04-2016 | Winter | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

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

Java ist vielseitig

JUnit 5
Das nächste große Release steht vor der Tür

Ansible
Konfigurationsmanagement auch für Entwickler

Spring Boot Starter
Komfortable Modularisierung und Konfiguration

4 191978 304903 04

iJUG Verbund

Wolfgang Taschner
Chefredakteur Java aktuell

Was ist mit Java EE 8 los?

Obwohl Oracle als Mitglied des Java Community Process (JCP) für die Weiterentwicklung von Java EE verantwortlich ist, hat das Unternehmen seit November 2015 seine Mitarbeit an dem Projekt eingestellt. Kunden, die Java EE als strategisches Projekt betrachten, machen sich große Sorgen. Mittlerweile hat sich hinter den Kulissen einiges getan.

Markus Eisele, stellvertretender Leiter der DOAG Java Community, hat die aktuelle Situation in einem Artikel zusammengefasst *(siehe „http://tinyurl.com/jj5wrhk")*. Nach dem momentanen Stillstand bieten sich mehrere Alternativen für die Zukunft von Java EE an. Doch keine davon ist dazu geeignet, Begeisterung für Java EE 8 zu wecken. Oracle hat für die im Herbst stattfindende JavaOne neue Informationen angekündigt.

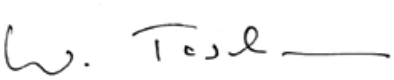
Markus Karg von der Java User Group Goldstadt (JUG PF) bringt es auf den Punkt: „Egal, welche Lösung Oracle auf der JavaOne aus dem Hut zaubert, letztendlich ist in jedem Fall der JCP und damit die Community brüskiert, da sich Oracle an keinerlei Regeln des JCP hält und der Welt seine Interessen ohne jegliche weitere Diskussion oder Abstimmung aufdrückt, ohne auf die zahlreichen Expert Groups zu hören." Er bringt in einem Interview eine Gruppe namens „Java EE Guardians" ins Spiel *(siehe „http://tinyurl.com/hlpzmnd")*.

Thomas Kurian, Oracle President für die Produkt-Entwicklung, gab vor einer Woche in einem Interview die Ziele von Oracle für Java EE bekannt *(siehe „http://tinyurl.com/jfryu2l")*. Für Markus Karg sind das alte Kamellen: „Diese Ziele sind bereits seit Jahren bekannt, aber bisher nicht in die Tat umgesetzt worden. Was wir umgehend brauchen, sind entsprechende JSRs, Zeitpläne, Early Drafts etc." Interessant sind in diesem Zusammenhang Äußerungen im JCP Executive Committee zum Thema „Java EE and microservices" *(siehe „http://tinyurl.com/hm94pn7")*.

Tobias Frech, stellvertretender Vorstandsvorsitzender des iJUG, betont: „Wenn Oracle zur JavaOne nicht umgehend ausreichend Ressourcen für den JCP und die JSRs vorsieht, dann wird die Community kein weiteres Vertrauen in Oracle setzen und nach anderen Alternativen suchen. Oracle vergibt sich dann die Möglichkeit, die Zukunft von Java EE, und damit auch die einer möglichen Basis für Cloud-Dienste, zu gestalten."

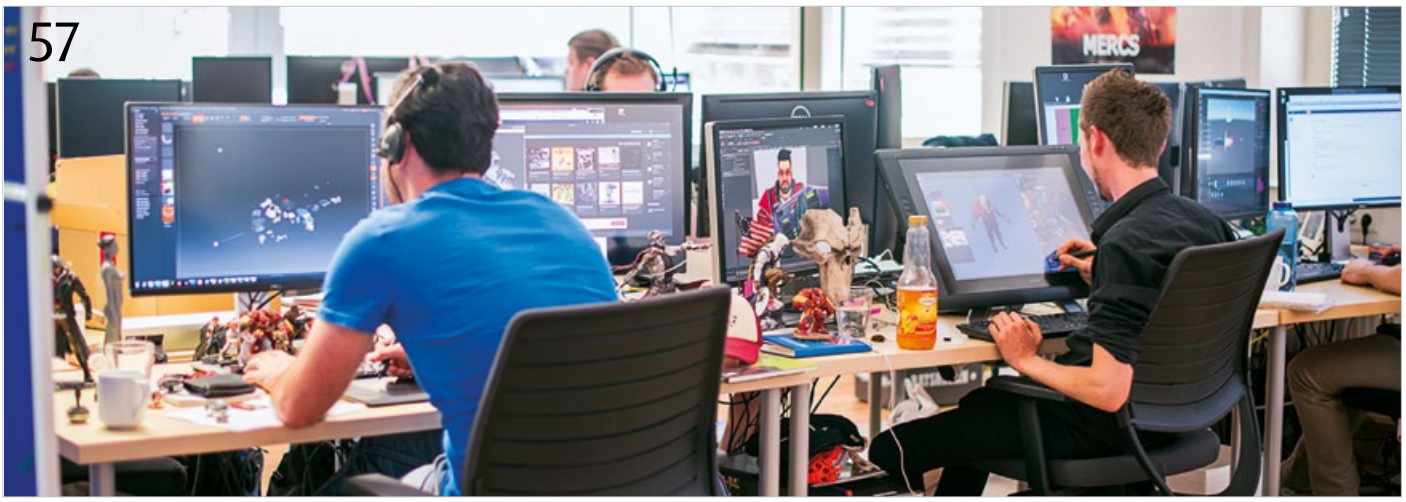
Ich finde, es ist an der Zeit, dass sich Oracle an seine Pflichten erinnert, die das Unternehmen als Mitglied des JCP übernommen hat, und Java EE im Sinne der Community weiterentwickelt. Sonst besteht zunehmend die Gefahr, dass sich die Community von Oracle unabhängige Lösungen sucht.

Ihr



Anwendung im laufenden Betrieb verwalten

Business Process Management auf Open-Source-Basis



Ein Blick hinter die Kulissen eines Spiele-Unternehmens

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

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 zweiten Quartal 2016.

30. April 2016

Java EE Guardians formieren sich

Nach seinem Ausstieg bei Oracle hat Java-Evangelist Reza Rahman die Java EE Guardians ins Leben gerufen, ein loser Zusammenschluss von Menschen, die Java EE wieder vorantreiben möchten. Die Kommunikation läuft hauptsächlich über ein Twitter-Handle („@javaee_guardian") und eine Google-Group („javaee-guardians"). Einer der ersten Tweets verweist auf eine Analyse im Blog von Josh Juneau, selbst Mitglied mehrerer Expert Groups zu Java-EE-8-JSRs: Dort präsentiert Josh Statistiken zu Commits und „Issue Resolutions" mehrerer JSRs und liefert damit objektive und für jeden nachvollziehbare Zahlen zum subjektiven Gefühl der Community, dass Oracle EE 8 momentan komplett vernachlässigt. Die Diskussionen laufen ziemlich heiß, wobei manchmal auch sehr emotional reagiert wird. Reza Rahmans Abgang bei Oracle ist wohl auf beiden Seiten noch nicht aufgearbeitet, um es vorsichtig zu formulieren. Im Sinne von Java wäre es gut, wenn die Diskussion schnell wieder nur auf der Sachebene geführt werden würde. Einige Guardians bringen einen Java-Fork ins Spiel. Insbesondere die rechtlichen Hürden (wie TCK-Lizenzen, IP-Rechte) dürften das aber zu einem fast aussichtslosen Unterfangen machen, so lange Oracle seine Kontrolle nicht freiwillig aufgibt – was kaum zu erwarten ist.

https://adtmag.com/blogs/waters-works/2016/03/java-ee-guardians.aspx

2. Mai 2016

Nodyn-Projekt wird wohl eingestellt

Node.js in der JVM – die Kombination scheint irgendwie kein Glück zu bringen. Nach dem Einstampfen des Project Avatar bei Oracle verfolgt nun RedHat seine Aktivitäten im Nodyn-Projekt nicht weiter. Es sei den Entwicklern nicht gelungen, den Node Package Manager darin zum Laufen zu bringen, berichtet der CTO von Red Hat Mobile – ohne diesen sei aber eine große Verbreitung kaum möglich.

