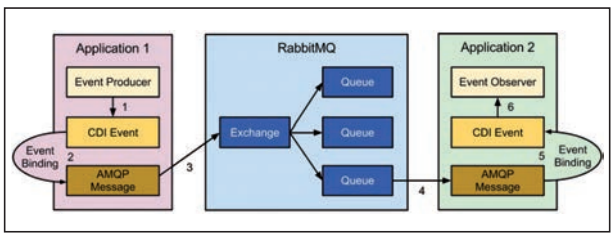


Abb. 1: Austauschen von CDI Events

erscheint es wenig erstrebenswert, diesen Ansatz weiterzuverfolgen. Es existieren durchaus weitere Möglichkeiten – z. B. mithilfe von Spring-Integration [2] –, eine Brücke zwischen JMS und AMQP zu schlagen. Doch alle haben signifikante Nachteile, sodass es sich lohnt, über Alternativen jenseits von JMS nachzudenken. Aus der Etablierung von Java EE 6 und der damit verbundenen Auseinandersetzung mit CDI (Contexts and Dependency Injection) entsprang deshalb die Überlegung, *CDI Events* als Ausgangspunkt für die Integration von RabbitMQ in Java EE zu nutzen.

Die Idee hinter diesem Ansatz ist folgende: CDI Events werden für das Publizieren zum Broker und das Konsumieren von selbigem an die entsprechenden Broker-Entitäten gebunden. Bei AMQP 0.9.1 werden AMQP-Nachrichten an *Exchanges* versendet und anhand des *Routing Keys* in die richtigen Queues weitergeleitet. Von diesen Queues können die Nachrichten wiederum von verschiedenen Applikationen konsumiert werden. Somit müssen gefeuerte CDI Events in AMQP-Nachrichten umgewandelt und an diese Exchanges mit korrektem Routing Key gesendet werden. Konsumierte Nachrichten müssen wieder zu entsprechenden CDI Events zurückgewandelt werden, die dann erneut lokal gefeuert werden, damit sie von den entsprechenden CDI Observern verarbeitet werden können. **Abbildung 1** veranschaulicht, wie CDI Events mithilfe von RabbitMQ über Applikationsgrenzen hinweg ausgetauscht werden.

Werden in der Applikation CDI Events gefeuert (1), so wird für zuvor gebundene Events (2) eine AMQP-Nachricht an den entsprechenden Exchange mit dem festgelegten Routing Key publiziert (3). Diese Nachrichten werden auf der Konsumentenseite von Queues konsumiert (4) und dann wiederum als gebundene CDI Events lokal propagiert (5). CDI Event Observer können dann auf diese Events lauschen (6).

Um das vorgestellte Konzept mit möglichst geringem Aufwand umzusetzen, wurde hierfür RabbitMQs Java-

Client erweitert: Das Resultat ist RabbitEasy [3], eine schlanke Open-Source-Bibliothek, die neben der Erweiterung für CDI-getriebenes Messaging weitere nützliche Funktionalität mitbringt, u. a. auch im Bereich der Testautomatisierung. Um die praktische Umsetzung des Ansatzes zu demonstrieren, wurde außerdem ein einfaches Beispielprojekt, RabbitOrdering, erstellt, das wie RabbitEasy auf GitHub zu finden ist [4].

Die praktische Umsetzung

Anhand von RabbitOrdering soll hier die praktische Umsetzung der Integration von RabbitMQ mithilfe von CDI veranschaulicht werden. Die Aufgabe, die das folgende kleine Beispielprogramm bewältigt, besteht darin, Bestellungen aufzunehmen, diese zu bezahlen und auszuliefern. Parallel dazu soll eine vollkommen unabhängige Statistikkomponente die aktuelle Anzahl der noch unbezahlten (unpayed), noch auszuliefernden (undelivered), bereits ausgelieferten (delivered) sowie die Gesamtzahl (total) der Bestellungen anzeigen. Die beiden Komponenten sollen dafür lediglich über CDI Events bzw. Messaging miteinander kommunizieren.

Um das Beispiel nachzuvollziehen, wird eine lokal installierte Version von RabbitMQ (2.7.x oder höher, Port: 55672) benötigt. Außerdem sollte Git installiert sein, um das Projekt zu klonen [5].

Das Beispiel wird gestartet, indem die Klasse *OrderProcessConsole* im Modul *rabbitordering-client* aus einer beliebigen IDE heraus ausgeführt wird. Dabei wird nicht nur die Bestellapplikation gestartet, sondern es werden auch alle notwendigen *Exchanges*, *Queues* und *Bindings* erstellt. Nach dem Starten sollte folgende Konfiguration auf dem Broker hinzugefügt worden sein:

Exchanges:
- *com.zanox.rabbitordering*

Queues:
- *com.zanox.rabbitordering.order.created-statistics*
- *com.zanox.rabbitordering.order.payed-statistics*
- *com.zanox.rabbitordering.order.delivered-statistics*

Dabei sind alle Queues an den Exchange *com.zanox.rabbitordering* mit den Routing Keys *com.zanox.rabbitordering.order.created/payed /delivered* gebunden (**Abb. 2**).

Darüber hinaus sollte sich Folgendes als Teil der Konsolenausgabe für *OrderCreatedEvent*, *OrderPayedEvent* und *OrderDeliveredEvent* wiederfinden:

```
...
Binding created between exchange com.zanox.rabbitordering and event type
OrderCreatedEvent
Routing key for event type OrderCreatedEvent set to
com.zanox.rabbitordering.order.created
Persistent messages enabled for event type OrderCreatedEvent
Publisher reliability for event type OrderCreatedEvent set to CONFIRMED
...
```

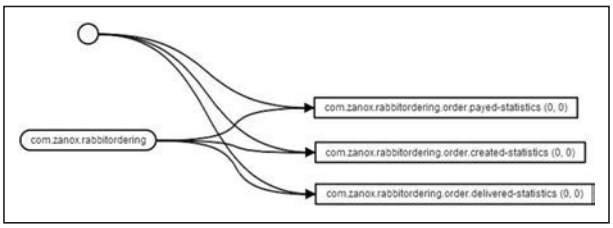


Abb. 2: Visualisierung der Broker-Konfiguration

Listing 1

```
public class OrderProcessEventBinder extends EventBinder {

@Override
protected void bindEvents() {

   bind(OrderCreatedEvent.class).toExchange("com.zanox.rabbitordering")
         .withRoutingKey("com.zanox.rabbitordering.order.created")
         .withPersistentMessages()
         .withPublisherConfirms();
   bind(OrderPayedEvent.class).toExchange("com.zanox.rabbitordering")
         .withRoutingKey("com.zanox.rabbitordering.order.payed")
         .withPersistentMessages()
         .withPublisherConfirms();
   bind(OrderDeliveredEvent.class).toExchange("com.zanox.
                                                      rabbitordering")
         .withRoutingKey("com.zanox.rabbitordering.order.delivered")
         .withPersistentMessages()
         .withPublisherConfirms();
  }
}
```

Listing 2

```
@Singleton
public class OrderProcess {

@Inject
   private ShopPersistence persistence;

   @Inject
   private Event<OrderCreatedEvent> orderCreatedEventControl;

   ...

   public Order create(String username, String articleId, int amount) {
      Article article = persistence.getArticle(articleId);
      Customer customer = persistence.getCustomer(username);
      Order order = new Order(customer, article, amount);
      persistence.createOrder(order);
      OrderCreatedEvent orderCreatedEvent = new OrderCreatedEvent();
      orderCreatedEvent.setId(order.id);
      orderCreatedEventControl.fire(orderCreatedEvent);
      return order;
   }

   ...
 }
```

Abb. 3: Übersicht über die Queues am Broker

Overview				Messages		
Name	Exclusive	Parameters	Status	Ready	Unacked	Total
com.zanox.rabbitordering.order.created-statistics			Idle	1	0	1
com.zanox.rabbitordering.order.delivered-statistics			Idle	0	0	0
com.zanox.rabbitordering.order.payed-statistics			Idle	0	0	0

Hier wird beim Initialisieren der Applikation das CDI Event *OrderCreatedEvent* an den Exchange *com.zanox.rabbitordering* mit Routing Key *com.zanox.rabbitordering.order.created* gebunden. Darüber hinaus werden noch Informationen über die Art der publizierten AMQP-Nachricht (*persistent*) und der Publizierungsmethode (*confirmed*) ausgegeben. Für die beiden anderen CDI Events *OrderPayedEvent* und *OrderDeliveredEvent* gilt das Gleiche.

Werfen wir einen Blick in die Klasse *OrderProcessEventBinder* (*rabbitordering-client*), so finden wir dort den hierfür verantwortlichen Code (Listing 1).
Die Klasse leitet von *EventBinder* ab und überschreibt dessen Methode *bindEvents()*. Beim Initialisieren der Applikation wird auch der Event Binder initialisiert und dabei der Konfigurationscode durchlaufen. Das Binden der geworfenen CDI Events an den jeweiligen Exchange erfolgt dabei in Form von möglichst sprechendem Code. Dabei bleiben die Konfigurationsmöglichkeiten, die AMQP zum Publizieren ermöglicht, vollkommen erhalten. Selbige Vorgehensweise findet sich auch auf der Konsumentenseite wieder. Zum Publizieren eines CDI Events muss dieses nun lediglich noch gefeuert werden. Dies unterscheidet sich in keiner Form vom üblichen Feuern eines CDI Events. Der entsprechende Code für das Erstellen einer Bestellung findet sich in der Klasse *OrderProcess (rabbitordering-client)*, siehe Listing 2.

Offensichtlich wird hier eine Bestellung für einen bestimmten Benutzer entgegengenommen, die außerdem eine Artikelnummer mit entsprechender Anzahl erwartet. Am Ende der Methode wird auf dem CDI Event *Control orderCreatedEventControl* die *fire()*-Methode aufgerufen. Durch das vorhergehende Binding wird auf dieses CDI Event nun gelauscht, es daraufhin direkt in eine AMQP-Nachricht umgewandelt und an den konfigurierten Exchange publiziert. Dieser Prozess lässt sich mit der zuvor gestarteten Konsolenapplikation ausprobieren, indem wir für Steph Jobs eine Bestellung von zwei Handschuhen aufgeben:

```
create stephj gloves 2
```

In der Konsole findet sich nun folgende Log-Ausgabe, die zeigt, wie das *OrderCreatedEvent* entsprechend der obigen Konfiguration an den Exchange *com.zanox.rabbitordering* mit dem angegebenen Routing Key publiziert wurde:

```
Publishing event of type OrderCreatedEvent
Publishing message to exchange 'com.zanox.rabbitordering' with routing key
  'com.zanox.rabbitordering.order.created' (deliveryOptions: NONE, persistent:
                                                                    true)
Order 456af25f-fb02-4953-9000-38e12cabd727 of 2 Gloves(s) for Steph Jobs
                                                                    created
```

Entsprechend der Konfiguration am Broker muss sich die Nachricht dort in der Queue *com.zanox.rabbitordering.order.created-statistics* befinden, da wir die konsumierende Applikation, *rabbitordering-statistics*, noch nicht gestartet haben. Ein Blick in die Queue-Übersicht im RabbitMQ-Management-Tool, sollte dementsprechend wie in **Abbildung 3** aussehen.

Zum Konsumieren der Nachricht starten wir jetzt die zweite Applikation, *OrderStatisticsConsole* im Modul *rabbitordering-statistics* aus der IDE heraus. Beim Initialisieren der Applikation finden wir wieder die Anbindung der drei CDI Events im Log. Diesmal werden sie jedoch als konsumierte Events an die entsprechenden Queues gebunden:

```
...
Binding created between queue com.zanox.rabbitordering.order.created-
                                    statistics and event type OrderCreatedEvent
...
```

Genau wie beim Publizieren ist hier eine Ableitung der Klasse *EventBinder* der Ort, an dem das Binding umgesetzt wird, in diesem Fall die Klasse *OrderStatisticsEventBinder (rabbitordering-statistics)*, siehe Listing 3.

Listing 3

```
public class OrderStatisticsEventBinder extends EventBinder {
 @Override
 protected void bindEvents() {
      bind(OrderCreatedEvent.class)
          .toQueue("com.zanox.rabbitordering.order.created-statistics");

      bind(OrderPayedEvent.class)
     .toQueue("com.zanox.rabbitordering.order.payed-statistics");

      bind(OrderDeliveredEvent.class)
      .toQueue("com.zanox.rabbitordering.order.delivered-statistics");
}

 }
```

Listing 4

```
@Singleton
public class OrderStatistics {
    private volatile int waitingForPayment;
    ...
    public void onOrderCreated(@Observes OrderCreatedEvent event) {
       waitingForPayment++;
    }

   ...
}
```

Konsumierte CDI Events werden genauso an Queues gebunden wie gefeuerte Events an Exchanges gebunden werden. Der Eventtyp wird mit dem Queue-Namen verknüpft. Infolgedessen werden für jedes *Binding* Konsumenten am Broker registriert, die einkommende AMQP-Nachrichten in das entsprechende CDI Event umwandeln und dann feuern. Auch an dieser Stelle können alle AMQP-spezifischen Konfigurationen, wie z. B. Auto-Acknowledgements genutzt werden.

