02-2014 | Sommer | www. ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community — für die Community

Java blüht auf

Praxis
Performance richtig bewerten

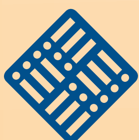
Technologie
HTML5 und Java

Java Server Faces
Umstieg auf 2.x

Oracle
Anwendungsentwicklung mit WebLogic

Java aktuell

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



iJUG Verbund

25.–26. März 2014
im Phantasialand
Brühl bei Köln

Zwei Tage lang das JavaLand besiedeln

Die Konferenz der Java-Community!

- Seien Sie mit dabei, wenn die neue Konferenz zum Zentrum der deutschen Java-Szene wird!
- Wissenstransfer, Networking und gute Gespräche treffen auf spannende Abenteuer, Spaß und Action.
- Vom Einsteiger bis zum Experten haben alle die Gelegenheit, zwei Tage im JVM-Kosmos zu leben.

"Know-how, Erfahrungen, Gespräche, Community Aktivitäten – was ist die größte Attraktion im JavaLand?!"

www.JavaLand.eu

Präsentiert von:

Community Partner:

Partner:

irian
THE JAVA EXPERTS

AZUL SYSTEMS
vaadin }>
BASIS International
PERFORCE
Version everything.

Wolfgang Taschner
Chefredakteur Java aktuell

Herzlich willkommen im JavaLand

Das JavaLand 2014 findet am 25. und 26. März im Vergnügungspark Phantasialand in Brühl statt. Um eine maßgeschneiderte Programmgestaltung für die Community zu gewährleisten, haben die Veranstalter die Java User Groups sehr stark eingebunden. Das Programm besteht aus rund 70 Vorträgen, eine Erweiterung ist in Planung. So soll in Zusammenarbeit mit der Eclipse Foundation unter anderem ein Eclipse-Track entstehen.

Für eine spannende Eröffnungsrede wird Heinz Kabutz Sorge tragen. Der Autor des in Java-Kreisen bekannten „Java Specialists' Newsletter" ist auf das Erkennen und Beheben von Fehlern in paralleler Programmierung spezialisiert. Dazu gehören Deadlocks, Race Conditions und Contentions sowie Fehler in der Java Virtual Machine. In seiner Keynote spricht er über Besonderheiten aus dem täglichen Entwickler-Leben.

Java Champion Andres Almiray wird sich um den Hackergarten kümmern. Dort besteht auch die Chance, mit Java-EE-Evangelist Arun Gupta zusammenzuarbeiten: Er wird zusätzlich zu seiner Session über die „50 Features in Java EE 7" den Kontakt zu den Teilnehmern halten. Für „Java FX"-Themen steht Gerrit Grunwald von der Java User Group Münster bereit. Darüber hinaus erhält das JavaLand Besuch von „Java Technology Ambassador" Stephen Chin. Er wird Interviews mit Persönlichkeiten aus der Java-Szene führen und live auf seine Webseite „nighthacking.com" übertragen.

Für zwei Java-intensive Konferenztage sorgen die „Early Adopters Area", die Aktion „Lambdafy Your Project" sowie das „Java Innovation Lab". Wenn Sie mit Spaß an die Thematik herangehen möchten, sind Sie bei der am Jeopardy-Spiel angelehnten „J-Party" gut aufgehoben. Auch für den körperlichen Ausgleich ist gesorgt: Am zweiten Konferenztag können die Teilnehmer beim JavaLand-Jogging im Phantasialand Luft holen, bevor es in die zweite Runde von JavaLand geht. Alle aktuellen Informationen sind auf www.javaland.eu zu finden.

Auch die Java aktuell ist im JavaLand 2014 präsent. Ich lade Sie herzlich in unser „offenes Redaktionsbüro" ein, um über Inhalte und Ausrichtung der Zeitschrift zu reden. Wer möchte, kann sich auch mit einem Beitrag an der JavaLand-Zeitung beteiligen, die am ersten Tag vor Ort entstehen und am zweiten Tag bereits an alle Teilnehmer verteilt werden wird.

Ich freue mich, Sie im Phantasialand persönlich zu treffen.

Ihr

Trainings für Java / Java EE

- Java Grundlagen- und Expertenkurse
- Java EE: Web-Entwicklung & EJB
- JSF, JPA, Spring, Struts
- Eclipse, Open Source
- IBM WebSphere, Portal und RAD
- Host-Grundlagen für Java Entwickler

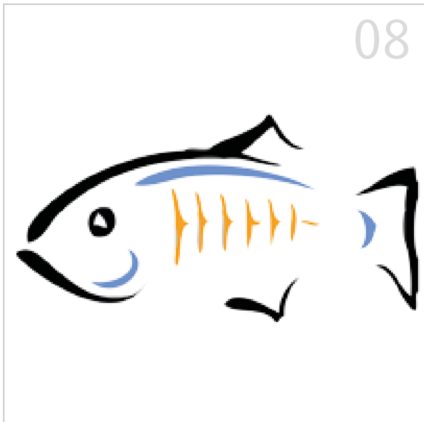
Wissen wie´s geht

Unsere Schulungen können gerne auf Ihre individuellen Anforderungen angepasst und erweitert werden.

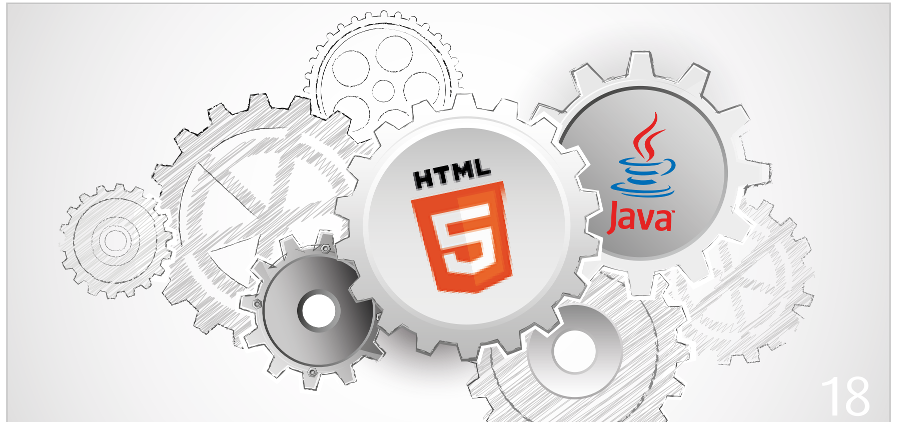
Weitere Themen und Informationen zu unserem Schulungs- und Beratungsangebot finden Sie unter www.aformatik.de

aformatik.®

aformatik Training & Consulting GmbH & Co. KG
Tilsiter Str. 6 | 71065 Sindelfingen | 07031 238070
www.aformatik.de



Die neue Oracle-Strategie zum GlassFish Server, Seite 8



HTML5 und Java-Technologien, Seite 18

3 Editorial

5 Das Java-Tagebuch
Andreas Badelt, Leiter der DOAG SIG Java

8 Die neue Oracle-Strategie zum Glass-Fish Server
Sylvie Lübeck und Michael Bräuer

10 Back to Basics: Wissenswertes aus „java.lang.*"
Christian Robert

13 Performance richtig bewerten
Jürgen Lampe

18 HTML5 und Java-Technologien
Peter Doschkinow

23 Java und funktionale Programmierung – ein Widerspruch?
Kai Spichale

26 Eingebettete DSLs mit Clojure
Michael Sperber

34 Lohnt sich der Umstieg von JSF 1.x auf JSF 2.x?
Andy Bosch

38 Anwendungsentwicklung mit dem Oracle WebLogic Server 12c
Michael Bräuer

46 Automatisierte Web-Tests mit Selenium 2
Mario Goller

51 Contexts und Dependency Injection – die grundlegenden Konzepte
Dirk Mahler

56 Distributed Java Caches
Dr. Fabian Stäber

59 Activiti in produktiven Umgebungen – Theorie und Praxis
Jonas Grundler und Carlos Barragan

63 Unbekannte Kostbarkeiten des SDK Heute: Vor und nach der „main()"-Methode
Bernd Müller

65 Java ist hundert Prozent Community
Interview mit Niko Köbler

66 Inserentenverzeichnis

66 Impressum

Java und funktionale Programmierung - ein Widerspruch?, Seite 23

Eingebettete DSLs mit Clojure, Seite 26

Das Java-Tagebuch

Andreas Badelt, Leiter der DOAG SIG Java

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java – in komprimierter Form und chronologisch geordnet. Der vorliegende Teil widmet sich den Ereignissen im vierten Quartal 2013.

16. Oktober 2013

JavaLand: Call for Paper mit hoher Beteiligung beendet
Die Zahl der Vortragseinreichungen war mit 400 sensationell hoch. Die neue, Community-getriebene Java-Konferenz geht Ende März 2014 im Phantasialand zum ersten Mal an den Start – und wie es aussieht, mit einem herausragenden Mix an Vorträgen. Wenn das insbesondere auf Interaktion mit und innerhalb der Community ausgerichtete Rahmenprogramm ähnlich einschlägt, wird es eine sehr erfolgreiche Konferenz werden.
http://www.javaland.eu

29. Oktober 2013

vJUG: Java-Vorträge auch auf der einsamen Insel
Simon Maple von der London Java Community hat auf der Community Night der JAX 2013 seine neugegründete, globale virtuelle JUG vorgestellt: die vJUG. Java-Fans, die keine JUG in ihrer Nähe haben (und nicht genügend Interessenten im Umkreis, um eine zu gründen), sollen nicht darauf verzichten müssen, regelmäßig gute Java-Vorträge zu hören und sich mit anderen Java-Fans auszutauschen. Natürlich innerhalb der Grenzen der Virtualität, aber das Setup mit Live Streaming über YouTube und einem IRC Channel für die Interaktion der Teilnehmer untereinander sowie mit Referenten und Moderatoren erscheint ausreichend professionell. Einziger Haken: Alle Meetings und Vorträge sind auf Englisch.
http://www.meetup.com/virtualJUG

Das JCP Executive Committee ist gewählt
Die diesjährigen Wahlen zum Executive Committee des JCP sind ohne Überraschungen beendet. Alle Kandidaten für „ratified seats" (Vorschlagsrecht allein bei Oracle) wurden angenommen: Red Hat, SouJava, IBM, Intel, Ericsson, Credit Suisse, Fujitsu und Hewlett-Packard für zwei Jahre sowie Gemalto M2M (vormals Cinterion Wireless Modules GmbH), Software AG, SAP, Freescale, Goldman Sachs, Nokia, V2COM und TOTVS für ein Jahr. Bei den „open election seats" setzten sich Eclipse Foundation, Twitter, London Java Community und CloudBees (für zwei Jahre) sowie ARM, Azul Systems, Werner Keil und die MoroccoJUG (für ein Jahr) durch. Die unterschiedlichen Wahlperioden sind durch die Reorganisation des JSP entstanden. Weitere JUGs aus China, Peru und Indonesien konnten sich nicht durchsetzen, aber allein die Tatsache, dass sie teilgenommen haben und dass drei JUGs im Executive Committee vertreten sind (die brasilianische SouJava sogar auf einem „ratified seat"), zeigt, dass die Community die Standardisierung von Java nicht als Aufgabe einiger weniger Firmen ansieht.

Henrik Ståhl: Oracle und ARM erweitern Zusammenarbeit
Oracle und ARM kooperieren bereits ist Längerem, um Java optimal auf ARM-Chips einzusetzen. Nun wollen sie diese Kooperation unter dem Stichwort „Internet of Things" erweitern. Diese Nachricht ist für die Java-Plattform-Gruppe bei Oracle bedeutsam genug, um Chef-Produktmanager Henrik Ståhl zu einem Blog-Post zu bewegen. Die auf der TechCon 2013 angekündigte Kooperation sieht unter anderem vor, dass ARM seine Implementierung des CoAP-Kommunikationsprotokolls („Constrained Application Protocol") zum OpenJDK beisteuert. Außerdem soll der mbed Hardware Abstraction Layer (HAL) als Portabilitätsschicht genutzt werden, um in Zukunft Java ME auf allen mbed-fähigen Geräten ohne weiteren Aufwand ausrollen zu können.
https://blogs.oracle.com/henrik/entry/armtechcon2013

6. November 2013

Oracle stellt kommerzielle GlassFish-Version ein
Ein Blog-Eintrag im „Aquarium" als Paukenschlag: Oracle stellt die kommerzielle Version von GlassFish ein! Anwender, die kommerziellen Support benötigen (und welche Betriebsabteilung wird sagen: „Brauchen wir nicht"?), sollen auf WebLogic migrieren. Die Open-Source-Variante von GlassFish bleibt zwar erhalten und soll auch Referenz-Implementierung für Java EE 8 werden. Trotzdem trauert die Szene bereits um den Tod des Applikations-Servers: Denn die Erfüllung des Java-EE-8-TestCompatibility-Kit bedeutet noch lange nicht, dass es weiterhin ein stabiler, weitestgehend fehlerfrei laufender Applikations-Server sein wird. Vieles spricht dafür, dass Oracle den bisherigen Aufwand deutlich herunterschraubt. Damit GlassFish eine Zukunft hat, müsste er zu einem echten Open-Source-Projekt gemacht werden. Dafür müssten aber die heutigen Hürden für interessierte Entwickler abgeschafft werden. Für alle Kunden, die GlassFish für kritische Anwendungen einsetzen und nicht auf Version 3.x sitzen bleiben wollen, bleibt neben WebLogic noch der (vermutlich steinigere) Umstieg auf einen der beiden verbleibenden Open-Source-Applikations-Server mit kommerziellem Support: Red Hat Wildfly (aka JBoss) oder Apache TomEE.
http://www.doag.org/home/aktuelle-news/article/oracle-stellt-kommerzielle-glassfish-version-ein.html

8. November 2013

JSR-Updates
Neues vom JCP: Eine ganze Reihe von JSRs, insbesondere Java SE 8 (der „Umbrella" oder auf Deutsch „Mantel-JSR") und mehrere darin enthaltene JSRs sind ins „Public Review" (neue Spezifikationen) beziehungsweise „Maintenance Review" (Aktualisierungen) gegangen. Darüber hinaus ist JCache nach über einer Dekade Laufzeit und der Reaktivierung im Jahr 2011 jetzt abgesegnet worden und hat den „Proposed Final Draft" veröffentlicht.

12. November 2013

Ceylon 1.0 und Reactor 1.0
Zweieinhalb Jahre nach der offiziellen Ankündigung ist die von RedHat beziehungsweise Gavin King getriebene Programmiersprache Ceylon jetzt in der Version 1.0 veröffentlicht worden. Die Sprache ist als (voll interoperable) Alternative zu Java gedacht und lässt sich für den Einsatz auf der JVM genauso wie auf JavaScript-Engines kompilieren. Die Motivation für das Projekt war eher, die Hauptschwächen von Java anzugehen, wie etwa fehlende Modularisierungsmöglichkeiten und der Bruch zwischen Client und Server. Ich bin gespannt, ob Ceylon seine angedachte Rolle als Java-Herausforderer einnimmt – entscheiden wird die Community. Die IDE für Ceylon 1.0 ist Eclipse, mit entsprechenden Plugins für sprachspezifische Unterstützung. Dazu kommen die Compiler und für die Kommandozeile ein einzelnes Executable „ceylon", das wie auch Git sämtliche Funktionen als Unterkommandos inklusive Autovervollständigung enthält und über einen Plug-in-Mechanismus erweiterbar ist. Der Ceylon-Quellcode ist übrigens komplett auf GitHub verfügbar.
http://ceylon-lang.org

Ebenfalls in der Version 1.0 ist heute Reactor – ein Framework für asynchrone Hochgeschwindigkeits-Anwendungen („fast data"), basierend auf dem gleichnamigen Design Pattern, veröffentlicht worden. Es setzt auf Java und Groovy, soll sich aber ohne großen Aufwand mit anderen JVM-Sprachen nutzen lassen, unter anderem gibt es schon Unterstützung für Clojure. Die Version 1.0 unterstützt bereits das neue JDK 8. Hinter Reactor steckt Pivotal, ein gemeinsames Spin-off von EMC und VMWare und auch der neue Hüter von SpringSource.
https://spring.io/blog/2013/11/12/it-can-t-just-be-big-data-it-has-to-be-fast-data-reactor-1-0-goes-ga

20. November 2013

Weitere JavaOne-Sessions online
Für diejenigen, die nicht zur JavaOne nach San Francisco reisen konnten oder sich die Vorträge nochmal in Ruhe zuhause anschauen wollen – eine Reihe weiterer Sessions ist jetzt online verfügbar.
https://blogs.oracle.com/java/entry/more_javaone_sessions_online

21. November 2013

BrianGoetzimInterview:Waswarsoschweran Lambdas?
Brian Goetz, Chief Architect of the Java Language, hat dem Oracle Technology Network auf der Devoxx ein Kurzinterview zu Lambdas gegeben, das auch auf YouTube verfügbar ist. Auf die Frage, warum es so lange gedauert hat, antwortet er, dass es im Wesentlichen zwei für Programmierer in der Regel unsichtbare Features waren, deren Zusammenspiel man grundsätzlich hätte überarbeiten müssen, um Lambdas wirklich in die Sprache und die Plattform zu integrieren: „Method Overload Selection" und „Type Inference". Ohne diesen Aufwand wären Lambdas nur aufgepfropft gewesen und die Programmierer hätten sie schnell wieder ignoriert. Herausgekommen sei aber etwas, das sich gut anfühlt, und er sei sich sicher, dass die Programmierer stattdessen fragen werden: „Wie konnte ich jemals ohne Lambda-Konstrukte in Java leben?" Die neuen Sprachkonstrukte zu lernen, sei auch gar nicht so schwer, zumal andere Sprachen ja bereits Vergleichbares haben, es werde allerdings ein bisschen dauern, ein Gefühl dafür zu entwickeln, wo und wie sie sich am besten einsetzen lassen.
https://blogs.oracle.com/java/entry/brian_goetz_offers_a_view

28. November 2013

Groovy 2.2
Gerade hatten wir noch das Stichwort „Groovy", da passt es doch gut, dass jetzt die Version 2.2 veröffentlicht wurde. Allerdings sind die Neuerungen nur sehr spärlich, unter anderem gibt es eine implizite „Closure Coercion", also die Umwandlung einer Closure in ein Interface oder einen abstrakten Typ ohne die „as type"-Angabe, ähnlich den neuen Lambda-Konstrukten im JDK 8.

13. Dezember 2013

Spring 4.0 ist da
Passend zu den Temperaturen im Dezember hierzulande ist Spring schon da – in der Version 4.0. SpringSource-CTO Adrian Colyer gibt sich in der Ankündigung ganz bescheiden: Das neue Release ist „positioniert, um die nächste Dekade JVM-basierter Innovation zu ermöglichen". Es unterstützt bereits Java 8, getestet mit Pre-Releases des JDK. Die Mindestversion ist jetzt Java SE 6; Java EE 6 und die enthaltenen Spezifikationen (insbesondere JPA 2.0 und Servlet 3.0) sind als „Baseline" definiert. Größere Änderungen hat es in der Gesamtarchitektur gegeben, Stichwort „Micro-Service-Architektur (MSA) und REST". Das Ziel lautet „Reactive, event-driven applications" – dabei soll auch die HTML5 und WebSocket-Unterstützung helfen. Das Annotations-getriebene Modell, mit Spring 3 anstelle endloser XML-Konfigurationsdateien eingeführt, ist auch deutlich erweitert worden.
http://spring.io/blog/2013/12/12/announcing-spring-framework-4-0-ga-release

18. Dezember 2013

OraclewirdweiterhinindasGlassFish-Projekt investieren
Oracle reagiert in einem deutschsprachigen Webcast auf die Kritik an der Einstellung der kommerziellen GlassFish-Version (siehe Seite 8). Fazit ist, dass Oracle weiterhin in das GlassFish-Projekt investieren wird, um die Open-Source-Edition bereitzustellen.

13. Januar 2014

James Gosling bewertet Oracle
James Gosling, der Urvater von Java, wurde vom Online-Magazin Infoworld zum vierten Jahrestag der Übernahme von Sun durch Oracle um eine Bewertung darüber gebeten, wie gut die übernommenen Technologien bei Oracle gediehen sind. Die Einschätzungen sind gemischt, ein „F-" (schlechter geht es im amerikanischen Notensystem nicht) gibt es für Solaris („totally dead"), zumindest noch ein „C" für MySQL. Interessanterweise sieht Gosling die Entwicklung bei Java mit ein bisschen Abstand recht positiv: Ein „B+" (also eine 2+) mit dem Kommentar „They've done surprisingly well with Java except for the 'growing pains' in figuring out how to deal with security issues". Auch für GlassFish gibt es noch ein „B-" – ob da die Einstellung der kommerziellen Variante schon eingeflossen ist? Die Begründung trifft es jedenfalls ganz gut: „It´s moved forward, but isn´t getting promoted much".
http://www.infoworld.com/t/technology-business/james-gosling-grades-oracles-handling-of-suns-technology-233924

JDK 8 kommt im März – auch mit Bugs
Mathias Axelsson, der Release Manager des JDK 8, hat in einer Mail die voraussichtliche Einhaltung der aktuell gesetzten Termine bestätigt: „We´re on track to have the release candidate built before the January, 23 deadline". Allerdings schreibt er auch, dass momentan nur P1-Bugs („Showstopper") für Fixes im initialen Release eingeplant sind. Damit soll der anvisierte Liefertermin am 18. März 2014 eingehalten werden. Alle weiteren Fixes sollen dann in späteren Bugfix-Releases folgen. Man darf dem initialen Release also mit gemischten Gefühlen entgegensehen.

14. Januar 2014

Community-Umfrage: Die Zukunft von Java EE
Heute hat Oracle den zweiten Teil einer Umfrage zur Zukunft von Java EE gestartet. Damit soll die Community bereits in einem frühen Stadium – die Expert Group für Java EE 8 existiert noch gar nicht – die Möglichkeit zur Mitsprache über die Themen erhalten. Der erste Teil der Umfrage war bereits im Dezember gestartet worden, ist aber inzwischen geschlossen. Mit ein bisschen Glück ist dieser zweite Teil bei Erscheinen des Tagebuchs noch geöffnet – es wurde kein Termin genannt.
http://glassfish.org/survey

16. Januar 2014

NetBeans 8 Beta verfügbar
Das NetBeans-Team hat die Beta-Version von Release 8 veröffentlicht – besser gesagt angekündigt, da es sich ja um Open-Source-Software handelt. Als roter Faden zieht sich natürlich die Unterstützung von Java 8 in allen Facetten durch die Release Notes. Für Java Embedded gibt es einiges Neues, unter anderem sollen Applikationen direkt aus der IDE auf Embedded Devices (wie ein Raspberry Pi) deployed, gestartet, debugged und „profiliert" werden können. Weitere Highlights sind Tomcat-8.0- und TomEE-Integration sowie eine wesentlich verbesserte JavaScript-Unterstützung, inklusive Debug-Unterstützung für die im JDK integrierte Nashorn-Engine. Übrigens hat Arun Gupta, ehemals GlassFish-Guru bei Oracle und seit Kurzem in Diensten von RedHat, in seinem Blog vor wenigen Tagen ein NetBeans-8-Plug-in für WildFly vorgestellt (vormals JBoss und schon länger bei RedHat). Dieses wird vermutlich kein Kernbestandteil von NetBeans werden, aber RedHat hat Interesse an einer guten Unterstützung für WildFly aus dieser wichtigen IDE heraus, und wird die Plug-in-Entwicklung sicher weiter vorantreiben.

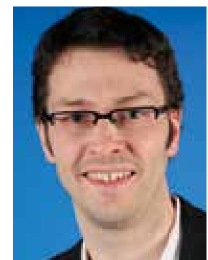
20. Januar 2014

Scala wird zehn
Wie die Zeit vergeht. Scala – laut vielen Umfragen und Statistiken mit Clojure und Groovy einer der Top-3-Herausforderer von Java auf der JVM – ist zehn Jahre alt. Am 20. Januar 2004 hat Martin Odersky vom Swiss Federal Institute of Technology die Fertigstellung der ersten Version bekanntgegeben und damit der funktionalen Programmierung zum Einzug in die JVM verholfen; Scala vereint Aspekte der objektorientierten mit denen der funktionalen Programmierung. Momentan wird an der Version 2.11 gearbeitet. In dieser fehlt allerdings unter dem Motto „Verschlanken" die parallele Unterstützung für .NET – für die reinen Java-Fans (oder besser gesagt JVM-Fans) ist das aber wohl nicht so relevant.

30. Januar 2014

DevCamp 2014: Ein neues Veranstaltungskonzept der DOAG kommt sehr gut an
In der Münchner Allianz Arena fand zum ersten Mal bei der DOAG eine Veranstaltung ohne festes Programm statt. Lediglich das Thema war bekannt: Moderne Softwareentwicklung im Oracle-Umfeld. Die Teilnehmer, die ein bestimmtes Thema im Sinn hatten, stellen ihren Vorschlag für eine Session vor und brachten ihn in die Planung ein. Im Nu waren alle zwanzig Slots gefüllt, die Themenvielfalt war enorm. Die Gruppen trafen sich in den jeweiligen Räumen und begannen mit der Arbeit. Das Feedback war durchgehend positiv: Manch einer war überrascht, dass viele Teilnehmer sich bereits im Vorfeld sehr intensiv mit dem Thema der Session beschäftigt hatten. In den meisten Fällen blieb man nicht nur an der Oberfläche sondern konnte sehr tief ins Detail gehen. Alle Teilnehmer haben sich eingebracht und es konnte jeder etwas für sich mitnehmen. Sie waren sich einig: Ein solches Veranstaltungsformat soll es wieder geben.
www.doag.org/go/devcamp

Andreas Badelt
Leiter der DOAG SIG Java



Andreas Badelt ist Senior Technology Architect bei Infosys Limited. Daneben organisiert er seit 2001 ehrenamtlich die Special Interest Group (SIG) Development sowie die SIG Java der DOAG Deutsche ORACLE-Anwendergruppe e.V.

UPDATE An dieser Stelle erhalten Sie in jeder Ausgabe ein Update über das Geschehen in der Java-Community

Die neue Oracle-Strategie zum GlassFish Server

Sylvie Lübeck und Michael Bräuer, ORACLE Deutschland B.V. & Co. KG

Am 4. November 2013 gab Oracle im GlassFish-Blog „Aquarium" [1] eine Änderung der bisherigen GlassFish-Server-Strategie bekannt.

Auf die Ankündigung folgte ein großes Echo in der Community. Aussagen wie „Der GlassFish Server ist tot" oder „Für den GlassFish Server wird es keinen kommerziellen Support mehr geben" gaben jedoch nur einen Teil des Announcement wieder. Oracle hat daraufhin unter anderem auch in einem Webcast in deutscher Sprache die Fakten [2] zusammengestellt.

Der GlassFish Server lebt weiter

Als maßgeblich Verantwortlicher für Java und Java EE wird Oracle weiterhin die Entwicklung für den GlassFish Server als strategische Open-Source-Plattform im Java-EE-Bereich sowie das GlassFish-Projekt unterstützen. Dies umfasst die Bereitstellung folgender Bestandteile:

- Die zukünftige Java-EE-Referenzimplementierung (RI)
  - Der Java Community Process verlangt vom Specification Lead eine Implementierung zum Nachweis der Machbarkeit/Umsetzbarkeit der verschiedenen Spezifikationen.
  - Der GlassFish Server dient als RI für Java EE
  - Plan: GlassFish 5.0 wird Java EE 8 implementieren
- Das Java EE SDK
  - Tutorials, Beispiele und Dokumentation für Entwickler, die sich mit Java EE beschäftigen
  - Plan: Bereitstellung eines Java EE SDK für die künftigen Java-EE-Releases
- Die GlassFish Server Open Source Edition
  - Kostenloser und Open-Source-basierter Server für das Deployment von Java-EE-Anwendungen
  - Plan: jährliches Release mit Bug-Fixes und Updates

Der Support für bestehende GlassFish Server wird nicht eingestellt

Der Support der Oracle GlassFish Server 3.0.x und 3.1.x läuft weiter. Es gilt weiterhin die Oracle Lifetime Support Policy [3]. Nach dem aktuellen Stand geht der Premier Support für den GlassFish Server 3.1.x bis März 2016 und der Extended Support bis März 2019. Für die GlassFish Server Open Source Edition sind reguläre Patch-Updates ohne kommerziellen Support geplant. Oracle ist hier anderen Unternehmen darin gefolgt, kommerziellen Support nur noch für einen Application Server im Java-EE-Umfeld anzubieten.

WebLogic kann eine preisgünstigere Alternative sein

Der GlassFish Server basiert auf der GlassFish Server Open Source Edition (OSE) mit zusätzlichen kommerziellen Funktionen. Lizenzseitig beinhaltet er auch Java-SE-Support und wird auf Prozessor-Basis lizenziert. Die WebLogic Server Standard Edition (SE) ist als kleinste Edition des WebLogic Server entsprechend der für verschiedene Produkte geltenden „Standard Edition"-Regelung pro Socket lizenziert. Nach der aktuellen Preisliste fährt man beim Einsatz einer CPU mit vier oder mehr Kernen mit der WebLogic SE günstiger. Zudem gibt es mit den verschiedenen WebLogic-Editionen entsprechend höherwertigen Support für die Java-Laufzeitumgebung und zusätzliche Java-Diagnosemöglichkeiten mittels „Java Mission Control" [4]. Für Entwickler wird eine kostenfreie Lizenz angeboten [5].

Die unterschiedlichen Editionen des WebLogic Server [4] bieten vielseitige Funktionen wie das WebLogic-Diagnostic-Framework, eine umfassende Script-Sprache namens „WLST" zur Script-basierten Administration, Work Manager zur Ressourcen-Priorisierung, „Side by Side"-Deployment, Oracle ADF und TopLink, Web-Tier (Oracle HTTP Server), Unterstützung für andere Fusion-Middleware-Produkte, Optimierungen für den Zugriff auf Oracle-Datenbanken wie optimale RAC-Integration, massives In-Memory Computing mittels Oracle Coherence, erweiterte Hochverfügbarkeit-Funktionalitäten wie Service Migration und Server Migration, Java Mission Control und Flight Recorder, Oracle-JDK-Unterstützung etc.

Support für Builds mit kommerziell angereicherten Features

Es gibt Support für die jeweiligen Open-Source Application-Server von entsprechenden Anbietern. Somit ist die Code-Basis im Allgemeinen nicht genau dieselbe wie die der Community-Builds. Auch Oracle hat nie kommerziellen Support für

GlassFish Server Open Source Edition Builds angeboten. Der kommerziell unterstützte Oracle GlassFish Server wurde von der OSE abgeleitet und mit zusätzlichen Funktionen angereichert; spezielle Downloads wurden zur Verfügung gestellt. Eine Ausnahme ist die Firma Tomitribe, die kommerziellen Support für Apache TomEE anbietet [6].

GlassFish und WebLogic Server nutzen viele gemeinsame Komponenten

Obwohl die beiden Produkte auf unterschiedlichen Code-Basen beruhen, nutzen GlassFish Server und WebLogic Server gemeinsame Komponenten. Unter anderem greifen beide auf die gleichen Implementierungen für JPA, JSF, WebSockets, CDI, Bean Validation, JAX-WS, JAXB, und WS-AT zurück.

Fazit

Oracle wird weiterhin in das GlassFish-Projekt investieren, um die Open-Source-Edition als eine entwicklerfreundliche Community-Distribution, das SDK und die Referenz-Implementierung für die Java-EE-Plattform bereitzustellen. Kunden bekommen entsprechend der Life Time Support Policy weiterhin Unterstützung.

Oracle ruft die Community zur aktiven Beteiligung durch „Adopt-a-JSR" und „Contributions to GlassFish" auf, wird sich – wie andere Anbieter auch – auf einen kommerziellen Java-EE-Application-Server fokussieren, der kommerziellen Support liefert, und damit dem Weg anderer Hersteller folgen.

Die Alternative WebLogic muss nicht teurer sein als der Oracle GlassFish Server. Zwar handelt es sich beim WebLogic Server um Closed-Source Software, aber Oracle bietet auch hier eine kostenfreie OTN-Lizenz für Entwickler an.

Wer vorhat, Open-Source-Produkte einzusetzen, sollte immer beachten, dass die kommerziell unterstützten Produkte oft um kommerzielle Features angereichert sind und sich dadurch von den Community Builds unterscheiden.

Weiterführende Links

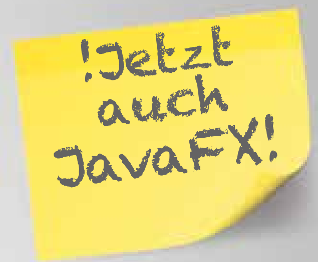
[1] https://blogs.oracle.com/theaquarium/entry/java_ee_and_glassfish_server
[2] https://blogs.oracle.com/brunoborges/entry/6_facts_about_glassfish_announcement
[3] http://www.oracle.com/us/support/library/lifetime-support-middleware-069163.pdf
[4] http://docs.oracle.com/middleware/1212/core/FMWLC/products2.htm#CHCJBBEG
[5] https://blogs.oracle.com/brunoborges/entry/weblogic_server_free_for_developers
[6] http://www.tomitribe.com/support

Sylvie Lübeck
sylvie.luebeck@oracle.com

Sylvie Lübeck ist in der ORACLE Deutschland B.V. & Co. KG in der Abteilung „Business Unit Server Technologies – Fusion Middleware", die deutschlandweit die Middleware-Themen technisch und vertriebsunterstützend verantwortet. Ihr Schwerpunktthema ist der Oracle WebLogic Server.

Michael Bräuer
michael.braeuer@oracle.com

Michael Bräuer ist leitender Systemberater der ORACLE Deutschland B.V. & Co. KG. Er nutzt seine langjährige Berufserfahrung in den Bereichen Anwendungsintegration, Middleware und Java EE, um kritische Geschäftsanwendungen auf das passende technologische Fundament zu stellen.



QF-TEST
Das GUI Testtool
für Java und Web

FX/Swing/SWT/RCP und Web
Capture/Replay & Skripting
Für Entwickler und Tester
System- und Lasttests
HTML und AJAX
Benutzerfreundlich
Robust und zuverlässig
Plattform- & Browserübergreifend
Etabliert bei 600 Kunden weltweit
Deutsches Handbuch
Deutscher Support

„Die Vollkommenheit besteht nicht in der Quantität, sondern in der QUALITÄT."
Baltasar Gracian v. Morales

www.qfs.de
Quality First Software GmbH
Tulpenstraße 41
82538 Geretsried
Deutschland
Fon: + 49. (0)8171. 38 64 80

Back to Basics: Wissenswertes aus „java.lang.*"

Christian Robert, SapientNitro

Das Package „java.lang." und dessen Unterpackages enthalten die grundlegendsten Bestandteile des Java-API. Auch wenn die Java-Plattform den Entwickler weitestgehend von den Interna des Betriebssystems und der Hardware abschirmt, ist es dennoch bei vielen Problemstellungen hilfreich, grundlegende Funktionsweisen der VM zu verstehen und unter die Haube blicken zu können.*

Gerade über die Klassen in „java.lang.*" ist die Schnittstelle zur Virtual Machine besonders greifbar. Oftmals bestehen hier aber Wissenslücken sowie kleinere und größere Fragezeichen über die genaue Funktionsweise bestimmter API-Bestandteile. Dieser Artikel zeigt anhand von Beispielen aus der Praxis, wie mit ein wenig (mehr) Hintergrundwissen reale Probleme einfacher beziehungsweise besser angegangen und effizienter gelöst werden können und wo potenzielle Stolperfallen lauern.

Speicherverwaltung

Dank der Java Virtual Machine und der Tatsache, dass Java mithilfe der Garbage Collection den höchst fehleranfälligen Code zur Allokierung und Freigabe von Speicher erspart, kommen Java-Entwickler nur selten mit den Details der Speicherverwaltung in Berührung. Spätestens wenn Anwendungen in den Kontakt mit der realen Welt treten, gibt es allerdings durchaus Situationen, in denen ein tieferes Wissen über die Art und Weise, wie die Virtual Machine mit Arbeitsspeicher umgeht, hilfreich, wenn nicht sogar zwingend notwendig ist. Abbildung 1 zeigt eine vereinfachte Übersicht darüber, wie die Virtual Machine den Speicher aufteilt, der von Java-Objekten eingenommen werden kann.

Alle Objekte, die von der Virtual Machine allokiert sind – in der Regel über „new XYZ()" oder „Reflection" – landen zunächst in der „Young Generation". Überleben sie dort genügend Garbage-Collector-Durchläufe, werden sie in die „Old Generation" verschoben. Die Generationen unterscheiden sich durch die unterschiedlichen Algorithmen, die der Garbage Collector zur Verwaltung verwendet.

Die „Permanent Generation" steht als dritter Speicherbereich relativ unabhängig daneben. Normale Objekte, die während der Programmlaufzeit entstehen, landen niemals dort. Stattdessen wird dieser Bereich für spezielle Objekte wie String-Konstanten sowie Metadaten (Informationen über Klassen, Felder, Methoden etc.) verwendet.

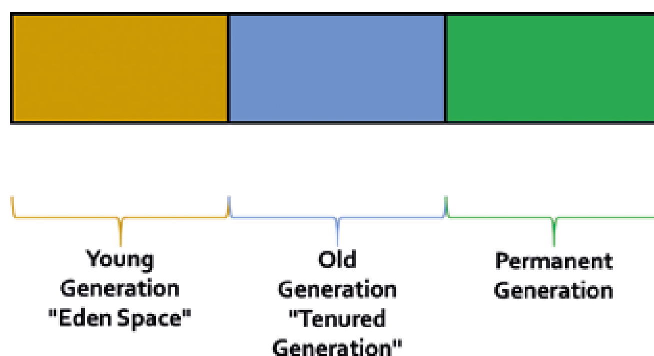


Abbildung 1: Speicheraufbau

OutOfMemory-Error: „PermGen space"

Jeder, der zu Beginn seiner Karriere als Java-Entwickler mehrere Anwendungen (oder auch nur eine relativ große Anwendung) in einem Servlet-Container oder Application-Server eingesetzt hat, wird früher oder später im Logfile einem OutOfMemory-Error mit der Ergänzung „PermGen space" begegnet sein. Die übliche Lösung für solche Fälle ist im 21. Jahrhundert eine schnelle Google-Suche, die einem ungefähr diese Problemlösung präsentiert: „Zum Start-Kommando ‚java' beziehungsweise ‚javaw' sollen innerhalb eines Start-Skripts die Argumente ‚JAVA_OPTS="–Xms512m –Xmx1024m –XX:NewSize=256m –XX:MaxNewSize=256m –XX:PermSize=256m –XX:MaxPermSize=512m"' hinzugefügt werden."

Es sind sechs verschiedene Optionen, die gesetzt werden sollen, um die unterschiedlichen Speicherbereiche zu konfigurieren. Nicht selten ist dies der Punkt, an dem vom Mitdenken zum Copy & Paste umgeschaltet wird. Es lohnt sich allerdings, die einzelnen Optionen ein wenig genauer anzuschauen und zu verstehen, welche einzelnen Bereiche hier mit welchen Werten definiert werden. In Abbildung 2 ist das Schaubild der Speichergenerationen um die entsprechenden Parameter, die der Virtual Machine mitgegeben werden können, erweitert. Jetzt sind die einzelnen Optionen aus dem Google-Beispiel besser zu verstehen.