http://www.infoworld.com/article/3063153/javascript/red-hat-ditches-effort-to-port-nodejs-to-java.html

4. Mai 2016

Hochverfügbarkeit mit Docker und Java EE

Bruno Souza, Präsident der brasilianischen JUG SouJava, und Elder Moraes haben eine Demo ausgearbeitet, in der sie zeigen, wie gut sich Java EE und Docker beim Erstellen hochverfügbarer und portabler Applikationen ergänzen. Das Ganze lässt sich in Minuten mit einer beliebigen eigenen Applikation nachvollziehen und zeigt sehr schön, wie Docker und „immutable container" die Schwächen von Java EE ausgleichen, wenn es um das Erzeugen und die nahezu beliebige Portierung einer umfangreichen Infrastruktur geht. Neben dem relativ einfach gehaltenen Beispiel (Cluster aus TomEE-Servern mit der Applikation in jeweils einem Container plus ein Load Balancer auf NGINX-Basis) geben sie einen Ausblick, was weitere mögliche Schritte sein könnten.

https://blogs.oracle.com/java/entry/step_by_step_high_availability

6. Mai 2016

TeamFX: Community treibt OpenJFX voran

Unter dem Namen „TeamFX" hat sich Ende 2015 eine Gruppe von deutschsprachigen JavaFX-Enthusiasten zusammengetan, um die Weiterentwicklung des UI-Frameworks zu unterstützen und zu beschleunigen. Entwicklern außerhalb des OpenJDK-Teams soll die Mitwirkung erleichtert werden (etwa das Melden von Fehlern, aber auch Code Contributions). Für Jemanden, der nicht regelmäßiger Committer im OpenJDK-Projekt und mit entsprechenden JIRA-Zugängen etc. ausgestattet war, stellte dies eine extrem hohe Hürde dar, und TeamFX hat offensichtlich harte Diskussionen mit Oracle geführt, um eine Einigung über das technische und organisatorische Setup zu erzielen. Als kleinsten gemeinsamen Nenner hat das Team nun ein GitHub-Projekt aufgesetzt. Dieses ist zunächst mal ein Mirror der offiziellen Repositories, Interessierte können hier aber ohne weiteren organisatorischen Aufwand forken und Pull Requests stellen. Die TeamFX-Mitglieder übernehmen dann die Koordination mit dem offiziellen Projekt. Zwei von ihnen, Markus Karg und Hendrik Ebbers, sind OpenJDK-Committer und haben hier entsprechend einen kurzen Draht.

https://github.com/teamfx

14. Mai 2016

Neue Version des Java Community Process: „Broadening Membership"

Seit vorgestern ist eine neue Version des Java Community Process gültig. Das Release 2.10 ist im JSR-364 „Broadening JCP Membership" entstanden, dem dritten JSR der JCP.Next-Initiative. Entsprechend dem Titel gibt es viele Änderungen, die eine Mitgliedschaft im JCP sowie die aktive Mitwirkung erleichtern sollen. So werden beispielsweise generell keine Mitgliedsbeiträge mehr erhoben und es wird generell mit elektronischen Unterschriften gearbeitet, um den Anmelde-Prozess zu beschleunigen. Arbeitnehmern, deren Arbeitgeber kein Mitglied im JCP ist, und der auch nicht das „Employer Contribution Agreement" unterzeichnen möchte (mit dem er gewisse Intellectual Property-Rechte abtritt, die sich aus der Mitwirkung des Arbeitnehmers in JSRs ergeben), bietet sich

nun der Ausweg einer individuellen „Associate Membership". Für diese neue Form der Mitgliedschaft sind eigene Sitze im Executive Committee vorgesehen, für die nur Associate Members wahlberechtigt sind (wobei sie umgekehrt auch nur für diese wahlberechtigt sind). Darüber hinaus gibt es eine Reihe von kleineren Überarbeitungen des Review- und „Ballot"-Prozesses (Abstimmung über JSRs), insbesondere was Fristen anbelangt. Aus technokratischer Sicht sind das eindeutig Verbesserungen, aber letztlich kommt es darauf an, dass alle Seiten inklusive Oracle den JCP aktiv mit Leben füllen – siehe die aktuelle Diskussion um Java EE.

https://blogs.oracle.com/jcp/entry/java_community_process_jcp_version

31. Mai 2016

Oracle gegen Google, die Neuauflage

Vor ein paar Jahren hatte ich im Java-Tagebuch ausführlich über den ersten Rechtsstreit „Oracle gegen Google" um Android berichtet. Die Klage war damals von einem Bezirksgericht in Nordkalifornien abgelehnt worden. Das amerikanische Bundesberufungsgericht hatte ihn dann aber an das Bezirksgericht zurückgegeben. Dabei ging es nicht mehr um Patentverletzungen, sondern nur noch um Urheberrechtsverletzungen, da Google eine Reihe von Java-APIs für Android wiederverwendet hat (wobei die Implementierungen bis auf eine Ausnahme neu sind). Anders als das Bezirksgericht war aber das Berufungsgericht der Auffassung, dass auch APIs dem Urheberrechtsschutz unterliegen. Google hatte noch versucht, den US Supreme Court zu einer Entscheidung zu bringen, dass die Nutzung der APIs von mit „fair use" bezeichneten Ausnahmen im amerikanischen Urheberrecht gedeckt ist (die etwa Standardisierung ermöglichen sollen). Dieser hatte aber eine Prüfung des Urteils abgelehnt.

Das neue Verfahren vor dem Bezirksgericht – bei dem Oracle inzwischen rund neun Milliarden Dollar fordert – ist gerade mit einem erneuten Sieg für Google zu Ende gegangen. Die Jury hat die API-Nutzung als „fair use" eingestuft. Ein für Europäer eher belustigendes Detail: Da Mitglieder von Jurys in den USA unvoreingenommen sein sollen, war niemand mit Sachverstand darin vertreten. Oracle hatte auch den einzigen Informatiker unter den Kandidaten abgelehnt – es scheint ihnen nicht geholfen zu haben. Oracle hat bereits verkündet, dass es mit diesem Urteil freie Software bedroht sieht, und will erneut in Berufung gehen. Anwältin Hurst verkündet in einem Kommentar auf Ars Technica: „Falls dieser Spruch Gesetz wird, könnt ihr euch von der General Public License (GPL) verabschieden." Die unabhängigen Kommentare lauten meist etwas anders: „Ein Sieg für Innovation und Gerechtigkeit" – aber: „So dreist-naiv, wie der Internetkonzern [Google] beim Prozess aufgetreten ist, das gehört eigentlich nicht auch belohnt" schreibt Alexander Neumann auf Heise Online. Es ist noch nicht vorbei …

http://arstechnica.com/business/2016/06/the-googleoracle-decision-was-bad-for-copyright-and-bad-for-software

31. Mai 2016

Ceylon lebt

Ceylon wird an prominenter Stelle in der Mai/Juni-Ausgabe des Java Magazine dargestellt. Die Ende 2013 mit Release 1 gestartete Programmiersprache, die sowohl in Java-Byte-Code als auch in JavaScript kompiliert werden kann, fristet bislang eigentlich eher ein Nischen-Dasein. Bezogen auf den TIOBE-Index und andere Rankings wie das von RedMonk, die die Popularität von Programmiersprachen zu messen versuchen, würde statt des Elefanten eher eine Mücke als Logo passen. Aber: „Popularität ist nicht alles" (und die Rankings sind eh allesamt umstritten). Ein Blick ins Java Magazine lohnt also für alle, die von Ceylon noch nichts gehört haben. Für alle, die es nutzen (oder eine andere neuere Sprache für die JVM): In der Java aktuell ist genau wie im Java Magazine immer Platz für „Adoption Stories".