Um die resultierenden CDI Events zu verarbeiten, muss – genau wie bei gewöhnlichem CDI – lediglich die *@Observes*-Annotation in einer Methodensignatur zusammen mit dem Eventobjekt verwendet werden (Listing 4).

Da die zuvor publizierte Nachricht über die erstellte Bestellung sofort nach dem Initialisieren der Statistikapplikation konsumiert wurde, muss dieser Code bereits einmal durchlaufen worden sein. Um dies zu testen, tippen wir *print* in die Konsole der *OrderStatisticsConsole* ein und sollten daraufhin folgende Ausgabe erhalten:

```
Total orders: 1
Unpayed orders: 1
Undelivered orders: 0
Delivered orders: 0
```

Das Ausführen weiterer Befehle in der Bestellapplikation verändert jetzt die Statistik entsprechend unmittelbar. Ab dieser Stelle sei ein wenig Experimentierfreudigkeit und der Einsatz des Debuggers ans Herz gelegt, um tiefere Erkenntnisse zur vergleichsweise simplen Implementierung des vorgestellten Ansatzes zu gewinnen.

Mehr als lesbarer Code

Wie das Beispiel zeigt, ist die Broker-spezifische Konfiguration von der fachlichen Implementierung sauber getrennt. Im fachlichen Code ist nur CDI präsent, also pures Java EE. Doch die Lösung bietet neben sauberem Code auch noch weitere Vorteile:

- Möglichkeit der Nutzung aller CDI-Features, z. B. von Qualifiers
- Austauschbarkeit der Event-Publizierung ohne Änderungen im fachlichen Code
- Verwendung auch in Java SE möglich (z. B. mit Weld)
- Einfache Erstellung von Prototypen ohne Message-Broker-Infrastruktur
- Einfaches Testen des fachlichen Codes ohne Message-Broker-Infrastruktur

Die letzten beiden Punkte sind besonders erwähnenswert, da sie einen ganz konkreten geschäftlichen Nutzen bieten: In der frühen Phase der Entwicklung eines Projekts kann eine einfache Applikation entwickelt werden, deren Module noch in derselben JVM „leben“ und die sich zunächst via CDI Events austauschen. In einer späteren Projektphase, wenn der Produktivbetrieb in greifbare Nähe gerückt ist und Skalierbarkeit wichtig wird, können die Module getrennt und mehrfach deployt werden. Die vormals lokalen CDI Events können dann via RabbitMQ ausgetauscht werden. Dies ist möglich, ohne den Code aus dem Prototypen grundlegend anpassen zu müssen, sondern geschieht durch Ergänzungen an zentraler Stelle.

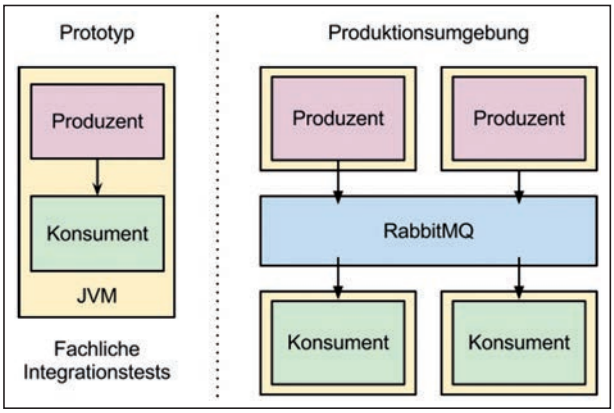


Abb. 4: Übersicht über die Queues am Broker

Dasselbe Vorgehen lässt sich zum Testen der fachlichen Logik auch umkehren: Module, die via RabbitMQ CDI Events austauschen, können zum integrativen Testen der fachlichen Logik wieder lokal zusammengeführt werden. Durch simples Abhängigkeitenmanagement (Maven, Gradle) kann die fachliche Logik in gewöhnlichen Unit Tests getestet werden, ohne dafür eine Messaging-Infrastruktur zu benötigen. Dies wird wieder durch die natürliche Kopplung über CDI ermöglicht, und ohne dass dafür Hilfsmittel wie Mocks notwendig wären.

Zusammenfassend lässt sich sagen, dass der Ansatz eine elegante Integration von RabbitMQ in Java EE ermöglicht, ohne dass dabei wesentliche Kompromisse auf der einen oder der anderen Seite eingegangen werden müssen. Im Gegenteil: Es entstehen neue Synergien zwischen den Technologien, die sich vom Anfangsstadium bis zur Reife einer Applikation positiv auf die Entwicklung auswirken. Dabei bleibt die Interoperabilität mit Clients anderer Programmiersprachen unangetastet, da diese nach wie vor als Produzenten oder Konsumenten der gesendeten und empfangenen AMQP-Nachrichten auftreten können.

Christian Bick arbeitet als Softwareentwickler und beschäftigt sich mit unterschiedlichsten Aspekten von Webanwendungen. Bei zanox ist er dabei insbesondere für die Architekturdomäne „Messaging mit RabbitMQ“ verantwortlich und veröffentlicht zu diesem Thema regelmäßig Beiträge in seinem Blog.

Links & Literatur

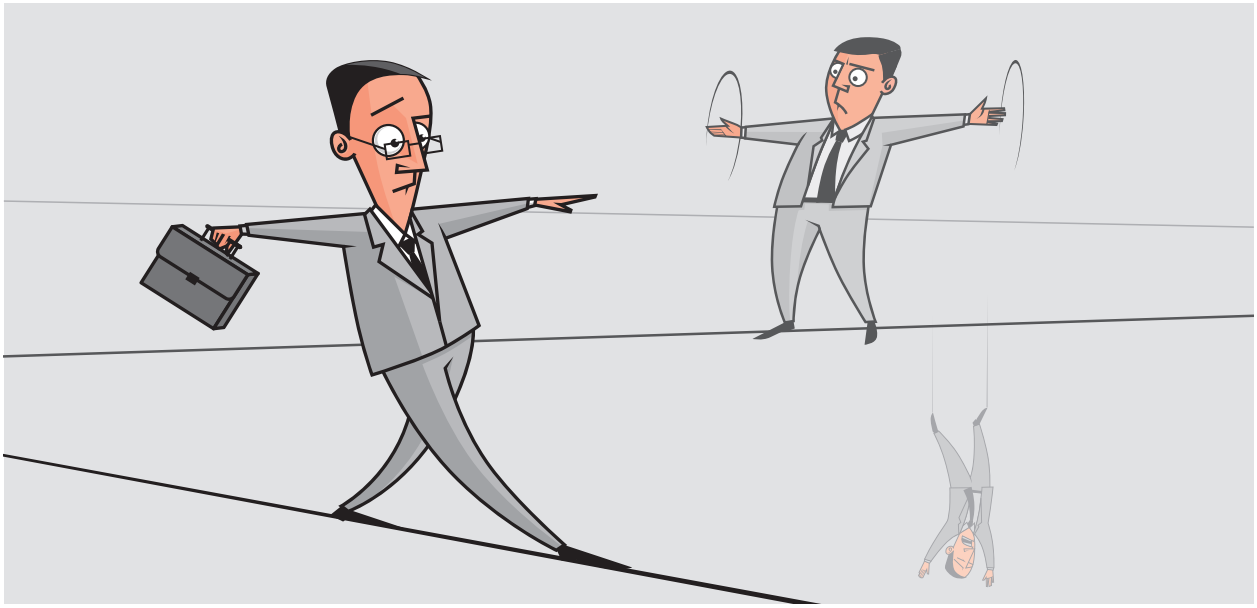
[1] RabbitMQ: http://www.rabbitmq.com

[2] www.springsource.org/spring-integration

[3] RabbitEasy: http://github.com/zanox/rabbiteasy

[4] RabbitOrdering: http://github.com/zanox/rabbitordering

[5] https://github.com/zanox/rabbitordering.git



Prozessverbesserung mit CMMI

Agile Reifeprüfung

Software bestimmt heutzutage weite Teile unseres Alltags. Die Erwartungen der Kunden zu erfüllen und qualitativ hochwertige Software zu liefern, sind in einem kundenorientierten Unternehmen die wichtigsten Ziele der Softwareentwicklung. Erreichen lassen sie sich durch leistungsstarke Technologien, motivierte und geschickte Mitarbeiter, vor allem aber durch reife Prozesse. Methoden zur Prozessverbesserung in der Softwareentwicklung wie das internationale Reifegradmodell CMMI (Capability Maturity Model Integration) sind dabei wichtige Wegbereiter. Wie sich der höchste CMMI-Reifegrad erzielen lässt, zeigt die folgende Fallstudie.

von Malgorzata Pinkowska, Cornelia Gilgen und Werner Müller

Am heutigen Markt gibt es Reifegradmodelle, Standards, Methoden und Richtlinien, die einer Organisation helfen können, ihre Geschäftsabläufe zu optimieren, um die Kundenzufriedenheit zu steigern. Das führende und weltweit verbreitetste Reifegradmodell für Qualitätsmanagement, insbesondere für die Softwareentwicklung, ist CMMI. Ein Schweizer Unternehmen, spezialisiert auf Individuallösungen mit Java und agilen Methoden, hat diesen Standard eingeführt und die Umsetzung bewerten lassen. Die Kultur der kontinuierlichen Prozessverbesserung war in der Firma bereits vorhanden. Sie sollte durch die Einführung von CMMI vertieft und messbar gemacht werden und so zum Erfolg der bereits laufenden Internationalisierung des Unternehmens beitragen.

CMMI ist ein Framework zur Prozessverbesserung. Es wird von kleinen und großen Unternehmen in einer Vielzahl unterschiedlicher Branchen verwendet, darunter IT, Elektronik, Gesundheitswesen, Finanzen, Versicherungen und Transport [1].

Das CMMI-Modell ist eine systematische Aufbereitung bewährter Praktiken, um die Verbesserung der Prozesse einer Organisation zu unterstützen und anhand von Reifegraden einzuordnen. Dabei können vorhandene Vorgehensweisen zur Prozessverbesserung eingebracht und verwendet werden.

Derzeit gibt es drei veröffentlichte CMMI-Modelle. Alle haben die gleiche Struktur und einen gemeinsamen inhaltlichen Kern. „CMMI für Entwicklung" (CMMI-DEV [1]), auf dem der hier vorgestellte Fall basiert, unterstützt die Prozessverbesserung in Organisationen, die Software, Systeme oder Hardware entwickeln. „CMMI für Akquisition" (CMMI-ACQ) zielt dagegen auf Organisationen ab, die Software, Systeme oder Hardware einkaufen, aber nicht selbst entwickeln. „CMMI für Services" (CMMI-SVC) widmet sich schließlich der Prozessverbesserung in Dienstleistungsunternehmen [1].

CMMI für Entwicklung V1.3 hat 22 Prozessgebiete, die einen konkreten Rahmen für ein Qualitätsmanagementsystem in vier Kategorien bilden: Projektmanagement, Entwicklung, Unterstützungsprozesse und Prozessmanagement. In einem so genannten CMMI Appraisal (Audit) wird bewertet, ob ein Unternehmen die Vorgaben bis zu einem bestimmten Grad erfüllt. Durch einheitliche Vorgaben und Reifegrade können Unternehmen verglichen werden. Vorgegebene Ziele und Praktiken regeln, was umgesetzt werden muss. Wie dies geschieht, bleibt aber jedem Unternehmen selbst überlassen. Für das Wie hat das Unternehmen in dieser Fallstudie im Zuge der Einführung ein Lösungspaket erarbeitet und in mimacom path integriert.

Die Umsetzung der Prozesse in einer Organisation verpflichtet jeden Mitarbeiter, diese auch anzuwenden. Zusätzlich wird von einer reifen Organisation verlangt, eine kontinuierliche Prozessverbesserungskultur zur Gewohnheit jedes Mitarbeiters zu machen. Es kann nur funktionieren, wenn sich jeder einzelne an der Definition der Prozesse, die ihn direkt beeinflussen, beteiligen kann und somit auch von deren Verbesserung profitiert.

Projektmanagement-, Prozessmanagement- und Unterstützungsprozessgebiete, die nur indirekt Einfluss auf die tägliche Arbeit der Ingenieure haben, wie z. B. *Konfigurationsmanagement* (Configuration Management, CM), müssen ebenfalls von ihnen verstanden und implementiert werden.