Die Option „–Xms512m" setzt den für die Young und Old Generation reservierten initialen Speicher, der von der Virtual Machine beim Start des Betriebssystems reserviert wird, auf 512 MB. „–Xmx1024m"

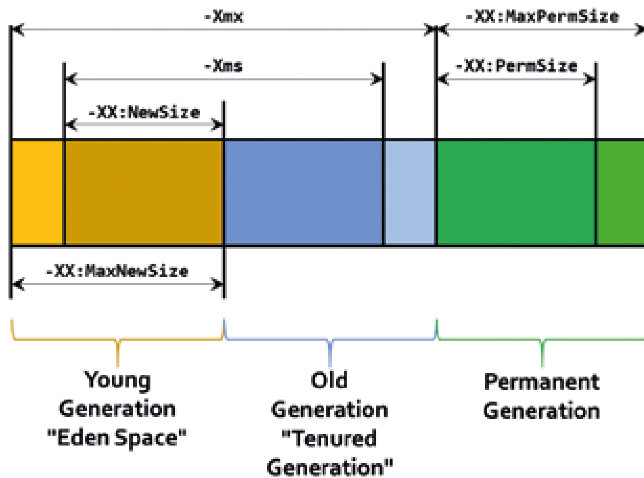


Abbildung 2: Speicheraufbau und Startoptionen

definiert als passenden Gegenwert den maximalen Speicher, den die Virtual Machine vom Betriebssystem anfordern kann, auf 1.024 MB. Über „–XX:NewSize“ und „–XXMaxNewSize“ wird die Verteilung dieser 512 bis 1.024 MB zwischen Young und Old Generation beeinflusst. Analog dazu definieren „–XX:PermSize=256m“ und „–XX:MaxPermSize=512m“ den initialen und den maximalen Wert der Permanent Generation. Was also zunächst nach einer schlecht überschaubaren Sammlung von Speicher-Parametern aussieht, lässt sich nun einordnen und verstehen.

„PermGen“ wird zu Metaspace

Der bisher beschriebene Aufbau der einzelnen Speicherbereiche ist bis einschließlich Java 7 gültig. Ab Java 8 wird es die Permanent Generation in dieser Art nicht mehr geben. Der erste Teil der Objekte, die bis Java 7 in der Permanent Generation verwaltet werden – die String-Konstanten – liegt ab Java 8 in der Old Generation. Der zweite Teil – die Meta-Informationen (Klassen, Methoden etc.) – werden in einem neuen Speicherbereich verwaltet, dem sogenannten „Metaspace“. Dieser ist auf den ersten Blick konzeptionell identisch mit der Permanent Generation, hat jedoch einen entscheidenden Unterschied: Er ist hinsichtlich seiner Größe unbeschränkt.

Während Permament Generation sowie auch Young und Old Generation bei jedem Start der Virtual Machine einen maximalen Wert erhalten – entweder explizit durch die entsprechende Option oder implizit durch einen Standardwert –, kann der Metaspace bis zum maximal vom Betriebssystem zur Verfügung gestellten Speicher anwachsen, ohne dass dafür eine zusätzliche Option notwendig ist.

Was zunächst wie eine Erleichterung ausschaut – es muss schließlich bei der Konfiguration etwa eines Servlet-Containers nicht mehr darauf geachtet werden, eine genügend große Permanent Generation zu definieren –, kann zu einem neuen und bisher für Java-Anwendungen noch nie dagewesenen Verhalten führen: das komplette Füllen des Speichers eines Systems.

Erfolgt zum Beispiel innerhalb eines Servlet-Containers der häufige Reload von Anwendungen, besteht die Möglichkeit, dass – sofern nicht alle Ressourcen sauber aufgeräumt werden können – einzelne Klassen-Definitionen und Metadaten im Speicher verbleiben. Dies führt früher oder später zum eingangs beschriebenen OutOfMemory-Error mit der Zusatz-Information „PermGen space“. Zukünftig wird dieser Fehler so nicht mehr auftauchen, da der Metaspace ja standardmäßig unbeschränkt ist. Ein potenzielles „Memory Leak“ wird deshalb möglicherweise erst sehr viel später entdeckt. Es kann sogar – wenn das Betriebssystem durch zunehmende Speicherbelegung auf Swapping ausweichen muss – einen extremen Einfluss auf die System-Performance haben.

Es gilt also in solchen Szenarien, ein zusätzliches Auge darauf zu haben, dass tatsächlich alle von einer Anwendung belegten Ressourcen korrekt freigegeben werden. Jedoch kann auch das alte Verhalten der Permanent Generation für den Metaspace aktiviert sein, das unbeschränkte Wachsen des Speichers lässt sich also über die Kommandozeilenoption „–XX:MaxMetaspaceSize“ abschalten. Bei Eingabe von „–XX:MaxMetaspaceSize=512m“ wird dann auch weiterhin ein OutOfMemory-Error ausgegeben, sobald die Größe des Metaspace 512 MB überschreitet.

java.lang.Runtime#exec

Nicht immer lassen sich alle Anwendungslogiken direkt innerhalb der eigenen Anwendung abbilden. Gerade in Integrationsprojekten ist es oftmals notwendig, bereits bestehende Anwendungen aufzurufen. Java bietet dafür die Möglichkeit, über die Methode „exec“ innerhalb der Klasse „java.lang.Runtime“ ein beliebiges

```
public void uploadContent(byte[] content,
  String fileName) throws IOException {

   File uploadedFile = uploadContentToFile(content, fileName);

   Runtime.getRuntime().exec(new String[] {
     “chown”,
     “test:test”,
     uploadedFile.getAbsolutePath()
   });

  }
```

Listing 1: Datei-Upload

externes Programm zu starten. Auch hier lauern jedoch Fallen und Probleme, die nicht unbedingt auf den ersten Blick erkennbar sind. Zwei davon werden nachfolgend näher betrachtet, um die hinter „Runtime#exec“ stehende Funktionalität zu verdeutlichen. In einem realen Projekt des Autors fand sich der folgende Code-Schnipsel (siehe Listing 1).

Hier wird ein von irgendwoher bezogenes Byte-Array in eine lokale Datei geschrieben. Anschließend soll diese Datei auf Betriebssystemebene (hier auf einem Solaris-System) dem Benutzer „test“ innerhalb der Gruppe „test“ zugewiesen werden. Da dies in dem Projekt-Szenario unter Java SE 6 nicht mit Java-Bordmitteln zu lösen war, kam ein externer Befehl zum Einsatz.

Der Effekt, der nun in der Praxis zu beobachten war, bestand darin, dass die Performance des Systems bei zunehmender Verwendung dieser Upload-Funktionalität extrem nachließ. Der als Versuch zur Behebung des Problems der Anwendung zusätzlich zur Verfügung gestellte Arbeitsspeicher trug jedoch nicht zur Lösung bei, sondern verschlimmerte das Problem nur noch. Nach aufwändiger Fehlersuche stellte sich heraus, dass nicht – wie zunächst vermutet – der IO-Durchsatz auf der Festplatte das Problem war, vielmehr der Aufruf des externen Programmes „chown“.

Erst durch intensive Recherche und grundlegendes Verständnis der Funktionsweise der Virtual Machine war das Problem identifiziert. Um zu verstehen, was genau in dem Szenario geschah, galt es zu verdeutlichen, was die Virtual Machine zur Ausführung des externen Programms macht: Um unter Unix-Systemen auf Betriebssystemebene einen neuen Prozess zu starten, setzt die Virtual Machine die „Fork“-Funktionalität ein. Dazu existiert folgende Beschreibung: „The fork operation creates a separate address space for the child. The child process has an exact copy of all the memory segments of the parent process, though if copy-on-write semantics are implemented actual physical memory may not be assigned.“

Im vorliegenden Fall (auf einer Solaris-Maschine unter Java 6) bedeutete dies, dass bei jedem Aufruf von „chown“ der komplette Adressspeicher des Quellprozesses (also der laufenden Java Virtual Machine, die den Application-Server beinhaltete) kopiert werden musste, bevor das – eigentlich harmlose – „chown“-Kommando gestartet werden konnte. Hiermit hatte sich auch der Effekt erklärt, dass eine Erhöhung des Arbeitsspeichers zu noch schlechterer Performance führte, da hier noch deutlich mehr kopiert werden musste.

Als Lösung für das Szenario wurde der Aufruf des „chown“-Kommandos in einen parallel neben dem Application-Server laufenden Daemon ausgelagert. Dieser konnte mit einem deutlich kleineren Speicherverbrauch gestartet werden und vermied damit die geschilderten Performance-Einbußen.

Die aktuellsten Virtual-Machine-Implementierungen nutzen inzwischen oftmals Verbesserungen, um diesen Effekt zu umgehen (etwa durch die Verwendung von „vfork“ anstatt „fork“), jedoch sollte für eine ganzheitliche Anwendungsarchitektur immer noch verstanden und darauf geachtet werden, welche Operationen die Virtual Machine im Hintergrund durchführt und welche Auswirkungen diese auf die eigene Anwendung haben können.

java.lang.Runtime#exec, Teil 2

Eine weitere Stolperfalle, die bei der Verwendung von „Runtime#exec“ auftreten kann, ist das Ignorieren der Ausgabe-Ströme „stdout“ und „stderr“, die von externen Kommandos verwendet werden, um Informationen an die aufrufende Umgebung zu transportieren. Listing 2 zeigt zunächst ein einfaches Beispiel.

Hier wird aus einer Java-Anwendung mittels „Runtime#exec“ ein externer Prozess gestartet. Anschließend wird über „waitFor“ darauf gewartet, dass dieser wie-

```
public static void main(String[] arg) {

  Process p = Runtime.getRuntime().exec(
    new String[] {
      "/bin/foo"
    }
  };

  p.wairFor();

}
```

Listing 2

```
ProcessBuilder processBuilder = new ProcessBuilder(
  "/bin/foo" // Liefert viel Output an stdout
);
processBuilder.redirectErrorStream(true);
Process process = processBuilder.start();

try(InputStream stdout = process.getInputStream()) {
  for(int d = stdout.read(); d > -1; d = stdout.read()) {
    doSomething(d); // Oder einfach ignorieren
  }
}

process.waitFor();
```

Listing 3

der endet. Bei dem Codeausschnitt kann jedoch – je nach ausgeführtem Kommando – die Situation eintreten, dass obwohl das eigentliche Kommando erfolgreich durchgeführt und beendet wurde, trotzdem der Aufruf von „waitFor" nicht terminiert und damit die Anwendung dauerhaft blockiert. Grund dafür sind die bereits erwähnten Streams „stderr" und „stdout" des externen Prozesses.

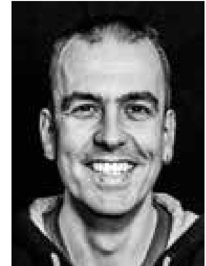
Im Javadoc von „Runtime#exec" findet sich dazu auch eine entsprechende Warnung: „Failure to promptly write the input stream or read the output stream of the subprocess may cause the subprocess to block, or even deadlock." Es sollte daher beim Aufruf von externen Kommandos immer sichergestellt sein, dass sowohl „stdout" als auch „stderr" immer komplett ausgelesen werden, bevor „waitFor" aufgerufen wird.

Zur einfachen Verarbeitung von „stdout" und „ stderr" ist anstatt von „Runtime#exec" ein „ProcessBuilder" hilfreich (siehe Listing 3). Damit wird der Puffer des Prozesses komplett ausgelesen und „waitFor" nach korrektem Ende des Prozesses nicht weiter blockieren.

Fazit

Auch und gerade in den grundlegendsten Java-Bibliotheken gilt es zu verstehen, was hinter den Kulissen abläuft. Nicht immer ist die Interaktion mit der Virtual Machine so klar und einfach, wie es auf den ersten Blick aussieht. Es lohnt sich, auch als langjähriger und erfahrener Java-Entwickler immer wieder mal nachzusehen, was dort geschieht – nicht zuletzt, um in schwierigen Projekt-Situationen dieses Wissen als Joker in den Ring werfen zu können.

Christian Robert
crobert@sapient.com



Christian Robert ist Senior Developer für mobile Lösungen bei SapientNitro in Köln. Seit mehr als zehn Jahren beschäftigt er sich mit der Konzeption und Entwicklung von Individual-Software im Java-Umfeld. Seine aktuellen Schwerpunkte liegen in der Entwicklung von pragmatischen und dennoch (oder gerade deswegen) effizienten Software-Lösungen im mobilen Umfeld. Außerdem interessiert er sich intensiv für die Ideen der Software-Craftsmanship-Bewegung. In seiner Freizeit ist er gern mit dem Fahrrad unterwegs oder genießt einen ruhigen Abend auf der Couch bei den neuesten Folgen der „Big Bang Theory".

Performance richtig bewerten

Jürgen Lampe, A:gon Solutions GmbH

Performance war und bleibt ein wichtiges Kriterium für die Software-Entwicklung. Die Fortschritte der Hardware und die Ausführung des Codes mithilfe virtueller Maschinen stellen neue Anforderungen an Verfahren zur Performance-Verbesserung. Ausgehend von einer Diskussion dieser Situation wird anhand eines sehr kleinen Code-Fragments exemplarisch ein mögliches Vorgehen demonstriert und diskutiert. Die vorgenommenen Messungen werden dargestellt und ihre Ergebnisse interpretiert, sodass ein Leitfaden für ähnliche Aufgabenstellungen entsteht.

Performance oder Geschwindigkeit ist eine der wichtigsten nichtfunktionalen Forderungen an Programme. Prinzipiell lässt sich eine höhere Leistung intensiv oder extensiv erreichen. Extensive Verbesserung erfolgt durch mehr beziehungsweise schnellere Hardware. Bei der Intensivierung wird die Steigerung durch verbesserte Nutzung des Vorhandenen erreicht.

In der Vergangenheit hat es mehrere Wechsel zwischen Perioden mit vorwiegend intensiver und solchen mit extensiver Verbesserung gegeben. In den letzten fünfundzwanzig Jahren sind die Technologiesprünge so kurz nacheinander erfolgt oder waren so groß, dass die Phase der Intensivierung nur ansatzweise erreicht wurde. Techniken und Fertigkeiten, um sparsame Programme zu entwickeln, sind deshalb aus dem Fokus geraten.

Die Hardware-Entwicklung stößt auf der bisherigen Basis an Grenzen. Kurzfristig ist nicht zu erwarten, dass es noch einmal zu einer derartig rasanten Erhöhung der reinen Verarbeitungsgeschwindigkeit kommt. Ein weiterer Grund, warum sparsamere Programme an Bedeutung gewinnen werden, ist der Energieverbrauch. Der massenhafte Einsatz hat zur Folge, dass selbst kleinste Einsparungen in der Summe ein relevantes Ausmaß erreichen können.

Es gibt also wichtige Gründe, sich mit der Performance von Java-Programmen zu befassen. Dabei soll gleich ein oft anzutreffendes Missverständnis ausgeräumt werden. Das Schreiben sparsamer und schneller Programme ist nicht die „premature optimization" aus Donald Knuths Spruch von 1974: „Premature optimization

is the root of all evil". Es geht vielmehr darum, sauber und ohne Verschwendung zu arbeiten.

Software-Produktion

Bei der Software-Produktion geht es fast immer um die Konstruktions-Effizienz. Das ist aus Herstellersicht vernünftig und bringt, solange es zu günstigen Preisen führt, auch den Kunden Vorteile. In dem Maße, wie sich das Angebot an Software-Produkten verdichtet, werden jedoch neben Funktionalität und Preis weitere Produkt-Eigenschaften für den Verkaufserfolg wichtig.

Die Annahme, dass die Effizienz von Code wichtiger wird, ist also nicht unbegründet. Natürlich gilt das nicht in gleicher Weise für jede Zeile eines Programms. Deshalb ist es wesentlich, die für das angestrebte Ziel entscheidenden Stellen zu identifizieren und jeweils angemessene Lösungen zu finden. Für Ingenieure ist es selbstverständlich, Konstruktions- beziehungsweise Entwicklungskosten gegen Gebrauchskosten abzuwägen. Die Gebrauchskosten eines Programms werden durch Laufzeit- und Speicheranforderungen bestimmt. Die Abwägung setzt jedoch voraus, dass man diese Kosten kennt. Jeder Programmierer sollte wissen, welche Aufwände einzelne Konstruktionen im Code verursachen.

Es ist deshalb erforderlich, Initiativen wie die Clean-Code-Bewegung um eine Komponente zu ergänzen, die sich in einem zweiten Schritt mit dem Material dieses Handwerks befasst, damit – um in diesem Bild zu bleiben – aus handwerklicher Fertigkeit und passender Materialauswahl echte Handwerkskunst entsteht.

J. Bloch [1] hat darauf hingewiesen, dass es immer schwieriger wird, vom Code auf die Performance zu schließen, weil sich der Abstand zwischen Quellcode und ausgeführten Instruktionen ständig vergrößert. Das hat auch eine gute Seite: Trickreiche und kryptische Optimierungen, auf die Knuth mit der zitierten Bemerkung zielte, sind weitgehend wirkungslos oder sogar kontraproduktiv. Einfacher wird das Schreiben von Code trotzdem nicht. Das ständige Abwägen von Varianten bleibt eine wichtige Aufgabe.

Regeln und Einzelfälle

Dieser Artikel zeigt an einem konkreten Beispiel, wie man sauberen und sparsamen Code schreiben kann. Das Ziel ist dabei nicht, die einzig richtige Lösung zu präsentieren. Die Lösung für alle Fälle gibt es meistens nicht – und wenn, ist ihre Darstellung nicht sonderlich interessant, weil man sie dann ja nur noch kopieren muss. Viel wichtiger ist der Weg, wie eine Lösung gefunden wird und welche Einflussfaktoren und Kriterien bei jedem Schritt berücksichtigt werden sollten.

Gerade Regeln, die anscheinend universell sind, verleiten dazu, die notwendigen Voraussetzungen aus den Augen zu verlieren. Es ist leichter und befreit von Verantwortung, sich auf eine allgemeine Regel zu berufen. Echte Meisterschaft zeigt sich allerdings darin zu wissen, wann man im Interesse eines übergeordneten Ziels gegen eine Regel verstoßen oder von einem Pattern abweichen muss.

Ein anderer wichtiger Aspekt ist die zeitliche Dimension von Regeln. Die Entwicklung sowohl der Hard- als auch der Software verläuft sehr schnell. Regeln, die für eine Sprachversion gültig waren, können mit dem nächsten Release bereits obsolet sein. Bei der Abwägung, ob einer bestimmten Regel gefolgt werden soll oder nicht, muss deshalb auch bedacht werden, wie wahrscheinlich es ist, dass diese Regel durch zukünftige Entwicklungen entwertet wird. Bei Sprachen, die wie Java in den Code einer virtuellen Maschine (VM) übersetzt werden, ist auch die Entwicklung der VM zu berücksichtigen.

Jede Optimierung beginnt mit der Frage: Wo und wodurch entstehen die Kosten? Hier heißt es, Sprachkonstrukte zu finden, die das Laufzeitverhalten negativ beeinflussen können. Wir beschränken uns auf drei wichtige Punkte, nämlich Objekt-Erzeugung, Datenbereichs-Kopien und den Aufruf virtueller Methoden.

Die Objekt-Erzeugung

Wenn ein Objekt neu erzeugt wird, muss zunächst der erforderliche Speicherplatz auf dem Heap allokiert werden. Danach wird die Objektinitialisierung ausgeführt, je nach Tiefe der Klassen-Hierarchie sind dazu mehrere Initialisierer aufzurufen.

Nicht unmittelbar sichtbar verursacht jedes Objekt weitere Aufwände, die gewöhnlich pauschal dem Garbage Collector (GC) zugerechnet werden. Auf das einzelne Objekt bezogen sind diese Kosten relativ gering. Wenn in (Hoch-)Lastsituationen jedoch so große Mengen von Objekten erzeugt werden, dass der GC aktiv werden muss, hat das erheblichen Einfluss auf die Gesamtleistung des Systems.

```
protected String getAttributeName(String name) {
  if (name.length() > 0) {
    if (name.length() > 1) {
      name = String.valueOf(name.charAt(0)).toLowerCase()
             + name.substring(1);
    } else {
      name = name.toLowerCase();
    }
  }
  return name;
}
```

Listing 1

```
   final char firstChar = name.charAt(0);
   char firstCharLower = Character.toLowerCase(firstChar);
   if (firstChar != firstCharLower) {
        // es gibt etwas zu tun
   } // sonst: fertig
```

Listing 2

Neuere Versionen der Java-HotSpot-VM versuchen, dieses Verhalten zu verbessern, indem durch ein „Escape-Analyse" genanntes Verfahren kurzlebige Objekte identifiziert und, wenn möglich, im Aufruf-Stack statt im Heap angelegt werden. Das Maß der Schwierigkeiten jedoch, die bei jedem Versuch, die Speicherverwaltung nachhaltig zu verbessern, zu bewältigen sind, lässt sich an der Diskussion um den seit dem Jahr 2006 entwickelten G1-Kollektor ablesen.

Kopieren von Datenbereichen

Aufgrund der Struktur der Hardware können größere Speicherbereiche nur durch Zerlegung in kleinere Elemente sequenziell gelesen und geschrieben werden. Die maximale Größe der Elemente ist durch den Speicherbus begrenzt. Unabhängig von Caches und Busbreiten ist das Kopieren von Datenbereichen eine Operation, deren Kosten ab einem bestimmten Schwellwert proportional zur Größe des Datenbereichs sind.

Im Java-Code treten diese Operationen direkt als explizite Schleifen oder durch Aufruf von „system.arraycopy" auf. Implizit kommen sie in allen dynamisch wachsenden Behälterobjekten vor: „StringBuffer", „ArrayList", „HashSet" etc.

Aufruf virtueller Methoden

Methoden-Aufrufe (außer von statischen Methoden) sind in Java immer virtuell. Das bedeutet, dass die anzuspringende Adresse im Code zum Zeitpunkt der Übersetzung unbekannt ist, weil sie von der konkreten Klasse des jeweiligen Objekts abhängt. Zur Laufzeit legt die VM für jede geladene Klasse eine Liste der implementierten Methoden einschließlich ihrer Einsprung-Adressen an. Über den tatsächlichen Typ eines Objekts kann dann auf diese Liste zugegriffen und die Adresse der auszuführenden Methode ermittelt werden. Im Fall einer abgeleiteten Klasse kann es vorkommen, dass die gesuchte Methode nicht überschrieben worden ist. Dann muss die Suche in der Liste des Supertyps gegebenenfalls rekursiv wiederholt werden. Dieses aufwändige „Dynamic Dispatching" wird bei folgendem Aufruf nicht verwendet:

- In der gleichen Klasse definierte Methoden
- Als „final" deklarierte Methoden
- Methoden, die in als „final" gekennzeichneten Klassen definiert sind

Für einige der verbleibenden Fälle versucht die VM, diesen Aufwand durch dynamische Analysen und Caching zu vermindern. Tief geschachtelte Klassen-Hierarchien machen also nicht nur den Code schwerer verständlich, sondern wirken sich nachteilig auf die Laufzeit aus.

Das Beispiel

Der Beispielcode entstammt einem Projekt, das XML-Daten erzeugt und wieder einliest. Dabei sind Objekt-Attribute der Form „attribute" als XML-Elemente „<Attribute></Attribute>" dargestellt. Um aus den Tags wieder den Attribut-Namen gewinnen zu können, muss das erste Zeichen in einen Kleinbuchstaben konvertiert werden. Dazu kommt die Methode „getAttributeName" zum Einsatz (siehe Listing 1). Beim Profiling war diese Methode als einer der Hotspots ermittelt worden.

Schritt 1: Analyse

Im ersten Schritt geht es darum, mögliche Ansatzpunkte für Verbesserungen zu finden. Der erste Blick gilt dabei der String-Erzeugung. Insgesamt werden pro Konvertierung vier (falls das erste Zeichen ein Kleinbuchstabe oder ein nicht zu konvertierendes Zeichen ist: drei) String-Objekte und ein „StringBuilder"-Objekt angelegt. Damit ist der Ansatz für die Optimierung klar:

1. Die Anzahl der Zwischen- beziehungsweise Hilfsobjekte verringern
2. Den Fall, dass nichts gemacht werden muss, identifizieren
3. Der Versuch, für die Zeichen von A bis Z den Aufruf von „toLowerCase" vermeiden

Schritt 2: Testfälle

Refactoring setzt eine ausreichende Testabdeckung zwingend voraus. Bei der Optimierung wird sehr oft noch ein Schritt weiter gegangen als beim Refactoring, indem man bewusst weitere Einschränkungen in Kauf nimmt, in diesem Fall die Beschränkung auf Einzelzeichen-weise Konvertierbarkeit zwischen Groß- und Kleinbuchstaben. Entsprechende Testfälle sind zu ergänzen.

Schritt 3: Implementierung

Nicht die optimale Lösung, sondern eine ausreichend gute ist gefragt. Zuerst ist zu prüfen, ob der übergebene String überhaupt verändert werden muss, ob also das erste Zeichen nicht bereits ein Kleinbuchstabe ist. Die „Character"-Klasse offeriert mit „isLowerCase" eine geeignete Methode. Allerdings wird das übergebene Zeichen darin bereits einmal konvertiert. Wenn diese Methode „false" liefert, also etwas zu tun ist, müsste die Konvertierung ein zweites Mal durchgeführt werden. Daher ist es besser, nur eine Konvertierung auszuführen und deren Ergebnis mit dem ursprünglichen Wert zu vergleichen (siehe Listing 2).

Noch wichtiger ist es, die Anzahl der erzeugten Objekte zu vermindern. Es liegt eine modifizierbare „String"-Repräsentation durch ein Feld von Zeichen vor. Darin wird das erste Zeichen durch seinen konvertierten Wert ersetzt. Danach verwendet man das modifizierte Feld für die Konstruktion des Ergebnis-Strings (siehe Listing 3).

Zum Vergleich wird noch eine weitere Variante besprochen. Fast immer sind die Großbuchstaben von A bis Z zu konvertieren. Diese können durch einen einfachen Ausdruck in Kleinbuchstaben umgewandelt werden: „firstCharLower = (char) (firstChar + ('a' - 'A'));". Damit sind die Möglichkeiten zur Optimierung der untersuchten Methode umrissen. Die folgenden Varianten werden als „getAttributName1" bis „getAttributName5" (Anmerkung: Das Listing kann unter http://www.ijug.eu/fileadmin/Daten/listings/2013_Java_aktuell_Listing_Lampe.txt heruntergeladen werden) genauer analysiert:

```
char[] chars = name.toCharArray();
// bzw. name.getChars(0, length, chars, 0);
chars[0] = firstCharLower;
name = new String(chars, 0, length);
```

Listing 3

1. Jeweils neu allokiertes Zeichenfeld, keine Sonderbehandlung von Zeichen
2. Globales Zeichenfeld, keine Sonderbehandlung von Zeichen
3. Jeweils neu allokiertes Zeichenfeld, Sonderbehandlung der Zeichen von A bis Z
4. Globales Zeichenfeld, Sonderbehandlung der Zeichen von A bis Z
5. Wie 4, redundanter Check entfernt

Schritt 4: Laufzeit-Messung

Die wichtigste Grundregel für Code-Optimierungen lautet: „Messen – nicht spekulieren!" Das ist sehr viel leichter gesagt als getan. Bei der Laufzeit-Messung für einzelne Methoden oder andere Code-Abschnitte – auch „Micro-Benchmarking" genannt – lauern viele Fallen [2]. Frameworks können helfen, wenigstens die gröbsten Fehler zu vermeiden. Hier kommt das „caliper"-Framework von Google [3] zum Einsatz. Die Mess-Szenarien für diese Untersuchung sind sechs unterschiedliche Strings. Die ersten beiden erfordern keine Veränderung. Es folgen drei Strings, die den Normalfall mit zunehmender Länge darstellen. Abschließend noch ein Beispiel für eine Umwandlung außerhalb des gängigen Zeichenbereichs. Als Hardware wird ein PC mit Intel 7-2600 CPU/3.40 GHz, 16 GB Hauptspeicher unter 64-Bit-Windows 7 verwendet.

Abbildung 1 zeigt die Ergebnisse mit Oracles Java 6 Runtime Environment. Die Verbesserung gegenüber der ursprünglichen Implementierung ist klar zu erkennen, während die Unterschiede zwischen den Varianten klein sind. Auffallend ist, dass sich die Verwendung des globalen Zeichenfeldes (Varianten 3, 4, 5) sichtbar auswirkt.

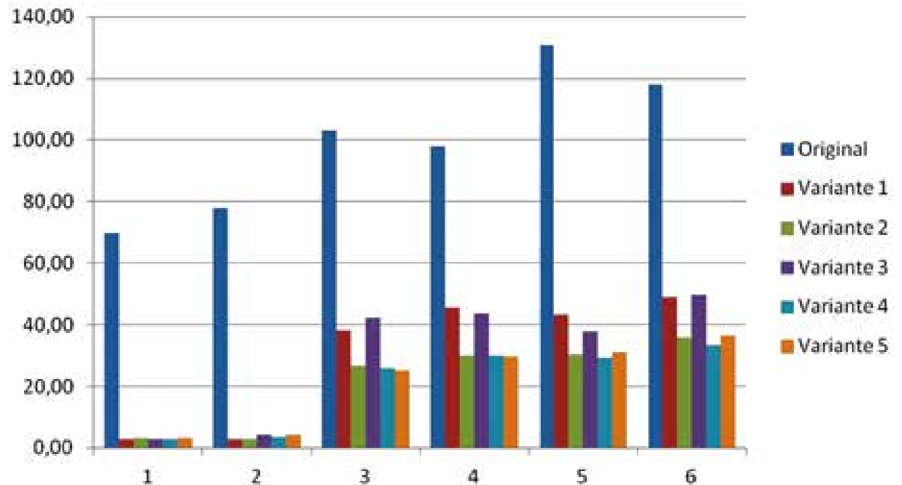


Abbildung 1: Laufzeiten in Millisekunden mit JRE 6 und HotSpot-VM

Schritt 5: Speicherverbrauch-Messung

Für die Messung des Speicherverbrauchs stellt „caliper" eine Erweiterung bereit. Die damit gewonnenen Werte sind in Abbildung 2 dargestellt. Alle Varianten verbrauchen keinen Speicher, wenn nichts verändert werden muss. Erkennbar ist der pro Aufruf erforderliche Puffer für die Varianten eins und drei.

Neben der Größe des belegten Heap-Bereichs ist die Anzahl der erzeugten Objekte interessant, weil sie ein direktes Maß für den Initialisierungsaufwand darstellt. In Abbildung 3 sieht man die Anzahl der pro Aufruf erzeugten Objekte. Die Originalversion benötigt für die Bearbeitung des fünften Strings noch ein zusätzliches Objekt durch den implizit benutzten „StringBuilder", dessen initiale Kapazität von 16 Zeichen nicht ausreichend ist.

Schritt 6: Zwischenbilanz

Nach dieser Analyse im Kleinen muss eine Variante für die Erprobung ausgewählt werden. Die Klarheit oder Lesbarkeit des Codes ist dabei nicht völlig zu vernachlässigen. Aus dieser Perspektive kann man auf die Zeichen-Sonderbehandlung verzichten und zwischen den Varianten eins und zwei wählen. Unter Java 7, wo der zusätzliche Speicherbedarf durch die Escape-Analyse obsolet wird, ist die Variante eins die Wahl. In den anderen Umgebungen spricht einiges für die Variante zwei, weil sie den Heap entlastet.

Es ist auch angebracht, sich über die Wechselwirkungen der vorgesehenen Änderungen mit anderen Teilen der Anwendung Gedanken zu machen. Vorsicht ist geboten, wenn kurzlebige Objekte durch solche ersetzt werden, deren Lebenszeit gerade einmal wenige GC-Läufe überdauert.

Schritt 7: Ergebnis überprüfen

Wenn die Zwischenbilanz zu einem vorläufigen Ergebnis geführt hat, ist es an der Zeit, die Modifikation in ihrem echten Umfeld zu erproben. Optimal wäre es, wenn die Ergebnisse im Rahmen eines produktionsnahen Lasttests überprüft werden könnten.

Falls die Ergebnisse positiv ausfallen, steht anschließend die Übernahme in die

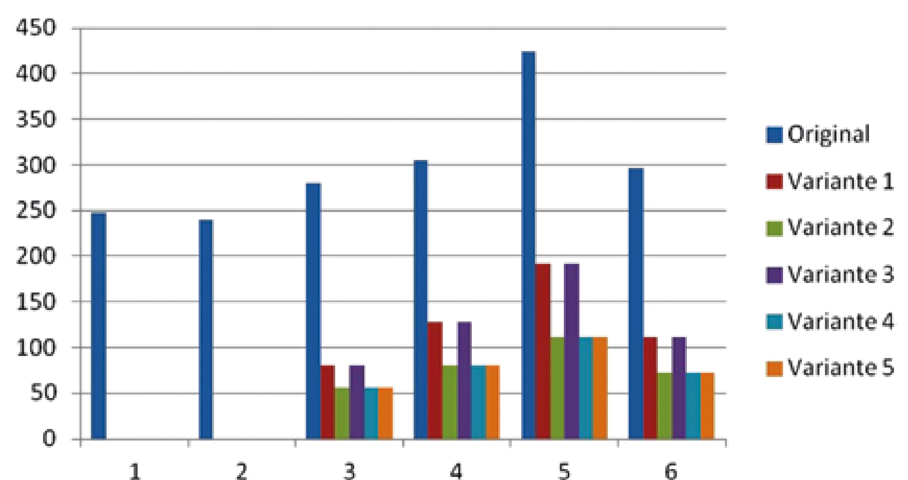


Abbildung 2: Speicherverbrauch pro Aufruf in Byte mit JRE 6 und HotSpot-VM

Produktion an. Selbst bei guter Vorbereitung sind dabei Überraschungen nicht ausgeschlossen. Mit den Worten von J. Bloch: „Die Performance von Programmen ist nicht länger vorhersehbar, und sie wird es aller Wahrscheinlichkeit nach auch nie wieder sein.“ [1, Übersetzung durch den Autor]. Das heißt letztlich aber auch, dass prinzipiell jede Änderung am Gesamtsystem, ganz gleich ob zusätzliche Hardware, Software-Updates oder geändertes Nutzerverhalten, unvorhersehbare Konsequenzen haben kann. Die Überprüfung des Ergebnisses ist mit der Produktivnahme nicht abgeschlossen, sondern sollte in ein ständiges Performance-Monitoring münden.

Die Ergebnisse des Tests können natürlich auch negativ oder indifferent ausfallen. Wirklich schwere Fälle, die eine sofortige Rücknahme der Änderungen erfordern, sind nicht zu erwarten und würden auf viel grundlegendere Defizite hinweisen. In den anderen Fällen muss man versuchen, die Ursachen für das unerwartete Verhalten einzugrenzen. Zuerst ist zu klären, ob der geänderte Code tatsächlich die Ursache des beobachteten Verhaltens ist. Gar nicht so selten kommt es nämlich vor, dass Ineffizienzen an einer Stelle solche an anderen verdecken. Dann war die Arbeit zwar erfolgreich, aber ohne das gesetzte Ziel zu erreichen. Immerhin braucht man nichts zurückzunehmen und kann darauf hoffen, dass sich der Aufwand später doch noch bezahlt macht.

Wenn der geänderte Code in der Produktion keinen spürbaren Fortschritt bringt, muss der hier beschriebene Prozess mit Schritt eins und der Suche nach einer neuen Hypothese von vorn begonnen werden. Eventuell kann es sich lohnen, die bei der Zwischenbilanz im Schritt sechs aussortierten Varianten nochmals kritisch zu bewerten und auf dieser Basis einen neuen Versuch zu starten. Normalerweise sollte dieser kleine Zyklus aber extrem scharfen Optimierungsaufgaben, die bereits an Tuning heranreichen, vorbehalten bleiben.

Im betrachteten Beispiel wurden für die ungefähr 5,5 Millionen Aufrufe in einer rund zwei Minuten dauernden Testsession unter JRE 6 mehr als 38 Millionen Objekte und knapp 40 MB Speicherplatz eingespart und die Laufzeit insgesamt spürbar gesenkt.

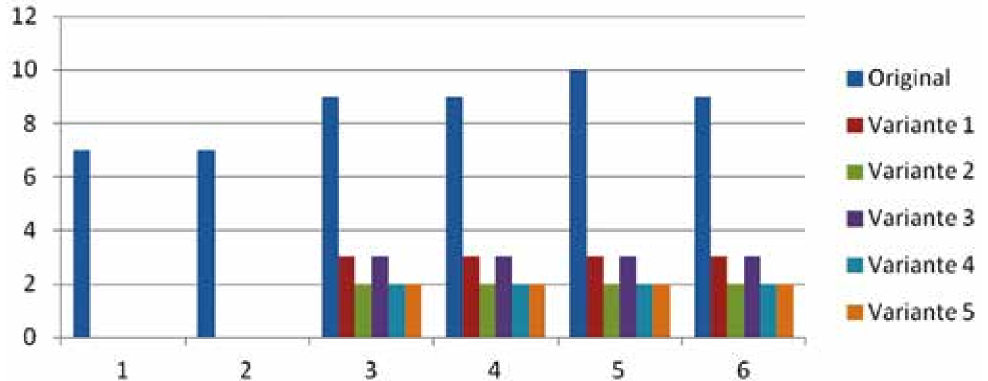


Abbildung 3: Anzahl der erzeugten Objekte mit JRE 6 und HotSpot-VM

Schlussfolgerungen

Trotz aller Fortschritte bei der Entwicklung von selbst-optimierenden Prozessoren und virtuellen Maschinen hat die Programmierung erheblichen Einfluss auf die Performance. Die wichtigste Voraussetzung, die Auswahl geeigneter Algorithmen, wird hier nicht betrachtet. Eine zentrale Schlussfolgerung lautet, dass es keinen Königsweg zu guter Performance gibt. Wegen der beschränkten Vorhersagbarkeit wird Performance-Engineering als Teil des Refactoring zukünftig eine wichtige Rolle spielen.

Eine der wichtigsten, aber derzeit auch schwierigsten Aufgaben des Performance-Engineering ist die Identifikation der kritischen Code-Bereiche. Dazu ist Erfahrung unverzichtbar. Deshalb sollte jeder Entwickler alle Gelegenheiten nutzen, solche Erfahrungen zu sammeln, indem beispielsweise verschiedene Varianten implementiert und in ihren Auswirkungen verglichen werden.

Fazit

Handwerklich sauber geschriebener Code lässt Performance-Eigenschaften nicht außer Acht. Seit der Entstehung von Java ist der Abstand zwischen dem Quellcode und den tatsächlich auf der Hardware ausgeführten Instruktionen stark gewachsen. Moderne CPU verwenden mehrere Cache-Ebenen, Parallelisierung oder spekulative Vorbereitung von Befehlen. Die Zeit für einzelne Operationen hängt dadurch erheblich vom jeweiligen (zufälligen) Ausführungskontext ab. Deshalb gibt es kein hinreichend beschreibbares Performance-Modell mehr. Der Hauptteil des Beitrags demonstriert in sieben Schritten, wie unter diesen Bedingungen eine zielstrebige Verbesserung kritischer Codeabschnitte durchgeführt werden kann. Großer Wert wird darauf gelegt zu zeigen, dass dieser Prozess wesentlich durch abgewogene Entscheidungen der Entwickler geprägt wird. Hinweis: Unter [4] findet sich eine ausführlichere Darstellung mit weiteren Messungen auch unter Java 7 und JRockit.