http://www.oracle.com/technetwork/java/javamagazine/index.html

3. Juni 2016

JavaFX auf Mobilgeräten mit Gluon

Die Oracle-Strategie für JavaFX steht ja aktuell unter verstärkter Beobachtung *(siehe Pressemitteilung des iJUG von 9.März unter „http://www.ijug.eu/presse.html")*. Aber zumindest hält das Open-Source-Projekt „Gluon" (hinter dem ein kommerzielles Support-Modell steckt) den „Write Once, Run Anywhere"-Gedanke hoch. Die Firma hat nach dem Ende des RoboVM-Projekts (das durch Aufkäufe bei Microsoft gelandet war) eine eigene „Gluon VM" für iOS und Android angekündigt, die auf dem OpenJDK basiert. Wie man eine OpenJFX-Anwendung für den Einsatz auf Smartphones entwickelt, ist in einem gemeinsamen Blog-Eintrag zweier Entwickler von Oracle und Gluon beschrieben.

https://blogs.oracle.com/java/convert-an-openjfx-app-to-mobile

Eine sehr ausführliche Demonstration, wie sich auf Grundlage von JavaFX hybride Desktop- und Web- Applikationen entwickeln lassen, ist im Oracle-Java-Blog zu finden. Mithilfe der JavaFX WebView und dem Meta-Framework Oracle Jet (ein Paket von CSS- und Java-Script Frameworks) demonstriert der Autor Thomas Kruse von der JUG Münster, wie leicht sich beide Welten integrieren lassen – aber auch, welche Herausforderungen noch warten.

https://community.oracle.com/docs/DOC-1001176

9. Juni 2016

Adam Bien und Markus Karg zu Java EE 8

Adam Bien, einer der bekanntesten deutschen Java-Entwickler, von Heise Online zu den aktuellen Entwicklungen um Java EE befragt, hat sich erstaunlich entspannt gezeigt: „Da ich schon mit Java EE 6 sehr zufrieden bin, sehe ich eine mögliche Verschiebung der Java EE 8 als unkritisch an." So kann man das natürlich auch sehen … Java EE hat (nicht nur) seiner Ansicht nach WebLogic zu großer Popularität und entsprechenden Verkaufzahlen verholfen – die aktuelle Situation würde die Kunden dann eher in die Arme von Wildfly und Payara treiben. Hier könnte dann der monetäre Anreiz für Oracle liegen, die Strategie wieder zu ändern. Aber: „Möglicherweise möchte Oracle die Aufmerksamkeit auf das eigene Cloud-Angebot richten." Vielleicht hat Oracle den Niedergang von WebLogic ja bereits in seine proprietäre Cloud-Strategie eingepreist.

http://www.heise.de/developer/artikel/Interview-mit-Adam-Bien-zur-aktuellen-Entwicklung-von-Java-EE-3228532.html

Etwas kämpferischer zeigt sich Markus Karg von der JUG Goldstadt, Mitstreiter bei den Java EE Guardians, in einem Interview mit JAXenter. Den Java Community Process als „runden Tisch der Industrie" hält er für die weitere Entwicklung nicht für geeignet und wirbt für eine „echte Entwickler-Organisation, die ein reines, freies und unabhängiges Community-Java-EE entwickelt" – nach

demokratischen Prinzipien. Auch er sieht die veränderten architektonischen Rahmenbedingungen hin zur Cloud und zu Microservices als die große Herausforderung an, der sich Java EE stellen muss – unter Beibehaltung der Rückwärtskompatibilität, was es noch schwieriger macht; aber es wäre sinnvoll, „neben JRMP und IIOP auch SOAP und letztendlich EJB als „deprecated" zu markieren."

https://jaxenter.de/java-ee-guardians-interview-41402

14. Juni 2016

Java EE Guardians: Website und Petition

Die Webseite der Java EE Guardians ist offiziell gestartet *(siehe „http://javaee-guardians.io")*. Der erste Aufhänger ist eine Petition auf „change.org", mit der Oracle zum Umdenken gebracht werden soll. Liebesbekundungen zu Java alleine werden aber sicher nicht ausreichen; eine große Masse an Unterzeichnern könnte eventuell helfen. Aber letztlich muss Oracle davon überzeugt werden, dass die Verunsicherung in der Community für das Unternehmen in Dollar messbare Konsequenzen hat. Oracle tickt anders als Sun. Die aktuelle Situation ist nicht den technisch Verantwortlichen bei Oracle zuzuschreiben, sondern mit ziemlicher Sicherheit eine bewusste unternehmerische Entscheidung.

http://tinyurl.com/he3fyc5

27. Juni 2016

Micro Profile für Java EE

Auf der DevNation-Konferenz ist das Java „Micro Profile" angekündigt worden. Es ist eine abgespeckte Java-EE-Spezifikation, entsprechend der mit Java EE eingeführten Profil-Definition (Web Profile ist sicherlich das bekannteste). Es handelt sich um keinen Standard an sich, sondern um eine Definition, was aus dem umfangreichen Java-EE-Standard dazugehört. In diesem Fall sind das bislang JAX-RS (inklusive Servlet), CDI, und JSON-P als der kleinste gemeinsame Nenner für Microservice-Implementierungen. Hinter dem Projekt mit eigener Website stecken vier Firmen, die mit Application-Servern Geld verdienen (IBM, RedHat, Payara, Tomitribe), sowie die London Java Community, aber grundsätzlich sind alle anderen zur Mitarbeit eingeladen.

http://microprofile.io

29. Juni 2016

Wahlen zum JCP Exceutive Committee

Die diesjährigen Wahlen zum Leitungsgremium des Java Community Process sind angekündigt worden. Da jetzt die neue JCP-Version 2.10 gilt, sind zwei neue Sitze für „Associate Members" zu vergeben. Nominierungen dafür werden bis Ende August entgegengenommen. Mitwählen darf jedes Associate Member, das bis zum 27. Oktober beitritt.

https://jcp.org/en/participation/committee

18. Juli 2016

ZeroTurnaround-Umfrage zu Trends und Werkzeugen

ZeroTurnaround hat im Jahr 2016 wieder eine Umfrage unter Java-Entwicklern durchgeführt. Der komplette Report ist frei verfügbar und teilweise sehr amüsant. Es gab 2.040 Antworten, also schon eine recht große Zahl für solch eine Umfrage. Natürlich ist das trotzdem nicht repräsentativ – vermutlich sind diejenigen, die veraltete Java-Versionen und Werkzeuge einsetzen, auch der Umfrage ferngeblieben. Aber ein paar Trends lassen sich trotzdem ablesen. Hier einige Details: Immerhin ein gutes Drittel der Teilnehmerinnen und Teilnehmer geben an, Microservice-Architekturen zu nutzen. Von den übrigen knapp zwei Dritteln planen wiederum nur zwölf Prozent, diese in Zukunft zu nutzen, während immerhin ein Drittel (also etwa 20 Prozent von allen) das nicht tun möchten. Diejenigen, die Microservices bereits nutzen, wurden gefragt, ob das ihren Job leichter oder schwerer gemacht hat. Die Tendenz geht schon in Richtung „leichter" oder „viel leichter" (40 Prozent), aber 18 Prozent haben auch mit „(viel) schwerer" geantwortet – es ist nicht immer gut, dem Hype zu folgen …

Auf die Frage, welche JVM-Sprache sie am meisten nutzen, haben 93 Prozent mit Java geantwortet, drei Prozent mit Groovy, und zwei Prozent mit Scala. „Welche Java-Version nutzt du für die Haupt-App in der Produktion" war eine weitere Frage: Hier antworteten schon 62 Prozent mit Java 8 (28 Prozent mit Java 7 und neun Prozent mit Java 6). Die meist-genutzte Java-EE-Version ist: Gar keine (42 Prozent), gefolgt von EE 7 (31 Prozent) und EE 6 (17 Prozent). Zehn Prozent nutzen EE 5 und Vorgänger-Versionen mit Oldtimer-Kennzeichen. Der Report enthält noch viel mehr Details (Maven liegt übrigens bei 68 Prozent vor Gradle mit 16 Prozent) – also bei Interesse einfach dem Link folgen.