Die Entwicklungsprozessgebiete

CMMI definiert fünf Entwicklungsprozessgebiete, die für einen reifen Entwicklungsprozess erforderlich sind [1]:

- Die **Anforderungsentwicklung** (Requirements Development, RD) ist mit der Formulierung der Kunden- und Produktanforderungen sowie deren Analyse und Validierung beschäftigt. Die Bedürfnisse, Erwartungen, Einschränkungen, Schnittstellen, Betriebs- und Produktkonzepte der Stakeholder werden ausführlich analysiert, harmonisiert, verfeinert und ausgearbeitet. Sie werden in Anforderungen an die Software und deren Bestandteile übersetzt, in einem Betriebskonzept zusammengefasst und in der Spezifikation detailliert aufgezeigt. Schließlich werden die Anforderungen validiert, um sicherzustellen, dass das resultierende Produkt in der Anwendungsumgebung wie beabsichtigt funktionieren wird.
- Die **technische Umsetzung** (Technical Solution, TS) umfasst die Konzeption und Entwicklung von Lösungskomponenten entsprechend den Anforderungen. Lösungen für Produkte oder Produktbestandteile werden hier aus alternativen Lösungen ausgewählt. Danach soll das Design den entsprechenden Inhalt nicht nur für die Umsetzung, sondern auch für andere Phasen des Produktlebenszyklus bereitstellen. Außerdem muss die Dokumentation für die Endanwendung entwickelt und gepflegt werden.
- Die **Produktintegration** (Product Integration, PI) beschäftigt sich damit, wie die einzelnen Komponenten zusammengesetzt werden, um ein Endprodukt für die Auslieferung zu bilden. Die Vorbereitung der Integration von Produktbestandteilen umfasst die Etablierung und Pflege einer Integrationsstrategie, der Umgebung zur Durchführung der Integration sowie Integrationsverfahren und -kriterien. Weiter wird die Kompatibilität von internen als auch externen Schnittstellen der Produktbestandteile sichergestellt. Verifizierte Produktbestandteile werden zusammengebaut, und das integrierte, verifizierte und validierte Produkt wird ausgeliefert [1]. Aber was bedeutet „verifiziert und validiert"?
- Die **Verifizierung** (Verification, VER) sorgt dafür, dass jede Lösungskomponente ihre spezifizierten Anforderungen erfüllt, also richtig umgesetzt wird. Zunächst werden Arbeitsergebnisse zur Verifizierung ausgewählt, Methoden und Kriterien definiert. Danach werden ausgewählte Arbeitsergebnisse anhand der spezifizierten Anforderungen verifiziert. Zusätzlich werden Peer-Reviews durchgeführt.
- Die **Validierung** (Validation, VAL) zeigt, dass ein Produkt in der Zielumgebung funktioniert und die Bedürfnisse der Endbenutzer erfüllt, also ob das richtige Produkt hergestellt wurde. Die Vorbereitungsaktivitäten umfassen die Auswahl von Produkten und Produktkomponenten für die Validierung sowie die Etablierung und Pflege der Validierungsumgebung, -verfahren und -kriterien. Die Produkte oder Produktbestandteile werden validiert, um sicherzustellen, dass sie für die Nutzung in der vorgesehenen Betriebsumgebung geeignet sind.

Neben den genannten Entwicklungsprozessgebieten sind auch Ingenieure wichtige Akteure in den Bereichen Projektmanagement und Organisation. In einem reifen Unternehmen können sie auch Anstoß für Verbesserungen in diesen Bereichen geben.

Als Teilnehmer an einem Sprint Planning sind die Entwickler beispielsweise an der Projektplanung beteiligt, bei einem Daily-Scrum-Meeting können sie bei Risikomanagement und Projekt-Controlling-Prozessen mitwirken. Außerdem sind alle Mitarbeiter an organisatorischen Prozessen beteiligt, wie zum Beispiel an der organisationsweiten Prozessentwicklung, organisati-

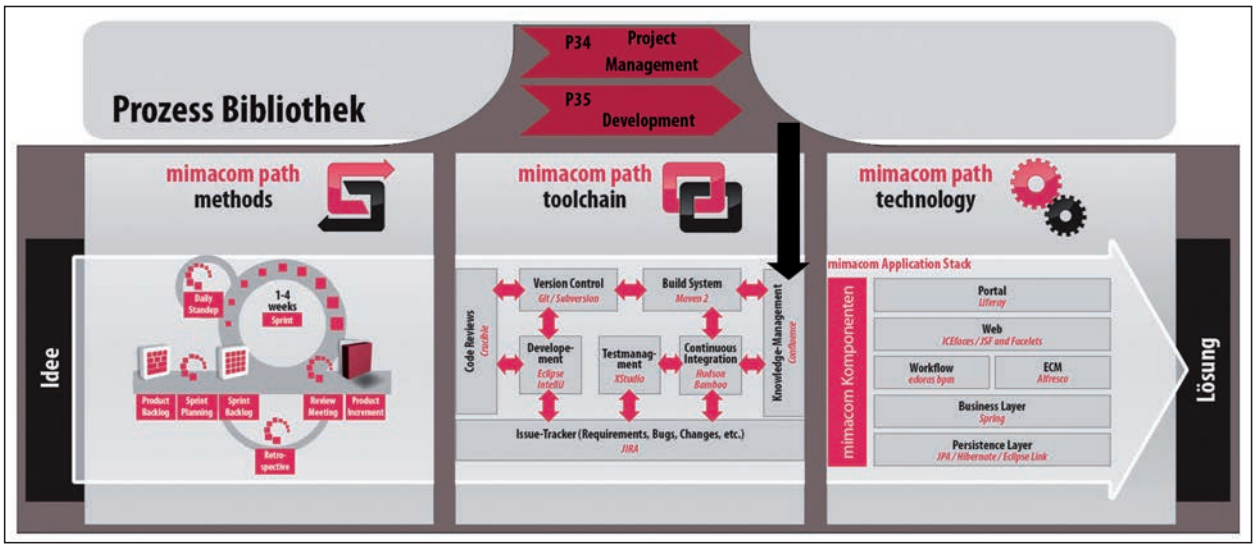


Abb. 1: mimacom path mit integriertem Qualitätsmanagementsystem

onsweiten Prozessausrichtung oder organisationsweiten Aus- und Weiterbildung.

Methoden, Werkzeuge, Technologie: mimacom path

Jeder Softwareentwickler würde seine Aufgaben gerne auf die ihm liebste und eigene Art und Weise erledigen. Damit die Teamarbeit in einem Unternehmen funktionieren kann, sind standardisierte Vorgehensweisen aber unerlässlich.

Der Schlüssel zum Erfolg im beschriebenen Fall ist die Kombination der richtigen Methoden, Werkzeuge, Technologien, einem ISO-zertifizierten Qualitätsmanagementsystem und der Kultur der kontinuierliche Prozessverbesserung (**Abb. 1**). Daran sind die Ingenieure maßgeblich beteiligt. Somit können die Ziele der CMMI-Entwicklungsprozesse sowie die weiteren Anforderungen der Reifegrade 2 und 3 erfüllt und diejenigen der Reifegrade 4 und 5 unterstützt werden.

Diese Erfolgsfaktoren werden zusammengefasst im mimacom path [2]. Ein für die Entwicklung zentraler Teil davon ist die mimacom path toolchain. Sie schafft die richtige technologische Umgebung für eine effiziente Entwicklung und Flexibilität. Mit Standardprodukten, die optimal aufeinander abgestimmt sind, werden alle Bereiche der modernen Softwareentwicklung umfassend abgedeckt. Der modulare Ansatz ist zentral, sodass sich die verschiedenen Komponenten mit bestehenden Systemen verbinden lassen und/oder austauschen lassen.

mimacom path setzt auf agile Methoden wie Scrum, ergänzt mit anerkannten Systemen zur Qualitätssicherung wie Total Quality Management (TQM). Dies ermöglicht auch in großen Projekten ein flexibles Vorgehen unter Einhaltung eines hohen Qualitätsstandards. Von Java ausgehend, enthält die mimacom path toolchain:

- Git/Subversion für die Versionskontrolle (CM)
- Apache Maven für zuverlässiges Application Building und Dependency Management (TS)
- IntelliJ/Eclipse als integrierte Entwicklungsumgebung mit den notwendigen Plug-ins (TS)
- JIRA für Issuetracking und agiles Management (RD, VAL)
- Confluence für ein strukturiertes Knowledge-Management (TS, TQMi)
- Bamboo für kontinuierliche Integration, Testing und Reporting (PI, VER)
- Crucible, FishEye und Sonar für Codereview und Codeinspektion (VER)
- XStudio für Testmanagement und Testautomation (VER)

mimacom path technology bietet einen flexiblen Rahmen für die verschiedenen modularen Komponenten. Dabei wird bei der Realisierung von Individualsoftware die Konzeptphase verkürzt, der Entwicklungsaufwand reduziert, und Risiken werden vermindert. Die einzelnen Komponenten können unabhängig voneinander und in beliebiger Kombination eingesetzt werden.

Bezogen auf die fünf Entwicklungsprozessgebiete von CMMI-Dev können diese Lösungskomponenten wie folgt eingesetzt werden:

- Die **Anforderungsentwicklung** findet hauptsächlich während der Entwicklung von Product Backlog und Sprint Backlog statt. JIRA ist dafür von zentraler Bedeutung. Die Anforderungen werden in Jira definiert und verfolgt. Zur Skizzierung der dazugehörigen GUI-Elemente können auch direkt im Tool Mockups erstellt werden. Über die Integration mit FishEye besteht die Möglichkeit, durch Links zum Code die Nachverfolgbarkeit (Traceability) von der Anforderung bis zur Applikation sicherzustellen.
- Die Basis für **technische Umsetzung** sind die Erarbeitung der Systemarchitektur, die in Sprint 0 stattfindet, und das Systemdesign. Beides wird kontinuierlich in den Entwicklungs-Sprints weiterbearbeitet. Die Grundlagen für Architektur und Design stellt mima-

com path technology zur Verfügung. Die nötige Dokumentation kann gemäß Vorlagen aus dem Qualitätsmanagementsystem einfach und effektiv erarbeitet werden. Erfahrungen, die während der Umsetzung gesammelt und in den Sprint-Retrospektiven zusammengetragen werden, können frühzeitig in die weitere Entwicklung einfließen und zur Verbesserung der Prozesse beitragen.

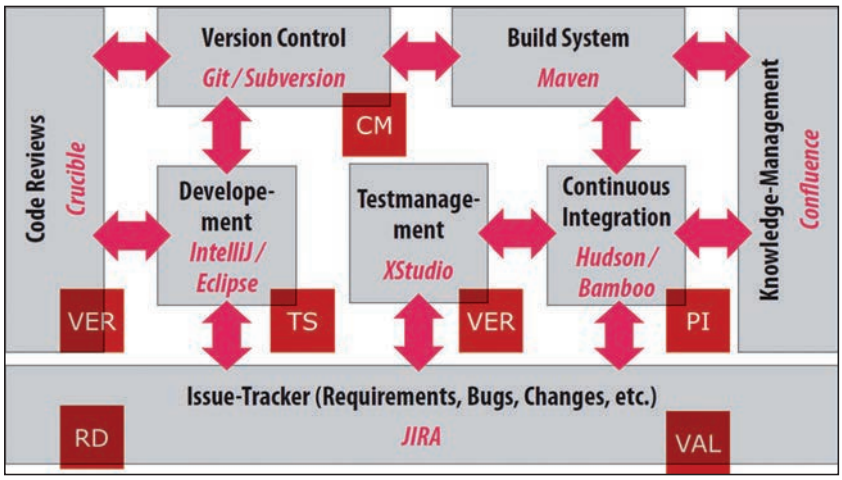


Abb. 2: mimacom path tools und CMMI

- Die Anforderungen der **Produktintegration** werden durch den Einsatz von Bamboo zur automatischen Integration größtenteils erfüllt. Die Continuous Integration geschieht bereits während der Sprints täglich und die Resultate werden somit ständig überwacht und dokumentiert. Erstellte Versionen werden mit Maven versioniert und archiviert.
- Die **Verifizierung** der spezifizierten Anforderungen wird durch Testing und Reviews sichergestellt, die im Scrum-Zyklus durchgeführt werden. Jira zum Tracking von Anforderungen, Testfällen und Bugs sowie XStudio zur Verwaltung und Dokumentation der Test-Sessions sind dabei eine große Hilfe.
- Auch die **Validierung** wird durch JIRA und das Vorgehen nach Scrum des mimacom path entscheidend unterstützt. Die durch Scrum übliche enge Zusammenarbeit mit dem Kunden und die frühen und häufigen Lieferungen von Produktinkrementen tragen entscheidend zur Erfüllung der Praktiken der Validierung bei.