Quellen

[1] J. Bloch: Performance Anxiety, Devoxx 2010, http://www.parleys.com/#st=5&id=2103&sl=15
[2] http://code.google.com/p/caliper/wiki/JavaMicrobenchmarks
[3] http://code.google.com/p/caliper/
[4] http://www.agon-solutions.de/downloads

Jürgen Lampe
juergen.lampe@agons-solutions.de

Dr. Jürgen Lampe ist IT-Berater bei der A:gon Solutions GmbH in Frankfurt. Seit mehr als fünfzehn Jahren befasst er sich mit Design und Implementierung von Java-Anwendungen im Banken-Umfeld. In dieser Zeit hat er sich immer wieder intensiv mit Performance-Problemen beschäftigt. An Fachsprachen (DSL) und Werkzeugen für deren Implementierung ist er seit seiner Studienzeit interessiert.

HTML5 und Java-Technologien

Peter Doschkinow, ORACLE Deutschland B.V. & Co. KG

HTML5 ermöglicht die Entwicklung moderner und portabler Web-Anwendungen für Desktops und verschiedene Endgeräte, die den neuen Anforderungen wie Offline-Betrieb und Real-Time-Kommunikation gerecht werden. Der Artikel zeigt, welche Java-Technologien für die Entwicklung von HTML5-Business-Anwendungen zum Einsatz kommen und welche Änderungen in der Anwendungsarchitektur dabei notwendig sind. In einem Beispiel, bei dem ein JavaFX-Client hybriden Java/JavaScript-Code verwendet, wird vorgeführt, wie nahtlos HTML5- und Standard-Java-API in End-to-End-Web-Anwendungen zusammen genutzt werden können.

Durch die immer bessere Unterstützung von HTML5, CSS3 und schnellen JavaScript-Engines haben sich moderne Browser zu einer leistungsstarken Anwendungs-Plattform entwickelt. HTML5 standardisiert ein neues semantisches Mark-up, das anstelle der generischen „<div>“- und „<span>“-Elemente zur besseren Strukturierung von HTML-Dokumenten beiträgt, sowie eine Fülle clientseitiger JavaScript-APIs. Sie eröffnen dem Entwickler umfangreiche Möglichkeiten, auf die nativen Funktionen moderner Endgeräte zuzugreifen und über verschiedene Protokolle mit Servern zu kommunizieren.

CSS3 bietet ein breites Spektrum an Layout und Styles, animierte Modifikationen bei der Darstellung des Document Object Model (DOM) und die dynamische Anwendung ganzer CSS-Sektionen, abhängig von der Bildschirmgröße und dem Gerätetyp des Clients. Die Evolution von JavaScript in Kombination mit mächtigen JavaScript-Frameworks wie JQuery, AngularJS und Knockout haben schließlich die Erstellung von komplexen Client-Anwendungen mit ansprechenden Benutzeroberflächen durch dynamische DOM-Manipulationen, eigenständiger Business-Logik und effizienter Backend-Anbindung deutlich vereinfacht.

Ende des Jahres 2012 hat HTML5 den Status einer W3C „Candidate Recommendation“ erreicht. Damit eine W3C-Spezifikation als „Recommendation“ anerkannt wird, muss es davon zwei hundertprozentig vollständige und interoperable Implementierungen geben. Dieses harte Kriterium erklärt, warum HTML5 dafür prädestiniert ist, eine sehr hohe Portabilität von kompatiblen Anwendungen auf HTML5-konformen Browsern zu gewährleisten [1]. Abbildung 1 gibt eine gute Vorstellung über den breiten Funktionsumfang des HTML5-JavaScript-API, angeordnet nach den Schichten einer typischen Anwendungs-Architektur. Daraus ist ersichtlich, dass HTML5 durch die Unterstützung von Offline-Web-Anwendungen und aller gängigen nativen Device-Funktionen auch hervorragend für den Einsatz auf mobilen Endgeräten aufgestellt ist.

HTML5-Anwendungs-Architektur

HTML5-Anwendungen haben gemeinsame Charakteristiken. Der Browser ist die Anwendungs-Plattform und die Anwendung selbst setzt sich typischerweise aus HTML5-Markup, JavaScript-Code, CSS3-Style-Sheets und Server-Ressourcen zusammen: „HTML5-Anwendung = UI (HTML5 + JavaScript + CSS3) + Server-Ressourcen".

Die einfache Implementierung und Bereitstellung der Server-Ressourcen ist genau die Stärke der bereits standardisierten Java-EE-7-Technologien, die in Kürze vorgestellt werden. Die Benutzeroberfläche und alle ihre Bestandteile (Model, View und Controller) laufen komplett auf dem Client, im Unterschied zu klassischen Java-EE-Anwendungen, die serverseitige Präsentations-Frameworks wie JSP, Struts oder JSF nutzen.

Die Rolle des Servers verschiebt sich von der Unterbringung der Logik des Anwendungs-Controllers und der Aufbereitung der Anwendungsseiten hin zur Bereitstellung von Daten-Services, die oft Business-Logik enthalten und die über REST-Schnittstellen und Standard-Protokolle wie HTTP, WebSocket und Server-Sent Events (SSE) unterschiedlichsten Clients (nicht nur Browser-basierten) zugänglich gemacht werden. Diese neue Architektur, bekannt unter dem Namen „Thin Server Architecture" (TSA), scheint am besten für HTML5- (und auch für Flash-, Silverlight- oder JavaFX-basierte RIA-) Anwendungen geeignet zu sein, die Backend-Dienste benötigen [2] (siehe Abbildung 2).

Die statischen Ressourcen der Anwendung (HTML, JavaScript, CSS) werden im ersten HTTP-Request heruntergeladen. Danach kümmert sich der JavaScript-Controller um den weiteren Ablauf, die Anzeige oder Verdeckung bestimmter DOM-Bereiche und den Zugriff auf die Backend-Dienste. Andere bekannte Variationen der TSA sind Service-Oriented Front-End Architecture (SOFEA) und Single Page Application (SPA). Zu den Vorteilen der TSA im Vergleich zu serverseitigen Presentation-Frameworks gehören:

- Bessere Performance, da Präsentationsdaten nicht immer wieder erneut übertragen werden
- Bessere Skalierbarkeit, weil der User-Interface-Zustand auf dem Client bleibt
- Geringere Komplexität, da die UI-Kontrolle nicht zwischen Client und Server verteilt ist und User-Interface-Events auf dem Client bleiben
- Offline-Anwendungen, die auf sporadische Kommunikation mit Servern an-

Abbildung 1: Einordnung des HTML5-JavaScript-API nach Architektur-Schichten

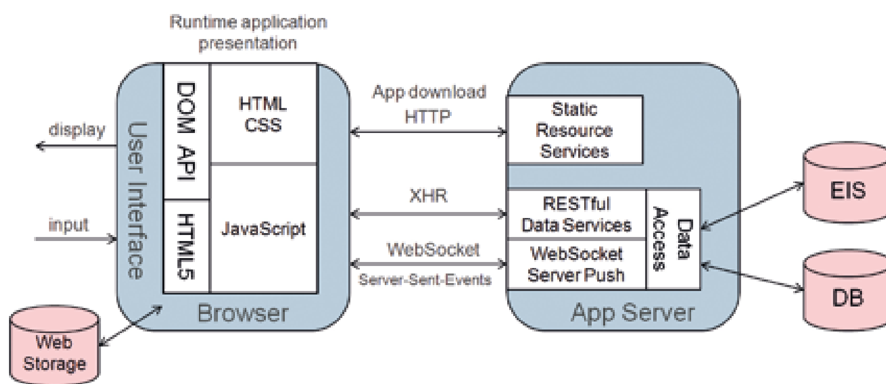


Abbildung 2: Thin Server Architecture (TSA) für HTML5-Anwendungen

```
@ServerEndpoint("/chat")
public class ChatBean {
    static Set<Session> peers = Collections.synchronized-
Set(...);
    @OnOpen
    public void onOpen(Session peer) {
        peers.add(peer);
    }
    . . .
    @OnMessage
    public void message(String message) {
        for (Session peer : peers) {
            peer.getBasicRemote().sendText(message);
        }
    }
}
```

Listing 1: Beispiel für einen serverseitigen WebSocket-Endpoint

gewiesen sind, lassen sich nur mit TSA realisieren

Serverseitige Java-Technologien für HTML5-Anwendungen

Die serverseitige Unterstützung von HTML5-Anwendungen ist der bedeutendste Schwerpunkt der neuen Java-EE-7-Spezifikation. Die einfache, effiziente und standardisierte Exposition von Server-Ressourcen liegt im Fokus des neuen beziehungsweise aktualisierten WebSocket-, JSON-, JAX-RS- und Servlet-API.

Das WebSocket-API ermöglicht die Nutzung von WebSockets, die ein Teil der HTML5-Spezifikationen sind und die Erstellung einer neuen, dynamischeren Klasse von Web-Anwendungen mit Server-Push-Fähigkeiten vereinfachen. Es ist ein neuer Grundbaustein in Java EE, ähnlich wie das Servlet-API, nur statt auf das HTTP-Protokoll auf die bidirektionale Kommunikation des WebSocket-Protokolls ausgerichtet. Mit WebSocket-Annotations lässt sich ein POJO leicht in einen WebSocket-Endpoint umwandeln (siehe Listing 1).

Mit dem neuen JSON-API sind Abhängigkeiten von Third-Party-JSON-Bibliotheken in Java-EE-Anwendungen überflüssig. Mit ihm lassen sich Java-Objekte auf ihre JSON-Darstellung abbilden, was für ihre Serialisierung/De-Serialisierung bei der Client-Server-Kommunikation notwendig ist, insbesondere wenn auf der Client-Seite HTML5/JavaScript zum Einsatz kommt. Das JSON-API besteht aus einem Low-Level-Streaming-API, ähnlich zum StAX-API in der XML-Welt, und einem einfach zu verwendenden Object-Model-API, das dem bekannten DOM-API für XML-Dokumente entspricht. Das JSON-API wird sehr häufig in Web-Anwendungen genutzt, die ihre Services über WebSockets oder JAX-RS exponieren.

JAX-RS ist ein Annotation-basiertes Framework, das im Wesentlichen HTTP-Requests auf Java-Methoden-Aufrufe abbildet und als eine Art „Domain Specific Language" (DSL) für den Umgang mit dem HTTP-Protokoll betrachtet werden kann. Mit JAX-RS ist es sehr einfach, durch die Verwendung weniger Annotationen in Java implementierte Anwendungsdienste als REST Web Services zu exponieren.

In JAX-RS 2.0, das Bestandteil von Java EE 7 ist, wird der Umgang mit Hypermedia vereinfacht, es gibt Unterstützung für die asynchrone Verarbeitung von Requests, ein Filter- und Interceptor-Framework, ein standardisiertes Client-API und die Möglichkeit der deklarativen Validierung der Request/Response-Daten durch die Bean-Validation-Spezifikation.

Eine Alternative zu WebSockets sind Server-Sent Events als Server-Push-Mechanismus, bei dem jedoch die Kommunikation über eine persistente HTTP-Verbindung abgewickelt wird und unidirektional ist. Sobald neue interessante Ereignisse auf dem Server stattfinden, schickt er die Event-Daten als „chunks" zum Client über die vom Client geöffnete HTTP-Verbindung, ohne sie zu schließen.

Clientseitig ist bereits im Rahmen von HTML5 ein JavaScript-API für den Empfang von Server-Sent Events standardisiert.

```
@Path("events")
public static class SseResource {
    @GET
    @Produces(SseFeature.SERVER_SENT_EVENTS)
    public EventOutput getServerSentEvents() {
        final EventOutput eventOutput = new EventOutput();
        // send the events using eventOutput in a separate
Thread
        …
        return eventOutput;
    }
}
```

Listing 2: Server-Sent Events mit Jersey

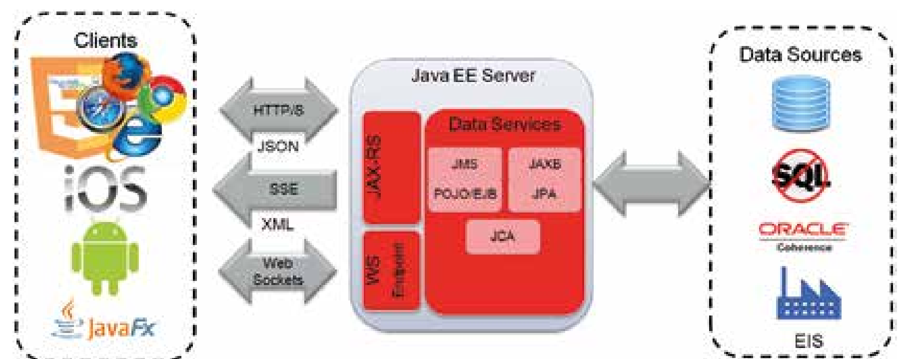


Abbildung 3: Thin Server Architecture (TSA) für HTML5-Anwendungen mit Java EE

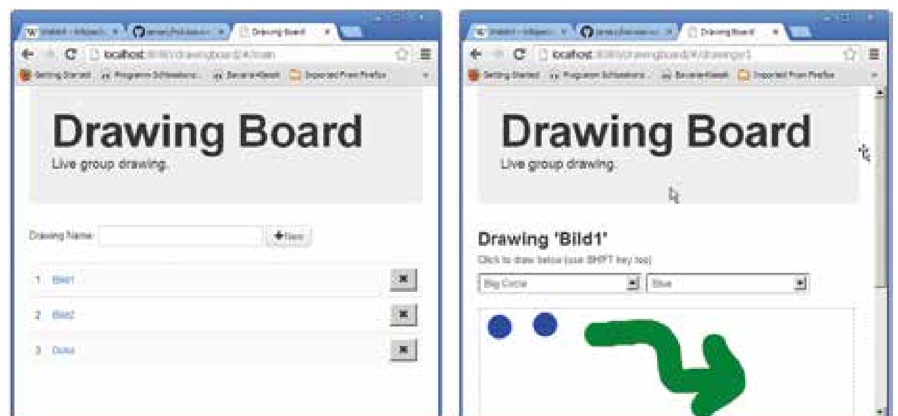


Abbildung 4: Drawing-Board-Demo

```
class DrawingsEventSource extends EventSource {
    private WebEngine engine;
    private String js_script = ... //JavaScript to redraw the list of drawings
    ...
    @Override
    public void onEvent(InboundEvent inboundEvent) {
            ...
            System.out.println("script to execute: " + js_script);
            Platform.runLater(new Runnable() {
                @Override
                public void run() {
                    engine.executeScript(js_script);
                }
            });
            ...
    }
}
```

Listing 3: Java-SSE-Client ruft JavaScript im WebView auf

Serverseitig existiert dafür noch kein Java-Standard. Eine solche Implementierung ist jedoch Jersey, die Referenz-Implementierung von JAX-RS 2.0, die über eine JAX-RS-Annotations-Erweiterung ein POJO in eine Server-Sent-Events-Ressource umwandeln kann [3] (siehe Listing 2).

Abbildung 3 zeigt eine TSA-Architektur für HTML5-Anwendungen, bei denen die Server-Seite in Java implementiert ist. Die Daten-Dienste können für ihre Implementierung das Standard-Java-EE-Programmier-Modell (ManagedBeans, EJB) und unterschiedliche Backend-Zugriffe (JMS, JPA, JCA, JAX-RS Client etc.) nutzen. Sie sind mit einem REST-Interface für klassische http- und SSE-Kommunikation sowie WebSocket-Endpoints für die WebSocket-Kommunikation mit dem Client ausgestattet. Auf der Datenleitung kommt JSON oder XML als interoperables Daten-Austauschformat zum Einsatz. Der folgende Abschnitt zeigt, warum JavaFX auch als HTML5-Client neben den bekannten Browser-Plattformen in Abbildung 3 auftaucht.

Clientseitige Java-Technologien für HTML5-Anwendungen

JavaFX besitzt in WebView [4] eine wichtige und leistungsfähige Komponente. Im Wesentlichen ist sie ein Wrapper von WebKit [5], eine Open-Source-Browser-Engine beziehungsweise der Browser-Engine-Ursprung vieler modernen Browser. WebKit enthält eine WebCore-Komponente, die das Rendering von HTML und das DOM-API implementiert, und eine JavaScript-Core-Komponente, die für die Ausführung von JavaScript zuständig ist.

JavaFX bietet somit die Möglichkeit, mit WebView einen vollwertigen Browser als JavaFX-Node im Java-Client einzubetten, um HTML5-/JavaScript-Seiten zu laden und auszuführen. Der Java-Container kann auch mit dem JavaScript-Code, der im WebView läuft, über eine bidirektionale Java/JavaScript-Schnittstelle kommunizieren und auf diese Art die Funktionalität des eingebetteten Browsers beziehungsweise der darin laufenden HTML5-Anwendung beliebig mit Java-Mitteln erweitern – ein Parade-Beispiel für eine hybride Java/JavaScript-Applikation.

```
    // mouseDown event handler
    $scope.mouseDown = function(e) {
        ...
        var msg = '{"x" : ' + posx +  ', "y" : ' + posy +
                ', "color" : "' + $scope.shapeColor +
                '", "type" : "' + $scope.shapeType + '"}';

        if (!javaFXClient)
            $scope.websocket.send(msg);
        else
            window.webSocketSend.send(msg);
    }
```

Listing 4: JavaScript im WebView ruft den Java-WebSocket-Client auf

Java-Entwicklungsumgebungen für die HTML5-Anwendungsentwicklung

Die führenden Java-Entwicklungsumgebungen Eclipse, NetBeans und IntelliJ bieten seit Langem auch Unterstützung für andere Web-Technologien wie HTML, CSS, JavaScript, PHP, Groovy, Ruby etc. an. So überrascht es nicht, dass sie im Zuge der zunehmenden Popularität von HTML5 ihren Werkzeugkasten weiter ausbauen.

Exemplarisch sind nachfolgend die neuen Features in NetBeans 7.4 zusammenge-

fasst, die unmittelbar die Entwicklung von HTML5-Anwendungen vereinfachen:

- Neuer Projekt-Typ für HTML5-Anwendungen mit integriertem JavaScript-Test-Support
- JavaScript-Editor mit Code-Completion und Unterstützung für die JavaScript-Frameworks AngularJS, Knockout und ExtJS
- JavaScript-Debugger
- Visueller Style-Sheet-Editor mit Page-Inspector und Unterstützung der Style-Sheet-Sprachen SASS und LESS
- Netzwerk-Monitor, eingebetteter Web-Kit-Browser, tiefe Integration mit Chrome, Android- und iOS-Browser
- Tiefgehender Support für Java EE 7

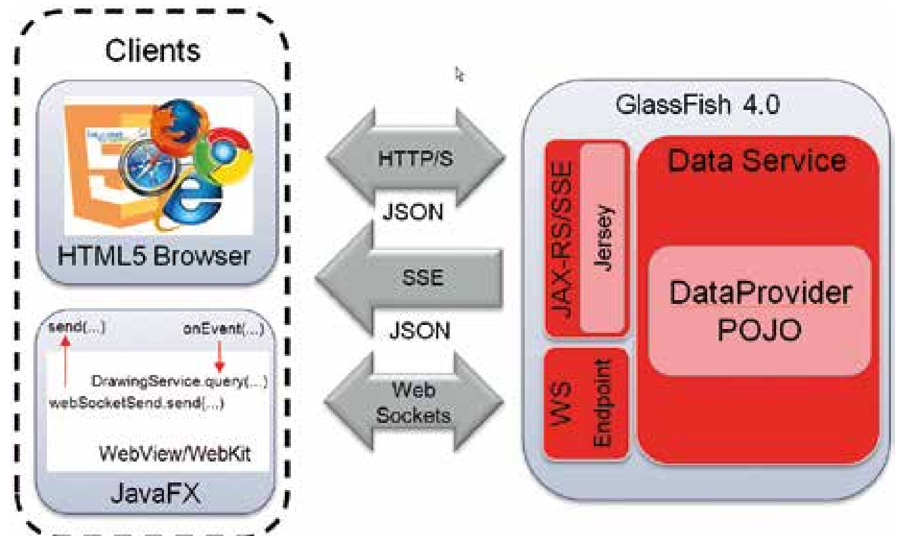


Abbildung 5: TSA-Architektur für das End-to-End Beispiel mit HTML5-Browser und JavaFX-Client

Ein End-to-End HTML5-Anwendungsbeispiel

Eine überschaubare und dennoch nicht triviale HTML5-Anwendung soll den Einsatz und das Zusammenspiel aller bisher genannten Technologien demonstrieren. Sie ermöglicht die gemeinsame Erstellung von Zeichnungsskizzen und wurde in zwei HTML5-Seiten mit AngularJS implementiert. Auf der ersten Seite kann man eine neue Zeichnung anlegen und eine bestehende löschen oder zur Zeichnung selektieren und auf der zweiten Seite dann die Skizze allein oder gleichzeitig mit allen anderen, die ausgewählt wurden, gemeinsam bearbeiten (siehe Abbildung 4). Das Beispiel verwendet folgende serverseitigen Java-Technologien:

- JAX-RS 2.0 zur Implementierung der CRUD-Operationen für die Liste der Zeichnungen
- Server-Sent Events für den sofortigen Server-Push der Modifikationen der Liste an alle angeschlossene Clients, ein Feature von Jersey, der Referenz-Implementierung von JAX-RS 2.0
- WebSocket zur Verteilung von Zeichnungsänderungen an alle Clients, die die gleiche Skizze bearbeiten
- JSON für die Implementierung von Encoder und Decoder des WebSocket-Server-Endpoint

Neben dem reinen HTML5- und AngularJS-basierten Client wurde auch ein JavaFX-Client mit derselben Funktionalität und dem gleichen Look & Feel erstellt. Er kann auch als ein nicht triviales Beispiel einer hybriden Java/JavaScript-Anwendung betrachtet werden. Die Benutzeroberfläche des HTML5-/AngularJS-Client wird wiederverwendet und läuft in einem WebView.

Da bei JavaFX 2.2 jedoch das eingebettete WebKit weder WebSocket noch SSE unterstützt, wurde diese Funktionalität mit dem Client-Java-API für WebSocket (Tyrus) und SSE (Jersey) implementiert. Entscheidend dabei ist die Nutzung der Java/JavaScript-Bridge in JavaFX, über die die bidirektionale Kommunikation zwischen dem Java-Code, der den Austausch mit dem Server abwickelt, und dem JavaScript-Code im WebView läuft. Listing 3 zeigt, wie der Java-SSE-Client den JavaScript-Code aufruft, während in Listing 4 zu sehen ist, wie JavaScript-Code die Zeichnungselemente an den Java-WebSocket-Client übermittelt.

Der komplette Quellcode dieses Beispiels sowie ein begleitender Hands-on-Guide können unter [6] heruntergeladen werden. Abbildung 5 zeigt die genutzte Architektur, die im Vergleich zu Abbildung 2 deutlich vereinfacht ist.

Fazit

HTML5 gehört neben Mobile, Grid und Cloud Computing sowie Big Data zu den großen Trends unserer Zeit, die zunehmend durch Standard-Java-Technologien unterstützt werden. Das ist kein Zufall, sondern das Ergebnis der Weitsicht und der unermüdlichen Arbeit zahlreicher JCP-Expert-Groups und Java-Community-Mitgliedern mit dem Ziel, die Java-Plattform laufend zu erneuern und ihre Relevanz auf dem Markt aufrechtzuerhalten.

Referenzen

[1] Browser-Kompatibilität mit HTML5, CSS3, SVG: http://caniuse.com/
[2] Thin Server Architecture: https://sites.google.com/a/thinserverarchitecture.com/home/Home
[3] Jersey Support für Server-Sent Events: https://jersey.java.net/documentation/latest/sse.html
[4] JavaFX WebView: http://docs.oracle.com/javafx/2/api/javafx/scene/web/WebView.html
[5] WebKit: http://en.wikipedia.org/wiki/WebKit
[6] HTML5 End-to-End Drawing Board Demo: http://github.com/jersey/hol-sse-websocket

Peter Doschkinow
peter.doschkinow@oracle.com

Peter Doschkinow arbeitet als Senior Java Architekt bei Oracle Deutschland. Er beschäftigt sich mit serverseitigen Java-Technologien und Frameworks, Web Services und Business Integration, die er in verschiedenen Kundenprojekten erfolgreich eingesetzt hat. Vor seiner Tätigkeit bei Oracle hat er wertvolle Erfahrungen als Java Architect and Consultant bei Sun Microsystems gesammelt.

Java und funktionale Programmierung – ein Widerspruch?

Kai Spichale, adesso AG

Eine wichtige Erweiterung von Java 8 sind Lambda-Ausdrücke und funktionale Schnittstellen. Lambda-Ausdrücke können eine Vielzahl der anonymen Klassen ersetzen und dank ihrer kompakteren Schreibweise die Lesbarkeit des Codes verbessern. Aber „syntactic sugar" war nicht die Haupt-Motivation für diese neuen Sprach-Features. Java holt verglichen mit anderen JVM-Sprachen kräftig auf, obwohl sich das Typ-System von Java kaum verändert hat. Der Artikel beleuchtet, welche Konsequenzen sich daraus ergeben und inwieweit funktionale Programmierung im Vergleich zu Sprachen wie Groovy und Scala möglich ist.

Das Konstrukt der anonymen Klassen dient in Java der Implementierung von namenlosen lokalen Klassen, die an der Stelle ihrer Definition einmalig instanziiert werden. Beliebt sind anonyme Klassen zur Implementierung von Schnittstellen, wie beispielsweise „ActionListener", um Ereignisse in graphischen Benutzeroberflächen zu behandeln, oder „FilenameFilter" zur Filterung von Verzeichnissen. Listing 1 zeigt ein Beispiel mit einer anonymen Klasse, die Dateien anhand ihrer Dateiendung filtert.

Mit den in Java 8 eingeführten Lambda-Ausdrücken lassen sich nun viele dieser anonymen Klassen ersetzen. Wie Listing 2 zeigt, ist der Filter mit einem Lambda-Ausdruck kompakter und leichter zu lesen, weil nicht erst umständlich eine konkrete Klasse, von der nur eine einzige Instanz gebraucht wird, definiert werden muss.

Lambda-Ausdrücke können allerdings nur in Kombination mit funktionalen Schnittstellen – zu denen „FilenameFilter" gehört – verwendet werden. Funktionale

```
FilenameFilter filter = new FilenameFilter() {
  public boolean accept(File file, String name) {
    return name.toLowerCase().endsWith(".java");
  }
};
File[] javaFiles = directory.listFiles(filter);
```

Listing 1: Implementierung der Schnittstelle „FilenameFilter" mit anonymer Klasse

Schnittstellen haben genau eine abstrakte Methode und eventuell weitere Default-Methoden, die aufgrund ihrer Standardimplementierung nicht abstrakt sind. Die Annotation „@FunctionalInterface“ führt zu Kompilierfehlern, falls sie nicht ausschließlich für Schnittstellen verwendet wird, die die beschriebenen Anforderungen an funktionale Schnittstellen erfüllen. Auf diese Weise ist sichergestellt, dass der Verwendungszweck als funktionale Schnittstelle nicht unbeabsichtigt durch Hinzufügen einer weiteren abstrakten Methode verloren geht. Die Annotation hat deswegen auch Dokumentations-Charakter.

Dank der Default-Methoden, die in Java 8 eingeführt wurden, ist es zudem auch möglich, ältere Schnittstellen des Java-API um zusätzliche Methoden zu erweitern, ohne die Kompatibilität zu existierendem Java-Code zu verlieren. So erhielt beispielsweise „Iterable“ die neue Methode „forEach“. Hätte diese keine Default-Implementierung, müsste jede Klasse diese nachträglich implementieren, um die Schnittstelle „Iterable“ zu erfüllen, was den Wechsel zu Java 8 immens erschwert hätte.

Das Typ-System

Ein Lambda-Ausdruck ist eine Art „anonyme Methode“ mit kompakter Syntax ohne Modifizierer, Rückgabe-Typ und in manchen Fällen sogar ohne Parameter-Typen. In jedem Fall handelt es sich um eine Instanz einer funktionalen Schnittstelle. Damit fügt sich dieses neue Sprach-Feature elegant in das bestehende Typ-System von Java ein.

Als Sub-Typ von „Object“ erbt ein Lambda-Ausdruck auch all dessen Methoden, besitzt aber nicht zwangsläufig eine eindeutige Identität, was der Semantik der geerbten Methode „equals“ widerspricht. Mit den vorsichtigen Änderungen in Java 8 wurde streng darauf geachtet, die langfriste Unterstützung von funktionalen Programmier-Konzepten nicht zu verbauen. Ob in späteren Java-Versionen tatsächlich vollwertige Funktionstypen folgen, wie sie Scala und Haskell bieten, bleibt abzuwarten. Als Objekte lassen sich Lambda-Ausdrücke jedenfalls von Variablen referenzieren. Fast noch wichtiger ist jedoch, dass sie als Methoden-Parameter anderen Objekten übergeben werden können.

Die Typ-Inferenz

In stark typisierten Programmiersprachen wie Java kann man an manchen Stellen auf die explizite Angabe von Typen verzichten, falls sich diese durch die verbliebene Information und die Typisierungsregeln rekonstruieren lassen. Ein Lambda-Ausdruck kennt seinen Typ nicht und ist nicht an diesen gebunden. So kann ein Ausdruck wie beispielsweise „x -> x*x“ für verschiedene funktionale Schnittstellen eingesetzt werden, denn „x“ könnte unter anderem vom Typ „double“, „long“ und „int“ sein.

Der Ziel-Typ des Lambda-Ausdrucks muss jedoch eine kompatible funktionale Schnittstelle sein. Der Lambda-Ausdruck darf also nur Exceptions werfen, die die funktionale Schnittstelle erlaubt, und die Typen der Parameter und des Rückgabewerts müssen kompatibel sein.

Der Namensraum

Lambda-Ausdrücke führen keinen neuen Gültigkeitsbereich ein, sodass die Schlüsselwörter „this“ und „super“ die gleiche Bedeutung haben – wie unmittelbar außerhalb des Lambda-Ausdrucks. Ganz allgemein haben Lambda-Ausdrücke den gleichen Zugriff auf lokale Variablen in ihrem Kontext, wie es lokale und anonyme Klassen an ihrer Stelle haben würden. Die einzige Besonderheit ist, dass Variablen-Definitionen innerhalb des Lambda-Ausdrucks keine anderen lokalen Variablen des umgebenden Gültigkeitsbereichs verdecken dürfen. Weil es einen abgeschlossenen Kontext mit Variablen gibt, der aus dem Inneren des Lambda-Ausdrucks referenziert werden kann, nennt man die Lambda-Ausdrücke auch „Closures“.

```
FilenameFilter filter = (file, name) -> name.toLowerCase().
endsWith(“.java”);
File[] javaFiles = directory.listFiles(filter);
```

Listing 2: Implementierung der funktionalen Schnittstelle mit einem Lambda-Ausdruck

Die Methoden-Referenzen

In der funktionalen Programmierung stehen nicht Objekte, sondern Funktionen im Vordergrund und repräsentieren die ausführbare Funktionalität. Diese Funktionen können nicht nur wie Methoden aufgerufen, sie können auch wie Objekte herumgereicht werden. In der funktionalen Programmierung spricht man daher auch von „Code as Data“. Passend dazu bietet Java 8 sogenannte „Methoden-Referenzen“ als weiteres Sprach-Feature. Es kann überall dort eingesetzt werden, wo ein Lambda-Ausdruck lediglich aus dem Aufruf einer einzelnen existierenden Methode besteht. Die Methoden-Referenz macht den Lambda-Ausdruck noch einfacher lesbar (siehe Listing 3).

Die Liste „jobList“ wird zunächst mit einem normalen Lambda-Ausdruck sortiert, dessen Notation schon im vorherigen Beispiel gezeigt wurde. Alternativ dazu kann als Lambda-Ausdruck auch die Methoden-Referenz „Job::compareBySalary“ zum Einsatz kommen. Hierbei handelt es sich um eine statische Methode.

Java 8 unterstützt außerdem Referenzen auf Konstruktoren und auf Instanz-Methoden. Bei Letzteren kann entweder die Methode eines bestimmten Objekts oder ein beliebiges Objekt eines bestimmten Typs verwendet werden (siehe http://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html).

Streams und Parallelisierung

Es gibt verschiedene Gründe, warum Lambda-Ausdrücke ein populäres Feature in modernen Programmiersprachen sind. Der wichtigste Grund für die Erweiterung in Java 8 war die Vereinfachung der nebenläufigen Verarbeitung von Collections. Das bisherige Verarbeitungsmuster einer Collection sah vor, dass diese einen Iterator bereitstellt, mit dem die Elemente der Collection nacheinander verarbeitet werden können. Falls die Verarbeitung in parallelisierter Form erfolgen sollte, war es nicht Aufgabe der Collection, dies zu organisieren, sondern die des aufrufenden Codes.

Mit Java 8 ändert sich die Aufgabenverteilung. Nun sind es die Collections, die Funktionen akzeptieren und diese auf die Elemente anwenden. Mithilfe der Lambda-Ausdrücke lassen sich diese Funktionen elegant definieren und übergeben. Die neue Methode „forEach“ der Schnittstelle „Itera-

```
Collections.sort(jobList, (j1, j2) -> { return Job.
compareBySalary(j1, j2); });
Collections.sort(jobList, Job::compareBySalary);
```

Listing 3: Implementierung der funktionalen Schnittstelle mit einem Lambda-Ausdruck

tor" ist ein Beispiel für die Verschiebung der Verantwortung von der Anwendung zur Bibliothek, denn die jeweilige Implementierung dieser Schnittstelle kann sich um die Verteilung auf mehrere Threads kümmern.

Die Schnittstelle „Collection" wurde ebenfalls erweitert. Sie erhielt die neuen Default-Methoden „stream" und „parallelStream" zur Erzeugung von Streams. Mit deren Hilfe lassen sich mehrere Kommandos zu einer Art Pipeline zusammensetzen.

Streams sind eine wichtige neue Abstraktion in Java 8, die nicht auf Collections beschränkt ist, denn es können beispielsweise auch Arrays und I/O-Kanäle als Datenquelle zum Einsatz kommen. Sie sind im Gegensatz zu Collections keine Daten-Strukturen zum Speichern von Daten, sondern Kanäle, die die Ausgabedaten einer Operation als Eingabedaten der folgenden Operation weiterleiten.

Zwischen-Operationen (engl. „intermediate operations") wie „limit", „map" und „filter" geben einen neuen Stream zurück. End-Operationen (engl. „terminal operations") wie „reduce" und „forEach" schließen den Verarbeitungskanal. Alle Operationen lassen sich sowohl sequenziell als auch parallel ausführen. Standardmäßig verwendet das JDK sequenzielle Streams, es sei denn, parallele werden explizit erzeugt.

Groovy

Groovy-Entwicklern sind viele der neuen Sprach-Features von Java 8 schon länger bekannt. Groovy unterstützt schon seit Jahren Lambda-Ausdrücke in Form von Closures, funktionale Schnittstellen und Methoden-Referenzen. Ein Closure entspricht in Groovy ebenfalls der Implementierung einer funktionalen Schnittstelle. Standardmäßig ist dies die „Callable"-Schnittstelle mit der „call"-Methode, aber auch andere Schnittstellen sind für Closures möglich. Die implizite „Return"-Anweisung, die bei einzeiligen Lambda-Funktionen in Java 8 üblich ist, findet man auch in Groovy.

Mit den Default-Methoden in Java 8 kann man Groovy-Features wie „Memoization" und „Tramplining" umsetzen. Memoization ist eine Technik zum Cachen von Funktions-Ergebnissen, um zur Performance-Verbesserung deren wiederholte Berechnung zu vermeiden. Tramplining erlaubt tiefe Rekursion, ohne auf Probleme mit dem Java-Call-Stack zu stoßen. Insbesondere bei der Verwendung eines funktionalen Programmierstils spielen solche rekursiven Algorithmen eine wichtige Rolle.

Das beschriebene Streaming-API von Java 8 kann Operationen an Collections ketten, so wie dies in Groovy mit „collect" und „sort" ebenfalls möglich ist. Auch Methoden-Referenzen mit etwas unterschiedlicher Syntax findet man in beiden Sprachen. Vergleicht man beide Sprachen miteinander, wird schnell deutlich, wie sehr sich Java 8 an Groovy orientiert. Und das ist gut so, denn die JVM-Sprache Groovy erfreut sich großer Beliebtheit und konnte die Praxistauglichkeit der beschriebenen Features schon unter Beweis stellen.

Scala

Scala ist eine weitere moderne JVM-Sprache, die Objektorientierung und funktionale Programmierung kombiniert. Auch hier zeigen sich im direkten Vergleich viele neue Gemeinsamkeiten. So sind Lambda-Ausdrücke in Form von Funktions-Literalen wichtige Sprach-Konstrukte in Scala. Sie können Variablen zugeordnet und als Parameter anderen Funktionen übergeben werden. Analog zum Streaming-API in Java bieten die Collections in Scala Operationen wie „filter", „map" und „sum". Da die Iterationen verborgen im Inneren der Collections erfolgen, kann dies potenziell auch nebenläufig geschehen. Seit mehreren Jahren unterstützt Scala daher die Möglichkeit, normale Collections wie Arrays und Maps mithilfe der Methode „par" in Parallel Collections umzuwandeln, sodass die Verarbeitung nebenläufig erfolgt.

Auch die Schnittstellen in Java ähneln nun den Traits in Scala. Mit deren Hilfe werden die Typen von Objekten definiert. Wie Java erlaubt auch Scala keine Mehrfach-Vererbung, aber die Verwendung mehrerer Traits ist möglich. Das Besondere an ihnen ist, dass sie neben einer Schnittstelle auch eine optionale Implementierung bieten können. Mit der Java-Terminologie könnte man sagen, dass Traits sowohl aus abstrakten als auch aus Default-Methoden bestehen. Darüber hinaus bieten sie abstrakte Felder, die in Java nicht möglich sind. Mit den neuen Features holt Java etwas auf, bietet aber keine Innovationen.

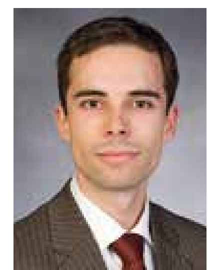
Fazit

Es wäre wohl übertrieben, Java 8 als funktionale Sprache zu bezeichnen, aber die vielen neuen Sprach-Features sind der Schlüssel zur nebenläufigen Verarbeitung von Collections beziehungsweise Streams auf modernen Multi-Core-Plattformen.

Die Lambda-Ausdrücke haben das Potenzial, die Entwicklung zukünftiger Anwendungen grundsätzlich zu verändern. Für die Einführung dieser Features war es höchste Zeit, denn andere JVM-Sprachen wie Groovy und Scala bieten diese schon lange. Mehr Konkurrenz müssen Scala & Co. nicht fürchten.

Wahrscheinlich wird das Interesse an anderen Programmiersprachen der JVM sogar noch wachsen, weil Closures und andere spannende Features mehr Aufmerksamkeit erhalten werden. Trotzdem kann man Java 8 als revolutionär bezeichnen. Die vielfältigen Möglichkeiten der Lambda-Ausdrücke ermöglichen völlig neue Algorithmen und die Umsetzung von Entwurfsmustern.