https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016

21. Juli 2016

Statement von Oracle zu Java EE

Mike Moeller, Vice President für Marketing und PR bei Oracle, hat sich gegenüber der britischen IT- Nachrichten-Plattform „The Register" zur Oracle-Java-Strategie geäußert: „Man arbeite eng mit wichtigen Partnern in der Java-Community an der Fertigstellung des Java EE 8 Proposal." Die nächste Java-EE-Version soll – keine Überraschung – auf Microservices in großen verteilten Systemen und auf Container-basierte Umgebungen ausgerichtet sein. Details wird es aber erst zur JavaOne geben. Wirklich beruhigen kann er die Diskussionen in der Community damit nicht. „Zeit schinden bis zur JavaOne ", lautet ein typischer Kommentar.

http://www.theregister.co.uk/2016/07/07/oracle_java_ee_8

WebSphere ist für Java EE 7 zertifiziert

WebSphere, die Vollversion von IBMs Application Server, ist jetzt auch Java-EE-7-zertifiziert. Für die abgespeckte Edition WebSphere Liberty gilt das ja schon länger.

https://blogs.oracle.com/theaquarium/entry/websphere_liberty_now_java_ee

Andreas Badelt
Leiter der DOAG SIG Java

Er organisierte von 2001 bis 2015 ehrenamtlich die Special Interest Group (SIG) Development sowie die SIG Java der DOAG Deutsche ORACLE-Anwendergruppe e.V. und war in dieser Zeit ehrenamtlich in der Development Community aktiv. Seit 2015 ist Andreas Badelt Mitglied in der neugegründeten Java Community der DOAG Deutsche ORACLE-Anwendergruppe e.V.

JUnit 5

Stefan Birkner, ThoughtWorks, und Marc Philipp, Citrix

Zehn Jahre nach Erscheinen von JUnit 4 steht das nächste große Release des weitverbreiteten Test-Frameworks vor der Tür. Dieser Artikel gibt einen Überblick über die Änderungen, die auf Entwickler zukommen, und die neuen Möglichkeiten, die sich dadurch ergeben.

Das Entwicklerteam von JUnit 5 *[1]* arbeitet darauf hin, die neue Version des Frameworks gegen Ende des Jahres zu veröffentlichen. Es gibt mittlerweile schon das zweite Milestone-Release, das alle wesentlichen Features enthält und mit dem jeder Entwickler JUnit 5 ausprobieren kann. Da dies jedoch nicht das finale Release ist, können sich Teile des API noch ändern und vereinzelt Bugs auftreten.

Wer JUnit 5 testen will, hat dazu mehrere Möglichkeiten. Seit dem Release 2016.2 unterstützt IntelliJ IDEA JUnit 5. Für Gradle und Maven stellt das JUnit-Team entsprechende Plug-ins zur Verfügung, bis die Build-Tools eine eigene Implementierung anbieten. Daneben gibt es den Console Launcher, mit dem sich Tests von der Kommandozeile starten lassen.

Für die Migration von JUnit 4 zu JUnit 5 gibt es zwei Strategien. Zum einen kann JUnit 5 auch JUnit-3- und JUnit-4-Tests ausführen. Andererseits lassen sich JUnit-5-Tests mit JUnit 4 ausführen. Dazu stellt das Team den JUnit-4-Runner „org.junit.platform.runner.JunitPlatform" zur Verfügung. Das Beispiel in *Listing 1* zeigt einen solchen Test, der sich in jeder IDE und mit jedem Build-Tool ausführen lässt.

```
@RunWith(JUnitPlatform.class)
public class SomeTest {
  //Test that uses new API and features
}
```

Listing 1

Tests schreiben

Mit dem neuen JUnit-Release werden die bisherigen JUnit-4-Tests durch JUnit Jupiter abgelöst. Dabei handelt es sich um eine Test-Engine zum Ausführen von Tests und das API zum Schreiben dieser Tests. Das API ist annotationsbasiert und ähnelt demjenigen von JUnit 4 *(siehe Listing 2)*.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

class FirstJUnit5Tests {
  @Test
  void myFirstTest() {
      assertEquals(2, 1 + 1);
  }
}
```

Listing 2

Der wesentliche Unterschied ist, dass die Annotation „@Test" aus dem neuen JUnit-Jupiter-Package kommt und die Klasse „org.junit.Assert" von „org.junit.jupiter.api.Assertions" abgelöst wird. Eine nette Kleinigkeit ist, dass Testklassen und -methoden nicht mehr „public" sein müssen.

Exceptions

Die erste größere Änderung betrifft das Testen von Exceptions. Die „@Test"-Annotation besitzt kein Attribut „expected" mehr. JUnit Jupiter nutzt stattdessen die Möglichkeiten von Java 8 und stellt eine spezielle Assertion für Exceptions bereit. Dieser Assertion wird der Lambda-Ausdruck übergeben, der die Exception wirft *(siehe Listing 3)*.

```
class LongTest {
  @Test void emptyStringCannotBeParsed() {
    assertThrows(
      NumberFormatException.class,
      () -> Long.parseLong("")
    );
  }
}
```

Listing 3

```
class LongTest {
  @Test
  void emptyStringCannotBeParsed() {
    Exception e = expectThrows(
      NumberFormatException.class,
      () -> Long.parseLong("")
    );
    assertEquals("For input string: \"\"", e.getMessage());
  }
}
```

Listing 4

Sollen neben dem Auftreten einer Exception auch deren Eigenschaften überprüft werden, lässt sich die geworfene Exception mittels „expectThrows" einer Variablen zuweisen. So kann diese mit beliebigen Assertions überprüft werden *(siehe Listing 4)*.

Will man nur einen Teil aller Tests ausführen, da manche etwa sehr lange brauchen, dann müssen die Tests kategorisiert werden. In JUnit Jupiter erfolgt eine Kategorisierung, indem man Tests mit Tags versieht *(siehe Listing 5)*.

```
@Tag("slow")
@Test
void somethingThatNeedsALotOfTime() {
  ...
}
```

Listing 5

Es ist auch möglich, einer Testmethode mehrere Tags zuzuweisen oder ganze Testklassen zu taggen. Die Ausführung lässt sich dann auf einzelne Tags beschränken oder es lassen sich einzelne Tags ausschließen.

Beim Auszeichnen einzelner Tests mit Tags fällt auf, dass sich bestimmte Kombinationen von Annotationen wiederholen. JUnit Jupiter erlaubt es daher, diese Annotationen zu einer einzigen Annotation zusammenzusetzen. So lassen sich beispielsweise die „@Tag"- und die „@Test"-

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("slow")
@Test
public @interface SlowTest {
}
```

Listing 6

```
@SlowTest
void somethingThatNeedsALotOfTime() {
  ...
}
```

Listing 7

```
class StackTest {

  @Test void isInstantiatedWithNew() {
    new Stack<Object>();
  }

  @Nested class WhenNew {
    Stack<Object> stack;

    @BeforeEach void createNewStack() {
      stack = new Stack<>();
    }

    @Test void isEmpty() {
      assertTrue(stack.isEmpty());
    }

    @Test void throwsExceptionWhenPopped() {
      assertThrows(
        EmptyStackException.class, () -> stack.pop()
      );
    }
   }
  }
}
```

Listing 8

```
class DynamicTestsDemo {
  @TestFactory
  Collection<DynamicTest> dynamicTestsFromCollection() {
    return Arrays.asList(
      dynamicTest(
        "1st dynamic test", () -> assertTrue(true)
      ),
      dynamicTest(
        "2nd dynamic test", () -> assertEquals(4, 2 * 2)
      )
    );
  }
}
```

Listing 9

```
public interface CollectionContract<T> {
  Collection<T> createEmptyCollection();

  @Test default void emptyCollectionHasSize0() {
    Collection<T> collection = createEmptyCollection();
    assertEquals(0, collection.size());
  }
}
```

Listing 10

Annotationen aus dem obigen Beispiel zu einer einzigen „@SlowTest"-Annotation kombinieren *(siehe Listing 6)*. *Listing 7* zeigt, wie sich damit das vorhergehende Beispiel vereinfachen lässt.