CMMI-Umsetzung – Erfahrungsberichte für die Entwickler und Tester

Die Erfahrung in der Beratung bei vielen Unternehmen hat gezeigt, dass man CMMI mit sehr viel Skepsis begegnet, da es in erster Linie zu einem Mehraufwand für (unnötige?) Dokumentation führen würde. Die Umsetzung im hier dokumentierten Fall hat aber im Gegenteil gezeigt, dass dadurch Stärken und Verbesserungspotenziale erkannt und besser genutzt werden konnten.

Die Einführung von CMMI in einer Organisation wird in der Regel insbesondere von den betroffenen Ingenieuren eher negativ aufgenommen. Hauptgründe dafür sind die sehr komplexe Sprache des Frameworks, die es schwierig macht, die Anforderungen in der täglichen Praxis effizient umzusetzen, Angst vor Umorganisation und Prozessveränderungen und der Mythos zusätzlicher bürokratischer Arbeit. Ein Ingenieur ist viel einfacher für technologische Neuerungen zu begeistern als für Unbekanntes außerhalb seines Themengebiets. Die gesammelten Erfahrungen zeigen aber, dass eine erfolgreiche Umsetzung der Entwicklungspraktiken durchaus möglich ist, wenn einige einfache Punkte beachtet werden:

- Es sollten so weit wie möglich die in der Unternehmung üblichen Begriffe zur Prozessdefinition verwendet werden und nicht die komplexe CMMI-Sprache. Dazu ist der Einsatz von internen Mitarbeitern nötig, die die Unternehmenskultur kennen, nach CMMI ausgebildet werden und danach eine „Übersetzung" vornehmen können.
- Es muss im Unternehmen an einer Kultur zur kontinuierlichen Verbesserung gearbeitet werden, so es diese noch nicht gibt. Ist diese bereits etabliert, vereinfacht sich die Umsetzung von CMMI erheblich. Die Mitarbeiter müssen über die Änderungen, die ihre Arbeit beeinflussen, informiert werden. Bei der Umsetzung eines Standards ist es sehr wichtig, die bestehende Praxis als Basis einzusetzen und die Richtlinien des Standards entsprechend zu adaptieren. CMMI sagt, **was** zu erfüllen ist, eine bestehende Umsetzung der agilen Ansätze kann klären, **wie** dies erfüllt werden kann.
- Der Codereview ist beschrieben und wird durchgeführt. Wichtig sind hierbei die Zufriedenheit jedes Ingenieurs und die Durchführung der Reviews auf einer regulären Basis. Außerdem werden Key Performance Indicators (KPIs) definiert und überwacht. Für die regelmäßige Durchführung ist das Projektteam verantwortlich.
- Der Mythos zusätzlicher bürokratischer Arbeit für Ingenieure bei der Umsetzung von CMMI sollte zu Beginn zerstört werden. Im vorliegenden Fall war zusätzliche Dokumentation praktisch nicht nötig, vor allem im Entwicklungsbereich. Die einzigen Dokumente, die berücksichtigt wurden, bestanden bereits oder wurden in bestehende eingefügt: Systemarchitektur, Testkonzept, Testbericht, Betriebshandbuch, Systemabnahmeprotokoll. Alle anderen Belege, die zum Nachweis der Umsetzung der CMMI-Praktiken benötigt wurden, konnten dem Appraisal-Team in

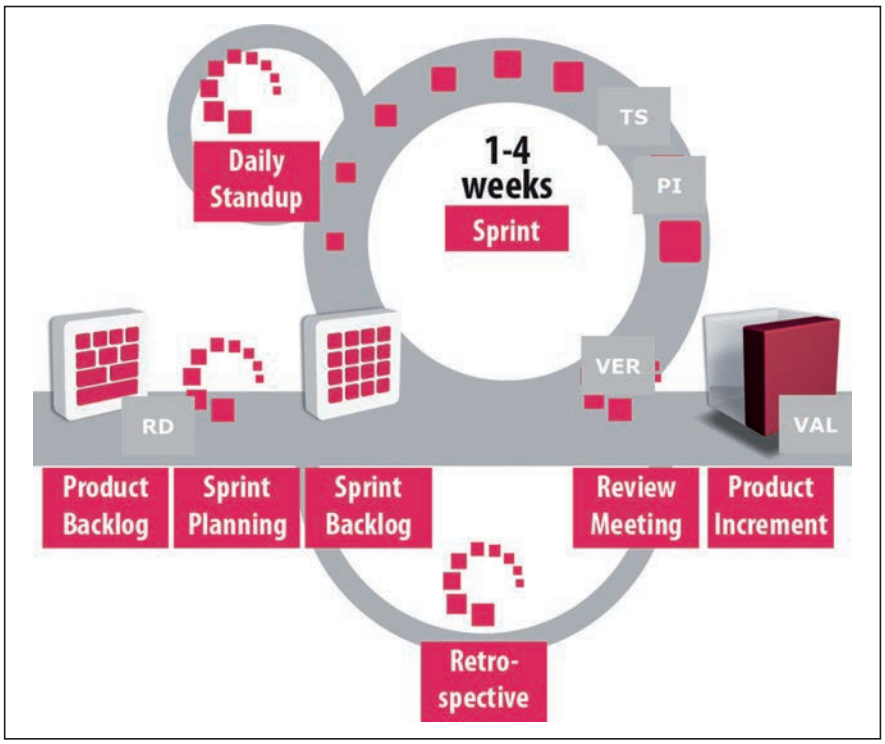


Abb. 3: mimacom path methods und CMMI

Form von Screenshots der Tools oder Quellverweisen geliefert werden, wie z. B. Anforderungsdefinitionen in JIRA, Mock-ups, Sourcecode etc.

Hat sich mit der Einführung von CMMI etwas im Projektalltag geändert?

Die Umsetzung von CMMI muss die Arbeit der Ingenieure nicht stören, wenn die nach CMMI verbesserten Prozesse sich nach dem Standardverbesserungsprozess des Unternehmens implementieren lassen.

Gute Kenntnisse und ein gutes Verständnis des CMMI-Frameworks sind nur von den Ingenieuren nötig, die dem Appraisal-Team angehören. Sie sind für die richtige und erfolgreiche CMMI-Umsetzung verantwortlich und leiten die Personen an, die während des Appraisals befragt werden.

Um erfolgreich zu sein, genügt technologisches Wissen allein für Softwareentwickler und Tester nicht mehr. Soziale Kompetenz und effizientes Arbeiten nach Prozessen sind ebenso entscheidende Faktoren. Jeder Ingenieur, der in einem CMMI-geprüften Unternehmen tätig ist, kann beweisen, dass er nach Prozessen arbeitet und die Bedeutung einer Kultur zur kontinuierlichen Verbesserung kennt.

Zudem ist CMMI ein internationales Framework und bei ca. 4000 Organisationen weltweit implementiert. Viele große Unternehmen wie z. B. Siemens AG, Alcatel-Lucent, Credit Suisse, Deloitte oder Bombardier haben Prozesse nach CMMI definiert und fordern von ihren Ingenieuren entsprechende Erfahrungen damit. Für Ingenieure, die nicht direkt an der CMMI-Umsetzung beteiligt waren, kamen oft nur kleine Aufgaben hinzu. Für die systematische Erfassung von Reviews oder Risikomanagement wurden eigene Issue Types definiert, die es fortan zu benutzen gilt. Das Erfassen von für CMMI wichtigen KPIs wird oft nur als Konfigurationsänderung in den Tools wahrgenommen. Die Arbeitsweise verändert sich also kaum. Dass im Gegensatz zu früher mehr gemessen wird, ist für den einzelnen Entwickler mitunter gar nicht ersichtlich. Wichtig sind aber das Einhalten der Arbeitsweise und der Umgang mit den Tools.

Abgesehen davon ist CMMI für den Entwickler ein weiteres Sicherheitsnetz, das aus den Erfahrungen und Praktiken gestrickt ist, die CMMI mitbringt. Speziell der Fokus auf agile Entwicklungsmethoden zog bislang einen Mangel an adäquater Projektdokumentation nach sich. Dies konnte mit CMMI korrigiert werden und führte zu einer Gesundung des Entwicklungsprozesses.

Am Ende wollen wir den Ingenieuren empfehlen, sich keinesfalls von der landläufigen Meinung entmutigen zu lassen, dass CMMI einem das Leben erschwert. CMMI verdient eine Chance.

Malgorzata Pinkowska (malgorzata.pinkowska@mimacom.com) arbeitet als Consultant bei der mimacom ag in Bern. Sie war verantwortlich für CMMI-Implementierung und die Prozessverbesserung in der mimacom ag.

Cornelia Gilgen (cornelia.gilgen@mimacom.com) ist Quality Manager und Scrum Master der mimacom ag in Bern.

Werner Müller (werner.mueller@mimacom.com) ist Softwarearchitekt bei der mimacom ag in Bern und ist für das Konfigurationsmanagement verantwortlich.

Links & Literatur

[1] CMMI für Entwicklung, Version 1.3, Technical Report, SEI, 2011

[2] http://www.mimacom.com/de/produkte/mimacom-path/, 2.2013

[3] Jenni, J.; Perriard, M.; Wandfluh, S.: „Das eine tun und das andere nicht lassen – Mit Agilität zum CMMI-Level 5“, OBJEKTspectrum, 2012

[4] CMMI or Agile: Why Not Embrace Both!, SEI, 2008

[5] Krebs, D.: „Besonnenes Experimentieren mit agilen Methoden hilft weiter“, in Netzwoche 12.2012

Memory Management mit Android

Wo ist der Speicher hin?

Gerade in der frühen Phase der Android-Ära hat sich der Benutzer oft über abstürzende Apps ärgern müssen. Denn oft wurden Apps vom System beendet, weil sie zu viel Speicher verbraucht hatten. Speziell in der Android-2.x-Welt war es für den Entwickler nicht ganz einfach, das Speichermanagement des Android-Systems zu verstehen. Zum Glück bietet das Android SDK eine Reihe von guten Tools, um Licht ins Dunkel der Speicherbereiche zu bringen.

von Dominik Helleberg

Viele Android-Entwickler haben Erfahrungen mit der Programmierung von Serverapplikationen auf Java-Basis gesammelt. Man kennt und schätzt die Vorzüge einer Virtuellen Maschine inklusive Garbage Collection und Speichermanagement. Memory Leaks, Segmentation Faults und aus dem Speicher laufende Applikationen gehören der Vergangenheit an. Life is good.

Erfahrene Java-Programmierer wissen, dass dies nicht die ganze Wahrheit ist, dennoch sind Speicherprobleme auf einer serverseitigen VM mit teilweise mehreren hundert Megabyte Heap eher selten. Im mobilen Bereich sieht das ganz anders aus. Was es hier zu berücksichtigen gilt und wie man Speicherproblemen auf die Spur kommt, erklären wir in diesem Artikel.

Typische Speicherprobleme auf Android

Auch auf Android gilt, dass man nicht zu paranoid auf mögliche Speicherbelegungen bei der Programmierung achten sollte, solange es keine Anzeichen für ein Problem gibt. Aber woran erkennt man Speicherprobleme? Der deutlichste und drastischste Fall ist mit Sicherheit der klassische *java.lang.OutOfMemory* Error der Dalvik VM, der zu einem unmittelbaren Absturz der Virtuellen Maschine führt. Dieses Verhalten ist vergleichbar mit Java-VMs auf dem Server: Das Programm hat mehr Speicher alloziert als der zulässige, maximale Heap der VM hergibt. Dies kann durch unbedachtes Erstellen von zu großen Objekten (Bilder sind hier ein klassisches Beispiel) geschehen, oder durch „leaken“ von zu vielen Objekten. Beides hat zur Folge, dass die VM keinen Speicher mehr hat und sich beendet.

Aber unter Android gibt es auch noch eine andere Art von Problemen mit dem Speicher, die z. B. zu Performanceproblemen führen können, und dies hängt mit dem Garbage Collector zusammen. Generiert die App sehr viele Objekte, führt das in der Regel zu einem Lauf des Garbage Collectors. Passiert dies beispielsweise während einer Animation, kann es zu einem kurzen Einbruch der Frame-Rate kommen, der vom Benutzer wie ein leichtes Ruckeln oder Stottern wahrgenommen wird. Dieses Problem ist grundlegend anders und sollte auch unterschiedlich behandelt und untersucht werden.