Kai Spichale
kai.spichale@adesso.de
http://spichale.blogspot.de



Kai Spichale ist Senior Software Engineer bei der adesso AG. Sein Tätigkeitsschwerpunkt liegt in der Konzeption und Implementierung von Java-basierten Software-Systemen. Er ist Autor verschiedener Fachartikel und hält regelmäßig Vorträge auf Konferenzen.

Eingebettete DSLs mit Clojure

Michael Sperber, Active Group GmbH

In letzter Zeit sind in der Software-Entwicklung domänenspezifische Sprachen – kurz als DSL für „domain-specific language" bezeichnet – populär geworden: Das Versprechen einer DSL ist es, für ein bestimmtes Anwendungsgebiet besonders kompakte und verständliche Programme zu ermöglichen. Eingebettete DSLs sind eine besondere Variante von DSLs, die in einer schon existierenden Sprache implementiert sind. Für die Realisierung solcher EDSLs eignen sich besonders gut funktionale Sprachen, von denen es auf der Java-Plattform eine erfreuliche Auswahl gibt. Dieser Artikel gibt eine Kurzeinführung in die Implementierung von EDSLs in der Programmiersprache Clojure anhand zweier Beispiele.

Das Entwickeln einer konventionellen DSL ist aufwändig, da zur Implementierung einer DSL alles dazugehört, was bei einer normalen Programmiersprachen-Implementierung ebenfalls fällig ist: Lexer, Parser, Compiler oder Interpreter sowie IDE-Support. Aus diesem Grund sind substanzielle Frameworks für die schnelle Entwicklung von DSLs entstanden, darunter Spoofax, das Eclipse Modeling Project oder Xtext.

In funktionalen Sprachen gibt es allerdings häufig einen einfacheren Weg: die Entwicklung einer eingebetteten DSL (EDSL steht für „embedded DSL"). Dabei wird eine schon existierende Host-Sprache so eingesetzt, dass es so aussieht, als ob innerhalb der Host-Sprache eine DSL entsteht. Dies hat eine Reihe von Vorteilen:

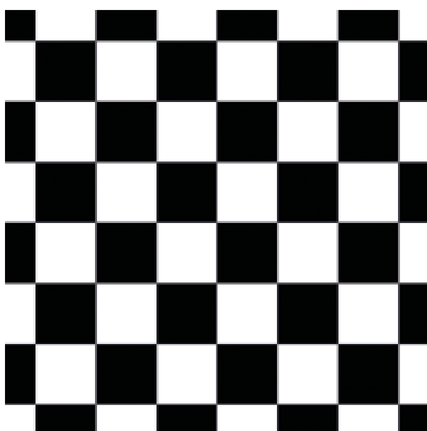


Abbildung 1: Bild mit Schachbrett-Muster, x, y = -3.5 … 3.5

- Die Entwicklung der DSL ist deutlich einfacher: Entwickler müssen kein komplettes Compiler-Frontend implementieren und können die schon bestehenden Sprachmittel der Host-Sprache für die Implementierung verwenden.
- Das Design einer Stand-Alone-DSL ist aufwändig und produziert oft suboptimale Ergebnisse. Bei EDSLs haben Entwickler einen syntaktischen und semantischen Rahmen und müssen nicht das komplette Design der Sprache stemmen.
- Benutzer können die EDSL zusammen mit der Host-Sprache verwenden.

Der EDSL-Ansatz ist prinzipiell vielen Sprachen zugänglich. In funktionalen Sprachen funktioniert er besonders gut, weil dabei die in diesen Sprachen typischen Sprachmittel wie Higher-Order-Funktionen, Makros, nicht-strikte Auswertung und Typklassen natürliche Einsatzgebiete finden.

In diesem Artikel erläutern wir den EDSL-Einsatz anhand zweier einfacher Beispiele: einer Sprache für Computergrafik-Bilder und einer für sogenannte „Stream-Prozessoren". Jede demonstriert unterschiedliche Aspekte der EDSL-Implementierung. Beide Beispiele betten wir in die funktionale Sprache Clojure ein, die eigens für die Java-Plattform entwickelt wurde und sich außerordentlich gut für die Einbettung von DSLs eignet.

Funktionale Bilder

Dieser Abschnitt orientiert sich an Conal Elliotts Pan-System für Bild-Synthese (siehe http://conal.net/Pan). Abbildung 1 zeigt einen Ausschnitt aus einem Bild, das unendlich in alle Richtungen weitergeht. Die Koordinaten sind beliebige reelle Zahlen, also keine Pixel. Wie könnte das Bild in einem Programm repräsentiert werden? Also nicht nur der Bildausschnitt, sondern das gesamte Bild. Wenn Bilder unsere Domäne sind, sollten wir eine möglichst direkte Repräsentation wählen. Ein Bild hat an jeder Stelle eine Farbe; bei Schwarzweiß reicht ein Boolean, um sie darzustellen. Wir brauchen also ein Objekt, das für gegebene Koordinaten die dortige Farbe zurückliefert. In einer funktionalen Sprache können wir dafür einfach eine Funktion nehmen. Listing 1 zeigt, wie diese in Clojure aussieht. Die dort definierte Funktion „checker" ak-

```
(defn checker
  [[x y]]
  (even? (+ (int (Math/floor x)) (int (Math/floor y)))))
```

Listing 1

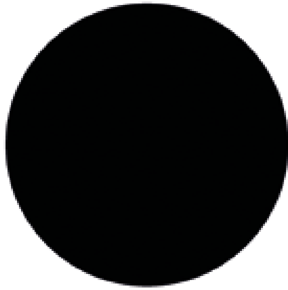


Abbildung 2: Einheitskreis, x, y = -1.5 … 1.5

```
(defn udisk
  [p]
  (< (distance-from-origin p) 1.0))
(defn distance-from-origin
  [[x y]]
  (Math/sqrt (+ (* x x) (* y y))))
```

Listing 2

zeptiert als Argument x- und y-Koordinate und liefert die Information, ob die Summe dieser Koordinaten – jeweils gerundet – gerade ist oder nicht.

Bei Clojure handelt es sich um einen Lisp-Dialekt, darum werden alle Funktionsaufrufe in Präfixschreibweise mit Klammern geschrieben. So ist „(+ x y)" die Summe von „x" und „y". „Math/floor" ist die statische Methode „floor" der Java-Klasse „java.lang.Math". Die Funktion „int" macht aus dem Ergebnis ein Integer. Ein „return" gibt es in Clojure nicht: Die Funktion gibt einfach den Wert des Ausdrucks im Rumpf der Funktion zurück. Die Schreibweise „[x y]" besagt, dass das Argument von „checker" ein einzelner Vektor mit zwei Elementen ist. Wie wir sehen werden, ist es praktisch, einen Satz kartesischer Koordinaten stets als ein Objekt zu betrachten.

Abbildung 2 zeigt ein weiteres einfaches Bild: einen Kreis mit Radius 1 (siehe Listing 2). Hier ist gut zu sehen, dass die Definition des Kreises der Definition eines Kreises aus der Formelsammlung entspricht: Diese Eigenschaft der entstehenden DSL folgt aus der direkten Repräsentation. Indirekte Repräsentationen durch Bitmaps oder Vektorgrafik hätten es da deutlich schwerer.

Wie sieht es aus, wenn kein Einheitskreis gefragt ist, sondern ein Kreis anderer Größe? Dazu bräuchten wir eine Funktion, um ein Bild zu skalieren. Das Skalieren eines Bildes fängt mit dem Skalieren eines Punktes an (siehe Listing 3).

Die erste Funktion „scale-point" skaliert einen Punkt relativ zum Ursprung in x-Richtung um den Faktor „sx" und in y-Richtung um „sy". Die zweite Funktion skaliert in beide Richtungen um den gleichen Faktor. Nun können wir ein Bild mit einer solchen Transformation komponieren: Ein Bild ist ja eine Funktion, die einen Punkt akzeptiert. Wir können also z.B. versuchen, das Bild „udisk" um den Faktor 2 zu skalieren: „(comp udisk (partial uscale-point 2.0))".

Da Bilder Funktionen sind, können wir die in Clojure eingebaute Funktion „comp" benutzen: „(comp f1 f2)" macht aus zwei Funktionen „f1" und „f2" eine Funktion, die „x" auf „(f1 (f2 x))" abbildet. Wir brauchen zunächst eine Funktion, die einen Punkt um den Faktor 2 skaliert, und komponieren diese dann mit „udisk". Diese Skalierungsfunktion bekommen wir mit „(partial uscale-point 2.0)": „partial" macht aus „uscale-point", die ja eigentlich zwei Parameter hat, eine Funktion mit einem Parameter, nämlich dem „p" von „uscale-point". Der erste Parameter „s" ist immer 2. Leider ist das Resultat nicht ganz wie erwartet: Da der Punkt skaliert wird, bevor er an „udisk" gefüttert wird, schrumpft der Kreis um den Faktor 2, anstatt dass er größer wird. Wir müssen für solche Kompositionen mit einer Punkt-Transformations-Funktion deshalb immer die inverse Transformation benutzen. Entsprechend definieren wir Funktionen „scale" und „uscale", die ein Bild skalieren (siehe Listing 4).

Und tatsächlich, (uscale 2.0 udisk) liefert einen doppelt so großen Kreis wie den Einheitskreis. Ähnlich können wir Funktionen für das Verschieben und Rotieren zunächst jeweils eines Punktes und eines Bildes definieren (siehe Listing 5).

Jetzt können wir zum Beispiel das Schachbrett drehen: Abbildung 3 zeigt das Bild „(rotate (/ Math/PI 4.0) checker)". Die Funktionen „scale", „uscale" und „rotate" sind allesamt Beispiele für Kombinatoren: Sie akzeptieren Bilder (also die Gegenstände der DSL) und liefern ihrerseits Bilder, die wiederum mit anderen Kombinatoren kombiniert werden können. Der Umstand, dass Funktionen in Clojure Objekte sind, für die Kombinatoren wie „comp" vorhanden sind, erleichtert ihre Definition.

Für Bilder ist eine Vielzahl von Kombinatoren denkbar. Zum Beispiel können wir ein Bild als „Region" betrachten, also als Menge der (schwarzen) Punkte, bei denen die Bildfunktion „true" liefert. Hier zwei primitive Regionen:

- (defn universe-region [p] true)
- (defn empty-region [p] false)

Entsprechend ist „universe-region" die Region, die aus allen Punkten und „empty-re-

```
(defn scale-point
  [sx sy [x y]]
  [(* sx x) (* sy y)])
(defn uscale-point
  [s p]
  (scale-point s s p))
```

Listing 3

```
(defn scale
  [sx sy img]
  (comp img (partial scale-point (/ 1.0 sx) (/ 1.0 sy))))
(defn uscale
  [s img]
  (comp img (partial uscale-point (/ 1.0 s))))
```

Listing 4

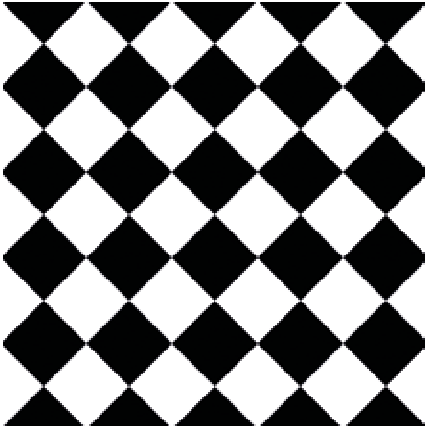


Abbildung 3: Gedrehtes Schachbrett, x, y = -3.5 … 3.5

gion" diejenige, die aus keinen besteht. In Clojure lassen sich diese Funktionen noch einfacher definieren:

- (def universe-region (constantly true))
- (def empty-region (constantly false))

Die in Clojure eingebaute Funktion „constantly" liefert eine Funktion, die – egal mit welchem Argument – immer den gleichen Wert liefert. Regionen können wir mit logischen Operationen punktweise verknüpfen (siehe Listing 6).

Das „fn" stellt – ähnlich wie das Lambda in anderen funktionalen Sprachen oder wie Lambda-Ausdrücke in Java 8 – eine anonyme Funktion her. Allerdings wird an Beispielen deutlich, dass beide Funktionen Instanzen eines allgemeineren Prinzips sind: Eine „innere" Funktion wird jeweils punktweise zu einer Funktion auf Bildern erweitert. Das Prinzip der punktweisen Fortsetzung einer Funktion nennt sich auch „Lifting" und tritt in vielen EDSLs auf. Es lässt sich allgemein für ein- und zweistellige Funktionen definieren (siehe Listing 7).

Die Funktion „lift1" akzeptiert eine Funktion „f", die geliftet werden soll. Sie liefert dann ihrerseits eine Funktion (mit „fn"), die ein Bild akzeptiert und ein Bild liefert, bei dem dann „f" auf die Werte der Punkte von „img" angewendet wird. Die zweistellige Funktion „lift2" funktioniert entsprechend. Es ist in Clojure sogar möglich, eine allgemeine Funktion „lift" zu definieren, die für beliebige Stelligkeiten funktioniert.

Mit Lifting können wir nun diverse logische Kombinatoren auf Bildern sehr einfach definieren (siehe Listing 8).

```
(defn translate-point
  [dx dy [x y]]
  [(+ x dx) (+ y dy)])
(defn translate
  [dx dy img]
  (comp img (partial translate-point (- dx) (- dy))))
(defn rotate-point
  [theta [x y]]
  [(- (* x (Math/cos theta))
      (* y (Math/sin theta)))
   (+ (* y (Math/cos theta))
      (* x (Math/sin theta)))])
(defn rotate
  [theta img]
  (comp img (partial rotate-point (- theta))))
```

Listing 5

Mithilfe dieser Operationen lässt sich zum Beispiel das Bild in Abbildung 4 erzeugen. Aber auch sogenannte „Op-Art" ist möglich (siehe Listing 9). Abbildung 5 zeigt ein Beispiel mit Op-Art.

So langsam entsteht also eine Bibliothek von „primitiven Bildern" wie „checker" oder „alt-rings" und von Kombinatoren wie „scale" oder „union-regions". Ein Detail fehlt noch, nämlich eine Funktion, die ein Bild auch anzeigt. Das ist mithilfe der Clojure-Bibliothek Seesaw ziemlich einfach – die Namespaces „seesaw.core" und „seesaw.color" werden benötigt (siehe Listing 10).

Mit den hier dargestellten Funktionen lassen sich direkt Bilder auf relativ abstrakter Ebene von Koordinaten, Geometrie und Regionen beschreiben und anzei-

```
(defn intersect-regions
  [img1 img2]
  (fn [p]
    (and (img1 p) (img2 p))))
(defn union-regions
  [img1 img2]
  (fn [p]
    (or (img1 p) (img2 p))))
```

Listing 6

```
 (defn lift1
  [f]
  (fn [img]
    (fn [p]
      (f (img p)))))
(defn lift2
  [f]
  (fn [img1 img2]
    (fn [p]
      (f (img1 p) (img2 p)))))
```

Listing 7

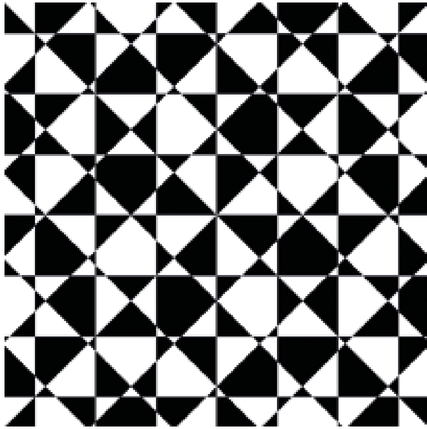


Abbildung 4: „(xor-regions checker (rotate (/ Math/PI 4.0) checker))“, x, y = -3.5 … 3.5

```
(def complement-region (lift1 not))
(def intersect-regions (lift2 (fn [a b] (and a b))))
(def union-regions (lift2 (fn [a b] (or a b))))
(def xor-regions (lift1 not=))
(defn difference-regions
  [r1 r2]
  (intersect-regions r1
                     (complement-region r2)))
```

Listing 8

gen: Es handelt sich also um eine DSL. Eine Stand-Alone-DSL für Bilder, die ähnlich funktioniert wie die hier dargestellte kleine Library, wäre äußerst aufwändig, da wir für die Definitionen von Funktionen auf Sprachmittel von Clojure zurückgegriffen haben; diese müssten wir erst nachbilden. Die Grenzen zwischen der DSL und der Host-Sprache Clojure sind dabei fließend: Kombinatoren sind ganz normale Funktionen und müssen darum auch nicht in „primitive“ und „user-definierte“ Operationen unterschieden werden.

Die Möglichkeiten der EDSL gehen weit über die hier gezeigten, simplen Beispiele hinaus (Mehr Material siehe http://conal.net/Pan). Die ersten Grundprinzipien für das Design von EDSLs in funktionalen Sprachen sind jetzt deutlich geworden:

- Definiere möglichst direkte Repräsentationen für die Entitäten des Problems
- Suche nach Kombinatoren, die aus Entitäten wieder Entitäten machen und implementiere diese

Dieser deklarative Ansatz steht im Gegensatz zu imperativen Ansätzen, wie sie gerade für die Beschreibung von Bildern typisch sind: Befehlsfolgen, die Bilder zum Beispiel in „java.awt.Graphics“ darstellen, eignen sich nur sehr eingeschränkt für die Definition von Kombinatoren. Natürlich kann dieser Ansatz auch in Java umgesetzt werden, wäre dort aber wegen des Fehlens von First-Class-Funktionen und der fälligen Typ-Annotationen so umständlich und aufwändig, dass er kaum als DSL durchgehen würde. Solche „Kombinator-EDSLs“ gibt es in funktionalen Sprachen inzwischen für vielfältige Anwendungen, von Finanzverträgen über Musik und Bühnenbeleuchtung zu VLSI-Layouts bis hin zur Animation.

Stream-Prozessoren

In diesem Abschnitt stellen wir eine EDSL vor, die von vornherein für die Einbettung bestimmt, also ohne Host-Sprache drumherum gar nicht sinnvoll ist. Außerdem ist die Notation schon vorgegeben.

Bei dieser EDSL machen wir von einem weiteren Sprachmittel von Clojure Gebrauch, das bei der Implementierung komplexerer EDSLs außerordentlich nützlich ist: Clojure erlaubt durch das leistungsfähige Makro-System, bei EDSL-Design und Implementierung systematisch vorzugehen.

Es geht um Stream-Prozessoren. Darunter versteht man ein kleines Programm, das einen Strom von Objekten als Eingabe einliest und einen Strom von Objekten als Ausgabe produziert. Beide Ströme können potenziell unendlich lang sein. Die Einlese-Operation heißt „get“, die Ausgabe-Operation „put“. Der Clou ist, dass zwei Stream-Prozessoren wie Gartenschläuche aneinander angeschlossen oder komponiert werden können: Die Ausgabe des ersten Prozessors ist die Eingabe des nächsten. Abbildung 6 illustriert das Prinzip. Nachfolgend die angestrebte Notation für einen Stream-Prozessor:

- (get x)
- (get y)
- (put (* x y))

Das sollte sinngemäß heißen: Lies eine Zahl ein und nenne sie „x“, lies eine weitere

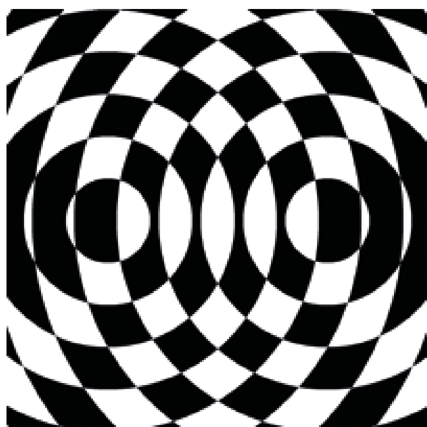


Abbildung 5: „(shift-xor 2.6 alt-rings)“, x, y = -5 … 5

```
(defn alt-rings
  [p]
  (even? (int (Math/floor (distance-from-origin p)))))
(defn shift-xor
  [r img]
  (let [tr (fn [d]
             (translate d 0 img))]
    (xor-regions (tr r)
                 (tr (- r)))))
```

Listing 9

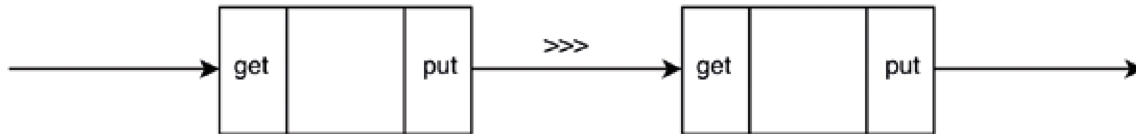


Abbildung 6: Stream-Prozessoren

Zahl ein und nenne sie „y“ und gib schließlich das Produkt beider Zahlen aus. Am schönsten wäre natürlich, wenn wir „get“ und „put“ direkt als Funktionen definieren könnten, etwa so:

- (defn put [x] ...)
- (defn get [var] ...)

Ganz so einfach ist es leider nicht, und zwar aus drei Gründen:

1. Der „get“-Befehl soll eine Variable binden, die nach dem Befehl verwendbar ist. Das geht nicht mit einem Funktionsaufruf.
2. Die einfache Hintereinanderausführung macht es schwer, zwei Prozessoren zu komponieren: Schließlich soll bei der Komposition zweier Prozessoren („sp1“ und „sp2“) jeweils „sp1“ bis zum nächsten „put“ laufen und dann „sp2“ die Gelegenheit haben, bis zum nächsten „get“ zu laufen und den Wert dort abzuholen. Wir müssen also irgendwie ermöglichen, die Ausführung eines Prozessors nach einem „get“ oder „put“ zu unterbrechen.
3. Um zwei Prozessoren komponieren zu können, sollten wir sie als Objekte behandeln. Oben steht aber nur eine einfache Anweisungsfolge.

Der erste Schritt zur Lösung des Problems ist Punkt 3: aus „get“ und „put“ Objekte zu machen. Dazu legen wir Record-Definitionen an:

- (defrecord Put [...])
- (defrecord Get [...])

Bei „...“ müssen wir noch Felder eintragen. Bei „Put“ brauchen wir auf jeden Fall den auszugebenden Wert „(defrecord Put [value ...])“.

Nun können wir uns Punkt 2 zuwenden: Um die Ausführung eines Programms zu unterbrechen (um später an der gleichen

```
(defn display-image!
  [img width height x-min x-max y-min y-max]
  (let [c (canvas :paint
                (fn [_ gr]
                  (let [xinc (/ (- x-max x-min)
                                (double width))
                        yinc (/ (- y-max y-min)
                                (double width))]
                    (doseq [x (range width)
                            y (range height)]
                      (.setColor gr
                             (if (img (+ x-min (* xinc x))
                                      (+ y-min (* yinc y)))
                               java.awt.Color/BLACK
                               java.awt.Color/WHITE))
                      (.fillRect gr x y 1 1)))))
        f (frame :width width :height height
                 :title „Functional Image“
                 :content c)]
    (show! f)))
```

Listing 10

```
(def sp1 (Put. 5 (fn [] (Put. 3 (fn [] stop)))))
(def sp2
  (Get. (fn [x] (Get. (fn [y] (Put. (+ x y) (fn [] stop)))))))
```

Listing 11

Stelle weiterzumachen), bietet es sich an, eine Funktion zu verwenden, die aufgerufen wird, wenn es weitergeht. Wir brauchen sie sowohl in „Get“ als auch in „Put“ und legen sie dort als Feld „next“ an:

- (defrecord Put [value next])
- (defrecord Get [next])

Die Funktion „next“ in „Put“ hat keine Parameter und liefert einfach nur den Prozessor, mit dem es weitergeht. Wenn also „sp“ ein Stream-Prozessor ist, können wir mit „((:next sp))“ den nächsten Stream-Prozessor bekommen.

Es bleibt noch Punkt 1: Wir wollen bei „get“ noch einen Bezeichner binden. Für das Binden sind jedoch in funktionalen Spra-

```
(defn sp-filter
  [p]
  (Get.
   (fn [x]
     (if (p x)
       (Put. x
           (fn []
             (sp-filter p)))
       (sp-filter p)))))
```

Listing 12

chen ebenfalls die Funktionen zuständig. Wir können die Funktion, die sowieso schon im „Get“-Record steckt, mit einem Parameter versehen; wir müssen sie dann mit dem

```
(defn run
  [sp]
  (cond
   (instance? Put sp)
     (lazy-seq (cons (:value sp) (run ((:next sp)))))
   (instance? Get sp) '()
   (instance? Stop sp) '()))
```

Listing 13

```
(defn >>>
  [sp1 sp2]
  (cond
   ;; put >>> get
   (and (instance? Put sp1) (instance? Get sp2))
   (>>> ((:next sp1)) ((:consume sp2) (:value sp1)))
```

Listing 15

gelesenen Wert als Argument aufrufen. Um die parameterlose Funktion im „Put"-Record von der Funktion in „Get" abzugrenzen, benennen wir sie in „consume" um:

- (defrecord Put [value next])
- (defrecord Get [consume])

Durch diese Darstellung entsteht ein kleines Problem: Es gibt keine Möglichkeit, den Stream-Prozessor zu beenden. Jedes „Put" beziehungsweise „Get" braucht eine Funktion, um weiterzumachen. Wir legen darum einen Singleton-Record-Typ für das Anhalten an:

- (defrecord Stop [])
- (def stop (Stop.))

„Stop." ist der Name des Konstruktors des Record-Typs „Stop". Jetzt können wir die ersten Beispiele aufschreiben (siehe Listing 11).

Auch hier sind „Put." und „Get." Konstruktor-Aufrufe. Der Prozessor „sp1" gibt die Zahlen „5" und „3" aus. Der Prozessor „sp2" entspricht dabei dem Beispiel von oben, gibt also die Summe zweier eingelesener Zahlen aus.

Wir können auch einen Prozessor definieren, der einen unendlichen Strom generiert:

- (defn sp-from [n] (Put. n (fn [] (sp-from (+ n 1)))))
- (def nats (sp-from 0))

Der Prozessor „nats" generiert die natürlichen Zahlen. Listing 12 zeigt eine Funktion für etwas komplexere Prozessoren, sogenannte „Filter".

Die Funktion „sp-filter" reicht diejenigen Eingaben an die Ausgabe durch, die ein bestimmtes Kriterium erfüllen: Sie akzeptiert ein Prädikat „p" (also eine Funktion, die einen Wert akzeptiert und „true" oder „false" liefert) und konstruiert den dazugehörigen Prozessor. Das „Get." liest einen Wert ein (das „fn" nennt ihn „x") und das „if" prüft das Kriterium; bei „true" gibt das „Put." den gelesenen Wert aus, ansonsten geht es ohne den Wert rekursiv weiter. Erfahrene Leser erkennen jetzt, dass die Stream-Prozessoren in Continuation-Passing-Style geschrieben sind.

Wir können jetzt also ziemlich beliebige Stream-Prozessoren herstellen. Doch leider …

1. … ist die Notation ziemlich umständlich und weit weg von dem, was wir uns ursprünglich vorgestellt hatten,
2. … haben wir noch keine Funktion, um die Ausgabe eines Prozessors aufzusammeln
3. … und auch noch keine Funktion, um zwei Prozessoren zu komponieren.

Beginnen wir mit Punkt 2, das geht recht einfach (siehe Listing 13).

Die „run"-Funktion akzeptiert einen Stream-Prozessor, unterscheidet dann

```
> (run sp1)
(5 3)
> (take 5 (run nats))
(0 1 2 3 4)
```

Listing 14

nach den drei Record-Typen („instance?", „Put" und „sp") und testet zum Beispiel, ob „sp" ein „Put"-Record ist: Bei jedem „put" steckt „run" den ausgegebenen Wert in eine „lazy sequence", also eine Folge, die auch unendlich sein kann. „(:value sp)" extrahiert den ausgegebenen Wert und „(run ((:next sp)))" macht weiter. „cons" konstruiert die Folge. Damit können wir schon zwei Beispiele aufrufen (siehe Listing 14).

Die „take"-Funktion extrahiert in diesem Fall die ersten fünf Elemente der Folge.

Aber es bleiben noch die Probleme 1 und 3, die umständliche Syntax und die Komposition. Machen wir uns erst einmal an die Komposition: Wir hätten gern einen Kombinator, der zwei Prozessoren akzeptiert und einen liefert. Eine Möglichkeit wäre, einen neuen Record-Typ für „>>>" (das ist in Clojure ein zulässiger Bezeichner) hinzuzufügen und diesen in „run" extra zu behandeln. Allerdings sollten wir bei solchen Erweiterungen erst einmal grundsätzlich versuchen, mit den existierenden DSL-Elementen auszukommen. Wir versuchen also, „>>>" als Funktion zu schreiben: „(defn >>> [sp1 sp2] ...)". Nun gibt es sowohl für „sp1" als auch für „sp2" jeweils die drei Möglichkeiten „Put", „Get" und „Stop", insgesamt also potenziell neun. Die Repräsentation, die wir gewählt haben, erlaubt uns jetzt einen tollen Trick: Wenn wir nur die „Put" und „Get" paarweise nebeneinanderstellen, dann heben sie sich jeweils gegenseitig auf (siehe Listing 15).

Der erste Zweig in der „cond"-Fallunterscheidung testet zunächst auf genau diesen Fall – also dass „sp1" ein „put" und „sp2" ein „get" sind. In diesem Fall können wir den ausgegebenen Wert vom „put" – das ist „(:value sp1)" – in die „consume"-Funktion vom „get" stecken. Auf der linken Seite machen wir mit der „next"-Funktion vom „put" weiter. Wir müssen noch die anderen Fälle abdecken: Dort, wo „sp2" ein „Put" ist, können wir den angegebenen Wert direkt ausgeben (siehe Listing 16). Bei zwei „Get" hintereinander ziehen wir das „>>>" nach innen (siehe Listing 17).

Damit sind alle Kombinationen von „Put" und „Get" abgedeckt. Es bleibt noch „Stop". „Stop" vorn bedeutet, dass der erste Prozessor fertig ist – wir machen mit dem zweiten weiter: „;; stop >>> sp2 (instance? Stop sp1) sp2". Wenn der zweite Prozessor fertig ist, spielt es hingegen keine Rolle, was der erste macht: „(instance? Stop sp2) stop". Damit ist auch „>>>" fertig. Wir können jetzt weitere Beispiele ausprobieren (siehe Listing 18).

Es bleibt also nur noch die umständliche Syntax. Hier kommt jetzt ein Makro ins Spiel, das die eigentliche DSL definiert. Dazu müssen wir der EDSL einen Namen geben, in diesem Fall bietet sich „stream-processor" an. Damit könnten die obigen Beispiel-Prozessoren so geschrieben werden:

Das Makro ist nichts anderes als eine Funktion, die vom Clojure-Compiler auf alle Formen angewendet wird, die mit „stream-processor" beginnen. An den Beispielen sieht man, dass die Form eine beliebige Anzahl von Operanden haben kann: einen für jede „Klausel". Die Makro-Definition steckt diese mithilfe der Parameter-Deklaration „& ?clauses" in eine Liste und nennt diese „?clauses". Das Fragezeichen ist reine Konvention.

Der Rumpf der Makro-Definition schaut dann nach, ob die Liste nicht leer ist (dann werden die erste Klausel und der Rest extrahiert) und wenn nicht, wird „stop" produziert. Der „accent grave" vor dem „stop", der sogenannte „Backquote", ist für die Generierung des Ausgabe-Codes zuständig. Jede Klausel fängt jetzt entweder mit „get" oder „put" an, was wir mit „case" unterscheiden (siehe Listing 20).

Innerhalb der Backquotes fügt „~?var" die Variable ein beziehungsweise „~@?rest" die restlichen Klauseln. Das Muster der obigen Beispiele ist damit erkennbar. Mit dieser Definition funktionieren schon einmal die Beispiele „sp1" und „sp2". „sp-from" enthält hingegen noch einen rekursiven Aufruf, müsste also wie in Listing 21 aussehen.

Das „stream-processor"-Makro kennt aber bisher nur „put"- und „get"-Klauseln. Wir müssen also noch einen Default-Fall einfügen (siehe Listing 22). Es bleibt noch das „sp-filter"-Beispiel, das eine Fallunterscheidung enthält. Listing 23 zeigt, wie diese aussehen könnte. Dann benötigen wir noch einen weiteren Fall im Makro für „when", der das benötigte „if" generiert (siehe Listing

```
;; sp >>> put
(instance? Put sp2)
(Put. (:value sp2) (fn [] (>>> sp1 ((:next sp2)))))
```

Listing 16

```
;; get >>> get
(and (instance? Get sp1) (instance? Get sp1))
(Get. (fn [i] (>>> ((:consume sp1) i) sp2)))
```

Listing 17

```
> (run (>>> sp1 sp2))
(8)
> (take 5 (run (>>> nats (sp-filter even?))))
(0 2 4 6 8)
```

Listing 18

```
(def sp1 (stream-processor (put 5) (put 3)))
(def sp2 (stream-processor (get x) (get y) (put (+ x y))))

Listing 19 zeigt den Anfang für die Definition des Makros.

(defmacro stream-processor
  [& ?clauses]
  (if (seq? ?clauses)
    (let [?clause (first ?clauses)
          ?rest (rest ?clauses)]
      ...)
    `stop))
```

Listing 19

```
(defmacro stream-processor
  [& ?clauses]
  (if (seq? ?clauses)
    (let [?clause (first ?clauses)
          ?rest (rest ?clauses)]
      (case (first ?clause)
        get
        (let [?var (second ?clause)]
          `(Get. (fn [~?var]
                   (stream-processor ~@?rest))))

        put
        (let [?val (second ?clause)]
          `(Put. ~?val (fn [] (stream-processor ~@?rest))))))
    `stop))
```

Listing 20

```
(defn sp-from
  [n]
  (stream-processor (put n) (sp-from (+ n 1))))
```

Listing 21

```
 (defmacro stream-processor
  [& ?clauses]
  (if (seq? ?clauses)
    (let [?clause (first ?clauses)
          ?rest (rest ?clauses)]
      (case (first ?clause)
        get ...
        put ...

        ?clause))
    `stop))
```

Listing 22

```
(defn sp-filter
  [p]
  (stream-processor
   (get x)
   (when (p x)
     (put x)
     (sp-filter p))
   (sp-filter p)))
```

Listing 23

```
 (defmacro stream-processor
  [& ?clauses]
  (if (seq? ?clauses)
    (let [?clause (first ?clauses)
          ?rest (rest ?clauses)]
      (case (first ?clause)
        get ...
        put ...

        when
        (let [?test (second ?clause)
              ?consequent (rest (rest ?clause))]
          `(if ~?test
             (stream-processor ~@?consequent)
             (stream-processor ~@?rest)))

        ?clause))
    `stop))
```

Listing 24

24). Selbst die viel schönere Definition von „sp-filter“ funktioniert (siehe Listing 25).

Auch bei den Stream-Prozessoren sind, wie bei den Bildern, Kombinatoren entstanden. Da diese aber noch syntaktisch umständlich zu benutzen waren, kommt zu den Schritten zum Implementieren von EDLs aus dem ersten Abschnitt noch ein weiterer hinzu, um die Syntax gegebenenfalls mit Makros zu vereinfachen. Ein Hinweis noch: Die Idee der Stream-Prozessoren stammt aus einem sehr lesenswerten Paper von John Hughes über Arrows (siehe http://www.haskell.org/arrows/biblio.html#Hug00).

```
> (take 5 (run (>>> nats
(sp-filter odd?))))
(1 3 5 7 9)
```

Listing 25

Fazit

Um noch einmal zusammenzufassen:

- DSLs sind oft nützlich
- EDSLs sind einfacher zu implementieren als Stand-alone-DSLs
- Funktionale Sprachen unterstützen die Implementierung von EDSL in besonderer Weise

Die Kombination von First-Class-Funktionen mit Makros in Clojure und anderen Lisp-Varianten unterstützt die systematische Entwicklung von EDSLs. Die hier vorgestellten EDSLs sind auch im Internet verfügbar.

Links

Pan: http://conal.net/Pan
Stream-Prozessor-DSL: http://www.haskell.org/arrows/biblio.html#Hug00
Source-Code funktionale Bilder: https://github.com/active-group/functional-images
Source-Code Stream-Prozessor-DSL: https://github.com/active-group/stream-processors

Michael Sperber
michael.sperber@active-group.de

Dr. Michael Sperber ist CTO der Active Group GmbH. Er ist seit mehr als fünfzehn Jahren in der Forschung über funktionale Programmierung und in deren industrieller Anwendung tätig. Er hat zahlreiche Fachartikel zum Thema verfasst, Anfänger-Ausbildungen in Programmierung an den Universitäten Tübingen, Freiburg sowie Kiel konzipiert und diese in Tübingen mehrfach durchgeführt. Er gehört zu den Mit-Initiatoren und Autoren des Blogs „funktionale-programmierung.de“.

Lohnt sich der Umstieg von JSF 1.x auf JSF 2.x?

Andy Bosch, JSF-Academy.de

In vielen großen und mittelständischen Firmen kommt häufig noch JSF 1.1 oder JSF 1.2 zum Einsatz. Aktuell verfügbar ist bereits die Version JSF 2.2. Dieser Versionssprung hört sich zunächst nicht allzu dramatisch an.

JSF 1.0 wurde im Jahr 2004 veröffentlicht und basiert auf Konzepten und Arbeiten aus den Jahren 2001 bis 2003. Das macht die Lage durchaus dramatisch, weshalb ein Umstieg auf die neueste JSF-Version in vielen Unternehmen aktuell auch eifrig diskutiert und evaluiert wird. Dieser Artikel geht auf die Argumente für einen Umstieg ein und zeigt dabei auch die vielen Verbesserungen, die dieser bringen kann. Darüber hinaus wird ein Migrationsszenario vorgestellt.

JSF blickt auf eine längere Entwicklungsgeschichte zurück. Nur die wenigsten wissen, dass die Anfänge von JSF auf das Jahr 2001 zurückführen. Damals wurde die Idee geboren, im Standard-Java-Stack ein webbasiertes UI-Framework zu erstellen. 2004 war es dann soweit, als JSF 1.0 öffentlich zum Download bereitstand. Wenig später gab es dann mit JSF 1.1 noch ein kleines Bugfix-Release.

Zwar wurde im Jahr 2006 eine neue Version 1.2 auf den Weg gebracht, allerdings bot diese nur unwesentliche Neuerungen. Der größte Punkt was damals die Harmonisierung der JSP Expression Language mit der damaligen JSF Expression Language zur Unified Expression Language. Doch dieser Vorgang fand überwiegend unter der Haube statt, sodass der Applikationsentwickler davon kaum etwas mitbekommen hat.

Bis zum Erscheinen von JSF 2.0 im Jahr 2009 hat sich JSF somit kaum merklich weiterverändert, was viele Kritiker von JSF damals auch zu Recht angeprangert hatten. Seit 2009 sind jedoch die Expert Groups sehr eifrig und haben sowohl JSF 2.1 als auch bereits JSF 2.2 herausgebracht (siehe Abbildung 1). JSF 2.3 wird intern bereits diskutiert. Da in diesen Versionen durchaus sehr viel passiert ist, sollten Unternehmen, die noch auf JSF 1.x stehen, einen Upgrade auf jeden Fall evaluieren, um die Vorteile der neuen Versionen zu nutzen und um Schwachstellen in der alten Version zu umgehen.