Verschachtelte Tests

Vielfach haben mehrere Tests in einer Klasse den gleichen Kontext. Dies bildet sich beispielsweise dadurch ab, dass sie den gleichen Setup-Code verwenden. In JUnit 4 ließ sich das durch die externe Erweiterung „HierarchicalContextRunner" *[2]* abbilden. JUnit Jupiter unterstützt eine solche Gliederung von Tests, bei der verschachtelte Testklassen mit der „@Nested"-Annotation versehen sind. Solche Klassen übernehmen unter anderem den Setup-Code aus den „@BeforeEach"-Methoden aller umschließenden Testklassen. Ein Beispiel hierfür ist der folgende Ausschnitt aus einer Testklasse für eine Stack-Implementierung *(siehe Listing 8)*. Die vollständige Testklasse findet sich in der JUnit-Dokumentation *[3]*.

Dynamische Tests

Ein komplett neues Feature von JUnit Jupiter ist die Möglichkeit, Tests dynamisch zu erstellen. Dazu wird eine Methode implementiert, die einen „Stream", ein „Iterable" oder einen „Iterator" mit Objekten vom Typ „DynamicTest" erzeugt und mit der Annotation „@TestFactory" versehen ist. Ein „DynamicTest" ist ein Tupel aus einem Namen und dem Testcode *(siehe Listing 9)*.

Contract-Tests für Interfaces

Interfaces definieren einen Contract zwischen demjenigen, der das Interface verwendet, und dem, der es implementiert. Beispielsweise hat eine leere Collection keine Elemente. Solche Contracts werden in der Regel nur in Javadoc-Kommentaren festgelegt. Mit JUnit Jupiter gibt es die Möglichkeit, Tests für solche Contracts in Form eines Interface mit Default-Methoden zur Verfügung zu stellen. Eine Testklasse kann dieses Interface und die entsprechenden Methoden zum Erzeugen von Objekten implementieren. Dadurch lässt sich einfach testen, dass der Contract erfüllt ist. Für die beiden oben genannten Eigenschaften einer Collection kann ein Contract-Test wie in *Listing 10* aussehen.

Testnamen

Es taucht immer wieder der Wunsch auf, Tests mit beliebigen Namen zu versehen.

Dazu lassen sich jeder Test und auch die Testklasse mit der „@DisplayName"-Annotation versehen, der ein beliebiger String zugewiesen werden kann *(siehe Listing 11)*.

```
@DisplayName("A stack")
class TestingAStackDemo {
  @Test
  @DisplayName("is instantiated with new Stack()")
  void isInstantiatedWithNew() {
    new Stack<>();
  }
}
```

Listing 11

Umbenennungen

Viele Annotationen aus JUnit 4 wurden auch in JUnit Jupiter übernommen. Vielfach wurden die Namen geändert, da sich über die Jahre gezeigt hat, dass die bisherigen Namen nicht eindeutig genug sind. *Tabelle 1* zeigt die Umbenennungen der Annotationen.

JUnit Jupiter erweitern

JUnit 4 hat zwei Erweiterungsmechanismen: „Runner" und „Rules". Es hat sich jedoch im Laufe der Zeit gezeigt, dass diese Schwächen aufweisen. So ist es nicht möglich, einer Testklasse zwei „Runner" zuzuweisen. JUnit Jupiter führt deshalb ein neues Erweiterungskonzept ein. Damit ein Test eine Erweiterung verwendet, wird die Testklasse oder -methode mit der Annotation „@ExtendWith" und einem Verweis auf die Erweiterungsklasse versehen *(siehe Listing 12)*.

```
@ExtendWith(MyFirstExtension.class)
class SomeTest {
  ...
}
```

Listing 12

Man kann auch mehrere Erweiterungsklassen angeben. Zudem lässt sich „@ExtendWith" mit anderen Annotationen zu zusammengesetzten Annotationen kombinieren. JUnit Jupiter bietet mehrere Einsprungspunkte für Erweiterungen. Jeder dieser Einsprungspunkte wird durch ein eigenes Interface repräsentiert. Darunter fallen:

- Einsprungspunkte für alle Phasen des Lebenszyklus eines Tests
- „ParameterResolver", die Objekte für Tests bereitstellen
- „ExecutionConditions" zum dynamischen Deaktivieren von Tests
- „TestExecutionExceptionHandler" reagieren auf „Exceptions"
- „TestInstancePostProcessor" zum Modifizieren der Test-Instanz

Eine Erweiterung kann mehrere dieser Interfaces miteinander kombinieren. JUnit Jupiter verwendet das Extension-Konzept für interne Features wie „@Disabled". Diese Features müssen allerdings nicht explizit mit „@ExtendWith" aktiviert sein.

```
class ServerExtension
  implements BeforeEachCallback, AfterEachCallback {

  @Override
  public void beforeEach(TestExtensionContext context) {
    Server server = new Server();
    server.start();
    context.getStore().put(
      ServerExtension.class.getName(), server
    );
  }

  @Override
  public void afterEach(TestExtensionContext context){
    Server server = getServer(context)
    server.stop();
  }

  private Server getServer(ExtensionContext context) {
    return context.getStore()
      .get(ServerExtension.class.getName(), Server.class);
  }
}
```

Listing 13

Lebenszyklen-Callbacks

Erweiterungen können auf bestimmte Phasen im Lebenszyklus eines Tests reagieren. Im Wesentlichen lässt sich damit Code in Klassen auslagern, der sich ansonsten in „@BeforeAll"-, „@Before"-, „@After"- und „@AfterAll"-Methoden befindet. Da JUnit Jupiter keine Zusicherung dahingehend macht, wie oft und wann eine Erweiterung instanziiert wird, haben alle in den Interfaces definierten Methoden einen Parameter vom Typ „ExtensionContext". Dieser stellt Informationen zum aktuellen Test bereit und bietet einen „Store" an, in dem sich Objekte hinterlegen lassen, auf die in anderen Callback-Methoden zurückgegriffen werden kann.

JUnit Jupiter	JUnit 4	Einsatzzweck
@Disabled	@Ignore	Der Test oder die Testklasse wird nicht ausgeführt
@BeforeAll	@BeforeClass	Diese Methode wird einmal vor allen Tests einer Testklasse ausgeführt
@AfterAll	@AfterClass	Diese Methode wird einmal nach allen Tests einer Testklasse ausgeführt
@BeforeEach	@Before	Diese Methode wird vor jedem Test ausgeführt
@AfterEach	@After	Diese Methode wird nach jedem Test ausgeführt