Der Umwelt zuliebe: Avoid Garbage

Die erste Indikation für ein Problem mit zu vielen Objekten sind die beschriebenen Einbrüche der Frame-Rate einer Applikation. Der erste Blick sollte dann in die Log-Ausgaben via *logcat* erfolgen. Tauchen hier zum kritischen Zeitpunkt (während der Animation) häufige Garbage-Collector-Ausgaben auf (Kasten: „GC-Logs“)? Dann kann das ein Grund für die schlechte Performance sein.

Um diesen Verdacht zu bestätigen, begibt man sich auf die Suche nach allozierten Objekten, die möglicherweise unbeabsichtigt den Speicher belegen und aufgeräumt werden müssen. Speziell Methoden, die häufig aufgerufen werden, sind hier verdächtig, z. B. die *onDraw()*-Methode. Listing 1 zeigt eine einfache Animation eines Rechtecks.

Während einer Animation wird die *onDraw*-Methode bis zu 60 Mal pro Sekunde aufgerufen. Um herauszufinden ob und wie viele Objekte hier erzeugt werden, bietet Android zwei gute Tools: den Allocation Tracker und die *android.os.Debug*-Klasse.

Der Allocation Tracker ist Teil der DDMS-Perspektive im Eclipse oder des Standalone-Tools „monitor“ (**Abb. 1**).

Es ist wichtig zu verstehen, dass der Allocation Tracker keinen Einblick in den aktuellen Zustand (Heap) der VM gewährt sondern auf das Aufspüren von neuen Objekten spezialisiert ist.

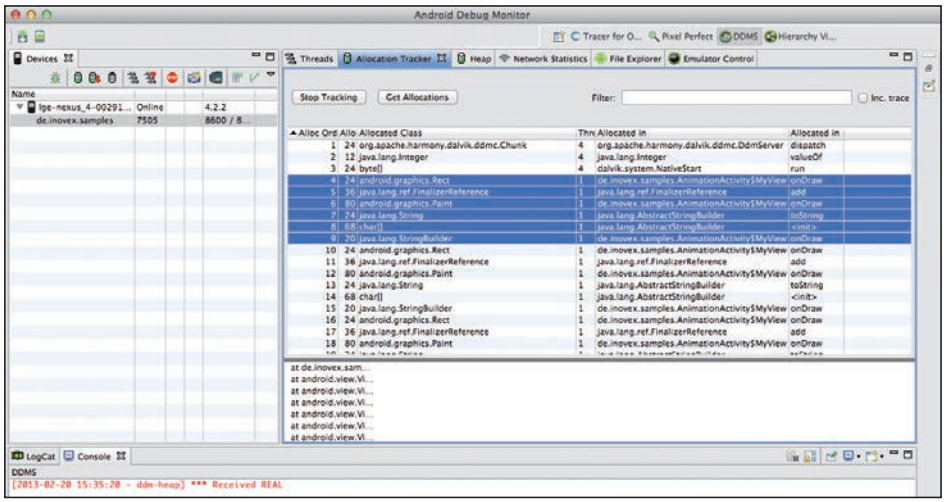


Abb. 1: Allocation Tracker

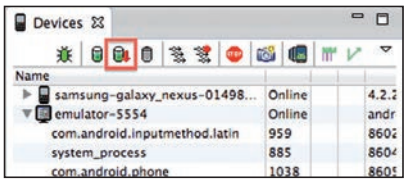


Abb. 2: Heap Dump mittels Eclipse erzeugen

Um einen Prozess zu inspizieren, wählen wir ihn links in der Liste aus und drücken zu einem geeigneten Zeitpunkt den Button „Start Tracking“. Ab jetzt zeichnet der Allocation Tracker alle neuen Objekte auf, bis zu dem Zeitpunkt an dem wir den Button „Get Allocations“ drücken.

Wendet man den Allocation Tracker auf das Codebeispiel in Listing 1 an, erhält man ein Ergebnis ähnlich dem in **Abbildung 1**. Hier werden alle Objekte gezeigt, die neu erzeugt wurden, auch wenn diese vielleicht schon wieder vom Garbage Collector eliminiert wurden. Ein Durchlauf der *onDraw*-Methode ist in **Abbildung 1** markiert. Erwartungsgemäß erzeugen wir drei Objekte direkt in unserem Code: Ein *Rect*-, ein *Paint*- und ein *StringBuilder*-Objekt (bedingt durch das Konkatenieren der Strings bei der Log-Ausgabe). Diese Objekte erzeugen allerdings selbst auch wieder neue Objekte, sodass wir auf insgesamt sechs Objekte pro Aufruf kommen. Bei einer Animation mit 60 Frames pro Sekunde sind das immerhin 360 Objekte pro Sekunde. Diese rufen dann schnell den Garbage Collector auf den Plan, der wiederum für eine kurze Unterbrechung der Animation sorgt.

Dieses Beispiel ist einfach zu fixen, wir verlagern einfach das Erstellen des *Rect*- und *Paint*-Objekts in die Initialisierung der View und schalten die Log-Ausgabe nur ein wenn wir einen Debug-Build durchführen oder verzichten ganz darauf.

Es ist nicht immer ganz einfach zum richtigen Zeitpunkt die Buttons im GUI zu bedienen, deshalb kann man die neu erzeugten Objekte auch programmatisch zählen. Listing 2 zeigt den Einsatz der Klasse *android.os.Debug* zu diesem Zweck.

Die Logcat-Ausgabe ist hier – wenig überraschend – „Allocs: 6“. Um diese recht naheliegenden Fehler zu vermeiden, gibt es mittlerweile einen Lint-Check, der davor warnt neue Objekte in Methoden wie *onDraw()* anzulegen.

Aber um es noch einmal deutlich zu sagen: Diese Objekte werden in der Regel einfach vom Garbage Collector aufgeräumt und beeinflussen lediglich die Performance, nicht aber die Stabilität der App.

GC-Logs

Jeder Android-Entwickler kennt die typischen Einträge des Garbage Collectors beim Lesen des Log-Puffers. Hier ein typisches Beispiel:

```
D/dalvikvm( 1064): GC_CONCURRENT freed 965K, 45% free 4314K/7815K, external
                                    2948K/4148K, paused 1ms+9ms
```

Während der Analyse von Speicherproblemen ist es hilfreich, die Informationen des GC auszuwerten, daher erklären wir das Format hier kurz:

Der Teil vor dem Doppelpunkt (*D/dalvikvm(1064)*) entspricht dem Standard Logcat-Output, d. h. 1064 ist die Prozess-ID der Dalvik-VM. Danach schreibt der GC folgendes Format:

```
[reason_gc] [freed_memory], [heap_stats], [external_memory],
                                                    [pause_time]
```

[reason_gc] (in diesem Fall GC_CONCURRENT) gibt den Grund des GC-Laufs an. Ab Android 4.0 findet man die folgenden Gründe im Source der Dalvik-VM [1]:

- *GC_FOR_ALLOC* – Nicht genug Speicher vorhanden, um ein neues Objekt anzulegen
- *GC_CONCURRENT* – GC aufgrund von Speicherauslastung der VM
- *GC_EXPLICIT* – Explizite GC z. B. über *Runtime.gc()*
- *GC_BEFORE_OOM* – Letzter Versuch, bevor ein Out-of-Memory-Fehler auftritt

In Gingerbread (Android 2.3) existieren zusätzlich diese Gründe:

- *GC_EXTERNAL_ALLOC* – GC aufgrund von Speicherallokation außerhalb der Dalvik-VM
- *GC_HPROF_DUMP_HEAP* – Spezialfall wenn ein Heap Dump angefordert wurde
- *[freed_memory]* – gibt den freigegebenen Speicher dieses GC-Laufs an
- *[heap_stats]* – Heap-Status nach dem GC: in unserem Beispiel sind 45 % des Speichers frei, 4314k sind belegt, die Heap-Größe ist 7815k
- *[external_memory]* – optional, nur wenn der Prozess Speicher außerhalb der Dalvik-VM belegt (nur in Android 2.3 und früher) aktueller Speicher/Softlimit
- *[pause_time]* – Die Zeit für die GC. Im Fall von GC_CONCURRENT und GC_EXPLIZIT sind es meistens zwei Zeiten, die man addieren muss, in den anderen Fällen ist es nur ein Wert.

„He's dead, Jim" – oder Out-of-Memory-Fehler vermeiden

Wesentlich kritischer als die zuvor beschriebenen temporären Speicherbelegungen ist der Fall wenn zu viele oder zu große Objekte den Speicher dauerhaft belegen. Dies führt in den meisten Fällen zu einem Absturz der Applikation. Aufgrund der restriktiveren Heap-Größen auf mobilen Geräten (teilweise nur 16 MB) passiert dieser Fall wesentlich häufiger als man es von der Serverprogrammierung gewohnt ist.

Grundsätzlich kann man zwischen zwei unterschiedlichen Ursachen für ein Speicherproblem unterscheiden: Im einfacheren Fall alloziert die Applikation zu viele Ressourcen (z. B. Bilder) gleichzeitig und belegt damit zu viel Speicher. Eine andere Ursache kann ein Memory Leak sein, bei dem die Applikation nicht benötigte Objekte nicht mehr frei gibt und immer wieder neu alloziert bis der Speicher voll ist. Dieses Problem ist meist wesentlich schwieriger zu finden und zu beheben.

Heap-Analyse

In beiden Fällen ist es meist hilfreich, eine Heap-Analyse durchzuführen. Dies kann auf verschiedene Weisen erfolgen. Ein einfacher Start, um sich den Zustand seiner Applikations-VM anzuschauen, ist das Tool „Dumpsys", das dazu dient verschiedene Informationen von Systemdiensten abzufragen. In unserem Fall fragen wir den Dienst „meminfo" kombiniert mit dem Package-Namen unserer Applikation. Dumpsys ist Teil des Android-Betriebssystems und wird auf dem Gerät selbst ausgeführt z. B. über a*db shell dumpsys meminfo de.inovex.samples*.

Das Ausgabeformat ist abhängig von der Android-Version des verwendeten Geräts und hat sich stetig weiter entwickelt. Listing 3 zeigt die Ausgabe auf Basis von Android 4.2.2.

Man erhält eine Übersicht über viele ressourcenrelevante Informationen der eigenen App. Den Zustand des Speichers entnimmt man der „Total Heap Size" und „Total Heap Alloc". Wachsen diese Zahlen während der Nutzung der App kontinuierlich an, kann dies ein Hinweis auf ein Memory Leak sein.

Ebenfalls interessant sind die Objekttypen (Views und Activities). Unter „Asset Allocations" erhält man einen

Listing 1: onDraw

```
protected void onDraw(Canvas canvas) {
  Log.v(TAG, "onDraw dx: "+dx+ " dy: "+dy);
  Paint p = new Paint();
  p.setColor(0xFFFF0000);
  Rect r = new Rect(0,0,60,60);

   canvas.translate(dx, dy);
  canvas.drawRect(r, p);
  //...
}
```

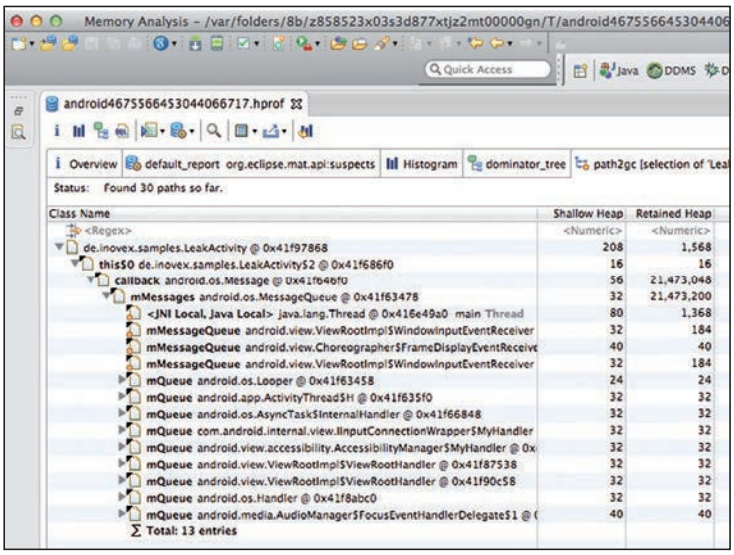


Abb. 3: Leak-Analyse via MAT

Überblick über Ressourcen, die von der App verwendet werden. Auch das sind wichtige Informationen, so führt ein Bug in Android 2.3 dazu, dass eigene Schriftarten mehrfach geladen werden, was irgendwann mit einem Speicherüberlauf der App endet. Dieses Fehlverhalten lässt sich hier transparent nachvollziehen, da die App dieselbe Font-Datei mehrfach alloziert.