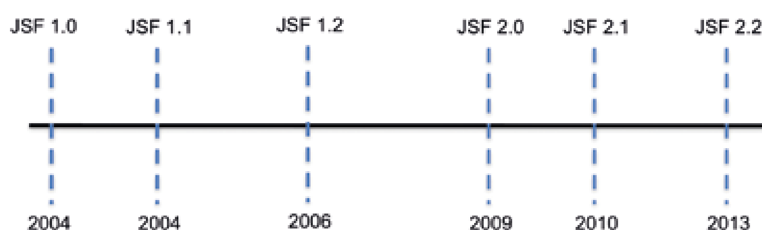


Abbildung 1: Entwicklungsgeschichte von JSF

Festgehangen in JSF 1.x

Mit Erscheinen von JSF 1.0/1.1 wurde eine JSF-Implementierung in vielen Application-Servern aufgenommen. Oftmals kam hier Mojarra, die Referenz-Implementierung von Oracle (damals noch Sun), zum Einsatz. MyFaces, die Open-Source-Alternative dazu, wurde damals sehr häufig im Umfeld von Servlet-Containern eingesetzt.

Da JSF als Standard konzipiert wurde, unterstützten viele Server- und IDE-Hersteller das neue Framework von Anfang an. Viele Projekte wurden basierend auf JSF gestartet und in Betrieb genommen. Es ist jedoch so, dass Projekte ab einer gewissen Größenordnung nicht einfach mal eben auf eine neue JSF-Version oder einen neuen Application-Server gebracht werden konnten. Auch wird in vielen großen und mittelständischen Firmen ein Versionswechsel von Application-Servern nicht jedes Jahr durchgeführt, sondern ein Server hält sich oftmals mehrere Jahre. Daher kommt es selbst im Jahr 2013 vor, dass viele Anwendungen auf JSF 1.0/1.1 basieren und ein Wechsel nicht abzusehen ist.

Genauso ist es Realität, dass selbst bei einem Upgrade des Application-Servers die neuere JSF-Version aufgrund von Zeit-, Budget- oder Know-how-Engpässen nicht verwendet wird, sodass noch Features und Schwachstellen von anno dazumal weiterhin im Einsatz sind. Der Autor kennt selbst viele Projekte, die nach fast zehn Jahren basierend auf JSF 1.0 immer noch nicht migriert werden konnten.

Probleme mit JSF 1.x

Im Zusammenhang mit JSF 1.x sind einige Probleme bekannt, weshalb pauschal immer zu einer Migration nach JSF 2.x geraten wird. Im Folgenden sind die verschiedenen Probleme und Nachteile der alten JSF-Versionen angesprochen, um Argumente für eine Migration liefern zu können.

Ursprünglich basierte JSF auf der JSP-Technologie für die View-Beschreibung. Java Server Pages (JSP) sind durchaus eine gute Sache, speziell im Zusammenspiel mit JSF existieren allerdings einige Nachteile. Daher sind sie im Umfeld von JSF 2.x (und nur hier) als „deprecated", also veraltet gekennzeichnet worden. JSPs bieten beispielsweise die Möglichkeit, Java-Code in der View selbst zu integrieren, die sogenannten „Scriptlets". Dies verursacht bei Architekten häufig Magenkrämpfe, da sie verschiedenste Design-Prinzipien auf das Gröbste verletzen (etwa „Separation of Concerns" – warum sollte Java-Code in die View integriert werden können?). Der Nachfolger

der View-Technologie innerhalb von JSF 2.x ist Facelets/XHTML. XHTML-Seiten werden im Gegensatz zu JSP interpretiert. Was sich zunächst als Bremse anhört, ist in der Praxis jedoch eher das Gegenteil, es wurde hier mächtig optimiert. So sind XHTML-Seiten aus Performance-Gesichtspunkten mindestens auf gleichem Niveau wie JSP, je nach Konstellation durchaus auch performanter. Viel wichtiger ist jedoch, dass einige Features aus JSP nicht unterstützt werden (etwa die erwähnten Scriptlets).

Deutlicher werden die Probleme sichtbar, wenn es um den Einsatz von Komponenten-Bibliotheken geht. Im Umfeld von JSP sind für eigene Komponenten vier Artefakte zu erzeugen: eine Komponenten-Klasse, eine dazugehörige Renderer-Klasse (es ist auch möglich, den Renderer in der Komponente zu haben) sowie eine Taghandler-Klasse zusammen mit einer Tag-Library. Dieses Zusammenspiel ist nicht immer einfach zu bauen und im Fehlerfalle zu analysieren.

Noch komplexer wird es, wenn JavaScript zum Einsatz kommt. Wie werden die JS-Dateien der Seite hinzugefügt? Soll ein weiteres Tag erschaffen werden, mit dem auf der Seite das Laden von JavaScript-Dateien (und natürlich gegebenenfalls auch CSS-Dateien) angestoßen wird? Oder soll zunächst die komplette Seite, sofern sie als Komponenten-Baum existiert, nochmals geparst und bei Bedarf weitere JS- und CSS-Dateien angefügt werden? Dafür gibt es keine generelle Antwort, selbst die bis dato etablierten JSF-Komponenten-Bibliotheken hatten unterschiedlichste Ansätze. Und genau das ist ein weiteres Problem. Jedes Framework lädt munter JS- und CSS-Dateien, ohne dass eine Abstimmung untereinander erfolgt.

Besonders hart trifft dieses Problem die Portal-Entwickler, die eine JSF-1.x-Anwendung im Portal-Container ablaufen lassen möchten. Basieren die eigenen oder hinzugenommenen JSF-Komponenten-Bibliotheken zum Beispiel auf einem Servlet-Lademechanismus (den es so im Portal-Umfeld nicht gibt), sind Schwierigkeiten vorprogrammiert.

Seien wir einmal optimistisch und unterstellen, dass alle diese Probleme gelöst wurden und die Anwendung im Single-User-Betrieb hervorragend läuft. Wie sieht es jedoch unter Last aus? Hier war JSF 1.x bekannt dafür, sehr Hauptspeicher-hungrig zu sein. Das Teure am Mechanismus von JSF ist der Komponenten-Baum. JSF hat dabei ein Abbild der aktuell im Browser gerenderten Seite als Objekt-Graph in der Regel serverseitig gespeichert. Dies beinhaltet alle Werte sämtlicher Attribute der UI-Komponenten, die auf der Seite vorhanden sind.

Um die Browser-Back-Thematik zu unterstützen, kann JSF sogar die letzten fünfzehn Seiten (und deren fünfzehn Ausprägungen) speichern. Dies ist ein Implementierungs-spezifischer Wert, der im Projekt durchaus auch höher oder niedriger sein kann. Gerade hier findet man bei Code-Reviews oftmals erstaunliche Dinge. Dass es bei falschen Konfigurationen zu Session-Größen von mehr als 50 MB kommen kann, liegt auf der Hand. Doch selbst bei bestmöglicher Konfiguration ist JSF in der Version 1.x recht Speicher-hungrig, kein Weg führt daran vorbei.

Wenn dann noch ein Session-Failover im Application-Server eingestellt ist, kann man das Thema „Performance" ad acta legen, da sie schlichtweg nicht vorhanden ist. Dies sind natürlich nicht alle Probleme, die im Zusammenhang mit JSF 1.x auftreten können, aber es sind doch die am meisten genannten.

Ist mit JSF 2.x alles besser?

JSF 2.x räumt in der Tat viele alte Schwachstellen auf und führt wichtige und lang erwartete Features ein. Eine Neuerung auf JSF 2.0 sind sogenannte „SystemEvents" und bei JSF 2.2 die „ViewActions" (siehe Abbildung 2). Damit lässt sich hervorragend eine „Preaction" realisieren, was mit JSF 1.x fast unmöglich war.
Hintergrund ist die Anforderung, beim Betreten einer Seite (oftmals der Startseite der Anwendung) eine Aktion triggern zu müssen.

JSF 1.x war ausschließlich reaktiv, man benötigte also zunächst einen Command-Button oder Command-Link, um darauf programmatisch reagieren zu können. Oftmals hat man sich in JSF 1.x mangels Alternativen für Lösungen mit „Lazy Init" (über „getter"-Methoden), Servlets oder Servlet-Filtern beholfen oder man hat eigene JSF-Tags dafür entworfen. Alle diese Lösungen waren streng genommen „Dirty Hacks". Mit JSF 2.0 kann dies viel eleganter über „System-Events" (PreRenderView) oder seit JSF 2.2 mittels ViewActions realisiert werden.

Auch dem Thema „Speicherverbrauch" wurde Rechnung getragen. Durch das neu eingeführte „Partial State Saving" wird nicht mehr der komplette Komponenten-Baum mit allen möglichen Attributen gespeichert, sondern lediglich diejenigen Informationen, die vom initialen View-Aufbau abweichen (also in der Regel direkt gesetzt wurden). Dies hat einen dramatischen Einfluss auf den notwendigen Speicherverbrauch im positiven Sinne. Als Anwender muss man hierzu nichts Besonderes tun, um dieses Feature zu nutzen. Alle JSF-Standard-Komponenten basieren bereits darauf.

Abbildung 3 zeigt zwei exemplarische Auswertungen. Der Autor hat zwei typische Anwendungsfälle aus der Praxis betrachtet und mithilfe einer einfachen Sessiongrößen-Ermittlung die Varianten JSF 1.2 und JSF 2.2 gegenübergestellt. Sicherlich sind die absoluten Zahlen in keiner Weise aussagekräftig, hängen diese doch vom konkreten Projekt und insbesondere von

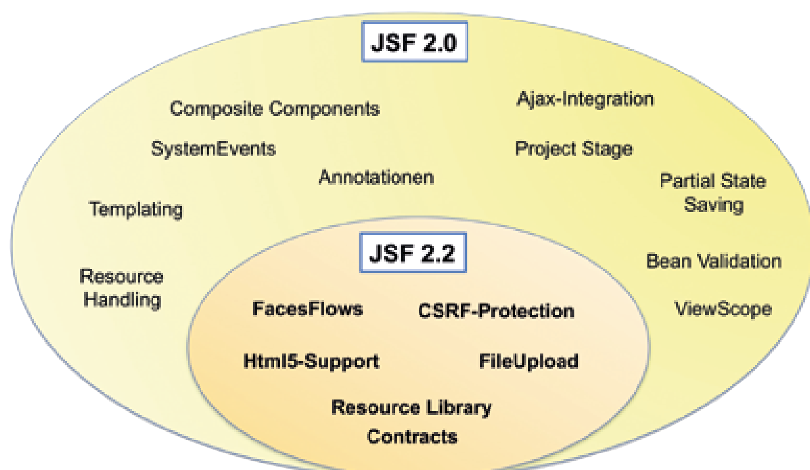


Abbildung 2: Neuerungen in JSF 2.0 und JSF 2.2

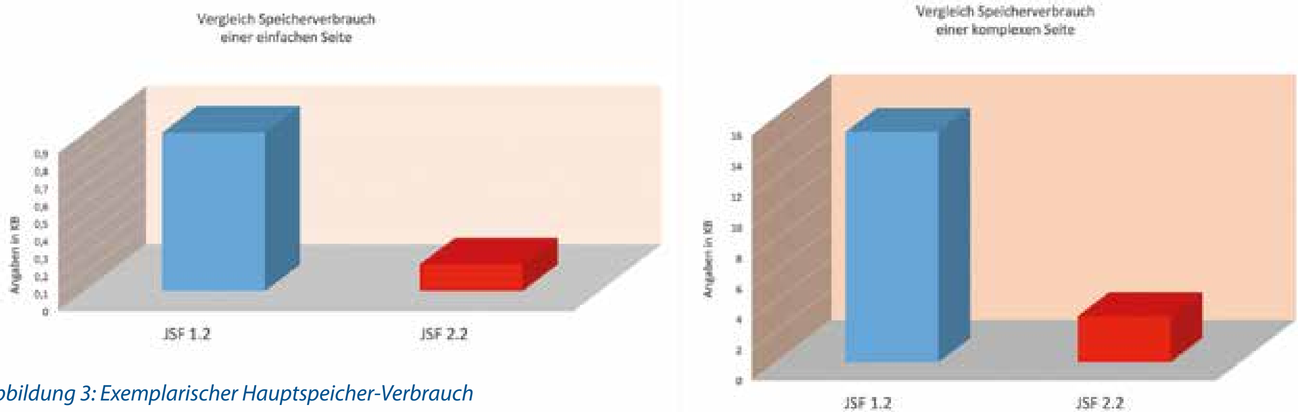


Abbildung 3: Exemplarischer Hauptspeicher-Verbrauch

der konkreten View ab. Allerdings zeigt der Vergleich der Seiten in zwei verschiedenen JSF-Umgebungen eindeutig einen Trend.

Der Unterschied im Speicherverbrauch ist hierbei drastisch zu bemerken. JSF 2.2 hat teilweise einen um Faktor fünf bis acht kleineren Bedarf an Hauptspeicher. Wie gesagt, diese Zahl lässt sich nicht verallgemeinern, es zeigt aber sehr deutlich den positiven Effekt von „Partial State Saving".

Ein weiteres Highlight der neuesten JSF-Version sind „FacesFlows" und „Composite Components". Letztere kennen viele Facelets-Nutzer schon seit einigen Jahren, da dieses Feature mit JSF 2.0 ist in den Standard aufgenommen wurde. Es geht darum, wiederverwendbare XHTML-Bausteine („Snippets") zu definieren. Man erzeugt sich somit eine eigene Komponente, indem vorhandene Bausteine zu einem größeren Ganzen verbunden werden.

Abbildung 4 zeigt ein einfaches Beispiel für eine „Composite Component". Eine Kreditkarten-Eingabe besteht aus verschiedenen Eingabe-Feldern, Listen und Labeln. Dies alles lässt sich über eine „Composite Component" zusammenfassen und durch ein einfaches Tag in eine Seite integrieren. Sollte sich die Eingabe von Kreditkarten-Daten künftig ändern, ist lediglich eine zentrale Änderung am „Snippet" erforderlich. Auch ist die eigentliche XHTML-Seite durch die Auslagerung wesentlich überschaubarer geworden.

„FacesFlows" gehen noch einen Schritt weiter. Sie fassen komplette Masken-Strecken zusammen und lagern sie aus. Das Interessante dabei ist, dass komplette Teilstrecken einer View-Abfolge sogar als „jar"-Datei hinzugeliefert werden können. Gerade bei größeren Projekten ist diese Art der Modularisierung sehr willkommen.

Dies waren natürlich nicht alle Neuerungen, die JSF 2.0 und JSF 2.2 aufzuweisen haben. Es lohnt sich auf alle Fälle, diese Veränderungen im Detail zu betrachten und den Mehrwert daraus für das eigene Projekt zu bewerten.

Ein Migrationspfad zu JSF 2.x

Nach dem Plädoyer für die Nutzung der neuen Möglichkeiten von JSF 2.x stellt sich aber oftmals die Frage, wie eine Migration von JSF 1.x zu JSF 2.x denn stattfinden kann. Bei Versions-Updates bestehen häufig die Bedenken, dass die gesamte Anwendung überarbeitet oder neu geschrieben werden muss. Im Falle von JSF ist dem nicht so.

Pauschal gesagt wird bei allen Standards im JCP sehr stark auf die Abwärts-Kompatibilität geachtet. Es ist von immenser Bedeutung, bei neuen Features und Konzepten diese Kompatibilität nach unten nicht zu gefährden. Auch JSF hat extrem darauf geachtet, dass keine (oder wenn, dann nur sehr wenige) Änderungen notwendig sind, wenn neue JSF-Bibliotheken im Klassenpfad zu finden ist. Somit wäre die einfachste Art der Migration, lediglich die entsprechenden „jar"-Dateien im Klassenpfad auszutauschen (im Falle eines Servlet-Containers wie Tomcat). Ist ein Application-Server im Spiel, hat der neue Server bereits die JSF-Bibliotheken im „System Classpath". Dann ist lediglich ein Deployment der JSF-Anwendung auf den Server notwendig und schon ist man auf JSF 2.x migriert.

Die Realität lehrt uns, dass es so einfach nicht immer funktioniert. Das Pareto-Prinzip bewahrheitet sich auch hier: Die ersten 80 Prozent einer Anwendung lassen sich meist ohne Änderungen sofort unter JSF 2.x zum Fliegen bringen. Es gibt jedoch ein paar Knackpunkte, an denen es häufig Schwierigkeiten gibt. Beginnen wir bei der View-Beschreibung. Basiert diese noch auf JSP, ist ein Wechsel auf Facelets/XHTML nicht zwingend notwendig.

JSF 2.x unterstützt noch JSPs, auch wenn sie als „deprecated" markiert sind. Allerdings können bei deren Einsatz keine neuen Features genutzt werden. Im Gegensatz zur reinen Facelets-Technologie zu JSF-1.x-Zeiten ist es mit JSF 2.x auch möglich, XHTML- und JSP-Seiten parallel im gleichen Projekt zu haben. Es besteht sogar die Möglichkeit, mit den JSF-Navigationsregeln zwischen XHTML- und JSP-Seiten hin und her zu springen. Somit ist es möglich, die einzelnen Views stückweise auf XHTML zu portieren, ohne gleich das gesamte Projekt überarbeiten zu müssen.

Aufwendiger wird es allerdings, wenn in den JSP-Seiten Komponenten-Bibliotheken zum Einsatz kamen. Hier sollte auf jeden Fall auf eine entsprechende Version mit JSF-2.x-Unterstützung gewechselt werden.

Häufig sind diese Upgrades allerdings nicht mehr abwärtskompatibel. Bei RichFaces wurde etwa eine komplette Überarbeitung mit der Version 4 (diese ist speziell für JSF 2.x) durchgeführt. Dies ist sicherlich der Code-Qualität zugutegekommen, bedingt jedoch auch häufig ein größeres Stück Arbeit bei der Migration von Richfaces 3 (für JSF 1.x).

Auch bei eigenen, speziell für das Unternehmen entwickelten Bibliotheken ist bei der Umstellung von JSP auf XHTML etwas Aufwand für die Migration einzuplanen. Zwar fällt mit den Taghandler-Klassen ein Artefakt weg, doch müssen oftmals ein paar Funktionalitäten aus den Taghandler-Klassen in die Komponenten-Klasse überführt werden.

Auch ist eine entsprechende Facelets-Taglib anzulegen, was ebenfalls mit etwas Zeit verbunden ist. Aus Erfahrung liegt ein solcher Aufwand meist im Bereich von Tagen.

Ganz besonders hart trifft die Migration jene Projekte, die mit „Component Binding" gearbeitet haben, was allerdings in Projekten nur sehr selten anzutreffen ist. Dieses bietet die Möglichkeit, den JSF-Komponenten-Baum nicht nur über die View-Beschreibung in JSF oder XHTML zu bauen, sondern programmatisch unter Verwendung der direkten UI-Komponenten-Klassen. Da sich hier die Klassen und Packages komplett geändert haben (dies ist nicht Bestandteil des Standards), fallen in dem Bereich die größten Aufwendungen an. Dieser Schritt sollte somit gut geplant werden, auch ein im Vorfeld durchgeführter Proof of Concept ist stark zu empfehlen.

Der Vollständigkeit halber sei noch erwähnt, dass auch die Konfigurations-Dateien („faces-config.xml" und „web.xml") eines kleinen Updates bedürfen, sei es auch nur in den Header-Angaben (Versionsnummer im XML-Dokument). Dies ist zwar nur ein Aufwand von wenigen Minuten, sollte aber tunlichst nicht vergessen werden.

Alle weiteren Features von JSF 2.x lassen sich nach und nach integrieren. Es besteht keine Notwendigkeit, sofort alle Möglichkeiten auszuschöpfen. Jedoch ist es oftmals angeraten, alte Zöpfe abzuschneiden und die verbesserten Konzepte auch zu nutzen. Dies kann allerdings auch schrittweise erfolgen, sodass die Anwendung auch zwischendurch aufgrund von fachlichen Erweiterungen jederzeit produktiv gehen kann.

Fazit und Empfehlung

Dass JSF 1.x eine veraltete Technologie ist, sollte keiner weiteren Diskussion mehr bedürfen. Ein mehr als zehn Jahre altes Framework entspricht bei weitem nicht mehr den heutigen Anforderungen an ein State-of-the-art UI-Framework. Es steht mit JSF 2.2 eine Version bereit, die sowohl in Bezug auf Stabilität und Performance als auch in Bezug auf wichtige Features eine große Weiterentwicklung darstellt. Gemäß dem Pareto-Prinzip kann man sagen, dass ein Großteil einer Migration in den meisten Projekten mit sehr überschaubarem Auf-

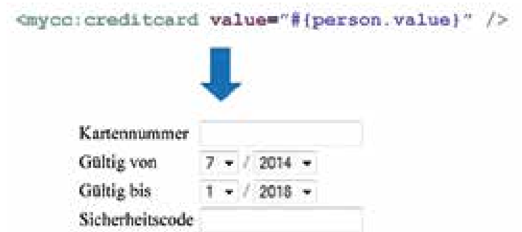


Abbildung 4: Beispiel einer „Composite Component"

LUST AUF VERÄNDERUNG?

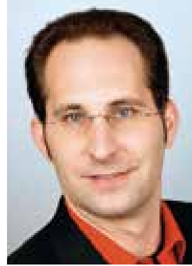
SOFTWAREENTWICKLER
SOFTWAREARCHITEKT
BERATER
PROJEKTLEITER
SCRUM MASTER

WIR SUCHEN DICH!
(VOM EINSTEIGER BIS ZUM PROFI)

Netpioneer GmbH • Ludwig-Erhard-Alle 20 • 76131 Karlsruhe
0721/92060-814 • jobs@netpioneer.de • www.netpioneer.de

wand durchgeführt werden kann. Spezielle Konstellationen müssen separat betrachtet werden, können aber meist ebenfalls mit überschaubarem Aufwand gelöst werden. Der größte Aufwand entsteht oftmals bei Einsatz von Component Binding und eigenen, benutzerdefinierten Komponentenbibliotheken.

Andy Bosch
andy.bosch
@jsf-academy.de



Andy Bosch ist als unabhängiger Trainer, Berater und Entwickler im Umfeld von JSF und Portalen seit vielen Jahren in Projekten unterwegs. Er ist Autor mehrerer Bücher zu JSF und Portlets und hält regelmäßig Vorträge auf nationalen und internationalen Konferenzen.

Anwendungsentwicklung mit dem Oracle WebLogic Server 12*c*

Michael Bräuer, ORACLE Deutschland B.V. und Co. KG

Während die ersten beiden Beiträge in den letzten zwei Ausgaben den WebLogic Server vorwiegend aus der administrativen Sicht zeigten, beleuchtet der dritte Teil die aktuelle Version aus Entwickler-Perspektive.

Das vorliegende Release 12.1.2 enthält neben dem Oracle HTTP Server und dem WebLogic Server als Laufzeitumgebung für serverseitige Java-Anwendungen auch jeweils eine neue Version von Oracle Coherence, TopLink und ADF. Da ADF 12.1.2 genügend Stoff für einen eigenen Beitrag bietet, geht dieser Artikel nur auf WebLogic Server, Coherence und TopLink ein.

Leichter Aufbau

Der Aufbau der Entwicklungsumgebung gestaltet sich sehr einfach. Voraussetzung für die Installation ist ein vorhandenes und zertifiziertes JDK 7 [1]. Speziell für die Entwicklung mit dem WebLogic Server gibt es eine 182 MB große „zip"-Distribution, die auf den Plattformen Linux, Windows und Mac OS X lauffähig ist. Zur Verwendung muss diese Datei entpackt werden, um dann eine Domäne per Skript zu er-

```
[oracle@wlsserver bin]$ ./startWebLogic.sh
Starting at 2013-12-03 09:24:54.387292724+01:00
JAVA Memory arguments: -Xms256m -Xmx512m -XX:CompileThreshold=8000 -XX:PermSize=128m
-XX:MaxPermSize=256m
...
<Dec 3, 2013 9:24:55 AM CET> <Info> <Management> <BEA-141107> <Version: WebLogic Server
12.1.2.0.0 Fri Jun 7 15:16:15 PDT 2013 1530982 WLS_12.1.2.0.0_GENERIC_130607.1100>
...
<Dec 3, 2013 9:25:02 AM CET> <Notice> <Server> <BEA-002613> <Channel "Default" is now liste-
ning on 10.10.10.10:7001 for protocols iiop, t3, ldap, snmp, http.>
<Dec 3, 2013 9:25:02 AM CET> <Notice> <WebLogicServer> <BEA-000331> <Started the WebLo-
gic Server Administration Server "AdminServer" for domain "mydomain" running in development
mode.>
<Dec 3, 2013 9:25:02 AM CET> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to
RUNNING.>
```

Listing 1

zeugen. Dabei erstellt das Domänen-Konfigurationsskript eine Domäne namens „mydomain“ mit genau einem Entwicklungsserver, der gleichzeitig die Rolle des Administrationsservers innehat. Sowohl die WebLogic-Administrationskonsole als auch das Class-Loading-Analysis-Tool sind sofort aufrufbar. Der Aufbau und das Verwalten der Domänenkonfigurationen kann auch hier durch die gängigen Werkzeuge erfolgen.

Alternativ zur „zip“-Distribution stehen der für Produktions-Umgebungen empfohlene generische Installer sowie spezielle Distributionen für Linux x86, Windows x86 und Mac OS X zur Verfügung, die neben WebLogic und Coherence auch das Oracle Enterprise Pack for Eclipse enthalten.

Die Startup-Zeit eines Entwicklungsservers der „zip“-Distribution ohne weitere Konfiguration und Deployments liegt bei einfachen Tests ab dem zweiten Start bei etwa acht Sekunden. Ist die Umgebungsvariable „CACHE“ auf „true“ gesetzt [2] und somit „Class-Caching“ angeschaltet, sinkt die Startzeit auf rund sechs Sekunden (siehe Listing 1). Zum Vergleich: Für einen Entwicklungsserver einer vollen Distribution (generischer Installer) lag die Startzeit mit Class-Caching ohne weitere Konfiguration und Deployments bei zehn Sekunden. Dabei wurde auch die mitgelieferte Derby-Datenbank gestartet. Gemessen wurde auf einem ThinkPad T430 Laptop mit Oracle Linux 6.4 (Oracle VM VBox Image mit vier zugewiesenen virtuellen Prozessoren). Nach dem Start stehen dann alle Dienste sofort zur Verfügung, es findet also keine nachgelagerte, bedarfsgesteuerte Initialisierung von Diensten statt.

Die Qual der Wahl: IDE-Unterstützung

Für die Entwicklung sind alle gängigen IDEs möglich. Das Oracle Enterprise Pack for Eclipse [3] erweitert Eclipse um Funktionalität bezüglich der Integration des WebLogic Server und Coherence. Neben spezifischen Projekt-Typen, der Einbindung der WebLogic Server Runtime mit JMX-Browser-Generatoren und grafischen Editoren für WebLogic-Deployment-Deskriptoren und Deployment-Pläne, gibt es auch eine WebLogic-Scripting-Tool-Konsole (WLST) und einen WLST-Skript-Editor mit Debugger, der nicht nur für Entwickler, sondern auch für Administratoren interessant ist.

Auch der JDeveloper 12.1.2 bietet eine Unterstützung für WebLogic Server 12.1.2.

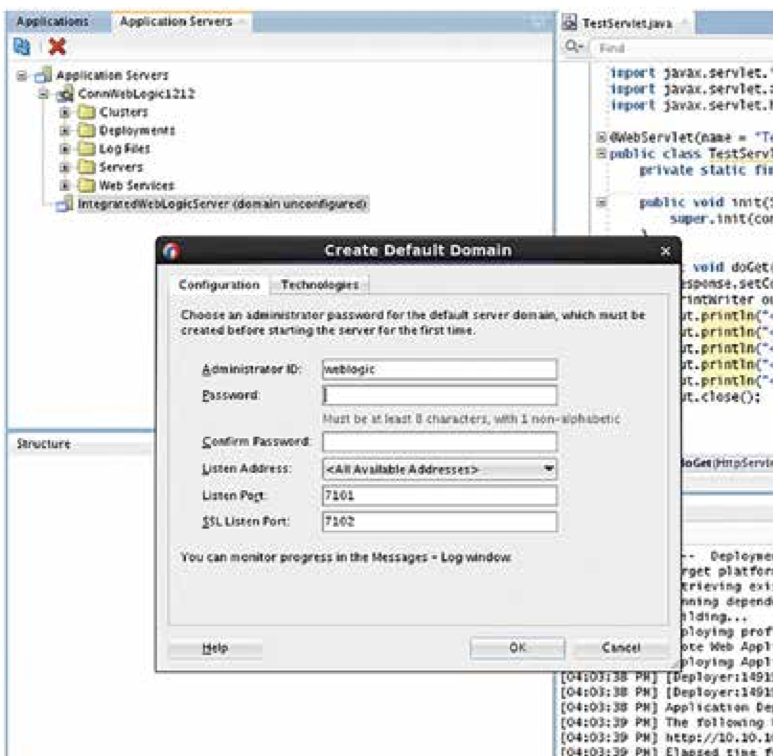


Abbildung 1: Der Dialog für die Eingabe-Parameter

Hier existieren grundsätzlich zwei Arten der Einbindung: Entweder man verwendet den integrierten WebLogic Server oder einen Stand-alone-Server. Für Letzteren muss explizit eine Application Server Connection angelegt sein. Über diese kann dann direkt ein Deployment aus dem JDeveloper heraus erfolgen. Um einen Stand-alone-Server aus der IDE heraus zu starten, kann man Start- und Stopp-Skripte als „External Tool“ sehr bequem einbinden.

Um Debugging mit einem Stand-alone-Server zu nutzen, ist dieser mit der entsprechenden Java-Option [4] zu starten. Beispielsweise würde WebLogic 12.1.2 mit der Option „-agentlib:jdwp=transport=dt_socket,address=8453,server=y,suspend=n“ den Port 8453 für Remote-Debugging zur Verfügung stellen. Auf der JDeveloper-Seite ist dann gemäß [5] vorzugehen.

Alternativ lässt sich der integrierte WebLogic Server nutzen. Die Konfiguration der Domäne wird bei der ersten Benutzung ausgeführt. Man erhält einen Dialog für die erforderlichen Eingabe-Parameter (siehe Abbildung 1). In der dann erzeugten Domäne stehen Runtime-Dienste für Technologien wie Oracle ADF sofort zur Verfügung.

Auch in NetBeans 7.4 kann WebLogic 12.1.2 integriert sein. Analog zu anderen Laufzeit-Umgebungen lassen sich WebLogic Server aus NetBeans heraus starten und stoppen, aktuelle Anwendungen und Ressourcen anzeigen und ein Deployment und Debugging direkt durchführen. Für die Benutzung von IntelliJ IDEA 13 sei auf [6] verwiesen. Dort wird Oracle WebLogic Server 12.1.2 als Server Runtime unterstützt.

Unterstützung für DevOps

Trends wie DevOps gehen auch an der WebLogic-Tool-Unterstützung nicht vorbei. Ziele von DevOps sind, operationale Prozesse effizienter, im Ausgang vorhersehbarer und insgesamt wartbarer zu machen. Diese Ziele sind durch einen starken Grad an Automatisierbarkeit erreicht [7], für den WLST sorgt. Das Tool basiert auf Jython und ist von jeher eine bewährte Methode, um Administrationsaufgaben durch Skripte bewerkstelligen zu lassen beziehungsweise durch eine interaktive Kommandozeilen-Konsole zu erledigen. WLST kann den kompletten Verwaltungsprozess automatisieren, was für Entwickler wichtig ist, denn das Erzeugen

Name	Kurzbeschreibung
appc	Erzeugt und kompiliert Klassen vor, die sonst erst zur Deployment-Zeit durch WebLogic erzeugt und validiert werden würden
create-domain	Erzeugt eine Domäne auf Basis eines Domänen-Templates
deploy	Deployment von Anwendungen und Modulen
distribute-app	Verteilung und Validierung von Deployment-Paketen auf Zielserver
install	Installation des WebLogic Server
list-apps	Listet die Namen aller Deployments auf
redeploy	Führt ein Redeployment einer laufenden Anwendung oder Teil einer laufenden Anwendung aus
start-app	Startet eine Anwendung
start-server	Startet WebLogic Server durch ein lokales Startskript; für Remote Server kann WLST in Zusammenhang mit dem Node Manager genutzt werden
stop-app	Stoppt eine Anwendung
stop-server	Stoppt WebLogic Server; für Remote Server kann WLST in Zusammenhang mit dem Node Manager genutzt werden
undeploy	Führt ein Undeployment einer Anwendung aus
uninstall	Führt eine Deinstallation der WebLogic Server Software durch (z.Zt. auf zip Distro beschränkt)
update-app	Update des Deployment-Plans einer Anwendung durch Überschreiben des Plans und Rekonfiguration der Anwendung, basierend auf dem neuen Deployment-Plan
wlst	WLST Wrapper für Maven

Tabelle 1

aller benötigten Ressourcen wie JDBC-Datasources, JMS-Destinationen, Connection Factories etc. geht mit dem Entwicklungsprozess einher. Dies löst der WebLogic Server sehr komfortabel: Ein Entwickler kann Ressourcen per Administrationskonsole GUI-basiert anlegen und die Erzeugung gleichzeitig aufzeichnen lassen. Ergebnis ist ein WLST-Fragment, das angepasst und in Skripten bequem wiederverwendet werden kann. Solche Skripte lassen sich im Gegensatz zu Ad-hoc-Kommandozeilen-Interaktionen in Version-Control-Systemen verwalten, die Ausführung ist wiederholbar, nachvollziehbar und testbar – im Gegensatz zu den genannten GUI-gesteuerten Tätigkeiten.

Ebenfalls automatisierbar ist das Anlegen von Domänen, zum Beispiel für verschiedene Umgebungen wie Entwicklung, Test und Produktion. Auch hier kommt ein in der Praxis oft bewährtes Konzept, die sogenannten „Domänen-Templates" zum Einsatz. Diese stellen parametrisierbare und erweiterbare Schablonen für Domänen-Konfigurationen dar. Sie lassen sich mittels WLST neu anlegen, auf Basis bestehender Domänen erzeugen und anpassen; es ist dann umgekehrt möglich, auf Basis eines Templates eine neue Domäne anzulegen oder eine bestehende zu erweitern. Somit ist auch hier der Domänenkonfigurations-Prozess komplett automatisierbar, nachvollziehbar und mit Versionskontrolle verwaltbar. Direkte Manipulationen in Konfigurationsdateien werden vollständig vermieden. Wird zudem Maven eingesetzt, steht für WLST ein spezielles WebLogic-Plug-in „Goal" zur Verfügung (siehe Tabelle 1).

Der komplette Deployment- und Test-Zyklus inklusive WebLogic-Installation und Domänen-Konfiguration ist somit mithilfe von Maven voll automatisierbar. Eine ausführliche Auflistung aller Goals inklusive Parameter ist in [8] und für die Produkte TopLink und Oracle Coherence in [9] beziehungsweise [10] zu finden.

Insgesamt wurde die Maven-Unterstützung für WebLogic Server, Oracle Cohe-

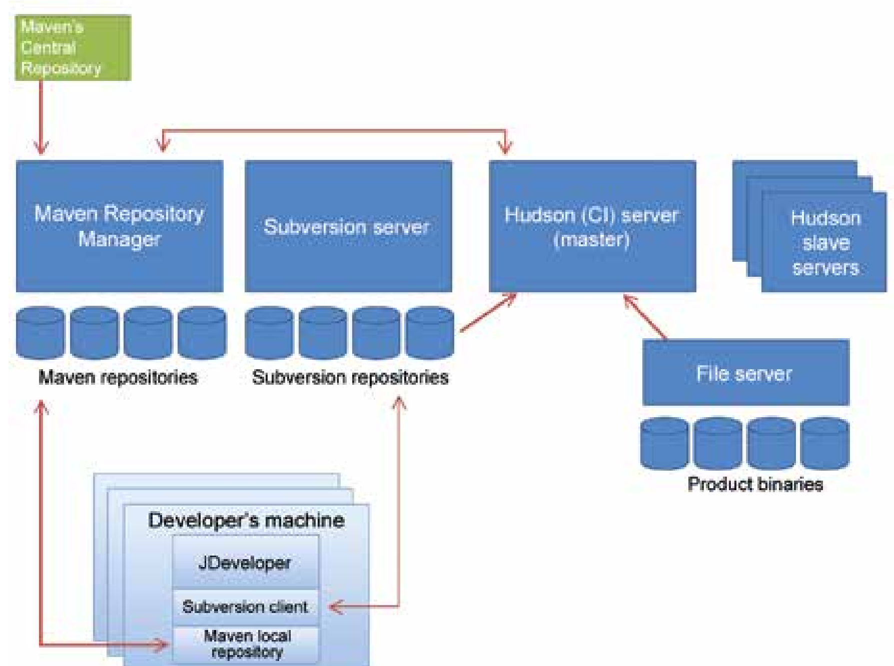


Abbildung 2: Aufbau einer Continuous-Integration-Umgebung

rence und Oracle TopLink erweitert und deren Handhabung gleichgezogen. Will man die produktspezifischen Maven Goals benutzen, muss man zunächst die Artefakte der jeweiligen Produkt-Installation in ein Maven Repository laden. Dazu wird das sogenannte „oracle maven sync Utility Plug-in" verwendet.

Es empfiehlt sich der Aufbau einer Continuous-Integration-Umgebung gemäß [11]. Abbildung 2 zeigt eine Übersicht (entnommen aus [11]) mit einem Maven Repository Manager wie Archiva, einem Version-Control-Tool (hier Subversion) und einem Continuous-Integration-Tool (hier Hudson). Der Aufbau einer solchen Umgebung und die Handhabung der einzelnen Komponenten werden in [11] sehr ausführlich dokumentiert. Auch eine Integration mit Chef oder Puppet ist möglich [12].

HTML5 & Co.

Mit Java EE 6 wurden einige Neuerungen eingeführt, um mit HTML5-Clients besser arbeiten zu können, darunter die Erzeugung von REST-Diensten auf Basis von POJOs mittels JAX-RS. Jedoch kennt erst Java EE 7 Web Sockets (RFC 6455) zur simultanen bidirektionalen Kommunikation. Diese Lücke wurde in WebLogic 12.1.2 durch ein proprietäres API [13] schon jetzt geschlossen. Damit ist es sehr einfach, mit Web Socket Clients aller Art zu interagieren. Um das API zu verwenden, ist zur Entwicklungszeit die Bibliothek „$ORACLE_HOME/wlserver/server/lib/wls-api.jar" einzubinden. Ein einfaches Beispiel für die Verwendung von Web Sockets ist mitgeliefert. Es nutzt die Klasse „WebSocketAdapter", die dem Entwickler alle Callback-Methoden zum Überschreiben zur Verfügung stellt. Im Beispiel wird nur die Methode „onMessage" überschrieben (siehe Listing 2).