Tabelle 1

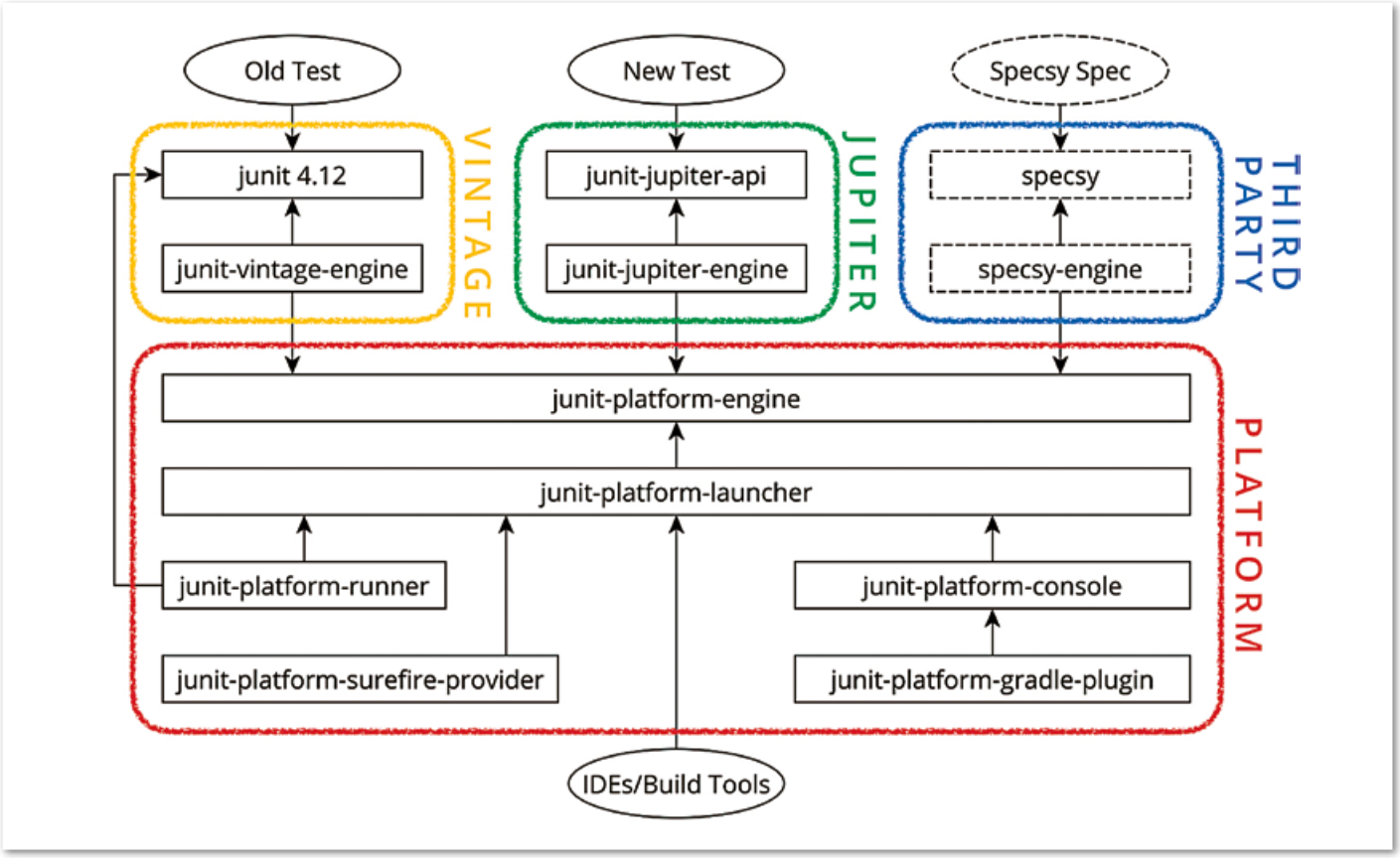


Abbildung 1: Die Architektur von JUnit 5

Das Beispiel in *Listing 13* zeigt die Verwendung anhand einer Extension, die vor dem Test einen Server startet und am Ende des Tests wieder stoppt.

Mithilfe von „ParameterResolvern" lassen sich Objekte in den Konstruktor der Testklasse sowie in „@Test"-, „@TestFactory"-, „@BeforeAll"-, „@Before"-, „@After"- und „@AfterAll"-Methoden injizieren. Die Erweiterung kann anhand des Parameter-Typs, -Namens und anderer Eigenschaften entscheiden, ob sie ein Objekt für den Parameter zur Verfügung stellt. Die obige Erweiterung „ServerExtension" kann so das Server-Objekt in Testmethoden injizieren, damit diese den Port auslesen können *(siehe Listing 14)*.

„ExecutionConditions" sorgen dafür, dass bestimmte Tests ignoriert werden. Ein Beispiel dafür ist das Feature zum Ignorieren von Tests aufgrund der „@Disabled"-Annotation. Mithilfe von „TestExecutionExceptionHandler" kann eine Erweiterung auf Exceptions reagieren, die von einem Test geworfen werden. Beispielsweise lassen sich dadurch Aufräumcode triggern oder bestimmte Exceptions ignorieren, sodass der Test erfolgreich ist.

Eine Erweiterung vom Typ „TestInstancePostProcessor" bekommt die Instanz der Testklasse übergeben. Solche Erweiterungen sind dazu gedacht, diese Instanz zu modifizieren. Der Hauptanwendungsfall ist das Setzen von Feldern. Ein Beispiel ist die „MockitoExtension" aus dem „junit5-samples"-Repository *[3]*. Diese Erweiterung besetzt alle mit „@Mock" annotierten Felder mit passenden Mocks.

```
class ServerExtension implements BeforeEachCallback, AfterEachCallback,
ParameterResolver {
  ...

  @Override public boolean supports(
      ParameterContext parameterContext,
      ExtensionContext context) {
    return parameterContext.getParameter()
      .getType().equals(Server.class);
  }

  @Override public Object resolve(
     ParameterContext parameterContext,
     ExtensionContext context) {
   return getServer(context);
  }
}

@ExtendWith(ServerExtension.class)
class MyTest {
  @Test void doSomething(Server server) {
    int port = server.getPort();
    //do something with the server
  }
}
```

Listing 14

Test-Engines

Neben neuen Features zum Schreiben von Tests ist eines der Hauptziele von JUnit 5 eine modularisierte, erweiterbare Architektur, die eine flexible Weiterentwicklung ermöglicht. Statt eines einzigen JARs ist JUnit 5 daher in mehrere Module unterteilt,

die sich in drei Gruppen gliedern: „Platform", „Jupiter" und „Vintage" *(siehe Abbildung 1)*.

In „Platform" ist mit dem Launcher die Schnittstelle zwischen JUnit und IDEs/Build-Tools enthalten. Der Launcher wird verwendet, um Tests zu finden, zu filtern und sie auszuführen. Allerdings weiß der Launcher nichts darüber, wie ein Test eigentlich ausschaut – geschweige denn, wie man ihn ausführt. Hier kommt ein weiteres neues Konzept ins Spiel: Test-Engines. Eine Test-Engine ist die Abstraktion eines Testing-Frameworks und dafür verantwortlich, Tests aufzufinden und auszuführen, die spezifisch für das jeweilige Programmiermodell sind. Zur Laufzeit werden alle verfügbaren Test-Engines vom Launcher angesteuert. Dazu wird ein Plug-in-Mechanismus, das Java ServiceLoader API, verwendet.

JUnit stellt zwei Test-Engine-Implementierungen bereit: JUnit Jupiter und JUnit Vintage. Neue Tests verwenden das Jupiter-API und werden von der Jupiter-Test-Engine ausgeführt. Zum Ausführen von Tests, die mit JUnit 4 oder JUnit 3 geschrieben wurden, benötigt man die Vintage-Test-Engine.

Third-Party-Testing-Frameworks können Platform ebenfalls verwenden, um Tests auszuführen. Sie müssen lediglich das „TestEngine"-Interface implementieren und sich über den „ServiceLoader"-Mechanismus registrieren. JUnit beinhaltet mit „HierarchicalTestEngine" sogar eine abstrakte Basisklasse, bei der ein Großteil der Ausführungslogik fertig implementiert ist. Es gibt bereits zwei Testing-Frameworks, die eine eigene Test-Engine zur Verfügung stellen: Specsy für Scala *[4]* und Spek für Kotlin *[5]*.

Fazit

JUnit 5 mit den beiden Teilen JUnit Jupiter und JUnit Platform stellt das JUnit-Projekt auf neue Füße, sodass es für die Zukunft gerüstet ist. Der Wechsel auf die neue Version wird dadurch erleichtert, dass die neue Version vorwärts- und rückwärtskompatibel ist. Die neuen Erweiterungspunkte bieten die Möglichkeit für JUnit-Erweiterungen, die mit JUnit 4 noch nicht möglich waren.

Das JUnit-Team freut sich über alle User, die die neue Version testen und Feedback geben. Jetzt ist es noch möglich, auf die Entwicklung Einfluss zu nehmen und Architekturprobleme einfach zu fixen. Das JUnit-Team ist besonders daran interessiert, ob Entwickler bisheriger Rules und Runner ihre Erweiterungen auch mit der neuen Version umsetzen können.

Die begonnene Integration von JUnit 5 in Build-Tools und IDEs zeigt die Akzeptanz, auf die das neue Release stößt. Projekte werden in Zukunft JUnit Jupiter verwenden und es wird spannend sein zu sehen, welche neuen Möglichkeiten in der Testautomatisierung sich dadurch eröffnen.