Erhärtet sich der Verdacht eines Speicherproblems, kann es sinnvoll sein den Heap der Applikation zu einem bestimmten Zeitpunkt „einzufrieren" und im Detail zu analysieren. Hierzu bietet die Dalvik VM die Möglichkeit eines Heap Dumps. Diesen erzeugt man entweder über DDMS oder dem Monitor-Tool mittels dem „Dump HPROF File"-Knopf (**Abb. 2**) oder programmatisch aus dem Sourcecode der App über *android.os.Debug.dumpHprofData()*.

Listing 2: Allocation Counting via android.os.Debug

```
protected void onDraw(Canvas canvas) {

   Debug.startAllocCounting(); //start counting

   Log.v(TAG, "onDraw dx: "+dx+ " dy: "+dy);
   Paint p = new Paint();
   p.setColor(0xFFFF0000);
   Rect r = new Rect(0,0,60,60);

   canvas.translate(dx, dy);
   canvas.drawRect(r, p);

   Debug.stopAllocCounting(); //stop
   Log.v(TAG, "Allocs: "+Debug.getThreadAllocCount());
   //...
  }
```

Listing 3: dumpsys meminfo

```
local:~ $ adb shell dumpsys meminfo de.inovex.samples
Applications Memory Usage (kB):
Uptime: 232614 Realtime: 232609

** MEMINFO in pid 4195 [de.inovex.samples] **
                   Shared  Private  Heap   Heap   Heap
                     Pss    Dirty    Dirty  Size   Alloc  Free
                   ------  ------   ------ ------ ------ ------
       Native        20      12       20   2084   1727    272
       Dalvik      1660   11772     1320   8908   8878     30
       Cursor         0       0        0
       Ashmem         0       0        0
    Other dev      9083    1160     7324
    .so mmap        596    2052      364
    .jar mmap         0       0        0
    .apk mmap        69       0        0
    .ttf mmap         0       0        0
    .dex mmap       148       0        0
   Other mmap       190      16       32
      Unknown      1012     652     1000
        TOTAL     12778   15664    10060  10992  10605    302

 Objects
               Views:        23         ViewRootImpl:        1
         AppContexts:         4           Activities:        1
              Assets:         2        AssetManagers:        2
      Local Binders:         8         Proxy Binders:       16
   Death Recipients:         0
    OpenSSL Sockets:         0

 SQL
         MEMORY_USED:         0
 PAGECACHE_OVERFLOW:         0          MALLOC_SIZE:        0

 Asset Allocations
    zip:/data/app/de.inovex.samples-1.apk:/resources.arsc: 67K
```

Listing 4: Leak Activity

```
public class LeakActivity extends Activity {

  private final Handler mLeakyHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
    }
  };

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_image_viewer);

      mLeakyHandler.postDelayed(new Runnable() {
        public void run() { }
      }, 60000);
    //...
```

Zum Auswerten des Heap Dumps bietet sich das Eclipse-basierte Tool MAT [2] oder das im JDK enthaltene Tool jhat [3] an. Zuvor muss der erzeugte Heap Dump allerdings noch konvertiert werden. Dies erledigt entweder Eclipse automatisch (im Fall von MAT) oder das Kommandozeilen-Tool „hprof-conv“ [4].

Die Auswertung eines Dalvik Heap Dumps folgt im Prinzip denselben Regeln wie die einer normalen Java-VM. Große Objekte werden ohnehin recht schnell und einfach gefunden, der Dominator-Tree hilft dabei heraus zu finden, welche Objekte durch bestehende Referenzen die (möglicherweise indirekte) Ursache für ein Speicherproblem sind. Ein Beispiel findet sich in **Abbildung 3**.

In diesem Beispiel haben wir bereits ein verdächtiges Objekt ausfindig gemacht (praktischerweise mit dem Namen „LeakActivity“) und im „Dominator View“ die Option Path to GC Roots | Exclude weak references ausgewählt. Das Ergebnis in **Abbildung 3** zeigt, dass es eine Referenz „this“ auf dieses Objekt gibt, welches verhindert, dass die Activity vom Garbage Collector aufgeräumt werden kann. Ein Blick in den Source der Klasse (Listing 4) zeigt die Ursache: Die Activity verwendet einen Handler als non-static Inner Class. Dieser hält damit eine Referenz auf die Klasse selbst. Erzeugt die Activity jetzt einen Eintrag in der Message Queue des Handlers (siehe *onCreate()*), verhindert die Referenz des Handler sowie die ebenfalls non-static Inner Class des Runnables, dass die Activity aufgeräumt werden kann. Solange die Message „lebt“, wird die Activity, sowie alle damit verbundenen Ressourcen wie Bilder etc., im Speicher gehalten.

Eine Lösung ist entweder, die betreffenden Klassen als *static* zu deklarieren oder in der *onDestroy*-Methode die Messages zu entfernen.

Dieses Beispiel zeigt wie einfach es sein kann eine Activity zu leaken. Da die Activity in der Regel ein recht teures Objekt ist, kann das schnell zu einem ernsthaften Problem werden. Immer wenn im Code Inner Classes verwendet werden, sollte man kurz überlegen ob die darin erzeugten oder verknüpften Objekte ein Problem für den Garbage Collector darstellen können. Nicht alle Memory Leaks lassen sich über die hier beschriebene Methode identifizieren, im Netz finden sich allerdings noch viele Beschreibungen und Berichte rund um das Thema[5], [6], [7].

Native vs. VM-Memory

Auf eine Besonderheit muss im Android-Kontext noch kurz eingegangen werden und die betrifft den Unterschied zwischen nativem und Dalvik-Heap-Speicher.

Erratum zum letzten Artikelteil

Im letzten Teil dieser Serie hat sich in der Abbildung 2 beim Setzen leider ein kleiner Fehler eingeschlichen. Auf www.javamagazin.de finden Sie unter „Erratum“ ein PDF, das als Download verfügbar ist und die Abbildung 2 in korrigierter Version zeigt.

Speziell in der Welt vor Android 3.0 spielt der native Speicher eine große Rolle, denn hier wurden Grafiken noch (für die Java-Welt intransparent) im nativen Speicher alloziert. Zusätzlich wurde die Summe des nativen Speichers auf den Heap der VM angerechnet. Dies führt dazu, dass der programmatisch abgefragte freie Speicher der VM unter Umständen einen zu hohen Wert zurückliefert, da zusätzlich noch Bilder im nativen Speicher auf das Applikation-Budget angerechnet werden. Ein weiterer Nebeneffekt ist, dass die Bitmap-Objekte in der VM nur sehr wenig Speicher belegen, obwohl sie für erheblich mehr (nativen) Speicher verantwortlich sind. Das erschwert die Analyse des Heaps erheblich, da Tools wie MAT nicht mit den korrekten Werten pro Objekt arbeiten können. Dieses Problem wurde zum Glück ab Android 3.0 behoben, jetzt zählen Bilder voll und ganz zum Heap der VM und die Objekte repräsentieren auch den real allozierten Speicher.

Fazit

Die Entwicklung von Apps für Android erfordert eine höhere Aufmerksamkeit des Entwicklers in Bezug auf das Speichermanagement. Wird dies vernachlässigt, können Performanceprobleme oder Abstürze die Folge sein. Daher ist es wichtig schon während der Entwicklung ein Auge auf verdächtige Stellen im Code (Inner Classes, häufig aufgerufene Methoden wie *onDraw()*) zu haben. Die Android Development Tools stellen dem Entwickler allerdings eine Reihe von guten Werkzeugen bereit, um Probleme im Code zu finden und zu beheben.

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] https://android.googlesource.com/platform/dalvik/+/master/vm/alloc/Heap.h

[2] http://www.eclipse.org/mat/

[3] http://docs.oracle.com/javase/6/docs/technotes/tools/share/jhat.html

[4] http://developer.android.com/tools/help/hprof-conv.html

[5] http://android-developers.blogspot.de/2009/01/avoiding-memory-leaks.html

[6] http://www.youtube.com/watch?v=_CruQY55HOk

[7] http://android-developers.blogspot.de/2011/03/memory-analysis-for-android.html

Tagträume und Lock Screen Widgets

Android 4.2: Traumfänger

Das Android 4.2 „Jelly Bean“ Release ist inzwischen einige Monate alt, die neuen Möglichkeiten werden von den Nutzern gut angenommen (Multiusersupport auf Tablets, Lock Screen Widgets etc.) und die komplette neue Generation der Nexus-Geräte (4, 7 und 10) ist inzwischen auch regulär verfügbar. Daher ist es an der Zeit, sich mit den Neuerungen aus Entwicklersicht zu beschäftigen, um die eigenen Apps mit den 4.2-Features sinnvoll zu ergänzen. Dazu werden wir in diesem Artikel zunächst die Entwicklung von Daydreams und Lock Screen Widgets unter die Lupe nehmen.

von Christian Meder

Im Gegensatz zur iOS-Welt trifft uns Android-Entwickler in aller Regel kein unmittelbarer Zwang, sich direkt nach der Freigabe die API-Änderungen eines neuen Release genauer anzuschauen. Der einfache Grund liegt darin, dass die Verbreitung einer neuen Version noch Monate nach Freigabe im einstelligen Prozentbereich liegt. Das hat sich auch mit Android 4.2 nicht geändert: Die Nexus-7- und Galaxy-Nexus-Geräte erhielten bereits im letzten Jahr das Update, Nexus 4 und 10 waren aber bis zum Frühjahr 2013 nur in geringen Stückzahlen verfügbar, damit faktisch eigentlich nicht bestellbar, von Herstellerupdates für die stark verbreiteten Android-Smarphones und -Tablets anderer Hersteller ganz zu schweigen. Zusätzlich gab es noch zahlreiche kleinere Fehler im 4.2 Release, die erst mit 4.2.2 im Februar 2013 größtenteils beseitigt wurden [1]. Aktuell ist nun ein guter Zeitpunkt, das Experimentieren mit den API-Erweiterungen zu beginnen: Die Verbreitung von 4.2 liegt im Februar bei 1,4 Prozent, d. h. es sind auf jeden Fall schon ausreichend 4.2-Endnutzer verfügbar, um Feedback zu sammeln, und die ersten Erfahrungen der API-Pioniere im Entwicklerumfeld liegen vor.

Die Renaissance des Bildschirmschoners

Etwas überraschend wurde mit Jelly Bean 4.2 die Möglichkeit so genannter *Daydreams* eingeführt, auf den ersten Blick eine Neuinterpretation des klassischen Bildschirmschoners. Wenn aktiviert, starten Daydreams typischerweise automatisch beim Laden oder beim Einstellen in die Dockingstation. Konfiguriert werden sie in den Bildschirmeinstellungen.

Nicht dass moderne Android-Geräte wirklich ihren Bildschirm schonen müssten. Was fängt man aber dann mit einem Daydream an? Auf den zweiten Blick zeigen sich die eigentlich interessanten Möglichkeiten:

- Voller Zugriff auf das Android-UI-Toolkit (Animationen, 3-D etc.)
- Dadurch Wiederverwendbarkeit bestehender App-Komponenten möglich
- Interaktivität durch Einbindung von Touchscreen Input

Damit ist es also sehr einfach, neben einem traditionellen Bildschirmschoner etwa auch einen Demomodus für die eigene App umzusetzen oder eine aggregierte Informationsübersicht, die in allen Fällen auch sehr interaktiv werden darf. Von künstlerisch verspielt (analog etwa zum Jelly Bean Home Screen Easter Egg oder Live Wallpaper) über die klassische Spieledemo bis hin zu seriösem Informationsticker im Magazinlayout à la Google Currents (**Abb. 1**) ist alles denkbar, eine Prise Widgets dazwischengestreut.