Im Zusammenhang mit HTML5 sind auch Funktionalitäten interessant, die TopLink 12.1.2 bietet. TopLink 12.1.2 basiert auf EclipseLink 2.4.2 und verfügt neben der Kernfunktionalität von EclipseLink zusätzlich über kommerzielle Features wie TopLink Grid und TopLink Data Services. TopLink Data Services ermöglichen es, JPA Persistence Units ohne zusätzliches Coding als Rest Services zu exponieren. Sie unterstützen die Ausführung von Named Queries und CRUD-Operationen als

```
package examples.webapp.html5.websocket;

import weblogic.websocket.ClosingMessage;
import weblogic.websocket.WebSocketAdapter;
import weblogic.websocket.WebSocketConnection;
import weblogic.websocket.annotation.WebSocket;

import java.io.IOException;

/**
 * This listener is used to receive messages from browser, encap-
sulate
 * the received messages, and send them back to the browser.
 *
 * @author Copyright (c) 2012,2013, Oracle and/or its affiliates.
All rights reserved.
 */

@WebSocket(timeout = -1, pathPatterns = {"/ws/*"})
public class MessageListener extends WebSocketAdapter {

  @Override
  public void onMessage(WebSocketConnection connection, String
  payload) {
    // send message from browser back to client
    String msgContent = "Message \"" + payload + "\" has been re-
    ceived by server.";
    try {
      connection.send(msgContent);
    } catch (IOException e) {

      // try to resend this message again
      try {
        connection.send(msgContent);
      } catch (IOException ex) {

        // close connection this time
        try {
          connection.close(ClosingMessage.SC_GOING_AWAY);
        } catch (IOException ioe) {
          System.err.println("Fail to close connection with cli-
          ent");
          ioe.printStackTrace();
        }
      }
    }
  }
}
```

Listing 2

REST-Operationen. Als Format für den Datenaustausch mit einem Client ist sowohl XML als auch JSON verwendbar. Alles, was gemacht werden muss, ist, eine JPA Persistence Unit als „war"-Paket zu verpacken und zuvor die von TopLink mitgelieferte Bibliothek toplink-dataservices-web.jar in das Verzeichnis WEB-INF/lib zu kopieren. Der Aufruf einer Ressource erfolgt dann per Konvention. Die Basis-URI lautet da-

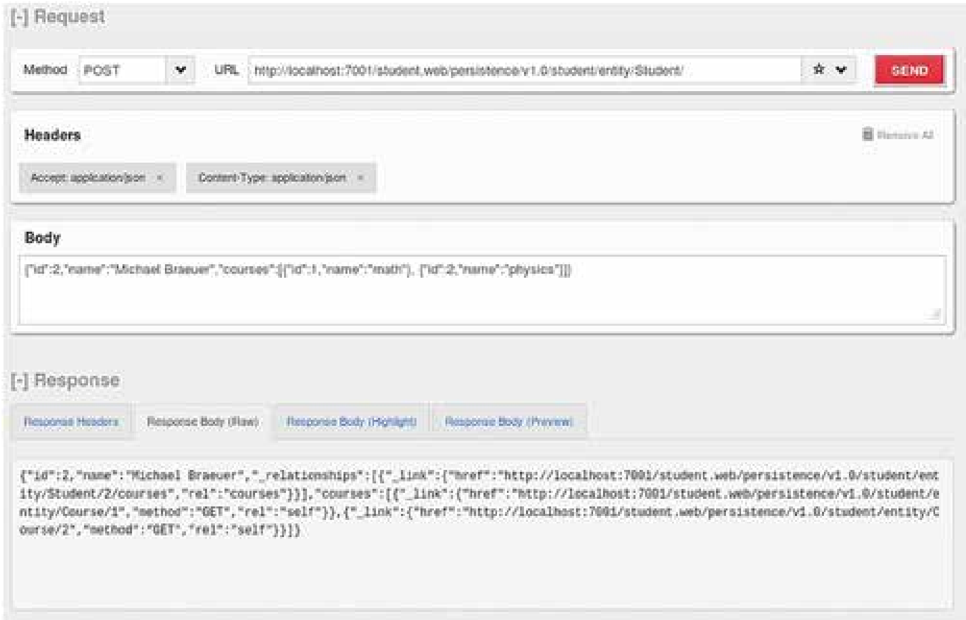


Abbildung 3: Speicherung eines Objekts

bei http://server:port/application-name/persistence/{version}, wobei für {version} in TopLink 12.1.2 „v1.0“ verwendet wird. Dann können – je nach Operation – folgende Anhänge erfolgen:

- Für Entity-Operationen „/{unit-name}/entity“
- Für Query-Operationen „/{unit-name}/query“
- Für Single Result Query „/{unit-name}/singleResultQuery“

Abbildung 3 zeigt die Speicherung eines Objekts vom Typ „Student“, der eine mehrwertige Beziehung zum Entity-Typ „Course“ besitzt. Beide gehören zu einer Persistence Unit namens „student“. Man sieht im Response das Ressourcenmodell, das zur weiteren Verarbeitung durch den Client genutzt werden kann.

Grundsätzlich sind zur Benutzung von TopLink-Data-Services neben den JPA-Annotationen keine zusätzlichen Annotationen erforderlich. Es können aber optional JAXB- und EclipseLink-spezifische Annotationen für Java-XML-Binding genutzt werden. Diese sind auch intern für das Generieren von Nachrichten im JSON-Format im Einsatz.

Natürlich kann EclipseLink JAXB und JSON-Binding auch sehr einfach für eigene Aufgaben genutzt werden. Sind Klassen mit JAXB-Annotationen versehen, so ist es einfach, daraus XML- oder auch JSON-Repräsentationen zu erzeugen und umge-

```
JAXBContext jc = JAXBContext.newInstance(Customer.class);

Marshaller marshaller = jc.createMarshaller();

// Output XML
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
marshaller.marshal(customer, System.err);

// Output JSON
marshaller.setProperty(MarshallerProperties.MEDIA_TYPE, «application/json»);
marshaller.setProperty(MarshallerProperties.JSON_INCLUDE_ROOT, true);
marshaller.setProperty(MarshallerProperties.JSON_WRAPPER_AS_ARRAY_NAME, true);
marshaller.marshal(customer, System.out);
```

Listing 3

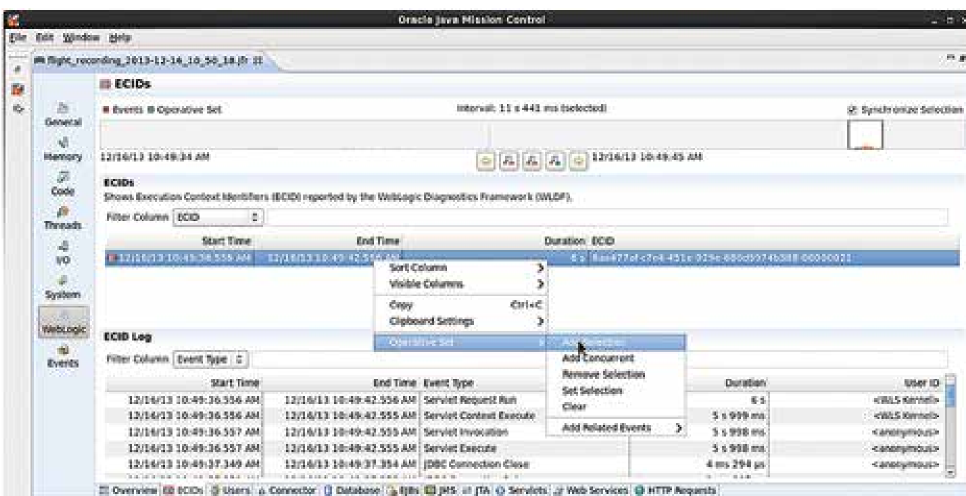


Abbildung 4: Zeitliche Auswertung

kehrt aus diesen wiederum Objektstrukturen (siehe Listing 3).

Die JAXB-Annotationen sind im selben Objektmodell möglich, das auch mit JPA-Annotationen versehen ist. EclipseLink/TopLink kann dabei mit Instrumentierung, zusammengesetzten Schlüsseln und zirkulären Beziehungen umgehen.

Diagnose und Analyse des WebLogic Servers mittels Java Mission Control

Ab JDK 7u40 steht Java Mission Control – das Gegenstück von JRockit Mission Control – für die Oracle Hotspot JVM zur Verfügung. Der JMX-Browser und der FlightRecorder von Java Mission Control bieten umfangreiche Funktionalität zur grafischen JVM-Analyse an. Dazu gehören Method Profiling, GC Profiling, Object Allocation Profiling, Latency Profiling, Thread- und Event-Analyse etc. Für WebLogic gibt es ein spezielles (zurzeit noch experimentelles) WebLogic-Plug-in [14]. Um dieses und den FlightRecorder mit WebLogic nutzen zu können, sind drei Dinge notwendig. Zunächst muss man das Plug-in der Management-Konsole explizit von der Default-mäßig eingestellten Update-Site herunterladen. Zweitens sind beim Start von WebLogic die Parameter „-XX:+UnlockCommercialFeatures-XX:+FlightRecorder“ mitzugeben.

Um Events über Laufzeitinformationen vom WebLogic Diagnostic Framework (WLDF) auszuwerten, ist drittens die Konfiguration des Servers bezüglich WLDF anzupassen. Dies erfolgt gemäß [15]. Hier sind drei Stufen für die Konfiguration der eingesammelten Informationen möglich – von „low“ (grundlegende Kennzahlen) über „medium“ bis „high“ (sehr umfangreiche Kennzahlen).

Welche der gesammelten WLDF-Event-Typen in der Management-Konsole zur eigentlichen Auswertung kommen, ist gra-

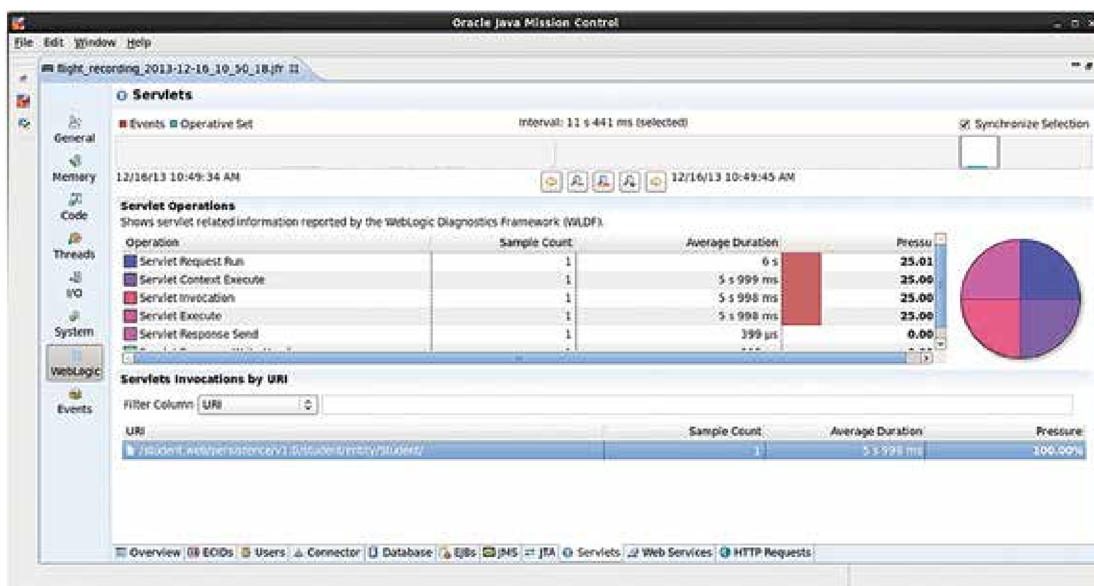


Abbildung 5: Das Beispiel

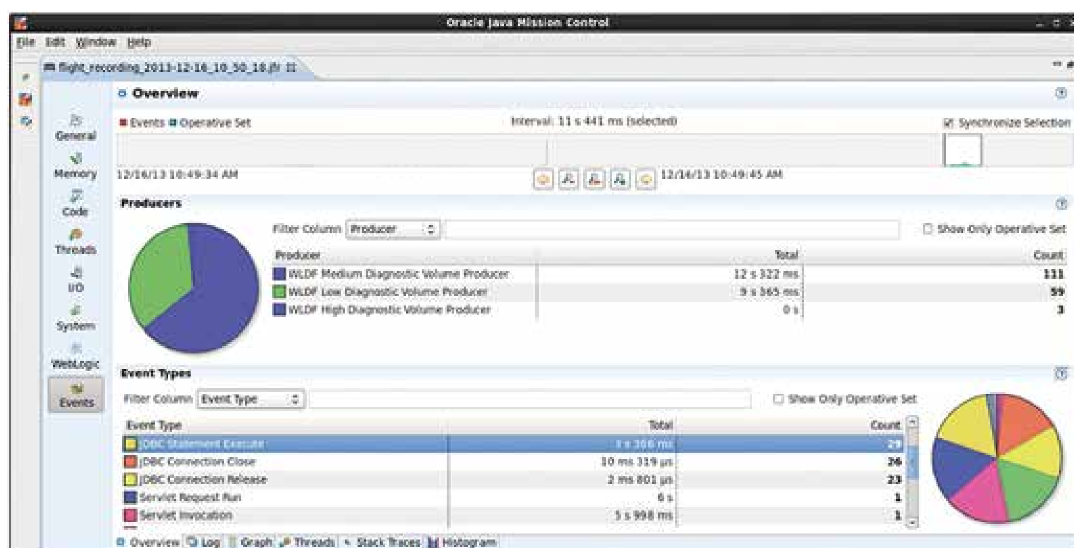


Abbildung 6: Die JDBC-Operationen

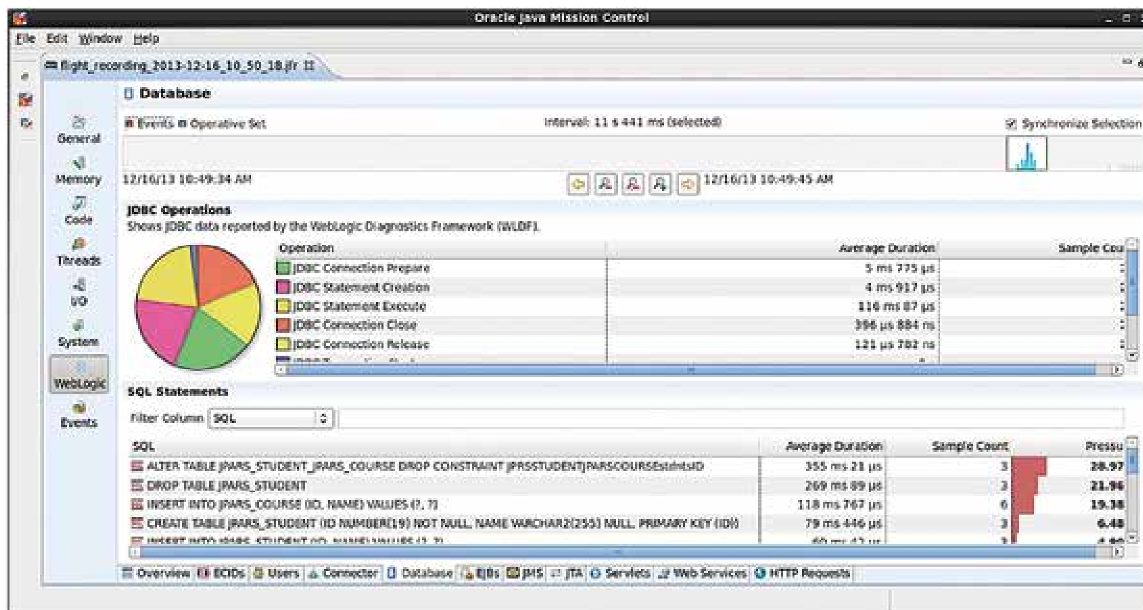


Abbildung 7: Die zugehörigen SQL-Statements

fisch auswählbar. So kann man sich anfänglich mit den Event-Typen beschäftigen, die für den konkreten Analysefall besonders interessant sind, und dann weitere, wenn benötigt, zur Auswertezeit dazunehmen, um Zusammenhänge zu betrachten. Grundsätzlich muss man nicht alle Events über einen Zeitraum zur Analyse nutzen. Die Auswertung kann zeitlich und inhaltlich („Operative Sets") eingeschränkt werden (siehe Abbildung 4). Für Korrelationen mit anderen Oracle-Produkten wie der Oracle-Datenbank wird die Execution Context ID (ECID) genutzt.

Das auf den Abbildungen 5 bis 7 dargestellte Beispiel zeigt die Analyse des erstmaligen Aufrufs einer REST-Ressource, die durch TopLink Data Services zur Verfügung gestellt wurde. Die wesentliche Zeit verbleibt in den JDBC-Operationen (siehe Abbildung 7). Analysiert man weiter die zugehörigen SQL-Statements (siehe Abbildung 8), so stellt man fest, dass am Anfang die Datenbankstrukturen neu angelegt wurden. Hier wurde wohl vergessen, die automatische Erzeugung der Datenbank-Strukturen durch EclipseLink bei Initialisierung der Persistence Unit abzuschalten.

Engere Verzahnung mit Oracle Coherence

Mit der Version 12.1.2 wurde das Zusammenspiel von Coherence-Knoten und WebLogic Managed Servern nochmals verbessert. Die Coherence-Knoten sind vollständig in die bewährte Domänen-Architektur integriert, sie können also analog zu herkömmlichen Managed Servern in einer Domäne konfiguriert und administriert werden.

Man kann dedizierte WebLogic-Knoten als Coherence Managed Server konfigurieren, indem man diesen einen zuvor erstellten Coherence Cluster zuordnet. Gewisse clusterweite Konfigurationseigenschaften werden dann den zugehörigen Managed-Server-Knoten vererbt; knotenspezifische Coherence-Eigenschaften können ebenfalls über die Administrationskonsole angepasst werden.

Als Alternative zur manuellen, GUI-basierten Konfiguration steht WLST zur Verfügung. Ebenso ist es möglich, die Coherence Managed Server über den Node-Manager zu verwalten und zu überwachen. Diese Harmonisierung scheint erst einmal für die Administration wichtig. Doch auch für Entwickler sind diese Konfigurationsvereinfachungen von großem Vorteil, denn Coherence-basierte Anwendungen sollten schon sehr früh auf Mehrknoten-Umgebungen getestet werden, um das Verhalten in verteilten Umgebungen bewerten zu können („The Network is your Enemy"). Für die Paketierung von Coherence-Anwendungen steht eine neuer Archivtyp namens „Grid Archive" (GAR) zur Verfügung. Dieser hat einen archivspezifischen Verzeichnisaufbau mit eigenen Deployment-Deskriptoren.

Referenzen und Links

[1] Oracle Fusion Middleware Supported System Configurations: http://www.oracle.com/technetwork/middleware/fusion-middleware/documentation/fmw-1212certmatrix-1970069.xls

[2] Configuring Class Caching: http://docs.oracle.com/middleware/1212/wls/WLPRG/classloading.htm#i1074000

[3] Oracle Enterprise Pack for Eclipse 12.1.2.1.1: http://www.oracle.com/technetwork/developer-tools/eclipse/downloads/index.html

[4] Java Remote Debugging: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/conninv.html

[5] JDeveloper Remote Debugging: http://docs.oracle.com/middleware/1212/jdev/OJDUG/run_debug_java.htm#BABBDICF

[6] Nutzung von WebLogic Server 12.1.2 mit IntelliJIDEA: http://www.jetbrains.com/idea/whatsnew/index.html

[7] DevOps auf Wikipedia: http://en.wikipedia.org/wiki/DevOps
WebLogic, Coherence und TopLink 12.1.2 Maven Goals und Benutzung:

[8] http://docs.oracle.com/middleware/1212/wls/WLPRG/maven.htm#CHEFFGFH

[9] http://docs.oracle.com/middleware/1212/core/MAVEN/coherence_project.htm
[10] http://www.oracle.com/technetwork/middleware/toplink/documentation/usingmavenwithoracle-toplink-1971073.pdf
[11] Developing Applications Using Continuous Integration: http://docs.oracle.com/middleware/1212/core/MAVEN/
[12] Hands-On zur Integration mit Puppet und Chef: http://blog.vennster.nl/2013/12/hands-on-lab-material-available-for.html
[13] WebLogic Web Socket API: http://docs.oracle.com/middleware/1212/wls/WLAPI/weblogic/websocket/package-summary.html und http://docs.oracle.com/middleware/1212/wls/WLPRG/websockets.htm#BABDCFDD
[14] WebLogic Experimental Plug-In for Java Mission Control: http://docs.oracle.com/javase/7/docs/technotes/guides/jmc/experimental.html#BABBBICD
[15] Konfiguration des WLDF Diagnostic Volume: http://docs.oracle.com/middleware/1212/wls/WLDFC/config_diag_images.htm#CJABCIFA

Michael Bräuer
michael.braeuer@oracle.com

Michael Bräuer ist für die ORACLE Deutschland B.V. & Co. KG in der Business Unit Server Technologies Fusion Middleware tätig. Zu seinen Schwerpunktthemen gehören serverseitige Java-Entwicklung und der Oracle WebLogic Server.

30.01.2014
DevCamp
Let's play together

Erstes DOAG DevCamp – Es hat gerockt.

Sicherlich konnte man es nicht mit einer Fußball-WM vergleichen, auch nicht mit der Finalrunde des DFB-Pokals. Das DOAG DevCamp 2014 war dennoch eine schöne Begegnung unter Gleichgesinnten; eine Veranstaltung mit Tempowechsel, ausgewogenem Ballbesitz und dynamischem Spielsystem; ein Derby, bei dem sogar APEX- und ADF-Anhängern Freundschaft geschlossen haben. Alle kamen gespannt in die Münchner Allianz Arena. Eine Veranstaltung ohne festes Programm mit dem Thema des Tages: Moderne Softwareentwicklung im Oracle-Umfeld.

Die Regeln für das DevCamp waren ganz simpel.

- Alle Teilnehmer waren für diesen Tag per du
- Jeder konnte in eine Session kommen und diese verlassen, wann immer er wollte
- Fand man keine Session, die einen interessiert, ging man zum Networking in den Catering-Bereich
- Was immer in einer Session geschah, das geschah
- Die Session beginnt, wann sie beginnt, und endet, wann sie endet

Man munkelt, es ging dann nach dem Abpfiff in die Verlängerung… Danke für das hervorragende Zusammenspiel! Wir treffen uns 2015 –gleiche Regeln, neues Spiel!

Für unser Mini-Wintermärchen haben wir eine elektronische Fanmeile ins Leben gerufen und die reguläre Spielzeit dokumentiert.

Wer das DOAG DevCamp verpasst hat oder wem es gefallen hat, der wird bestimmt auch mit JavaLand und der DOAG 2014 Development warm.

Da wird's noch rocken:
Zwei Tage für Java-Nerds mit vollem Java-Programm und Unmengen an Community-Aktivitäten. www.javaland.eu

Automatisierte Web-Tests mit Selenium 2

Mario Goller, Trivadis AG

„Online first!" ist ein wesentlicher Gedanke, der in der Software-Entwicklung Einzug gehalten hat. Der Trend zu immer komplexeren Web-Anwendungen zeigt auch das Bedürfnis nach ausreichender Qualitätssicherung. Es gibt am Markt mittlerweile eine Reihe kommerzieller Tools zur Testautomatisierung, die umfangreiche Features für den Anwender bieten. Der Artikel blickt speziell auf den Open-Source-Bereich und geht der Frage nach, welche Möglichkeiten der Java-Entwickler hat, um eigene Web-Tests zu definieren und diese auch in den Build-Prozess zu integrieren. Die Vorstellung des Test-Frameworks Selenium beantwortet unter anderem diese Fragestellung.

Wenn man heute moderne Web-Applikationen betrachtet, so bestehen diese kaum mehr nur aus statischem HTML und etwas Scripting auf der Server-Seite. Vielmehr ist auch der Client, also der Web-Browser, die Laufzeit-Umgebung für komplexe und hoch dynamische Webseiten. Es muss hier auf eine Vielzahl von unterschiedlichen Benutzer-Interaktionen reagiert, Business-Logik ausgeführt oder das Aussehen und Layout der Seite entsprechend der Auflösung angepasst werden können.

Der Vormarsch von Rich Internet Applications (RIA) verdeutlicht diese Entwicklung. Die Herausforderung besteht nicht nur in der eigentlichen Implementierung solcher Applikationen, sondern schließt natürlich auch den Testprozess mit ein. Eine Qualitätssicherung muss in diesem Kontext nun unterschiedliche Aspekte berücksichtigen.

Zum einen steht Web-Entwicklern heute eine Vielzahl von Programmiersprachen, Frameworks und Bibliotheken zur Verfügung, die die Basis einer Applikation bilden und unterschiedlich arbeiten. Zum anderen muss eine Web-Anwendung heute verschiedene Web-Browser in unterschiedlichen Versionen unterstützen, also deren Besonderheiten in der Interpretation und Darstellung von Web-Inhalten berücksichtigen. Darüber hinaus darf man auch nicht vergessen, dass diese Browser nicht nur auf einer, sondern auf verschiedenen Plattformen und Betriebssystemen verfügbar sind. Durch die zunehmende Bedeutung mobiler Endgeräte ist hier eine zusätzliche Umgebung für die Ausführung einer Web-Anwendung entstanden, die nicht zu vernachlässigen ist.

Betrachtet man all diese Aspekte, so erscheint es wichtig, unterschiedliche Browser und Plattformen beim Testen zu berücksichtigen, um aussagekräftige Resultate über die Qualität und Kompatibilität der eigenen Web-Anwendung zu erhalten. Schließlich soll diese für möglichst viele Benutzer verfügbar sein und natürlich auf jeder Plattform fehlerfrei funktionieren.

Testen mit Selenium

Selenium [1] ist ein Framework und zugleich ein API, das ursprünglich von Programmie-

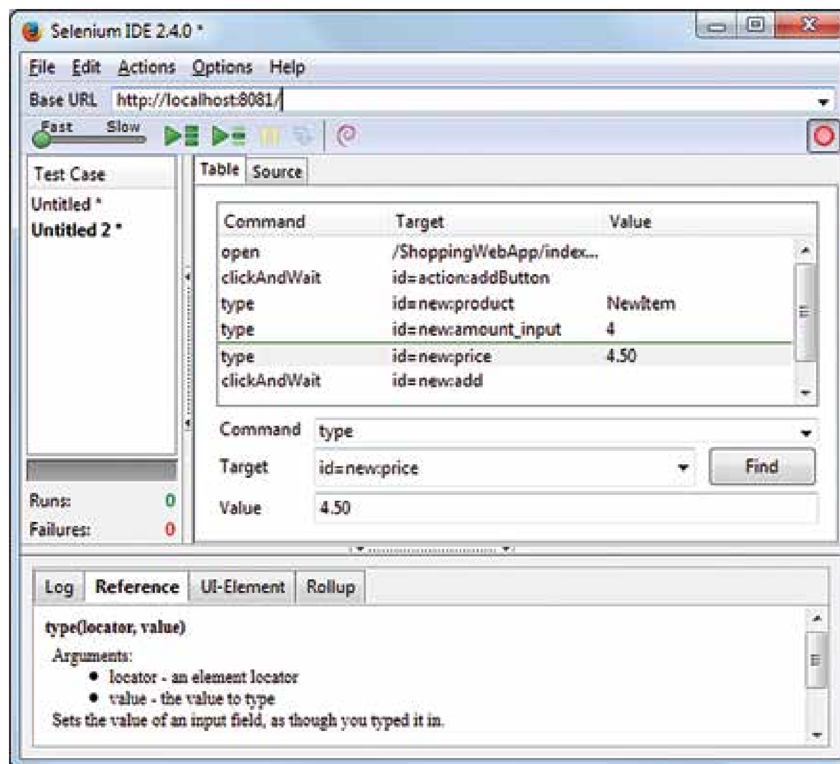


Abbildung 1: Das Firefox-Plug-in der Selenium IDE

rern der Firma ThoughtWorks entwickelt wurde und heute von Google vorangetrieben wird. Es vereint mehrere Komponenten und kann damit als Basis für automatisierte Tests von Web-Applikationen dienen. Besonders Entwickler und agile Entwicklerteams können von diesen Funktionen zur Testautomatisierung profitieren.

Selenium stellt sein Interface zur Test-Definition und -Ausführung für unterschiedliche Programmier- und Skript-Sprachen zur Verfügung, darunter Java, C#, Python und PERL. Mit Version 2 ist dieses API unter dem Namen „WebDriver" bekannt und für alle gängigen Web-Browser implementiert.

Im Wesentlichen umfasst Selenium folgende vier Hauptkomponenten, die auch unabhängig voneinander genutzt werden können:

- Selenium IDE
- WebDriver
- Selenium RemoteControl (RC)
- Selenium GRID

Selenium IDE

Für den Einstieg in Selenium und dessen Funktionalitäten empfiehlt sich die Installation der Selenium IDE. Diese ist als Plug-in für den Firefox Browser [4] verfügbar und kann dort nach der Installation unter dem Menüpunkt „Tools" gestartet werden (siehe Abbildung 1).

Mithilfe eines Record-Buttons lassen sich alle Interaktionen auf der gerade geöffneten URL aufzeichnen. Sie werden als Kommandos abgespeichert, tabellarisch aufgelistet und sind anschließend durch die Play-Funktion erneut ausführbar. Auf diesem Weg kann man sich auf sehr einfache Weise die Tests zu seiner Web-Applikation ohne eigene Programmierung erstellen und anschließend auch in unterschiedliche Formate exportieren. So lässt sich zum Beispiel mit der Selenium IDE aus den aufgezeichneten Aktionen eine vollständige JUnit-Testklasse in Java erstellen und diese dann sofort in den eigenen Test-Code integrieren.

```
<dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-firefox-driver</artifactId>
      <version>2.35.0</version>
</dependency>
```

Listing 1: Ausschnitt Maven-Konfiguration („pom.xml")

```
@Before
public void setUp() {
  driver = new FirefoxDriver();
  baseUrl = "http://localhost:8080/";
  driver.manage().timeouts()
.implicitlyWait(5,TimeUnit.SECONDS);
}
```

Listing 2: WebDriver-Initialisierung

```
<form name="loginForm">
    <label for="username">UserName: </label>
    <input type="text" id="username" /><br/>
    <label for="password">Password: </label>
    <input type="password" id="password" /><br/>
    <input name="login" type="submit" value="Login" />
</form>
```

Listing 3: Beispiel einer HTML-Login-Form

```
@Test
public void testLogin(){
  url = baseUrl + "/login.html";
  driver.get(url);
  WebElement username = driver.findElement(By.id("username"));
  assertTrue(username.isDisplayed());
  List<WebElement> labels =
      driver.findElements(By.tagName("label"));
  assertEquals("2 Labels erwartet", 2, labels.size());
}
```

Listing 4: JUnit-Selenium-Test

Projekt Setup

Um Selenium einzusetzen, sind je nach Programmiersprache unterschiedliche Bibliotheken erforderlich. Im Falle von Java wird ein Set von „Jar"-Files benötigt, die in den „Classpath" aufzunehmen sind. Für Projekte, die mit Maven arbeiten, ist dies einfach durch die Angabe der entsprechenden WebDriver-Implementierung zu erledigen. Die zusätzlich benötigten Selenium-Libraries werden dann automatisch aufgelöst. Listing 1 zeigt die entsprechende Konfiguration in Maven für einen Firefox-WebDriver.

Der Ausgangspunkt für jeden Selenium-Test ist die Erzeugung und Bereitstellung eines WebDriver. Je nach Browser, auf dem die Web-Applikation getestet werden soll, ist ein anderer WebDriver zu instanziieren. Dies ist nur einmal erforderlich und geschieht typischerweise in der globalen SetUp-Me-

thode für alle Tests. In unserem Beispiel werden die Tests mit einem installierten Firefox ausgeführt und ein entsprechender Firefox-WebDriver erzeugt (siehe Listing 2).

WebDriver im Einsatz

Ist der WebDriver bereitgestellt, so steht mit ihm eine Art „Fernbedienung" des Browsers zur Verfügung. Man kann damit URLs laden oder alle möglichen Benutzer-Interaktionen auf der im Browser geladenen Webseite durchführen, wie einen Button betätigen, ein Textfeld mit Inhalt füllen oder aus einer Drop-Down-Liste einen Eintrag auswählen (siehe Listing 3).

Voraussetzung für solche Interaktionen ist jedoch, dass der WebDriver die entsprechenden Elemente auf einer HTML-Seite genau lokalisieren kann. Dem WebDriver muss also mitgeteilt werden, wie er die HTML-Elemente finden kann. Hierzu ruft man die Methode „findElement" auf und wählt eine sogenannte „Locator-Strategie". Konkret bedeutet dies, dass man der Methode eine Instanz der Klasse „By" übergibt, die über eine statische Factory-Methode erzeugt werden kann.

Soll jetzt zum Beispiel das Element über dessen ID gesucht werden, so wird die Methode „By.id" mit der jeweiligen Element-ID aufgerufen. Listing 4 zeigt einen solchen Aufruf. Als Ergebnis liefert uns die Methode eine Instanz der Klasse „WebElement" zurück, die somit ein Element innerhalb der Webseite repräsentiert.

Es ist zu beachten, dass im Falle einer nicht eindeutigen Selektion immer das erste Element auf der Seite zurückgeliefert wird, das dem Suchkriterium entspricht. Erwartet man mehrere passende Elemente, beispielsweise mit der Selektion per „Class-" oder „Tag-Name", so bietet sich die Methode „findElements" an, die immer eine Liste von „WebElement" liefert. Eine Dokumentation zum Selenium-Java-API und eine Übersicht aller Locator-Strategien findet man unter [3].

Bei der Wahl der richtigen Strategie ist immer zu berücksichtigen, dass die Selektoren im Browser unterschiedlich schnell arbeiten. Eine Selektion per Identifier ist (wann immer möglich) einer Suche mit „Tag-Name" oder einem „XPath"-Ausdruck vorzuziehen. Der Grund liegt in den Browser-Implementierungen, die einen effizienten Zugriff per „ID"-Attribut ermöglichen.

```
@Test
public void testLoginSubmit(){
 WebElement username = driver.findElement(By.id("username"));
 username.sendKeys("admin");
 WebElement password = driver.findElement(By.id("password"));
 password.sendKeys("letMeIn");
 WebElement button = driver.findElement(By.id("submit"));
 button.click();
 ...
 Actions builder = new Actions(driver);
 Action dragAndDrop = builder.clickAndHold(sourceElement)
.moveToElement(targetElement)
.release(otherElement)
.build();
 dragAndDrop.perform();
 ...
}
```

Listing 5: Benutzer-Interaktionen

```
public class LoginPage {
      @FindBy(id = "username")
      public WebElement usernameField;
      @FindBy(id = "password")
      public WebElement passwordField;
      @FindBy(name = "login")
      public WebElement loginButton;
      public LoginPage(WebDriver driver) {
          PageFactory.initElements(driver, this);
      }
public void doLoginAs(String username, String password) {
usernameField.sendKeys(username);
passwordField.sendKeys(password);
loginButton.click();
      }
}
```

Listing 6: Page-Objekt-Definition

Und … Action!

Nachdem man bestimmte Elemente auf unserer Web-Seite mit dem WebDriver lokalisieren kann, geht es im nächsten Schritt darum, mit diesen Elementen zu interagieren. Das Ziel ist, möglichst genau das Verhalten einer Benutzer-Interaktion mit dem Web-Frontend im Browser zu simulieren. Zu diesem Zweck stellt das WebDriver-API verschiedene Methoden zur Verfügung, die man je nach gewünschter Interaktion auf dem WebElement anwenden kann. Im Wesentlichen unterscheidet man zwischen Keyboard- und Mouse-Action:

- Keyboard-Interface
  - sendKeys(CharSequence … keysToSend)
  - pressKey(Keys keyToPress)
  - releaseKey(Keys keyToRelease)
- Mouse-Interface
  - click(WebElement onElement)
  - doubleClick(WebElement onElement)
  - mouseDown(WebElement onElement)
  - mouseUp(WebElement onElement)
  - mouseMove(WebElement toElement)

```
@Test
public void testLoginSubmit(){
  WebDriver driver = new FirefoxDriver();
  url = baseUrl + "/login.html";
  driver.get(baseUrl);
  LoginPage loginPage = new LoginPage(driver);
  loginPage.doLoginAs("admin", "letMeIn");
  ...
}
```

Listing 7: JUnit-Test mit Page-Objekten

```
$ java -jar selenium-server-standalone.jar -role node
  -hub http://hubHost:4444/grid/register
  -browser browserName="firefox",platform=LINUX
```

Listing 8: Start-Parameter

```
< Selenium_GRID.tif>
Abbildung 2: Übersicht Selenium GRID

@Before
public void setUp(){
DesiredCapabilities capability =
       DesiredCapabilities.firefox();
// Browser auf dem wir testen
capability.setBrowserName("firefox");
// die zugehörige Plattform auf der wir testen wollen
capability.setPlatform("LINUX");

driver = new RemoteWebDriver(new URL(
"http://hubHost:4444/wd/hub"), capability);

driver.get("http://localhost:8080/login.html");
...
}
```

Listing 9: RemoteWebDriver

Listing 5 zeigt deren Anwendung an einem einfachen Beispiel. Sind komplexere Interaktionen notwendig, bei denen zum Beispiel mehrere Aktionen hintereinander ausgeführt werden müssen, kann man die im API vorhandene „Actions"-Klasse nutzen. Damit lassen sich durch verkettete Methoden-Aufrufe die bereits bekannten Aktionen hintereinanderschalten und zu einer Interaktion verbinden. So ist unter anderem eine „Drag & Drop"-Aktion möglich.

Page Object Model

Wenn man die Implementierung der Selenium Tests betrachtet, fällt auf, dass eine Mischung vorliegt von eigentlichem Testcode, der das Verhalten der Web-Applikation testet, und Code zur Steuerung des Browsers mithilfe des WebDriver. Es ist zudem sichtbar, dass an vielen Stellen immer wieder die gleichen Aktionen ausgeführt werden müssen, wie das Suchen eines Elements und das Starten von Interaktionen mit diesem. Schöner wäre es hier, diesen Code für den eigentlichen Test zu verbergen und in eigene Klassen auszulagern, die so auch wiederverwendet werden können.

Genau dieser objektorientierte Ansatz setzt das Page Object Model Pattern um. Konkret bedeutet das, dass jede zu testende Applikationsseite als eigene Klasse („Page Object") modelliert wird. Die vorhandenen WebElemente werden darin gekapselt und Methoden angeboten, die der Seite die möglichen Funktionen und Interaktionen zur Verfügung stellen.

Listing 6 zeigt die Definition eines solchen Page Object. Hier kann man zudem die „FindBy"-Annotation von Selenium nutzen, die an die jeweiligen WebElemente gesetzt ist. Als Parameter wird die bekannte Locator-Strategie angegeben, die für die Elementsuche erforderlich ist. Um eine automatische Initialisierung der Elemente anzustoßen, nutzt man die Klasse „PageFactory" und die Methode „initElements", die dem aktuellen WebDriver samt dem jeweiligen Page-Objekt als Parameter übergeben werden. Listing 7 zeigt, dass die Testklasse durch die Verwendung der erzeugten Page-Objekte übersichtlicher geworden ist.