Links und Literatur

[1] JUnit 5, *http://junit.org/junit5*

[2] Hierarchical Context Runner, *https://github.com/bechte/junit-hierarchicalcontextrunner*

[3] JUnit 5 Samples, *https://github.com/junit-team/junit5-samples*

[4] Specsy, *http://specsy.org*

[5] Spek, *https://jetbrains.github.io/spek*

Stefan Birkner
mail@stefan-birkner.de

Stefan Birkner ist Software-Entwickler bei ThoughtWorks Deutschland. Er beschäftigt sich seit vielen Jahren damit, die Qualität von Software im Web-Umfeld zu verbessern. Dazu entwickelt er Test-Tools und arbeitet seit langer Zeit an JUnit mit.

Marc Philipp
mail@marcphilipp.de

Marc Philipp arbeitet als Software-Entwickler bei Citrix in Karlsruhe. Er ist langjähriger Maintainer von JUnit, Mitinitiator der Crowdfunding-Kampagne „JUnit Lambda" und JUnits „Keeper of the Green Bar" für das Jahr 2016.

A Fool with a Tool is still a Fool

Marco Schulz, Independent

Auch wenn zur Qualitätssteigerung der Software-Projekte in den letzten Jahren ein erheblicher Mehraufwand für das Testen betrieben wurde [1], ist der Weg zu kontinuierlich wiederholbaren Erfolgen keine Selbstverständlichkeit. Stringentes und zielgerichtetes Management aller verfügbaren Ressourcen war und ist bis heute unverzichtbar für reproduzierbare Erfolge.

Es ist kein Geheimnis, dass viele IT-Projekte nach wie vor ihre liebe Not haben, zu einem erfolgreichen Abschluss zu gelangen. Dabei könnte man durchaus meinen, die vielen neuen Werkzeuge und Methoden, die in den letzten Jahren aufgekommen sind, führten wirksame Lösungen ins Feld, um der Situation Herr zu werden. Verschafft man sich allerdings einen Überblick zu aktuellen Projekten, ändert sich dieser Eindruck.

Der Autor hat öfter beobachten können, wie diese Problematik durch das Einführen neuer Werkzeuge beherrscht werden sollte. Nicht selten endeten die Bemühungen in Resignation. Schnell entpuppte sich die vermeintliche Wunderlösung als schwergewichtiger Zeiträuber mit einem enormen Aufwand an Selbstverwaltung. Aus der anfänglichen Euphorie aller Beteiligten wurde schnell Ablehnung und gipfelte nicht selten im Boykott einer Verwendung. So ist es nicht verwunderlich, dass erfahrene Mittarbeiter allen Veränderungsbestrebungen lange skeptisch gegenüberstehen und sich erst dann damit beschäftigen, wenn diese absehbar erfolgreich sind. Aufgrund dieser Tatsache hat der Autor als Titel für diesen Artikel das provokante Zitat von Grady Booch gewählt, einem Mitbegründer der UML.

Oft wenden Unternehmen zu wenig Zeit zum Etablieren einer ausgewogenen internen Infrastruktur auf. Auch die Wartung bestehender Fragmente wird gern aus verschiedensten Gründen verschoben. Auf Management-Ebene setzt man lieber auf aktuelle Trends, um Kunden zu gewinnen, die als Antwort auf ihre Ausschreibung eine Liste von Buzzwords erwarten. Dabei hat es Tom De Marco bereits in den 1970er-Jahren ausführlich beschrieben *[2]*: Menschen machen Projekte *(siehe Abbildung 1)*.

Wir tun, was wir können, aber können wir etwas tun?

Das Vorhaben, trotz bester Absichten und intensiver Bemühungen ein glückliches Ende finden, ist leider nicht die Regel. Aber wann kann man in der Software-Entwicklung von einem gescheiterten Projekt sprechen? Ein Abbruch aller Tätigkeiten wegen mangelnder Erfolgsaussichten ist natürlich ein offensichtlicher Grund, in diesem Zusammenhang allerdings eher selten. Vielmehr gewinnt man diese Erkenntnis während der Nachbetrachtung abgeschlossener Aufträge. So kommen beispielsweise im Controlling bei der Ermittlung der Wirtschaftlichkeit Schwachstellen zutage.

Gründe für negative Ergebnisse sind meist das Überschreiten des veranschlagten Budgets oder des vereinbarten Fertigstellungstermins. Üblicherweise treffen beide Bedingungen gleichzeitig zu, da man der gefährdeten Auslieferungsfrist mit Personal-Aufstockungen entgegenwirkt. Diese Praktik erreicht schnell ihre Grenzen, da neue Teammitglieder eine Einarbeitungsphase benötigen und so die Produktivität des vorhandenen Teams sichtbar reduzieren. Einfach zu benutzende Architekturen und ein hohes Maß an Automatisierung mildern diesen Effekt etwas ab. Hin und wieder geht man auch dazu über, den Auftragnehmer auszutauschen, in der Hoffnung, dass neue Besen besser kehren.

Wie eine fehlende Kommunikation, unzureichende Planung und schlechtes Management sich negativ auf die äußere Wahrnehmung von Projekten auswirkt, zeigt ein kurzer Blick auf die Top-3-Liste der in Deutschland fehlgeschlagenen Großprojekte: Berliner Flughafen, Hamburger Elbphilharmonie und Stuttgart 21. Dank ausführlicher Berichterstattung in den Medien sind diese Unternehmungen hinreichend bekannt und müssen nicht näher erläutert werden. Auch wenn die angeführten Beispiele nicht aus der Informatik stammen, finden sich auch hier die stets wiederkehrenden Gründe für ein Scheitern durch Kostenexplosion und Zeitverzug.

Der Wille, etwas Großes und Wichtiges zu erschaffen, allein genügt nicht. Die Ver-

Abbildung 1: Problemlösung – „A bisserl was geht immer", Monaco Franze

antwortlichen benötigen auch die notwendigen fachlichen, planerischen, sozialen und kommunikativen Kompetenzen, gepaart mit den Befugnissen zum Handeln. Luftschlösser zu errichten und darauf zu warten, dass Träume wahr werden, beschert keine vorzeigbaren Resultate.

Große Erfolge werden meist dann erzielt, wenn möglichst wenige Personen bei Entscheidungen ein Vetorecht haben. Das heißt nicht, dass man Ratschläge ignorieren sollte, aber auf jede mögliche Befindlichkeit kann keine Rücksicht genommen werden. Umso wichtiger ist es, wenn der Projektverantwortliche die Befugnis hat, seine Entscheidung durchzusetzen, dies jedoch nicht mit aller Härte demonstriert.

Es ist völlig normal, wenn man als Entscheidungsträger nicht sämtliche Details beherrscht. Schließlich delegiert man die Umsetzung an die entsprechenden Spezialisten. Dazu ein kurzes Beispiel: Als sich in den frühen 2000er-Jahren immer bessere Möglichkeiten ergaben, größere und komplexere Web-Anwendungen zu erstellen, kam in Meetings oft die Frage auf, mit welchem Paradigma die Anzeigelogik umzusetzen sei. Die Begriffe „Multi Tier", „Thin Client" und „Fat Client" dominierten zu dieser Zeit die Diskussionen der Entscheidungsgremien. Dem Auftraggeber die Vorteile verschiedener Schichten einer verteilten Web-Applikation zu erläutern, war die eine Sache. Einem technisch versierten Laien aber die Entscheidung zu überlassen, wie er auf seine neue Applikation zugreifen möchte – per Browser („Thin Client") oder über eine eigene GUI („Fat Client") –, ist schlicht töricht. So galt es in vielen Fällen, während der Entwicklung auftretende Missverständnisse auszuräumen. Die schmalgewichtige Browser-Lösung entpuppte sich nicht selten als schwer zu beherrschende Technologie, da Hersteller sich selten um Standards kümmerten. Dafür bestand üblicherweise eine der Hauptanforderungen darin, die Applikation in den gängigsten Browsern nahezu identisch aussehen zu lassen. Das ließ sich allerdings nur mit erheblichem Mehraufwand umsetzen. Ähnliches konnte beim ersten Hype der Service-orientierten Architekturen beobachtet werden.