Viele Anwendungsfälle, die traditionell unter dem Namen Kioskmodus zusammengefasst werden, sind mit Daydreams abbildbar, da diese hinter dem *secure keyguard* laufen und damit in privilegierten und unprivilegierten Zugriff unterteilt werden können. Der Google

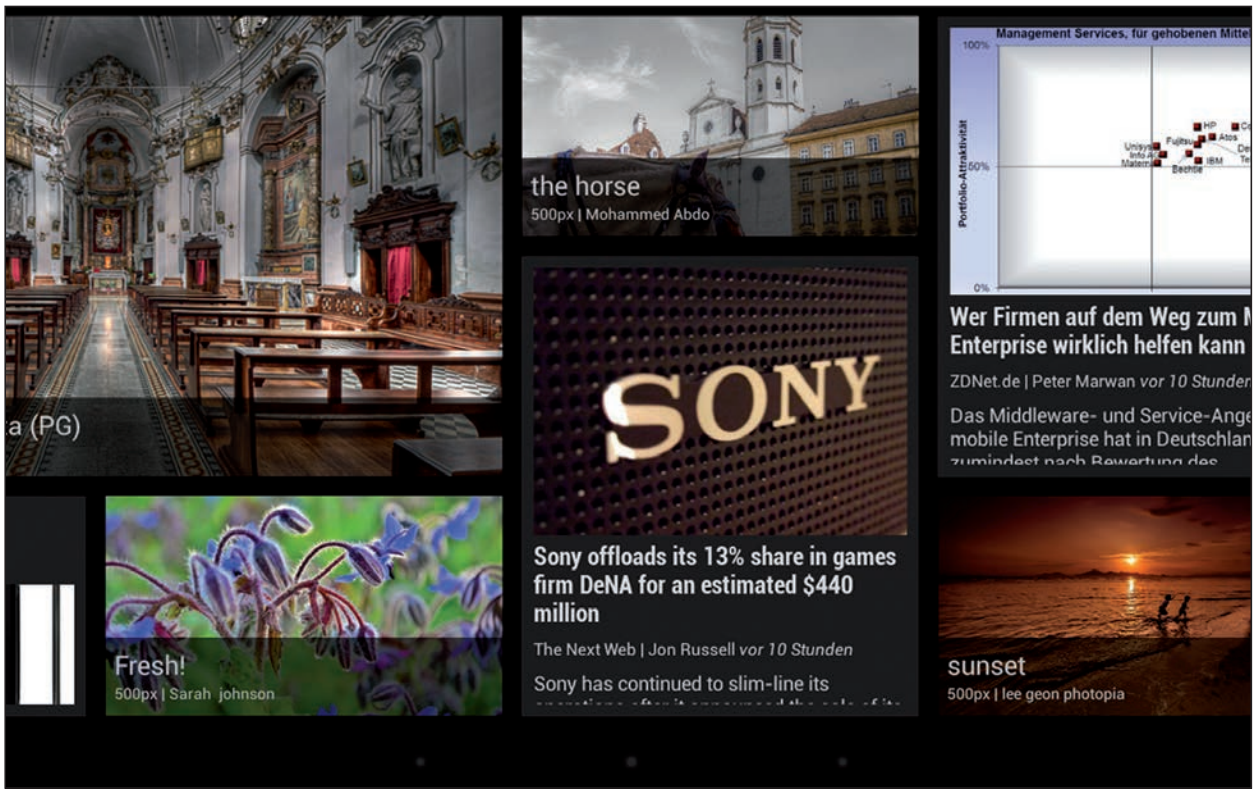


Abb. 1: Daydream der Google-Currents-App

Current Daydream nutzt dies beispielsweise, um dem unprivilegierten Nutzer die Möglichkeit zu bieten, einzelne Nachrichten im Vollbildmodus anzuzeigen, erst die komplette Detailansicht in der App erfordert ein Entsperren des Geräts.

Traumfänger

Daydreams leiten sich von *android.service.dreams.DreamService* ab, sind daher eigentlich in der *Service*-Hierarchie angesiedelt, bieten aber einen vereinfachten, *Activity*-ähnlichen Lebenszyklus an. In *onAttachedToWindow()* wird der Daydream initialisiert, dabei kann auch die *ContentView* gesetzt werden. Das Gegenstück bildet *onDetachedFromWindow()*, hier werden die verwendeten Ressourcen wieder freigegeben. Verbleiben noch *onDreamingStarted()* und *onDreamingStopped()*, in denen die dynamischen Anteile des Daydreams gestartet und angehalten werden, typischerweise Animationen und Timer (ausführlich beschrieben in [2]). In der *AndroidManifest.xml* wird der *DreamService* einfach als *Service* mit einem entsprechenden *Intent* konfiguriert.

Modifizieren wir doch einfach zur Übung eines der Beispiele aus [3], die tanzenden Androiden, und ersetzen die Androiden-Bilder durch die neuesten Bilder aus der Medienbibliothek des Geräts. Die *AndroidManifest.xml* bleibt dann wie in Listing 1 dargestellt.

Die *Bouncer*-Animation übernehmen wir ebenfalls, diese sorgt selbstständig für das Starten und Anhalten der Animation, in diesem Fall in *onAttachedToWindow()* bzw. *onDetachedFromWindow()*.

Die Methode *onDreamingStarted()* in unserem Daydream muss allerdings um das Laden der Bilder aus der Medienbibliothek ergänzt werden.

Wie in Listing 2 zu sehen, holen wir uns daher einen *Cursor* über die Bilderbibliothek, nach neuesten Einträgen sortiert. Diese Bilder werden dann mithilfe der *Bouncer*-Animation angezeigt. Das Resultat sollte schließlich ähnlich wie **Abbildung 2** aussehen.

Listing 1: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.daydream.bouncer"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-sdk android:targetSdkVersion="17" android:minSdkVersion="17"/>

    <application android:label="Bouncing Images">
        <service
        android:name=".BouncerDaydream"
        android:exported="true"
        android:label="Bouncing Images">
        <intent-filter>
            <category android:name="android.intent.category.DEFAULT" />
            <action android:name="android.service.dreams.DreamService" />
        </intent-filter>
      </service>
    </application>
</manifest>
```

Über verschiedene Schalter kann der Daydream konfiguriert werden. *setInteractive(true)* erlaubt die Verarbeitung von Eingabegesten, im anderen Fall beendet jedes Touchevent den Daydream. *setFullscreen(true)* lässt auch die Statusleiste verschwinden und *setScreenBright(true)* schaltet die standardmäßige Abdunkelung des Bildschirms während des Daydreams ab.

Dies führt zu einem weiteren, nicht zu vernachlässigenden Aspekt der Verwendung von Daydreams: Das Gerät sollte auch neben der Darstellung immer noch in der Lage sein, aufgeladen zu werden. Daydreams, die zu viel Energie verbrauchen und somit das Laden verhindern, werden vom System beendet.

Weiterhin ist es möglich, über einen *meta-data*-Eintrag im *AndroidManifest.xml* auf eine zusätzliche XML-Datei mit einer *SettingsActivity* zu verweisen. Diese wird dann in den Einstellungen der Daydreams mit dem Konfigurationssymbol neben der Auswahl angezeigt und erlaubt spezifische Einstellungen (in unserem Beispiel könnte das etwa die Geschwindigkeit der Animation oder die Anzahl der Bilder sein). Gerade bei den Konfigurationsmöglichkeiten gilt aber in der Regel: weniger ist mehr. Nicht unerwähnt soll bleiben, dass auch mehrere Daydreams pro App möglich sind. Standardmäßig bringt beispielsweise die Bildergallerie zwei unterschiedliche *Daydreams* mit, einen interaktiven Fototisch und einen digitalen Bilderrahmen [4].

Ein dunkles Kapitel ist aktuell noch das im Android-Umfeld leidige Zubehörthema. Das *Daydream*-Konzept ist ja vor allem für den Betrieb von Tablets in Dockingstationen gedacht, die allerdings noch selten für die 4.2-Geräte anzutreffen sind. Für das Nexus 7 gab es sehr kurzzeitig vor zwei Monaten eine Dockingstation, seitdem ist sie allerdings ausverkauft [5]. Ein Weihnachtsvideo von Google brachte eine Vorschau auf eine Dockingstation für das Nexus 10, der Zeitpunkt der Verfügbarkeit ist aber noch völlig unklar [6]. Es ist aber damit zu rechnen, dass sich dieses Problem im Laufe des Jahres mit der größeren Verfügbarkeit von 4.2-Geräten erledigen wird.

Listing 2: onDreamingStarted() des Daydream

```
public void onDreamingStarted() {
    final FrameLayout.LayoutParams lp = new FrameLayout.LayoutParams(200, 300);

    mBouncer = new Bouncer(this);
    mBouncer.setLayoutParams(new ViewGroup.LayoutParams(MATCH_PARENT,
                                                        MATCH_PARENT));
    mBouncer.setSpeed(200);

    String[] projection = {MediaStore.Images.Media.DATA};
    Cursor cursor = getContentResolver().query( MediaStore.Images.Media.EXTERNAL_
                                                        CONTENT_URI,
            projection, // Which columns to return
            null,       // Return all rows
            null,
            MediaStore.MediaColumns.DATE_MODIFIED + " DESC");
    int columnIndex = cursor.getColumnIndexOrThrow(MediaStore.Images.Media.DATA);
    int counter = 0;

    if (cursor != null) {
            while (cursor.moveToNext() && counter++ < 5) {
            final Bitmap bm = BitmapFactory.decodeFile(cursor.getString(columnIndex));
            final ImageView image = new ImageView(this);
            image.setImageBitmap(bm);
            image.setBackgroundColor(0xFF004000);
            mBouncer.addView(image, lp);
            }
    }
    setContentView(mBouncer);
}
```

Listing 3: Widget, das auch auf dem Lock Screen angezeigt werden kann