Selenium GRID

Nachdem bisher immer davon ausgegangen wurde, dass die Applikationstests zusammen mit dem Web-Browser auf der gleichen Maschine gestartet werden, wird im Folgenden gezeigt, wie mit dem Selenium GRID [2] ein solcher Test auch auf Browsern möglich ist, die auf einem anderen Host laufen. Dies bietet zum einen die Möglichkeit, einen Web-Test auf unterschiedlichen Plattformen beziehungsweise Betriebssystemen und Browsern auszuführen, und zum anderen kann diese Infrastruktur benutzt werden, um ein großes Set von durchzuführenden Tests auf verschiedene Hosts zu verteilen und somit parallel auszuführen.

Selenium GRID unterscheidet grundsätzlich zwei Arten von Server-Instanzen: den Selenium Hub und eine oder mehrere Selenium-Remote-Control-Instanzen (RC). Der Hub übernimmt hierbei die zentrale Vermittlungsrolle zwischen dem auszuführenden Test und einem Selenium RC, der den eigentlichen Browser ausführt und steuert. Ein Selenium Hub wird auf einem beliebigen Host über das „selenium-standalone.jar" gestartet, das in aktueller Ver-

sion unter [1] bezogen werden kann. Der Aufruf auf der Kommandozeile sieht dann folgendermaßen aus: „$ java –jar selenium-server-standalone.jar –role hub".

Das Starten eines Selenium RC geschieht ähnlich. Als Parameter muss zusätzlich der Name des Host mit dem gestarteten Selenium Hub angegeben werden sowie optional die auf diesem Node vorhandenen Browser-Konfigurationen (siehe Listing 8).

Der Selenium RC verbindet sich nun automatisch mit dem Hub und wird dort mit der angegebenen Konfiguration registriert. Auf dem Hub kann dies über die Web-Konsole „http://hubHost:4444/grid/console" geprüft werden. Möchten man nun im Test nicht auf einem lokalen Browser arbeiten, sondern eine der im GRID vorhandenen Instanzen nutzen, so ist die Initialisierung des WebDriver etwas anzupassen (siehe Listing 9).

Es muss nun eine „RemoteWebDriver"-Instanz erzeugt werden, die man mit der gewünschten Browser und der Plattform konfiguriert. Der WebDriver kommuniziert nun über den Selenium Hub, der entsprechend der im WebDriver angegebenen Konfiguration einen passenden, am Hub registrierten Node sucht. Ist ein solcher Node registriert, so wird die eigentliche Browsersteuerung dann an diesen weitergeleitet. Abbildung 2 zeigt das Zusammenspiel aller GRID-Komponenten. Für die Verwendung eines RemoteWebDriver muss also die eigentliche Test-Implementierung nicht angepasst werden. Somit kann der gleiche Testcode sowohl lokal als auch im Selenium GRID ausgeführt werden.

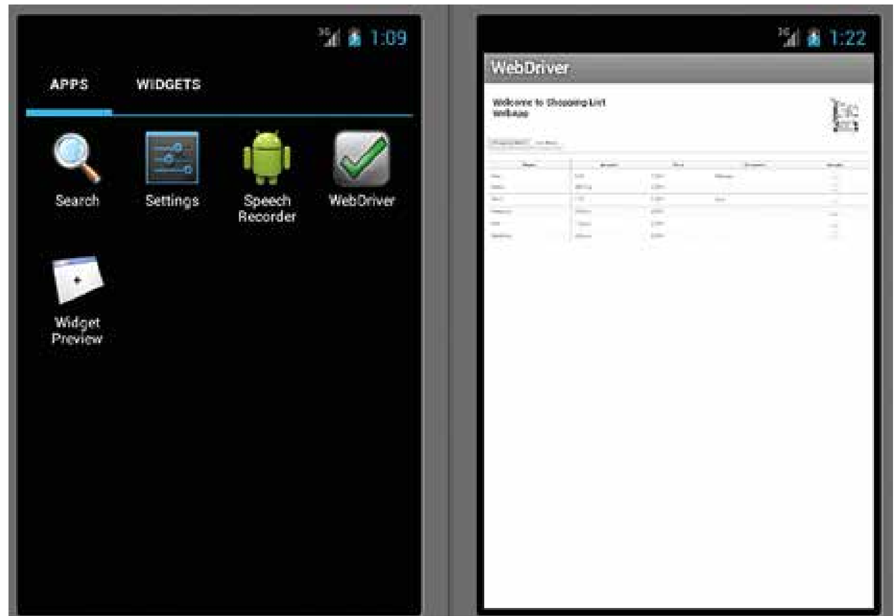


Abbildung 3: WebDriver auf dem Android-Gerät

Selenium goes Mobile

Wie bereits erwähnt, bietet Selenium nicht nur die Basis für automatisierte Tests einer Web-Applikation, sondern ermöglicht auch das (parallele) Testen auf unterschiedlichen Browsern und Plattformen. Zu den relevanten Plattformen zählen mittlerweile auch Mobile Devices. Für diesen Fall bietet Selenium mit „AndroidDriver" und „IphoneDriver" zwei weitere WebDriver-Implementationen an.

Zur Verwendung muss dazu auf dem Gerät ein entsprechendes Paket – ähnlich einer App – installiert sein [5]. Dabei spielt es keine Rolle, ob es sich um ein echtes Device oder einen Emulator handelt. Als Resultat kann über die WebDriver-App nun ein Browser auf dem Gerät gestartet werden, der wiederum über unseren Testcode gesteuert werden kann. Auch hierbei ist die Implementierung des Tests nicht anzupassen, es wird lediglich eine andere Instanz des WebDriver zur weiteren Verwendung initialisiert (siehe Abbildung 3).

Fazit

Mit Selenium steht dem Entwickler ein mächtiges und flexibles Framework zur automatisierten Testausführung zur Verfügung. Besonders durch das einfach zu verwendende WebDriver-API und die Nutzung von Page-Objekten ist es möglich, sehr schnell und objektorientiert Tests zu implementieren, die die relevanten Aspekte der eigenen Web-Applikation abdecken.

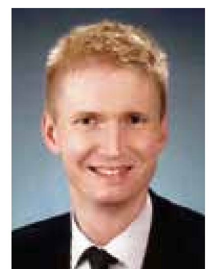
Das Selenium GRID kann zudem unterschiedliche Umgebungen (Browser und Plattformen) für die Testausführung definieren und sogar verteilt auf mehrere Hosts ausführen. Dies ist besonders für die agile Entwicklung relevant, um schnell Feedback aus dem letzten Build zu erhalten.

Anwender ohne Programmier- oder Scripting-Know-how können lediglich auf Selenium-IDE zurückgreifen, werden hier allerdings kein umfangreiches Tool zur Erstellung und Verwaltung von Tests vorfinden. Hier sind andere (kommerzielle) Testsuiten sicher im Vorteil.

Links

[1] http://code.google.com/p/selenium
[2] http://docs.seleniumhq.org
[3] http://selenium.googlecode.com/git/docs/api/java/index.html
[4] http://release.seleniumhq.org/selenium-ide
[5] http://code.google.com/p/selenium/wiki/AndroidDriver

Mario Goller
mario.goller@trivadis.com



Mario Goller arbeitet als Senior Consultant bei der Trivadis AG in Zürich und ist seit dem Jahr 2004 Software Engineer im Java-Umfeld. Schwerpunkt seiner Tätigkeit ist die Konzeption und Entwicklung von Web-Applikationen und Business-Anwendungen auf Basis von JEE.

Contexts und Dependency Injection – die grundlegenden Konzepte

Dirk Mahler, buschmais GbR

Die Voraussetzung zur Nutzung von Contexts und Dependency Injection (DI/CDI) ist natürlich die Verwendung eines entsprechenden Containers. Dies kann ein vollständig Java-EE-6-kompatibler Application-Server sein (etwa JBoss AS 7.x), für kleinere Ansprüche stehen JBoss Weld (Referenz-Implementierung) und Apache OpenWebBeans als Stand-Alone-Bibliotheken zur Verfügung. Als Einstieg in den Artikel empfiehlt sich der Beitrag „Contexts und Dependency Injection – der lange Weg zum Standard" aus der letzten Ausgabe.

Ein Container durchsucht während der Deployment- beziehungsweise Initialisierungs-Phase den Klassenpfad der Anwendung nach sogenannten „Bean Archives". Das sind JAR- beziehungsweise WAR-Archive, die durch die Existenz eines XML-Deskriptors namens „beans.xml" im Verzeichnis „META-INF/" beziehungsweise „WEB-INF/" für WAR-Archive gekennzeichnet sind (siehe Listing 1). Damit wird signalisiert, dass der Container die Instanzen der darin enthaltenen Klassen als Beans betrachten und verwalten soll. Das betrifft sowohl die Erzeugung als auch die Injizierung benötigter Abhängigkeiten. Der Deskriptor selbst kann leer sein, er dient in erster Linie als Marker.

Bean und Injection-Point

Jede in einem Bean-Archive enthaltene Klasse ist potenziell eine Bean, die durch den Container verwaltet werden kann, sofern sie vorgegebenen Konventionen entspricht. Dies ist bereits der Fall, wenn sie über einen nicht privaten Default-Konstruktor verfügt. Sogenannte „Plain Old Java Objects" (POJO) erfüllen im Normalfall diese Bedingung – im einfachsten Fall sind keine weiteren Annotationen oder Deskriptoren notwendig. Benötigt eine Bean weitere Abhängigkeiten (wie Dienste), können diese mit der Annotation „@ Inject" als sogenannte „Injection Points" definiert werden. Dazu gibt es verschiedene Möglichkeiten, am gebräuchlichsten ist die Verwendung von Instanz-Variablen („Field-Injection", siehe Listing 2).

Fordert „ClientImpl" eine Instanz am Container an, wird sie dort erzeugt und die darin mit „@Inject" annotierten Variablen werden mit verfügbaren Instanzen der jeweiligen Typen initialisiert. Dazu muss in der Gesamtheit aller Bean-Archive des Klassenpfads exakt eine zu diesem Typ zuweisungskompatible und durch den Container verwaltete Bean existieren. Im einfachsten Fall gibt es schlicht eine Klasse, die das Interface „IService" implementiert.

Sie kann natürlich wiederum über Injection Points verfügen. Der so entstehende – oft recht komplexe – Abhängigkeitsgrad wird durch den Container aufgelöst. Er ist übrigens dazu verpflichtet, bereits

```
serviceApi.jar:
      /com/buschmais/service/api/IService.java
      /META-INF/beans.xml
serviceImpl.jar:
      /com/buschmais/service/impl/ServiceImpl.java
      /META-INF/beans.xml
client.war:
      /WEB-INF/beans.xml
      /WEB-INF/web.xml
      /WEB-INF/classes/com/buschmais/client/ClientImpl.java
```

Listing 1

```
ClientImpl.java:
public class ClientImpl implements IClient {
    @Inject
    private IService service;
    …
}
```

Listing 2

während der Initialisierung der Anwendung das Vorliegen möglicher Inkonsistenzen zu prüfen. Der Sinn besteht darin, Probleme so früh wie möglich – also beim Deployment und nicht erst zur Laufzeit – zu erkennen.

Das wäre im beschriebenen Beispiel der Fall, wenn keine oder mehr als eine Implementierung von „IService“ als Bean zur Verfügung stehen. Der Container würde die Initialisierung mit einer Fehlermeldung abbrechen, die auf eine unbefriedigte Abhängigkeit („unsatisfied dependency“) beziehungsweise eine mehrdeutige Abhängigkeit („ambiguous dependency“) hinweist.

Qualifier

In größeren Anwendungen lassen sich die letztgenannten, Typ-bezogenen Mehrdeutigkeiten nicht vermeiden. Sie müssen durch „Qualifizierung“ aufgelöst werden. Dazu definiert man im Anwendungscode entsprechende Annotationen, die an den zu erzeugenden Beans und Injection Points verwendet werden. Die Annotationen sind mit der Meta-Annotation „@Qualifier“ zu versehen (siehe Listing 3).

Die Qualifier-Annotationen ergänzen also Typ-Definitionen, sind aber von ihnen unabhängig. Es kommt daher häufig vor, dass ein und dieselbe Qualifier-Annotation auf verschiedene Vererbungs-Hierarchien angewendet wird. Dies ist etwa der Fall, wenn mehrere technische Dienste (wie EntityManager und Datasource) jeweils für unterschiedliche fachliche Belange zur Verfügung stehen sollen. Die fachliche Trennung erfolgt dann anhand entsprechender Qualifikation (wie „@UserData“ und „@ShopData“). Darüber hinaus ist es möglich, Attribute in Qualifier-Annotationen zu deklarieren, deren Werte ebenfalls zur Auflösung herangezogen werden (siehe Listing 4).

Producer

Oft ist es notwendig, Instanzen benötigter Abhängigkeiten im Anwendungscode selbst erzeugen zu können. Ein Beispiel dafür könnten die erwähnten Instanzen eines JPA-EntityManager oder einer JDBC-Datasource sein, für die keine Beans als eigenständige Klassen in einem Bean-Archiv zur Verfügung stehen. In diesem Fall kommen sogenannte „Producer-Felder“ beziehungsweise „-Methoden“ zum Einsatz. Der Container erkennt sie an der Annotation „@Produces“.

```
InMemory.java:
      @Qualifier
      public @interface InMemory {
      }
Persistent.java:
      @Qualifier
      public @interface Persistent {
      }
InMemoryServiceImpl.java:
      @InMemory
      public class ServiceImpl implements IService {
          ….
      }
PersistentServiceImpl.java:
      @Persistent
      public class ServiceImpl implements IService {
          ….
      }
ClientImpl.java:
      public class ClientImpl implements IClient {
          @Inject
          @InMemory
          private IService service;
          …
}
```

Listing 3

```
InMemory.java:
      @Qualifier
      public @interface InMemory {
          boolean transactional();
      }
TransactionalInMemoryServiceImpl.java:
      @InMemory(transactional=true)
      public class ServiceImpl implements IService {
          ….
      }
NonTransactionalInMemoryServiceImpl.java:
      @InMemory(transactional=false)
      public class ServiceImpl implements IService {
          ….
      }
ClientImpl.java:
      public class ClientImpl implements IClient {
          @Inject
          @InMemory(transactional=true)
          private IService service;
          …
      }
```

Listing 4

```
EntityManagerProducer.java:
public EntityManagerProducer {
    @Produces
    public EntityManager getEntityManager(EntityManagerFactory factory) {
        return entityManagerFactory.getEntityManager();
    }
    …
    public void close(@Disposes EntityManager entityManager) {
        entityManager.close();
    }
}
DataSourceProducer.java:
public DataSourceProducer {
    @Produces
    @UserData //application defined qualifier annotation
    @Resource(name="jdbc/UserDataSource")
    private DataSource dataSource;
}
```

Listing 5

Listing 5 macht den Einsatz deutlich: Die Klasse „EntityManagerProducer" stellt eine entsprechend annotierte Methode bereit, die durch den Container aufgerufen wird, sobald eine Instanz eines „EntityManager" benötigt wird. Die Methode selbst kann – wie gezeigt – parametrisiert sein, für die benötigten Instanzen (im konkreten Fall vom Typ „EntityManagerFactory") muss eine entsprechende Bean beziehungsweise ein Producer verfügbar sein – die beschriebene Producer-Methode wird damit selbst zum Injection-Point („Parameter-Injection").

Die Klasse „EntityManagerProducer" enthält darüber hinaus die Methode „close()", deren Parameter mit „@Disposes" annotiert ist und die im vorliegenden Fall dazu dient, die erzeugte „EntityManager"-Instanz zu schließen, wenn sie nicht mehr benötigt wird. Wann genau dies der Fall ist, wird später erläutert. Interessant ist hier zunächst der Umstand, dass man auf diesem Weg einfach und elegant den Lebenszyklus einer Ressource kontrollieren kann. Dispose-Methoden sind ebenfalls Injection-Points, sie können also analog zu Producer-Methoden parametrisiert werden.

Die Klasse „DataSourceProducer" demonstriert die Verwendung von Producer-Feldern, wie sie typischerweise im Umfeld von EJBs vorkommen. Mittels „@Resource" und der Angabe eines JNDI-Namens wird eine DataSource injiziert, die durch die Annotation „@Produces" wiederum für CDI-Injection-Points zur Verfügung steht. Dieses Vorgehen erscheint auf den ersten Blick etwas widersinnig. Der Vorteil besteht allerdings darin, die Streuung des JNDI-Namens über verschiedene Klassen, die diese DataSource verwenden, zu vermeiden. Es existiert also nur noch eine zentrale Stelle in der Anwendung, an der er deklariert sein muss. Eine Umbenennung oder der Ersatz durch eine andere DataSource (mit oder ohne JNDI) kann nun relativ einfach durch eine Anpassung oder den Austausch des Producer erfolgen.

Das Producer-Feld ist beispielhaft mit einem Qualifier („@UserData") versehen. Dies ist natürlich auch für Producer- und Dispose-Methoden möglich. Auf diesem

```
InMemoryServiceImpl.java:
      @InMemory
      @ApplicationScoped
      public class ServiceImpl implements IService {
           @Inject
          private ConnectionFactory connectionFactory

          private Connection connection;

          @PostConstruct
    public void init() {
        this.connection = connectionFactory.createConnection();
    }
          @PostConstruct
          public void destroy() {
             this.connection.close();
          }
      }
```

Listing 6

Wege können verschiedene DataSource-Instanzen voneinander unterschieden werden. Der Einsatz der Qualifier-Annotation bietet gegenüber einem JNDI-Namen den Vorteil einer höheren „Refactoring-Sicherheit", da eine Umbenennung im Zweifelsfall zu Fehlern während des Kompilierens führt und nicht erst zur Laufzeit.

Scope und Lebenszyklus

Am Beispiel der Producer-Methoden hat sich bereits gezeigt, dass eine existierende Dispose-Methode zum geeigneten Zeitpunkt vom Container aufgerufen wird. Doch wann ist dieser Zeitpunkt?

Jede durch den Container erzeugte Bean-Instanz ist eindeutig einem sogenannten „Scope" zugeordnet. Dieser verfügt über einen Lebenszyklus (also Erzeugung und Zerstörung), der durch Ereignisse innerhalb des Containers bestimmt ist, also etwa Start und Stopp einer Applikation. Beim Zugriff auf eine injizierte Bean an einem Injection-Point („@Inject") wird geprüft, welchem Scope sie zugeordnet werden soll und ob in diesem bereits eine Instanz existiert. Ist dies der Fall, wird diese verwendet, anderenfalls eine neue erzeugt und im entsprechenden Scope abgelegt. Erreicht dieser das Ende seines Lebenszyklus (etwa durch Stopp der Applikation), werden die darin enthaltenen Bean-Instanzen ebenfalls zerstört. Im Falle der Erzeugung der Instanzen über eine Producer-Methode wird also zu diesem Zeitpunkt eine gegebenenfalls existierende Dispose-Methode aufgerufen.

Wird eine Bean durch den Container direkt verwaltet (also ohne den Umweg über

```
UIController.java:
      public class UIController {

          @Inject
          private Conversation conversation;

          public void begin() {
        conversation.setTimeout(TimeUnit.MINUTE.asMillis(5));
        conversation.begin();
        … // load conversation data
    }
    public void end() {
              … // store conversation data
              conversation.end();
         }
      }
```

Listing 7

Producer), können in ihr sogenannte „Lifecycle-Callbacks" (also mit „@PostConstruct" beziehungsweise „@PreDestroy" annotierte Methoden ohne Parameter beziehungsweise Rückgabewert) implementiert werden, die bei Erzeugung beziehungsweise Zerstörung der Instanz durch den Container aufgerufen werden. Sie eignen sich hervorragend, um benötigte Ressourcen wie JMS-Connections zu allokieren beziehungsweise wieder freizugeben. Listing 6 zeigt ihre Verwendung.

Der zu verwendende Scope für eine Bean wird durch eine entsprechende Annotation auf Klassenebene oder am jeweiligen Producer (Methode oder Feld) festgelegt, im Beispiel wurde bereits „@ApplicationScoped" verwendet. Folgende Scopes sind durch CDI vordefiniert:

- *@RequestScoped*
  Dauer einer Anfrage (wie HTTP-Request, Konsumieren einer JMS-Nachricht)
- *@SessionScoped*
  Lebensdauer einer Sitzung (wie HTTP)
- *@ConversationScoped*
  Durch die Anwendung bestimmte Dauer einer Konversation/Interaktion im Rahmen einer Sitzung, die sich über mehrere Anfragen (Requests) erstrecken kann (wie Wizard oder modaler Dialog in einer Web-Anwendung)
- *@ApplicationScoped*
  Start und Stopp einer Applikation im Container (wie Deployment/Undeployment einer WAR/EAR-Datei)
- *@Singleton*
  Definiert ein klassisches Java-Singleton,

```
UIController.java:
      @Named // can be referenced by EL expressions using #{userSettings}
      @SessionScoped
      public class UserSettings {
        ...
      }
      public class SystemSettings {
        @Produces
        @Singleton
        @Named(„systemLocale") // can be referenced by EL expressions using #{systemLocale}
        public Locale getDefaultLocale() {
          ...
        }
      }
```

Listing 8

es ist also im Java-EE-Container an den Lebenszyklus des Applikations-Classloader gekoppelt. Damit ergibt sich eine große Ähnlichkeit zu „@ApplicationScoped", es existiert allerdings ein Unterschied bei der Injizierung in andere Beans, der noch erläutert wird.

- *@Dependent*
  Übernahme des Scope vom jeweiligen Injection-Point einer erzeugten Instanz. Dies ist bei Injizierung in ein Feld einer Bean-Instanz beziehungsweise im Falle einer Producer-Methode der jeweils für diese Bean deklarierte Scope. Dieser kann selbst wiederum „@Dependent" sein, auf diese Art können ganze Abhängigkeitsketten entstehen. Verfügt eine Bean oder ein Producer über keine der aufgezählten Annotationen, werden die entsprechenden Instanzen implizit als „@Dependent" betrachtet.

Eine Besonderheit für „@Dependent" und „@Singleton" besteht darin, dass die erzeugten Instanzen im Gegensatz zu allen anderen sogenannten „normalen Scopes" direkt, also ohne Verwendung dynamischer Proxies, in den Konsumenten injiziert werden und so beispielsweise eine Verwendung von Interzeptoren nicht möglich ist.

Es bleibt noch die Frage offen, wie der Lebenszyklus einer Konversation zur Arbeit mit „@ConversationScoped"-Beans gesteuert werden kann. Eine Konversation ist an eine umgebende Sitzung gekoppelt und überlebt mehrere Anfragen. Sie muss explizit begonnen und beendet werden (siehe Listing 7). Pro Sitzung existiert nur jeweils eine aktive Konversation. Wird eine Konversation begonnen, aber nicht beendet, wird sie nach Ablauf eines konfigurierbaren Timeout automatisch verworfen.

Enterprise Java Beans

Die CDI-Spezifikation sieht die Integration mit anderen Technologien aus dem Umfeld von Java EE 6 vor. Das betrifft vor allem die Arbeit mit Enterprise Java Beans. Sie können über ihr Business-Interface via „@Inject" oder „@EJB" in CDI-Beans injiziert werden. Dies gilt ebenso für technische Ressourcen („@Resource", „@PersistenceContext").

Umgekehrt kann „@Inject" in EJBs verwendet werden, im Falle von Stateful Session Beans (SSB) ist darüber hinaus die Deklaration von Scopes möglich. Letzteres ersetzt die Verwendung einer mit „@Remove" annotierten Methode, die SSB-Instanz wird automatisch bei der Zerstörung des jeweiligen Scope verworfen.

Diese beidseitige Integration sorgt am Anfang oftmals für Verwirrung: Wann sollte man nun EJBs und wann CDI-Beans verwenden? Ein kleiner Perspektiv-Wechsel birgt die Antwort in sich: EJBs können als CDI-Beans aufgefasst werden, die mit speziellen Eigenschaften ausgestattet sind – sie können also Transaktionen öffnen beziehungsweise schließen, Autorisierung erzwingen etc. Dieser Mehrwehrt ist in der Regel nur an wenigen definierten Stellen innerhalb einer Anwendung notwendig.

In der klassischen Drei-Tier-Architektur ist dies die Schnittstelle der Anwendungslogik zu anderen Schichten beziehungs-

TEAM - Ihr Partner für innovative IT-Lösungen

Als Oracle Platinum Partner bieten wir ein umfassendes Dienstleistungsspektrum rund um die Oracle-Technologien.

- ADF Entwicklung Best Practices
- Von Forms zu Java
- Workshop und Coaching
- Oracle WLS/GlassFish Server Installation/Administration
- Individualentwicklung
- TEAM ADF-Tools

TEAM GmbH
www.team-pb.de

Hermann-Löns-Str. 88
33104 Paderborn

Mail oracle@team-pb.de
Fon +49 5254 8008-0

ORACLE Platinum Partner

weise Modulen – das Stichwort lautet „EJB-Facade". Im bildlichen Sinne kommen darüber (etwa JSF-Controller- und Model-Beans) und darunter (etwa DAOs, Berechnungslogik) wiederum nur CDI-Beans zur Anwendung, die in ihrer Natur wesentlich leichtgewichtiger als EJBs sind.

Java Server Faces

Ein maßgeblicher Treiber für die Entwicklung von CDI war die Schaffung eines einfacheren Programmiermodells für JSF. Dies lässt sich gut an den vordefinierten Scopes („@RequestScoped", „@SessionScoped", „@ConversationScoped") erkennen. Leider besteht auch hier wieder genügend Raum für Verwirrung, da die JSF-Spezifikation eigene Annotationen für sogenannte „Managed Beans" und „Managed Properties" vorsieht, die sich aber nicht mit CDI integrieren und bei denen darüber hinaus sogar Namensdopplungen mit CDI-Scopes bestehen. Das bedeutet, dass für die Konzeption einer JSF-basierten Anwendung im Java-EE-6-Kontext eine Entscheidung für den einen oder den anderen Weg getroffen werden muss. Aufgrund des deutlich einfacheren Programmiermodells fällt die Empfehlung klar für CDI aus, obwohl hierfür leider kein „@ViewScoped" vordefiniert ist. Dieser durchaus nützliche Scope kann aber durch eine entsprechende CDI-Erweiterung leicht nachgerüstet werden.

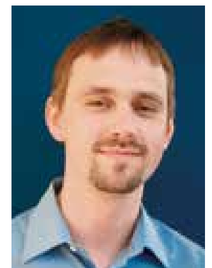
Um CDI-Beans oder EJBs für die Referenzierung in Expression-Language (EL)-Ausdrücken in Views zugänglich zu machen, kommt die Annotation „@Named" zum Einsatz. Diese kann an Bean-Deklarationen oder Producer-Feldern beziehungsweise -Methoden eingesetzt werden (siehe Listing 8).

Fazit

CDI definiert ein relativ abstraktes, aber sehr konsequentes Modell zur Arbeit mit Dependency Injection und verlangt dafür von seinem Nutzer Verständnis für die zu lösenden Probleme und angebotenen Konzepte. Wer sich darauf einlässt und das notwendige Vertrauen in die Container-Infrastruktur entwickelt, gewinnt ein hohes Maß an Flexibilität und kann seinen Code weitestgehend frei von komplizierten technischen Konstrukten zur Entkopplung von Diensten und deren Konsumenten halten.

Darüber hinaus erhält er weitere, noch nicht näher benannte Boni: Angerissen wurden bereits die Möglichkeiten zur Arbeit mit Interceptoren und Delegates sowie die Umsetzung des Observer-Patterns zum Feuern und Konsumieren von Ereignissen (Events). Darüber hinaus definiert CDI eine umfangreiche Erweiterungsschnittstelle, die unter anderem die Definition eigener Scopes ermöglicht. Auf diese Extras wird in den folgenden Ausgaben im Detail eingegangen.

Dirk Mahler
dirk.mahler@buschmais.com



Dirk Mahler ist als Senior Consultant auf dem Gebiet der Java-Enterprise-Technologien tätig. In seiner täglichen Arbeit setzt er sich mit Themen rund um Software-Architektur auseinander, kann dabei eine Vorliebe für den Aspekt der Persistenz nicht verleugnen.

Distributed Java Caches

Dr. Fabian Stäber, ConSol* Consulting & Solutions Software GmbH

Mit Ehcache, Infinispan und Hazelcast gibt es gleich drei Java-Produkte, die Cluster-fähige Caches zur Verfügung stellen. Alle drei Implementierungen bieten eine ähnliche Schnittstelle, unterscheiden sich jedoch erheblich in den zugrunde liegenden Architekturen. Dieser Artikel stellt diese vor und erklärt, welche Architektur sich für welche Anforderungen am besten eignet.

Horizontale Skalierbarkeit und Cluster-Fähigkeit gehören inzwischen zu den Standardanforderungen bei der Entwicklung neuer Java-Server-Anwendungen. Es wird erwartet, dass bei steigenden Zugriffszahlen die höhere Last durch das Starten neuer Server-Instanzen abgefangen werden kann. Diese Entwicklung bringt auch neue Anforderungen an Caches mit sich: Es genügt nicht mehr, Caches als einfache, lokale Speicher zu implementieren. Im Cluster-Betrieb müssen „gecachte" Daten synchronisiert werden, sodass Cluster-weite Daten-Konsistenz über alle Instanzen hinweg gewährleistet werden kann. Um diese Anforderungen zu erfüllen, haben sich moderne Java-Caches zu mächtigen In-Memory-Data-Management-Systemen entwickelt, die verteilte atomare Operationen sowie verteilte Transaktionen unterstützen. Nachfolgend sind drei ver-

breitete Java-Cache-Implementierungen vorgestellt:

- *Ehcache/Terracotta*
  Von der Software AG herausgegeben und in einer Open-Source-Variante sowie einer erweiterten kommerziellen Variante verfügbar, die speziell auf Anwendungen mit großen Datenmengen ausgerichtet ist
- *Infinispan*
  Ein Open-Source-Projekt von Red Hat
- *Hazelcast*
  Die Cache-Implementierung einer gleichnamigen Firma, die ebenfalls in einer Open-Source-Variante und einer erweiterten kommerziellen Variante angeboten wird.

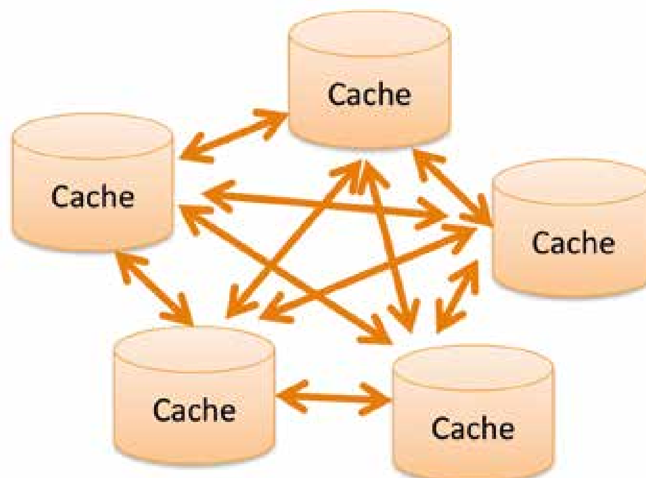


Abbildung 1: Full-Replication-Peer-to-Peer-Topologie

ConcurrentMap Interface

Alle hier vorgestellten Cache-Implementierungen bieten ein ähnliches Interface, das sich an „java.util.concurrent.ConcurrentMap" orientiert. „ConcurrentMap" definiert die Methoden „putIfAbsent", „remove" und „replace", die atomare Updates der Daten erlauben. Mit deren Hilfe ist es möglich, „thread-safe" zu programmieren, ohne „synchronized"-Blöcke zu verwenden. Diese Eigenschaft macht „ConcurrentMap" zu einem geeigneten Interface für Cluster-fähige Caches. Während synchronized-Blöcke jeweils nur innerhalb einer VM Thread-Safety garantieren, können atomare Operationen mithilfe geeigneter Algorithmen auch Cluster-übergreifend implementiert werden.

Peer-to-Peer-Cluster

Als „Peer-to-Peer-Cluster" werden Architekturen bezeichnet, in denen jede beteiligte Instanz gleichzeitig Client und Server ist. Das heißt, jede Instanz, die den Cache nutzt, dient gleichzeitig zur Datenhaltung. Es gibt zwei grundsätzlich verschiedene Topologien von Peer-to-Peer-Netzen.

Abbildung 1 zeigt eine Full-Replication-Peer-to-Peer-Topologie. Hier sind alle Daten an alle Instanzen verteilt. Das ermöglicht eine sehr hohe Effizienz beim lesenden Zugriff auf Daten, da lesender Zugriff immer lokal bedient werden kann. Beim schreibenden Zugriff erzeugt das Verteilen der Daten allerdings hohen Netzwerk-Overhead. Die Implementierung von atomaren Schreib-Operationen in Full-Replication-Netzwerken ist sehr ineffizient, da bei jeder Schreib-Operation auf Acknowledgements aller anderen Instanzen gewartet werden muss.

Die Alternative zu Full-Replication-Topologien sind „Distributed Hash Tables". Wie Abbildung 2 zeigt, wird die Menge an Keys partitioniert, sodass immer eine definierte Gruppe von Cache-Instanzen für eine definierte Menge von Keys verantwortlich ist. Ausfallsicherheit wird durch Backup-Instanzen realisiert.

Lese-Operationen sind in Distributed Hash Tables langsamer als in voll replizierten Topologien, da die Daten meistens über das Netzwerk von der verantwortlichen Instanz geladen werden müssen. Die Implementierung von verteilten atomaren Operationen ist jedoch sehr effizient möglich, da jeweils die für den Key verantwortliche Instanz die atomare Schreiboperation koordinieren kann.

Die Tabelle zeigt, welche Cache-Implementierung welche Peer-to-Peer-Architektur unterstützt. Es ist zu sehen, dass Ehcache seinen Fokus auf voll replizierten Installationen hat, während Hazelcast eine verteilte Hash-Tabelle implementiert. Infinispan unterstützt je nach Konfiguration beides, ist also die vielseitigste, aber auch die komplexeste Lösung.

Client-Server-Clustering

Die beschriebenen Peer-to-Peer-Topologien haben gemeinsam, dass es keine Trennung zwischen datenhaltenden und datennutzenden Instanzen gibt. Das bedeutet, dass die gecachten Daten direkt in den Application-Servern gehalten werden. Will man getrennte Hardware für gecachte Daten und für Application-Server einführen, muss man die Caches im Client-Server-Modus betreiben. Dabei wird ein dedizierter Cache-Server-Cluster zur Datenhaltung aufgesetzt, der dann von den Application-Servern genutzt wird.

Infinispan und Hazelcast nutzen im Server-Modus dieselbe Code-Basis wie im Peer-to-Peer-Modus: Die beschriebenen Peer-to-Peer-Topologien lassen sich Stand-Alone als Cache-Server-Cluster betreiben. Auf diesen wird dann von den Application-Servern aus zugegriffen.

Bei Ehcache wird der Server-Teil von einem eigenen Projekt namens „Terracotta" bereitgestellt, das ursprünglich unabhän-

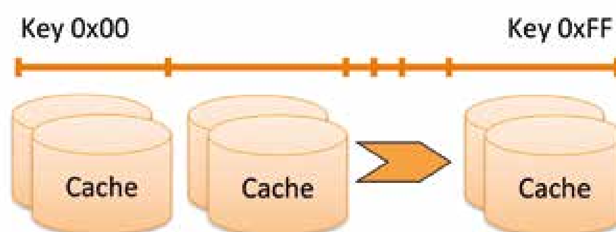


Abbildung 2: Distributed-Hash-Table-Peer-to-Peer-Topologie

gig von Ehcache entwickelt wurde. Während Ehcache im Peer-to-Peer-Modus eine Full-Replication-Topologie implementiert, stellt Terracotta im Server-Modus eine verteilte Hash-Tabelle bereit.

Wie auch im Peer-to-Peer-Modus ist Infinispan im Client-Server-Modus die vielseitigste, aber auch die komplexeste Lösung: Der Server-Cluster kann mit Infinispan sowohl voll repliziert als auch als verteilte Hash-Tabelle betrieben werden.

Die verteilten Hash-Tabellen unterscheiden sich unter anderem dadurch, dass bei Hazelcast und Infinispan die Zahl der Instanzen elastisch erweitert werden kann, während die verfügbaren Instanzen bei Terracotta fest konfiguriert sind. Dadurch ist es bei Terracotta möglich, statisch festzulegen, auf welcher Hardware jeweils das Back-up für welche andere Hardware liegen soll. Andererseits ist es mit Terracotta nicht möglich, zur Laufzeit dynamisch neue Server-Instanzen hinzuzufügen.

	Full Replication	Distributed Hash Table
Ehcache	X	
Hazelcast		X
Infinispan	X	X

Tabelle

Fazit

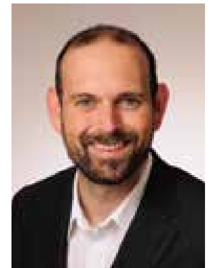
Solange man sich als Programmierer auf die in „ConcurrentMap" definierten Methoden beschränkt, sind die drei Cache-Implementierungen aus API-Sicht fast identisch und leicht austauschbar. Die zugrunde liegenden Netzwerk-Topologien der Caches unterscheiden sich jedoch zum Teil erheblich.

Je nach Anwendungsfall sind die verschiedenen Netzwerk-Topologien unterschiedlich gut geeignet. Kommt eine Anwendung ohne atomare Operationen aus, kann die Performance erheblich gesteigert werden, wenn man eine zugrunde liegende Cache-Architektur ohne Atomizitätsgarantien verwendet. Beispiele dafür sind „WORM"-Anwendungen: Write-Once-Read-Many. Hier werden die Daten nicht aktualisiert, aber oft gelesen.

Wenn atomare Operationen benötigt werden, skaliert der Full-Replication-Ansatz schon bei einer geringen Zahl von Instanzen nicht mehr. In diesem Fall sollte eine Lösung gewählt werden, die eine verteilte Hash-Tabelle implementiert. Weitere grundsätzliche Unterschiede zeigen sich in den Möglichkeiten, große Datenmengen zu verarbeiten. Ehcache/Terracotta sowie Hazelcast bieten hierbei erweiterte Optionen, die gespeicherten Daten in erweitertem Speicherraum („off-heap") abzulegen, um maximale vertikale Skalierbarkeit zu ermöglichen.

Aufgrund solcher Unterschiede ist es nicht möglich, mithilfe von Benchmarks pauschal eine beste Cache-Implementierung zu ermitteln.

Dr. Fabian Stäber
fabian.staeber@consol.de



Dr. Fabian Stäber ist Senior Software Architect bei der ConSol* Consulting & Solutions Software GmbH. Er arbeitet als Entwickler sowie Consultant und entwickelt seit zehn Jahren Java-Web-Anwendungen. Während seiner akademischen Laufbahn beschäftigte er sich hauptsächlich mit Peer-to-Peer-basierten Netzwerk-Architekturen in industriellen Anwendungen. Dr. Fabian Stäber begeistert sich für JEE, geclusterte Back-Ends, Big-Data-Anwendungen und verteilte Architekturen.

www.ijug.eu

Sichern Sie sich 4 Ausgaben für 18 EUR

FAXEN SIE DAS AUSGEFÜLLTE FORMULAR AN
0700 11 36 24 39

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

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

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

Java aktuell

Activiti in produktiven Umgebungen – Theorie und Praxis

Jonas Grundler und Carlos Barragan, NovaTec Consulting GmbH

Die Open-Source-Process-Engine Activiti kann für JEE-basierte Prozess-Management-Lösungen in produktiven Umgebungen erfolgreich eingesetzt und dabei mit Komponenten wie Tomcat, Atomikos oder Weld integriert werden – ein Projekt für die Installation von Ladestationen für Hybrid-Fahrzeuge hat das am praktischen Beispiel bewiesen.