```
<appwidget-provider
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="110dip"
  android:minHeight="110dip"
  android:updatePeriodMillis="3600000"
  android:previewImage="@drawable/preview"
  android:initialLayout="@layout/widget_layout"
  android:initialKeyguardLayout="@layout/widget_layout"
  android:widgetCategory="keyguard|home_screen"
  android:autoAdvanceViewId="@id/stack_view"
  android:resizeMode="horizontal|vertical">
</appwidget-provider>
```

Mehr Informationen, schneller, zum Zweiten

Mit Android 4.2 wurde die Möglichkeit ergänzt, jetzt auch auf dem Lock Screen Widgets anzuzeigen, womit es prinzipiell möglich ist, viele Widgets zu vielen Lock-Screen-Seiten hinzuzufügen. Diese zweite Ebene von weiteren Widget Screens vor dem eigentlichen Home Screen fand nicht überall uneingeschränkte Zustimmung [7]. Einen besonderen Schub bekam das Konzept trotzdem durch das designtechnisch gut eingepasste Widget von Google-Mitarbeiter Roman Nurik: *DashClock* [8] (**Abb. 3**).

Dieses Lock Screen Widget zeichnet sich durch einfache Konfigurierbarkeit, ein Plug-in-API und seinen offenen Sourcecode aus [9]. Durch das API ist das Widget leicht zu erweitern [10] und es existieren bereits eine ganz erkleckliche Anzahl von Plug-ins [11]. Dabei ist es aus Entwicklersicht technisch sehr einfach, ein Widget zu einem Lock Screen Widget zu machen.

In der XML-Datei zur Widget-Beschreibung wird ein *widgetCategory* mit *keyguard* und/oder *home_screen* ergänzt und ggfs. noch ein *initialKeyguardLayout*. Damit ist ein Old School Widget schon zu einem modernen Hipster Lock Screen Widget geworden. Inhaltlich sollte sich der Entwickler noch Gedanken machen, ob er wirklich sicherheitstechnisch alle Informationen aus dem Widget bereits auf dem Lock Screen anzeigen will,

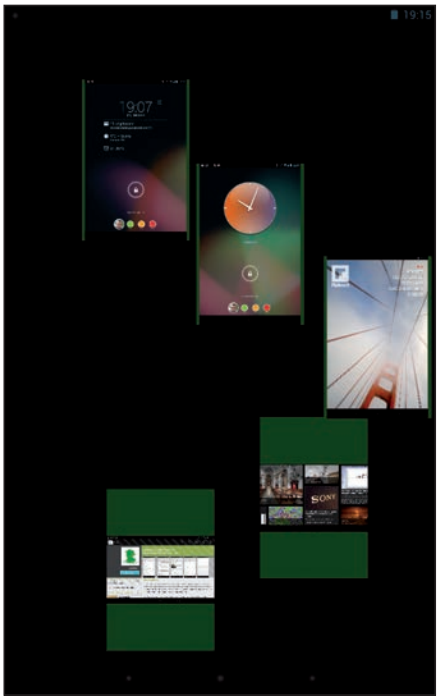


Abb. 2: Bouncing-Images-Beispiel

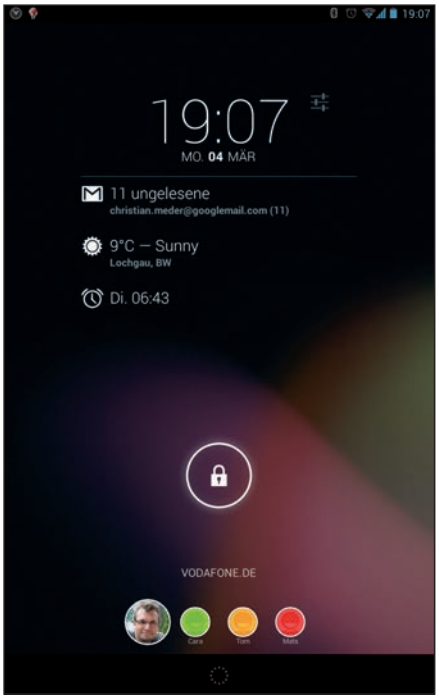


Abb. 3: DashClock

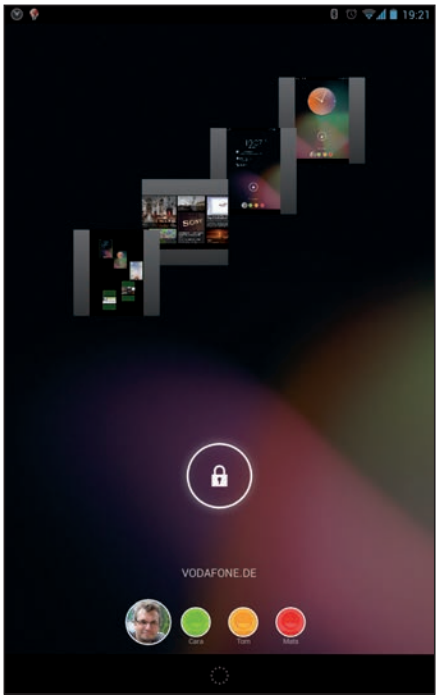


Abb. 4: StackWidget als Lock Screen Widget

aber aus reiner Implementierungssicht sind keine weiteren Änderungen notwendig.

Wenn wir also das *StackWidget*-Beispiel aus dem SDK-Fundus (*android-sdk/docs/resources/samples/StackWidget*) analog zu unserem Daydream-Beispiel als Lock Screen Widget abwandeln, sieht das Ergebnis aus wie in **Abbildung 4**.

Eine ganze Reihe von Apps unterstützt bereits Lock Screen Widgets [12], von Wetter, Nachrichten und soziale Netzwerke über ToDos, Kalender, Stundenplan und Fernsehprogramm bis zu Fernbedienungsfunktionalitäten. Ein Beispiel für die letzte Kategorie ist die Yatse-App, eine ausgezeichnete Fernbedienung-App für das relativ weit verbreitete Open-Source-XBMC-Mediencenter [13]. Mithilfe des Lock Screen Widgets ist es nun bei Yatse möglich, die wichtigsten Fernbedienungsfunktionen direkt auf dem Lock Screen nutzen zu können, komplett mit Cover- bzw. Filmplakatanzeige des gerade abgespielten Mediums als Hintergrund.

Fazit

Obwohl die aktuelle Reichweite von Android-4.2-Features in einer App noch im niedrigen einstelligen Prozentbereich aller am Google Play Store bekannten Geräte liegt, wird die sinnvolle Einbindung von Daydreams und Lock Screen Widgets sehr wohl von der Nutzergemeinde bereits positiv in den Rezensionen der Apps honoriert.

Es wird wohl noch eine Weile dauern, bis die neuesten Zahlen und Reports als Daydream im Hintergrund in den Büros der Vorstände über die Tablets laufen und digitale Fotorahmen durch Tablets ersetzt sind. Aber in der Zwischenzeit freue ich mich darüber, meine Fernbedienungen nach und nach durch Lock Screen Widgets abzulösen und mehr Informationen mit weniger Aufwand, personalisiert aufbereitet in der Dockingstation und auf dem Lock Screen, serviert zu bekommen.

Zumindest bis es auf Uhren und in Brillen verfügbar ist.

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://bit.ly/12JPjEJ

[2] http://bit.ly/XmYBU9

[3] http://code.google.com/p/android-daydream-samples/

[4] http://developer.android.com/about/versions/android-4.2.html

[5] http://www.n-droid.de/nexus-7-die-offizielle-asus-dockingstation-im-kurztest.html

[6] http://bit.ly/WB6iTT

[7] http://www.androidnext.de/schwerpunkt/android-4-2-lockscreen-meinung/

[8] https://play.google.com/store/apps/details?id=net.nurik.roman.dashclock

[9] http://code.google.com/p/dashclock/

[10] http://gmariotti.blogspot.de/2013/02/how-to-write-dashclock-extension.html

[11] https://play.google.com/store/search?q=DashClock&c=apps

[12] http://www.mobiflip.de/android-lockscreen-widgets-apps/

[13] http://bit.ly/S31bvF

Security-Features in Android Jelly Bean

Sicher ist sicher

Das Betriebssystem Android legt von Anbeginn an großen Wert auf Security und bietet daher etliche sinnvolle Features zur Absicherung mobiler Anwendungen. Angefangen bei der Android Application Sandbox über ein feingranulares Permission-System bis hin zu Security-Frameworks für Encryption & Co. wird dem Entwickler so einiges an Hilfsmitteln an die Hand gegeben, um Anwendungen vor unerwartetem Zugriff zu schützen. Dass hierbei noch lange nicht das Ende der Fahnenstange erreicht zu sein scheint, zeigen die neuen Security-Features in Jelly Bean.

von Lars Röwekamp und Arne Limburg

Wie unter [1] zu lesen, warten die beiden Android-Versionen 4.1 und 4.2 – aka Jelly Beans – gleich mit einer ganzen Reihe losgelöster, neuer Security-Features auf. Unabhängig davon, ob die eigenen Apps für dieses Android Release geplant sind oder nicht, ist es auf jeden Fall sinnvoll, einen Blick auf das eine oder andere neue Feature zu werfen. U. a. lassen sich so auch Rückschlüsse auf das aktuelle Security-Verhalten der eigenen Anwendungen treffen.

Content-Provider

Unter den Android-Kernkomponenten gibt es die eine oder andere, deren Default-Verhalten – unter Sicherheitsaspekten – in der Vergangenheit des Öfteren einmal Grund zur Diskussion gegeben hat. Einen Paradekandidat dieser Fraktion stellt der Content-Provider dar. Er dient per Definition zur Datenteilung zwischen Anwendungen. Es liegt somit in seiner Natur, dass er anderen Komponenten und Anwendungen Zugriff auf ihre Daten erlaubt.

Die Grundphilosophie eines Content-Providers sollte es sein, anderen Anwendungen soviel Zugriffsrechte wie notwendig und so wenig wie möglich zu vergeben, damit diese sinnvoll mit den Daten des Content-Providers arbeiten können. Möchte man zum Beispiel nur lesenden Zugriff erlauben, so kann dies durch folgende Konfiguration erreicht werden:

```
<provider android:name="com.example.ReadOnlyDataContentProvider"
        android:authorities="com.example"
        android:exported="true"
        android:readPermission="com.example.permission.READ_DATA" />
```

Ein Problem ergibt sich bei Android-Anwendungen bis zur Version 4.1 aus der Tatsache heraus, dass das optionale Attribut *exported* bisher als Standard den Wert *true* hatte. Folgende unscheinbare Konfiguration innerhalb der eigenen Anwendung würde somit jeder anderen Anwendung lesenden und schreibenden Zugriff auf die Daten des Content-Providers erlauben:

```
<provider android:name="com.example.DataContentProvider"
        android:authorities="com.example"/>
```

Dieses Verhalten dreht sich nun mit der Android-Version 4.2 um. Ist also die *minSdkVersion* oder die *targetSdkVersion* auf 17 (dies entspricht Android 4.2) oder höher eingestellt, steht zukünftig der Content-Provider nur noch innerhalb der Anwendung selbst zur Verfügung. Ein kleiner aber wichtiger Schritt in Richtung verbesserter Sicherheit. Es gilt im Übrigen als Best Practice, immer das *exported*-Attribut anzugeben, um so die Ausrichtung des Content-Providers explizit zu machen – und dies nicht erst seit Android 4.2.

Der zufällige Zufall

Mit Android 4.2 existiert auch eine neue, auf OpenSSL basierende Implementierung von SecureRandom [2] zur Generierung pseudozufälliger Zahlen. Da sich lediglich die Basis der APIs, nicht aber das API selbst geändert hat, ergibt sich für den Entwickler zunächst einmal kein Bedarf zur Änderung des bestehenden Codes. Problematisch wird es aber, wenn die alte, auf Bouncy Castle basierende Implementierung verwendet wurde, um bewusst deterministisches Verhalten zu erzeugen:

```
SecureRandom secureRandom = new SecureRandom();
byte[] b = new byte[] { (byte) 1 };
secureRandom.setSeed(b);
// Prior 4.2, next line always return same number!
int nextInt = secureRandom.nextInt();
```

Das aufgeführte Anti-Pattern bzw. das damit verbundene Problem zur Verschlüsselung von Strings & Co. ist recht gut unter [3] beschrieben – ebenso wie die eine oder andere deutlich elegantere Lösung.

Risikofaktor WebView

Ein weiteres, potenzielles Sicherheitsloch wurde im Bereich der WebView gestopft. War es bisher möglich, mittels JavaScript Interface, beliebige – als *public* markierte – Java-Methoden eines via *addJavascriptInterface* eingebetteten Java-Objekts aus der WebView heraus aufzurufen, ist dies zukünftig nur noch für Methoden möglich, die explizit mit *@JavascriptInterface* annotiert wurden [4]:

```
// Make method accessible for Android 4.2+
@JavascriptInterface
public void doSomething(String input) {
   ...
  }
```

Auch wenn die bis Android 4.1 gewählte Variante extrem flexibel daher kommt, birgt sie doch die Gefahr, dass „untrusted" Content, der innerhalb der WebView eingebunden wird, mittels Reflection potenziell aufrufbare Methoden der via *addJavascriptInterface* eingebetteten Java-Objekte ausfindig macht und unerlaubt nutzt.

Fazit

Android 4.1 und 4.2 bringen eine ganze Reihe neuer Sicherheitsfeatures mit sich. Die meisten von ihnen sind für den Entwickler transparent und bedürfen daher keiner oder nur geringer Änderungen am bestehenden App-Code. In der Regel geht es bei den meisten Verbesserungen darum, sicherheitsrelevante Informationen und Einstellungen zukünftig explizit anzugeben und nicht weiter hinter Konventionen zu verstecken, die bei Nichtkenntnis zu unerwünschten Sicherheitslücken führen können. Genau aus diesem Grund lohnt sich auch für Entwickler von Android-2.x- und -4.0-Anwendungen ein Blick auf die neuen Features. Wer sich etwas intensiver mit der gezielten Absicherung seiner eigenen Android-Anwendungen beschäftigen möchte, der findet unter [5] einen guten Startpunkt mit etlichen Security Best Practices.

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] Android Jelly Bean – Neue Features: http://developer.android.com/about/versions/jelly-bean.html

[2] SecureRandom: http://developer.android.com/reference/java/security/SecureRandom.html

[3] Android Dev Blog: http://android-developers.blogspot.de/2013/02/using-cryptography-to-store-credentials.html

[4] WebView: http://developer.android.com/reference/android/webkit/WebView.html

[5] Android-Security-Tipps: http://developer.android.com/training/articles/security-tips.html

Vorschau auf die Ausgabe 6.2013

The State of Scala

Vor zwölf Jahren wollte Martin Odersky in Lausanne ein „besseres Java“ entwickeln, das Ergebnis war Scala. Die objektorientierte Sprache hat seitdem ihren Siegeszug durch die Java-Welt angetreten, um sie herum entstand ein ganzes Ökosystem an Projekten. Eines davon haben Sie in den letzten Monaten hier bereits kennengelernt: das Play-Framework. Nächsten Monat wollen wir uns der Sprache selbst widmen und schauen, wie es um ihre Features bestimmt ist. Und wir werfen einen Blick auf das Aktoren-Framework Akka, mit dem sich hochskalierbare Anwendungen bauen lassen und das komplett in Scala geschrieben ist.

Aus redaktionellen Gründen können sich Themen kurzfristig ändern.

Die nächste Ausgabe erscheint am 7. Mai 2013

Querschau

eclipse MAGAZIN — Ausgabe 3.2013 | www.eclipse-magazin.de

- **Tip, Tap, Test: Automatisierte Tests im mobilen Umfeld**
- **Fernsehprogramm aus Eclipse: Anwendungen für Samsung Smart TVs erstellen**
- **Zehn kleine Formate: Vereinheitlichung von Dokumentenschemata über XSD**

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**
- **Wider den Schluckauf: Asynchrone Requests und Race Conditions in GWT**

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:
Christian Bick, Christian Brandenstein, André von Deetzen, Jacob Fahrenkrug, Danno Ferrin, Wolf-Dieter Fink, Cornelia Gilgen, Gerrit Grunwald, Dominik Helleberg, Dierk König, Angelika Langer, Arne Limburg, Bernhard Löwenstein, Klaus Kreft, Christian Meder, Björn Müller, Michael Müller, Werner Müller, Lars Röwekamp, Malgorzata Pinkowska, Thomas Scheuchzer, Remo Schildmann, Gregor Schrägle, Sandro Sonntag, Marc Teufel, Wolfgang Weigend

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