Die Erkenntnisse und Erfahrungen aus diesem auf der Grundlage von JEE und Activiti realisierten Projekt sind von genereller Relevanz und lassen sich grundsätzlich auf jede Open-Source-BPM-Plattform übertragen. Programmier-Umwege, die an verschiedenen Stellen noch beschritten werden mussten, werfen die spannende Frage auf, welche Möglichkeiten die Weiterentwicklung von Activiti unter dem Dach von Camunda BPM bereitstellen wird.

Der Weg von den theoretischen Anforderungen an prozessgesteuertes Business-Management bis zur praktischen Umsetzung wird von zahlreichen Herausforderungen und Detail-Problemen markiert, die sich aus der Realisierung ergeben. An einem konkreten Projekt aus der Praxis der Autoren können aus erster Hand Erfahrungen und Erkenntnisse gewonnen und Schlussfolgerungen für mögliche Alternativen gezogen werden. Im vorliegenden Fall suchte ein Hersteller von Hybrid-Automobilen mit Verbrennungs- und elektrischem Antrieb auf der Basis des BPMN-2.0-Standards [1] eine Lösung für die prozessgesteuerte Installation von Ladestationen bei den Käufern seiner Fahrzeuge. Im Rahmen der Anwendung, deren Entwicklung rund fünfhundert Personentage in Anspruch nahm, waren mehr als dreißig JPA-Entitäten und mehr als zwanzig Services als Schnittstellen zwischen User-Interface und Business-Ebene zu realisieren. Um den Prozess mit seinen drei Teilprozess-Ebenen abzubilden, mussten darüber hinaus etwa fünfzig JSF-Seiten und Komponenten eingebunden werden.

Die Theorie: Anforderungen und Vorgaben

Der Ablauf des zu steuernden Prozesses stellt sich wie folgt dar: Wird ein Hybrid-Fahrzeug verkauft, startet der Vertrieb zugleich den Bereitstellungsprozess für die Ladestation. Entweder wählt der Vertrieb schon zu diesem Zeitpunkt den Dienstleister aus – also den Elektriker, der die Ladestation beim Kunden installieren wird, oder er nimmt zunächst Rücksprache mit dem Kunden. In diesem Fall wird die Auswahl des Dienstleisters später vorgenommen.

Für den Kunden bietet die prozessgesteuerte Auftragsverarbeitung entscheidende Vorteile. Per E-Mail kann er auf Wunsch über jeden einzelnen Prozessschritt benachrichtigt werden und ist so über den Fortschritt immer auf dem Laufenden. Hakt es im Prozess, wenn sich beispielsweise die Bereitstellung der Ladestation verzögert oder der beauftragte Dienstleister sich nicht meldet, kann umgekehrt der Vertrieb entsprechende Rückfragen des Kunden effektiv und punktgenau beantworten, da er jederzeit den Überblick über den Stand des Prozesses hat.

Ein weiterer Prozess-Teilnehmer ist der Dienstleister, der die Ladestation installieren soll. Sobald er den Auftrag angenommen hat, ist er für den Prozessablauf verantwortlich. Dabei sind länderspezifische Besonderheiten zusätzlich zu berücksichtigen: In manchen Ländern ist für die Installation der Ladestation in einem Privathaushalt eine behördliche Genehmigung einzuholen, zudem muss in einigen Ländern die Ladestation nach der Installation zusätzlich von den Behörden abgenommen werden.

Die Behörden sind ebenfalls direkter Prozess-Teilnehmer. Für die Entwicklung der Anwendung, die weltweit eingesetzt werden soll, war das eine der komplexesten Herausforderungen. In manchen Fällen ist zu Beginn des Prozesses bekannt, ob eine Erlaubnis benötigt wird, in anderen Fällen ändert sich diese Anforderung im Laufe des Prozesses, sie kann wegfallen oder nachträglich wieder hinzukommen. Die aus Entwicklersicht notwendigen nicht-unterbrechenden Ereignis-Teilprozesse waren in Activiti 5.10 [2] allerdings noch nicht enthalten und mussten aufwändig nachmodelliert werden.

Zu den Anforderungen, die sich aus dem Prozessablauf ergeben, treten weitere Vorgaben des Auftraggebers hinsichtlich der IT-Architektur. Activiti 5.10 als Prozess-Engine war ebenso vorab festgelegt wie Tomcat 7.0 [3] als Serverlösung und die Datenbank eines internationalen großen Anbieters. Aus Gründen von Datenschutz und Datensicherheit forderte der Auftraggeber ferner, dass im Prozess selbst nur für die Prozess-Navigation relevante Daten wie Terminvereinbarungen oder die Einholung von Genehmigungen gespeichert werden, während im Prozess benötigte fachliche Daten über JPA außerhalb des Prozess-Kontextes abgelegt und über eine eindeutige ID entsprechend verknüpft

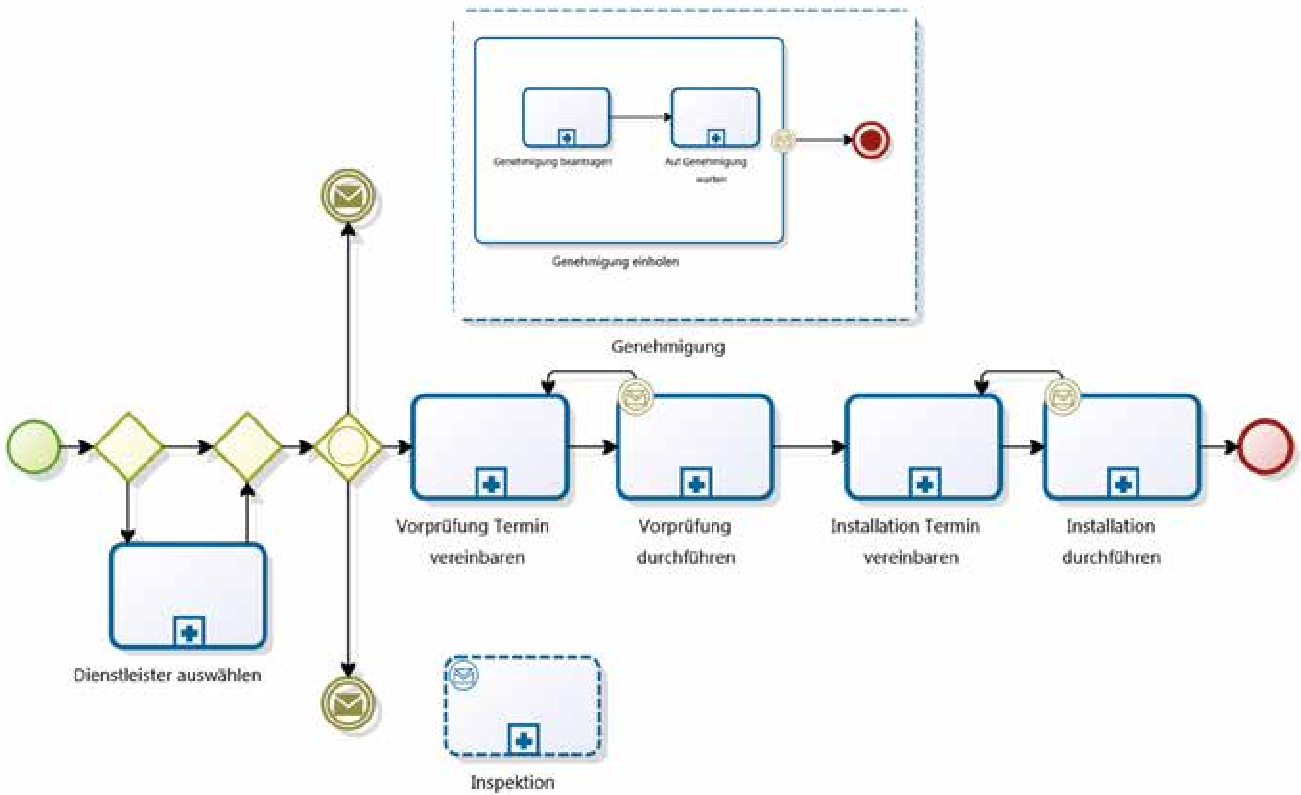


Abbildung 1: Hauptprozess „Installation der Ladestation durch Dienstleister“ gemäß BPMN-2.0-Spezifikation

werden. Zudem enthält der fachliche Kontext nur Referenzen auf die Umsysteme, die die Inhalte wie Kunden-Kontaktdaten verwalten. Diese Vorgabe verlangt, dass bei Datenänderungen Prozessdaten und fachliche Daten in beiden Bereichen, also in globalen Transaktionen, synchronisiert werden müssen. Wegen des vorgesehenen weltweiten Einsatzbereiches musste die Anwendung in zehn Sprachen ausgeliefert werden und auf bis zu hunderttausend Prozesse im Jahr ausgelegt sein.

Die Praxis (I): das Prozessmodell

Modelliert wurde der Prozess, den Vorgaben entsprechend nach der BPMN-2.0-Spezifikation. Ein Mantelprozess (Wrapper) deckt die vom Vertrieb auszuführenden Schritte ab – Anlegen des Prozesses, Eingabe der Basisdaten und Einlesen der Basiskonfiguration wie etwa Länder-Zugehörigkeit. Diese Informationen werden nicht manuell eingegeben, sondern über einen Service-Task („Read Config“ in Abbildung 1) aus der Einordnung des jeweiligen Verkäufers im Organisationsmodell entnommen.

Der Hauptprozess – also der Teil, der vom Dienstleister durchgeführt wird – findet in einem Teilprozess mit drei Ebenen statt: Der Hauptpfad, der die Installation der Ladestation abbildet, ist für eine erfolgreiche Installation immer zu durchlaufen; die Einholung einer Behördengenehmigung (Permit) und die Abnahme (Inspection) sind länderabhängige, optionale Pfade und müssen flexibel aktiviert und deaktiviert werden können (siehe Abbildung 1).

Bei jedem Task, der menschliches Handeln verlangt, ist ein wiederkehrendes Muster für die Versendung von Info-, Erinnerungs- und Eskalations-Mails zu erkennen. In Abbildung 1 ist dieses Pattern jeweils hinter den Aufrufaktivitäten (engl. „Call Activities“, mit „+“ gekennzeichnete Aktivitäten) zu finden. Im Feld „Genehmigung beantragen“ ist dieses Pattern auch detailliert dargestellt (siehe Abbildung 2).

In der Praxis ist die Umsetzung unter Activiti 5.10 allerdings erheblich komplexer als die Anforderung in der Theorie. Zusätzliche Elemente im Prozessmodell sind vor allem deswegen notwendig, weil Activiti zum Zeitpunkt der Entwicklung keine nicht-unterbrechenden Ereignisteilprozesse unterstützt hat – ein Problem, das

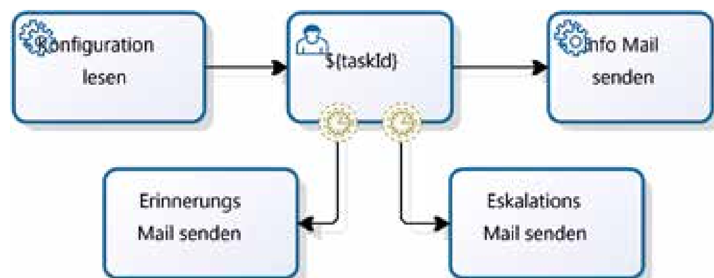


Abbildung 2: Pattern für „Benutzer-Tasks“

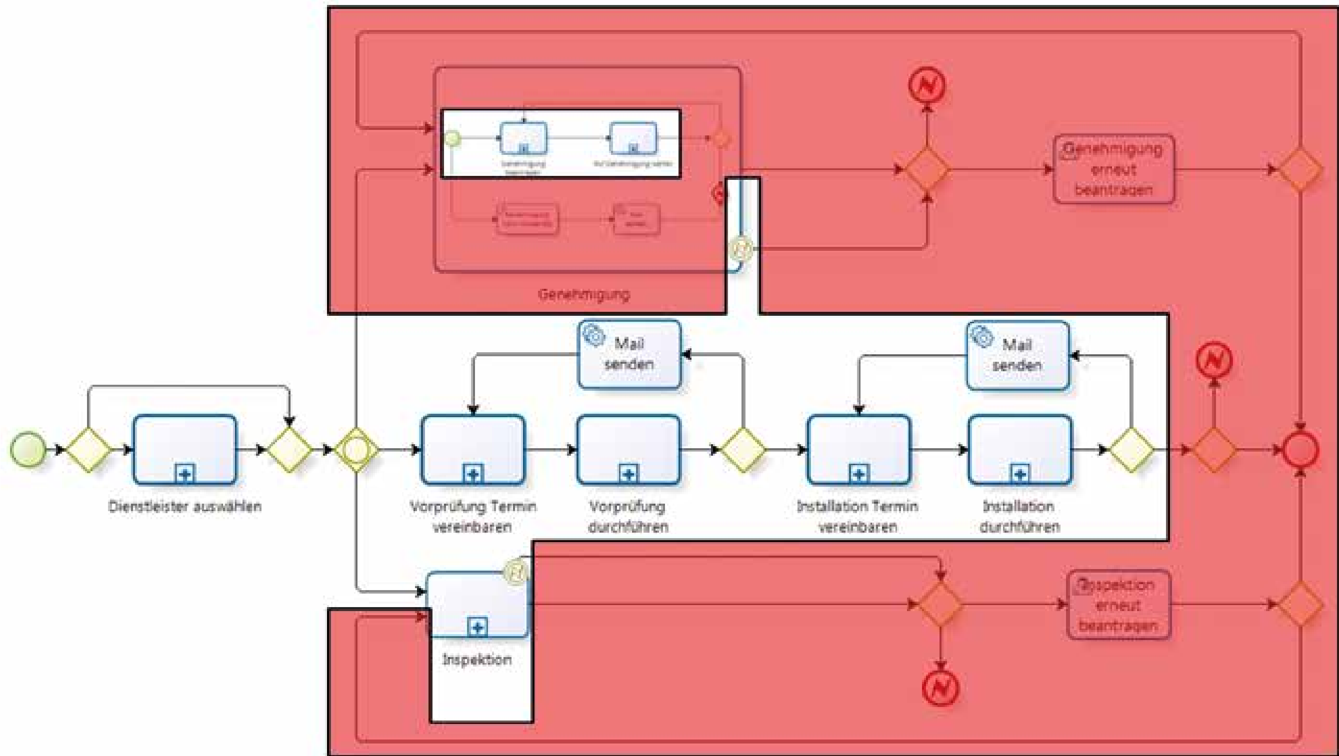


Abbildung 3: Nachmodellierung nicht-unterbrechender Ereignisteilprozesse unter Activiti (Zusatzschritte sind rot markiert)

zum Zeitpunkt des Redaktionsschlusses für diesen Beitrag noch fortbesteht. Diese Funktion wäre vor allem für die Nebenpfade „Genehmigung" und „Inspektion" erforderlich gewesen. Diese Ebenen müssen zum Hauptprozess beliebig hinzu- oder weggenommen werden können, und zwar zu jedem Zeitpunkt, solange der Hauptprozess noch nicht abgeschlossen ist. Die fehlende Unterstützung solcher nicht-unterbrechender Ereignisteilprozesse machte eine komplexe Nachmodellierung des an sich einfachen BPMN-2.0-Modells für die Nebenpfade erforderlich und steigerte den Entwicklungsaufwand (siehe Abbildung 3).

Darüber hinaus sind auch nicht-unterbrechende Rand-Ereignisse auf Teilprozessen mit Activiti 5.10 noch nicht möglich gewesen und mussten explizit modelliert werden. Das bedeutete auch für den Hauptpfad zusätzlichen Entwicklungsaufwand: Da jeder Benutzer-Task einen externen Input vom installierenden Dienstleister benötigt, wenn beispielsweise Termine vereinbart, geändert oder verschoben werden, muss bei Unterbrechung oder Wiederholung die Aufgabe abgebrochen und neu gestartet werden. Im Prozess wird das durch weitere Zusatzschritte abgebildet.

Die Praxis (II): Erweiterung des Activiti API

Eine erfreulich hohe Übereinstimmung von idealem Modell und praktischer Umsetzung ergibt sich dagegen beim Einsatz von wiederverwendbaren Subprozessen für wiederholte Anforderungen (siehe auch Abbildung 2). Jeder Benutzer-Task erfordert nämlich, dass nach dessen Abschluss eine E-Mail versandt wird. Ferner muss für jeden Benutzer-Task die Zeit bis zum Versand einer Erinnerungs- oder Eskalationsmail an den Vertrieb konfigu-

```
public Object execute (CommandContext cc) {
  JobManager jm = Context.getCommandContext().getJobMana-
ger();
  List<Job> jobs = jobManager.findJobsByExecutionId(id);
  for (Job job : jobs) {
    JobEntity je = (JobEntity) job;
    if (REMINDER.equals(je.getJobHandlerConfiguration())) {
      je.setDuedate(newDueDateReminder);
    }
    if (ESCALATION.equals(je.getJobHandlerConfiguration()))
{
      je.setDuedate(newDueDateEscalation);
    }
  }
}
```

Listing 1: Einfacher Einbau eines neuen Kommandos in den Prozessablauf

riert werden können. Dank wiederverwendbarer Teilprozesse lassen sich alle manuellen Schritte, die der Dienstleister abschließen muss, durch denselben Teilprozess realisieren – aber eben parametrisiert.

Je nach Land werden die Tasks für Genehmigung und Abnahme der Ladestations-Installation gleichzeitig gestartet. Oft ist es allerdings erforderlich, das bereits im Task vorgegebene Datum für die Erinnerungs-Mail drei Tage nach Terminvereinbarung nachträglich anzupassen, wenn etwa das Genehmigungsverfahren noch nicht erfolgreich war.

Dieses Problem beim Erstellen von Tasklisten mit fachlichen Daten ist durch eine einfache Erweiterung des Activiti API elegant lösbar. Die Flexibilität von Activiti und die Erweiterungsmöglichkeiten der Open Source Process Engine konnten dabei optimal genutzt werden: Die zusätzliche Funktionalität ließ sich auf einfache Weise in das Framework einbauen, ohne dass dafür der Code selbst angepasst werden musste (siehe Listing 1).

Die Praxis (III): Komponenten und Alternativen

Ausgehend von den Vorgaben des Auftraggebers haben die Autoren eine Infrastruktur mit folgenden Produkten und Komponenten entwickelt:

- Eine Benutzerschnittstelle für den Endanwender (hauptsächlich der Dienstleister) und eine für die Administration der Anwendung. Diese verwenden JSF 2 und das Open-Source-Framework PrimeFaces 3.1.
- Die Anbindung der Datenbank bereitete keine wesentlichen Probleme. Wichtig ist allerdings, dass in der Entwicklungsumgebung eine Datenbank vom gleichen Hersteller wie in der Produktion verwendet wird, um Komplikationen, bedingt durch abweichendes Verhalten (unterschiedlicher Datenbanken), zu vermeiden.
- Für Prozesse, Geschäftsobjekte, Geschäftslogik und Services kommen Activiti 5.10 und JPA 2.0 auf Hibernate 4.1 zum Einsatz.
- Weitreichende Konsequenzen hatte die Vorgabe von Tomcat 7 als Serverlösung. Um unter Tomcat nicht verfügbare Funktionen wie die eines Transaktions-Managers bereitzustellen, mussten unter beträchtlichem Einsatz von Ressourcen zusätzliche Komponenten integriert werden: So können mithilfe von Weld [4] moderne Technologien wie Contexts and Dependency Injection (CDI) eingesetzt werden, während der Open-Source-Transaktionsmanager Atomikos 3.8 [5] den transaktionalen Layer, der in JEE-Application-Servern Voraussetzung ist, in Tomcat ersetzt. Darüber hinaus wurde Seam-Transactions für die Demarkation von Transaktionen eingesetzt.

Die Integration von CDI und Transaktionen in eine Activiti-Infrastruktur mit Tomcat-Server funktioniert, allerdings muss dafür verschiedentlich tiefer als vorgesehen in die Konfiguration eingegriffen werden. Der hohe Aufwand wirft die Frage nach Alternativen auf: So hätte – wäre nicht Tomcat als Server vorgegeben gewesen – der Einsatz eines JEE-Application-Servers wie GlassFish anstelle eines Webcontainers einiges an Programmierzeit ersparen können, da sich die Einbindung wesentlich einfacher gestaltet hätte. Weld und Atomikos hätten dann beispielsweise nicht konfiguriert und deren Einsatz nicht getestet werden müssen.

Grundsätzlich ist nach den Erfahrungen mit dem hier vorgestellten Projekt festzuhalten, dass BPM auf Open-Source-Basis funktioniert und ein gut getestetes sowie dokumentiertes Framework wie Activiti sich dabei bewährt und gut in eine Java-Infrastruktur einpassen lässt. Allerdings ist dabei zu bedenken, dass die Aussage „eine bestimmte Aufgabe kann mit Activiti umgesetzt werden“ verschiedene Bedeutungen annehmen kann: Umsetzbarkeit kann zum einen heißen, dass Activiti diese Funktionalität selbst anbietet, es kann aber auch meinen, dass Activiti gut mit einem Open-Source-Framework umgehen kann, das diese Funktionalität anbietet. Die dritte mögliche Interpretation ist, dass Activiti einen Ansatzpunkt bietet, der es dem Programmierer erlaubt, die benötigte Funktionalität selbst umzusetzen. Diese Aspekte müssen gerade im Hinblick auf die weitere Entwicklung von Activiti, das in Camunda BPM [6] seine logische Fortsetzung gefunden zu haben scheint, sorgfältig im Auge behalten werden.

Links

[1] http://www.iso.org/iso/catalogue_detail.htm?csnumber=62652
[2] http://www.activiti.org
[3] http://tomcat.apache.org
[4] http://www.camunda.org
[5] http://www.atomikos.com
[6] http://www.seamframework.org/Weld

Jonas Grundler
jonas.grundler@novatec-gmbh.de

Jonas Grundler ist seit dem Jahr 2002 als Berater im BPM-Umfeld tätig und seit 2011 Managing Consultant bei der NovaTec Consulting GmbH mit den Beratungsschwerpunkten „Business Process Management“ und „Application Performance Management“.

Carlos Barragan
carlos.barragan@novatec-gmbh.de

Carlos Barragan arbeitet seit dem Jahr 2002 als Java-Software-Entwickler und ist seit 2010 Senior Consultant und Leiter der Kompetenzgruppe „Enterprise Application Development JEE“ bei der NovaTec Consulting GmbH.

Unbekannte Kostbarkeiten des SDK Heute: Vor und nach der „main()"-Methode

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir stellen in dieser Reihe solche Features des SDK vor: die unbekannten Kostbarkeiten.

Java-Programmieranfängern wird beigebracht, dass die „main()"-Methode der Einstiegspunkt eines Java-Programms ist und dass nach deren Ende die JVM beendet wird. Dies ist zwar prinzipiell richtig, jedoch gibt es die Option, sowohl vor als auch nach der „main()"-Methode beliebigen Code ausführen zu können. Der Artikel stellt diese beiden Optionen vor.

Instrumentierung

Unter Byte-Code-Instrumentierung, kurz „Instrumentierung", versteht man das Ändern von Byte-Code, etwa um Methoden-Aufrufe zu loggen oder deren Ausführungszeit zu messen. Frameworks, die den vom Anwendungsentwickler geschriebenen Code ändern oder erweitern müssen, um eine bestimmte Funktionalität zu realisieren, machen ebenfalls von der Instrumentierung Gebrauch. Ein Beispiel dafür sind JPA-Provider, die etwa das Lazy-Loading von Assoziationen realisieren müssen. Dies ist bereits in [1] detailliert beschrieben.

Um die unterschiedlichen Ansätze der Instrumentierung zu vereinheitlichen, wurde in der Version 5 des SDK das Package „java.lang.instrument" eingeführt. Zitat direkt aus der API-Doc des Package [2]: „Provides services that allow Java programming language agents to instrument programs running on the JVM."

Das Ziel dieses Artikels ist jedoch nicht die Einführung in das Instrumentation-API, sondern die Möglichkeit, beliebigen Code vor der „main()"-Methode ausführen zu können. Der im API-Doc erwähnte „Agent" ist hier das Mittel zum Zweck.

Java-Agenten

Ein Java-Agent ist eine Sammlung von Klassen, die in einem JAR gepackt sind. Die Klasse, deren Methoden vor der „main()"-Methode ausgeführt werden, ist die sogenannte „Pre Main"-Klasse. Die Methoden sind nicht beliebig wählbar, sondern durch Signaturen festgelegt (siehe Listing 1).

Die JVM versucht beim Programmstart zunächst, die erste und – falls diese nicht existiert – die zweite Methode auszuführen. Aber wie wird der JVM überhaupt der Java-Agent bekannt gemacht? Dies erfolgt zum einen über die Manifest-Datei des JAR, die für den Eintrag „Premain-Class" die Pre-Main-Klasse enthält, etwa „Premain-Class: de.pdbm.SimpleAgent". Außerdem muss der Aufruf der JVM mit der Option „javaagent" erfolgen. Die Option enthält durch Doppelpunkt getrennt den Namen der JAR-Datei des Agenten, also beispielsweise „java -javaagent:/path-to/agent.jar de.pdbm.Main".

Da wir nicht instrumentieren wollen, genügt für einen einfachen Test die zweite der oben genannten, überladenen „premain()"-Methoden. Listing 2 zeigt die Klasse „SimpleAgent".

Bei Aufruf der JVM mit obiger Agentenoption erfolgt die Ausgabe der „premain()"-Methode vor dem Laden der Main-Klasse und dem Aufruf der dortigen „main()"-Methode. Der Parameter „agentArgs" der Methode enthält den optionalen String, der nach dem Agenten-Jar angegeben werden kann, also etwa „java -javaagent:/path-to/agent.jar="para1 para2" de.pdbm.Main".

Anwendung mit Sinn

Das vorangegangene Beispiel bewegte sich auf „Hello World"-Niveau. Wir werden deshalb etwas Sinnvolleres realisieren. Java-Agenten wurden eingeführt, um die Instrumentierung von Klassen zu ermögli-

```
public static void premain(String agentArgs, Instrumentation inst);
public static void premain(String agentArgs);
```

Listing 1

```
package de.pdbm;

public class SimpleAgent {
      public static void premain(String agentArgs) {
            System.out.println("Ausgabe vor 'main()'");
      }
}
```

Listing 2

```
public class Agent {

  static List<Class> classesBeforeMain;
  static List<Class> classesAfterMain;

  public static void premain(String agentArgs,
      final Instrumentation instrumentation) {
    Runtime.getRuntime().addShutdownHook(new Thread() {

      public void run() {
        classInfoAfterMain(instrumentation);
      }
    });
    classInfoBeforeMain(instrumentation);
  }
  private static void classInfoBeforeMain(Instrumentation instrumentation) {
    classesBeforeMain = Arrays.asList(instrumentation.getAllLoadedClasses());
    System.out.println("Anzahl geladener Klassen: " + classesBeforeMain.size());
  }
  private static void classInfoAfterMain(Instrumentation instrumentation) {
    classesAfterMain = Arrays.asList(instrumentation.getAllLoadedClasses());
    System.out.println("Anzahl geladener Klassen: " + classesAfterMain.size());
    for (Class clazz : classesAfterMain) {
      if (!classesBeforeMain.contains(clazz)) {
        System.out.println(clazz.getCanonicalName());
      }
    }
  }
}
```

Listing 3

chen. Dieser Artikel führt jedoch nicht das Instrumentation-API ein, sodass wir uns auf die folgende Anforderung festlegen: Zählung der geladenen Klassen vor und nach dem Laden der Main-Klasse und der Ausführung der „main()"-Methode. Wir sind uns darüber im Klaren, dass diese Anforderung in der täglichen Projektarbeit eher selten auftaucht und man sich eigene, sinnvollere Anforderungen überlegen kann.

Wir haben bereits gesehen, dass die „premain()"-Methode vor der „main()"-Methode ausgeführt wird. Wie kann man dann etwas nach der „main()"-Methode ausführen? Hier steht die Methode „addShutdownHook()" der Klasse „Runtime" zur Verfügung. Laut Dokumentation registriert die Methode in der JVM einen Shutdown-Hook. Dies ist ein initialisierter, aber noch nicht gestarteter Thread. Er wird gestartet, wenn die JVM heruntergefahren wird.

Obwohl wir nicht instrumentieren wollen, kommt uns der „Instrumentation"-Parameter der „premain()"-Methode beim Erreichen unseres Ziels zu Hilfe. Das Interface „Instrumentation" enthält unter anderem die Methode „getAllLoadedClasses()", die alle geladenen Klassen zurückgibt. Die Definition des Java-Agenten gestaltet sich damit recht einfach (siehe Listing 3).

System	Geladen vor „main()"	Geladen nach „main()"
Oracle Hot Spot 1.7.0_25	420	430
Oracle Hot Spot 1.7.0_40	421	429
OpenJDK 1.7.0_45	446	454
IBM 1.7.0 SR4	460	485
SAP 1.7.0_25	511	522

Tabelle

Die Methode „Instrumentation#getAllLoadedClasses()" liefert als Antwort ein Array von Klassen. Wir haben dieses Array in eine Liste konvertiert, um es bei der Berechnung der Differenz der beiden Mengen einfacher zu haben. Ansonsten ist die Klasse selbsterklärend.

Zu den geladenen Klassen gehören selbstverständlich die Main-Klasse, aber auch andere, nicht so offensichtliche Klassen wie etwa „java.lang.Void" oder „java.util.IdentityHashMap.KeySet". Erstaunlicherweise ist die Anzahl der geladenen Klassen von SDK zu SDK verschieden. Die Tabelle zeigt einige Ergebnisse.

Fazit

Mit einem Java-Agenten ist es möglich, beliebigen Code auszuführen, noch bevor die „main()"-Methode aufgerufen wird. Java-Agenten dienen primär der Instrumentierung von Klassen, lassen sich jedoch auch anderweitig verwenden. Mit einem Shutdown-Hook kann Code nach der „main()"-Methode ausgeführt werden.

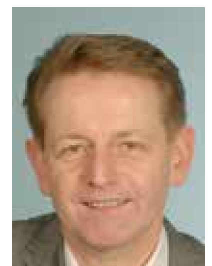
Literatur

[1] Marc Steffens und Bernd Müller. Weaving, Instrumentation, Enhancement: Was ein JPA-Provider so alles macht. Java aktuell, Ausgabe 1/2012.

[2] http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html

Bernd Müller

bernd.mueller@ostfalia.de



Bernd Müller ist Professor für Software-Technik an der Ostfalia. Er ist Autor des Buches „JavaServer Faces 2.0" und Mitglied in der Expertengruppe des JSR 344 (JSF 2.2).

„Java ist hundert Prozent Community …"

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

Wie ist die JUG Darmstadt organisiert?

Niko Köbler: Die JUG Darmstadt ist eine reine Non-Profit-Organisation, ohne dass wir in einem Verein oder Ähnlichem organisiert sind. Die „Leitung" besteht wir sieben Personen (neben mir noch Gerd Aschemann, Marcel Bruch, Jörn Hameister, Sebastian Rose, Falk Sippach und Jan Westerkamp), die sich mehr oder weniger die Organisation teilen. So hat jeder Einzelne recht wenig Arbeit und dennoch können wir ein interessantes und abwechslungsreiches Programm auf die Beine stellen.

Was zeichnet die JUG Darmstadt aus?

Niko Köbler: Durch unser großes Orga-Team haben wir Beziehungen in allen Bereichen der Java-Welt und können so unseren Gästen ein sehr unterschiedliches und facettenreiches Programm anbieten. Unsere Zuhörer kommen nicht nur aus Darmstadt und der näheren Umgebung, wir haben durchaus den einen oder anderen Gast, der ein paar Kilometer auf sich nimmt, nur um bei unseren Talks anwesend zu sein. Eines der Highlights ist der Besuch einer benachbarten Studentenkneipe im Anschluss an die Vorträge. Dort ergibt sich für die Besucher und den Speaker die Möglichkeit, über den Vortrag und die zahlreichen Java-Themen zu fachsimpeln und zu diskutieren.

Wie viele Veranstaltungen gibt es pro Jahr?

Niko Köbler: Wir planen einen Vortrag im Monat, was bislang recht gut funktioniert hat. Zudem ist unser „Backlog" an Themen und Speakern gut gefüllt. Im letzten Jahr haben wir außerdem ein Code-Retreat mit ausgerichtet und im Jahr 2014 planen wir die eine oder andere Sonderveranstaltung.

Wie bist du zur JUG Darmstadt gekommen?

Niko Köbler: Im Jahr 2012 hatte ich dort selbst einen Vortrag gehalten und bin so mit dem damaligen Organisator in Kontakt gekommen. Als ich dann während eines anderen Vortrags hörte, dass er aufhört und die Organisation abgibt, habe ich ihn einfach darauf angesprochen, da ich mich sowieso mehr in der Community engagieren wollte. Letztendlich kam dann so das größere Orga-Team zusammen.

Was motiviert dich besonders, bei der JUG Darmstadt mitzuarbeiten?

Niko Köbler: Es macht Spaß, für andere Java-Begeisterte interessante Vorträge zu organisieren. Das Feedback von unseren Zuhörern zeigt uns, dass wir das Richtige tun. Nicht zuletzt ist es natürlich auch für mich eine Form der Weiterbildung, um stets am Ball zu bleiben und zu wissen, was gerade im Java-Land angesagt ist.

Was bedeutet Java für dich?

Niko Köbler: Java ist schon lange mehr als nur eine Programmiersprache. Durch die Möglichkeit, viele verschiedene (Skript-)Sprachen zur Laufzeit auf der VM ausführen zu können, hat Java als Thema eine unheimlich große Verbreitung gefunden.

Welchen Stellenwert besitzt die Java-Community für dich?

Niko Köbler: Java ist hundert Prozent Community! Durch die weite Verbreitung der Sprache kann die Weiterentwicklung des gesamten Ökosystems auch nur aus der Community kommen. Wenn ein Hersteller den Anwendern seine Ideen hier aufdrücken wollte, würde er das ziemlich schnell zu spüren bekommen. Das hat meines Erachtens auch Oracle dazu veranlasst, den Java Community Process stark zu verschlanken und stärker zu promoten. Außerdem begrüße ich die Entscheidung sehr, alle zukünftigen JDKs auf dem durch die Community entwickelten OpenJDK basieren zu lassen.

Wie sollte sich Java weiterentwickeln?

Niko Köbler: An vielen Stellen ist Java beziehungsweise sind viele APIs noch zu schwergewichtig, speziell im Java-EE-Umfeld. Zwar gibt es hier seit Java EE 6 stetige Verbesserungen, dennoch wünsche ich mir eine weitere starke Verschlankung.

Die Verzögerung des Java-8-Launch war sicherlich auch ein großes Minus für Java, speziell weil darin lange erwartete Features wie Lambdas enthalten sind, die es in anderen Sprachen schon seit einiger Zeit gibt.

Insgesamt muss Java schneller auf Anforderungen im Markt reagieren. Aber hier gilt wie überall: Wenn viele Parteien mitreden wollen und dürfen und zudem noch eine Abwärtskompatibilität für bestehende Anwendungen berücksichtigt werden muss, dauert es lange, bis ein Konsens gefunden ist.

Niko Köbler
niko@n-k.de
http://jugda.wordpress.com

Zur Person: Niko Köbler

Niko Köbler ist freiberuflicher Software-Architekt, Developer und Coach für (Java-)Enterprise-Lösungen, Integrationen und Web-Development. Er ist Mitbegründer der Qualitects Group, berät und unterstützt Kunden verschiedenster Branchen, hält Workshops und Trainings und führt Architektur-Reviews durch. Neben seiner Arbeit bei der Java User Group Darmstadt (JUG DA), schreibt er Artikel für Fachzeitschriften und ist regelmäßig als Sprecher auf Fachkonferenzen anzutreffen.

Die iJUG-Mitglieder auf einen Blick

Java User Group Deutschland e.V.
http://www.java.de

DOAG Deutsche ORACLE-Anwendergruppe e. V.
http://www.doag.org

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

Java User Group Köln
http://www.jugcologne.eu

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

Java User Group München (JUGM)
http://www.jugm.de

Java User Group Metropolregion Nürnberg
http://www.source-knights.com

Java User Group Ostfalen
http://www.jug-ostfalen.de

Java User Group Saxony
http://www.jugsaxony.org

Sun User Group Deutschland e.V.
http://www.sugd.de

Swiss Oracle User Group (SOUG)
http://www.soug.ch

Berlin Expert Days e.V.
http://www.bed-con.org

Java Student User Group Wien
www.jsug.at

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

Java User Group Hannover
https://www.xing.com/net/jughannover

Java User Group Augsburg
http://www.jug-augsburg.de

Java User Group Bremen
http://www.jugbremen.de

Java User Group Münster
http://www.jug-muenster.de

Java User Group Hessen
http://www.jugh.de

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

www.ijug.eu

Impressum

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

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

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

Redaktionsbeirat:
Ronny Kröhne, IBM-Architekt;
Daniel van Ross, NeptuneLabs;
Dr. Jens Trapp, Google; André Sept, InterFace AG

Titel, Gestaltung und Satz:
Alexander Kermas,
DOAG Dienstleistungen GmbH
Foto Titel © Danussa/Fotolia.com
Foto S. 18 © vectomart/Fotolia.com
Foto S. 23 © wacomka/Fotolia.com

Anzeigen:
Simone Fischer
anzeigen@doag.org

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

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

Unsere Inserenten

aformatik Training und Consulting GmbH & Co. KG, www.aformatik.de — *S. 3*

DOAG e.V. www.doag.org — *U 2, U 4*

Hahn-Littlefair communication www.hahn-littlefair.de — *S. 9*

Heise-Verlag www.heise.de — *U 3*

Netpioneer GmbH www.netpioneer.de — *S. 37*

TEAM GmbH www.team-pb.de/ — S. 55

DOAG 2014 Development

4. Juni 2014 in Düsseldorf

Datenbank-Design

Geodaten

BPM

SOA Suite

Weblogic

Kein Superheld ...

... nur das richtige

Handwerkzeug

Forms und ADF

PLSQL und APEX

Mobile

RAD

Frühbucherrabatt
11.April 2014

DOAG
Deutsche ORACLE-Anwendergruppe e.V.

development.doag.org

MAGAZIN FÜR PROFESSIONELLE INFORMATIONSTECHNIK

Entwickler-Konferenzen 2014

Karlsruhe, IHK – 5. bis 7. Mai 2014

para//el 2014

Softwarekonferenz für Parallel Programming, Concurrency und Multi-Core-Systeme.

www.parallel2014.de

30. Juni bis 2. Juli 2014
KOMED im MediaPark, Köln

Die neue Konferenz für Enterprise JavaScript

Zeitgemäße Webanwendungen entwickeln – professionelle Infrastrukturen betreiben

enterJS 2014

www.enterjs.de

continuous lifecycle 2014

Rosengarten, Mannheim
10. bis 12. November 2014

Prozesse – Tools – Erfahrungen

Die Konferenz zu Agile ALM, Continuous Delivery und DevOps

www.continuouslifecycle.de