Die Konsequenz aus diesen Beobachtungen zeigt, dass es unverzichtbar ist, vor dem Projektstart eine Vision zu erarbeiten, deren Ziele auch mit dem veranschlagten Budget übereinstimmen. Eine wiederverwendbare Deluxe-Variante mit möglichst vielen Freiheitsgraden erfordert eine andere Herangehensweise als eine „We get what we need"-Lösung. Es gilt, sich weniger in Details zu verlieren, als das große Ganze im Blick zu halten.

Besonders im deutschsprachigen Raum fällt es Unternehmen schwer, die notwendigen Akteure für eine erfolgreiche Projektumsetzung zu finden. Die Ursachen dafür mögen recht vielfältig sein und könnten unter anderem darin begründet sein, dass Unternehmen noch nicht verstanden haben, dass Experten sich selten mit schlecht informierten und unzureichend vorbereiteten Recruitment-Dienstleistern unterhalten möchten.

Getting things done!

Erfolgreiches Projektmanagement ist kein willkürlicher Zufall. Schon lange wurde ein unzureichender Informationsfluss durch mangelnde Kommunikation als eine der negativen Ursachen identifiziert. Vielen Projekten wohnt ein eigener Charakter inne, der auch durch das Team geprägt ist, das die Herausforderung annimmt, um gemeinsam die gestellte Aufgabe zu bewältigen. Agile Methoden wie Scrum *[3]*, Prince2 *[4]* oder Kanban *[5]* greifen diese Erkenntnis auf und bieten potenzielle Lösungen, um IT-Projekte erfolgreich durchführen zu können.

Gelegentlich ist jedoch zu beobachten, wie Projektleiter unter dem Vorwand der neu eingeführten agilen Methoden die Planungsaufgaben an die zuständigen Entwickler zur Selbstverwaltung übertragen. Der Autor hat des Öfteren erlebt, wie Architekten sich eher bei Implementierungsarbeiten im Tagesgeschäft gesehen haben, anstatt die abgelieferten Fragmente auf die Einhaltung von Standards zu überprüfen. So lässt sich langfristig keine Qualität etablieren, da die Ergebnisse lediglich Lösungen darstellen, die eine Funktionalität sicherstellen und wegen des Zeit- und Kostendrucks nicht die notwendigen Strukturen etablieren, um die zukünftige Wartbarkeit zu gewährleisten. Agil ist kein Synonym für Anarchie. Dieses Setup wird gern mit einem überfrachteten Werkzeugkasten voller Tools aus dem DevOps-Ressort dekoriert und schon ist das Projekt scheinbar unsinkbar. Wie die Titanic!

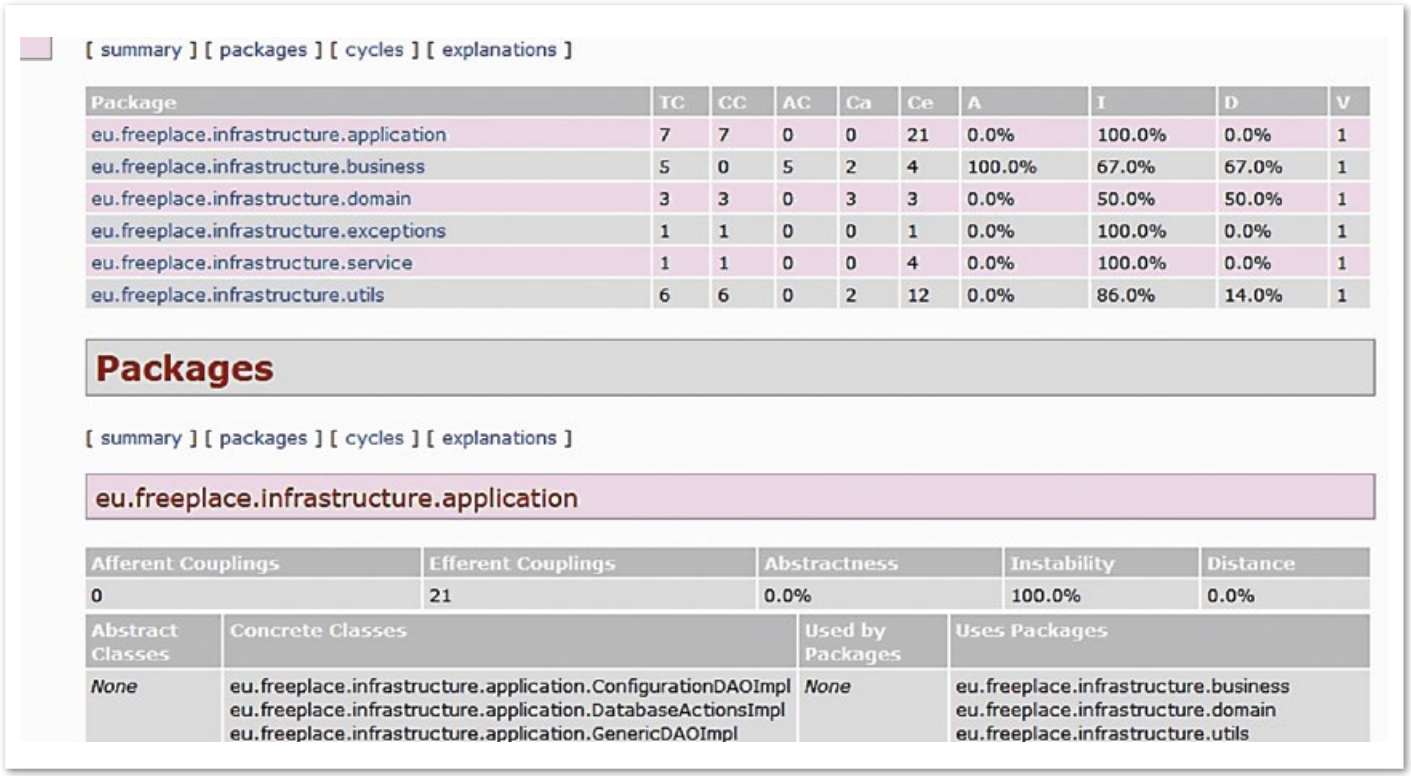
Nicht ohne Grund empfiehlt man seit Jahren, beim Projektstart allerhöchstens drei neue Technologien einzuführen. In diesem Zusammenhang ist es auch nicht ratsam, immer gleich auf die neuesten Trends zu setzen. Bei der Entscheidung für eine Technologie müssen im Unternehmen zuerst die entsprechenden Ressourcen aufgebaut sein, wofür hinreichend Zeit einzuplanen ist. Die Investitionen sind nur dann nutzbringend, wenn die getroffene Wahl mehr als nur ein kurzer Hype ist. Ein guter Indikator für Beständigkeit sind eine umfangreiche Dokumentation und eine aktive Community. Diese offenen Geheimnisse werden bereits seit Jahren in der einschlägigen Literatur diskutiert.

Wie geht man allerdings vor, wenn ein Projekt bereits seit vielen Jahren etabliert ist, aber im Sinne des Produkt-Lebenszyklus ein Schwenk auf neue Techniken unvermeidbar wird? Die Gründe für eine solche Anstrengung mögen vielseitig sein und variieren von Unternehmen zu Unternehmen. Die Notwendigkeit, wichtige Neuerungen nicht zu verpassen, um im Wettbewerb weiter bestehen zu können, sollte man nicht zu lange hinauszögern. Aus dieser Überlegung ergibt sich eine recht einfach umzusetzende Strategie. Aktuelle Versionen werden in bewährter Tradition fortgesetzt und erst für das nächste beziehungsweise übernächste Major-Release wird eine Roadmap erarbeitet, die alle notwendigen Punkte enthält, um einen erfolgreichen Wechsel durchzuführen. Dazu erarbeitet man die kritischen Punkte und prüft in kleinen Machbarkeitsstudien, die etwas anspruchsvoller als ein „Hallo Welt"-Tutorial sind, wie eine Umsetzung gelingen könnte. Aus Erfahrung sind es die kleinen Details, die das Krümelchen auf der Waagschale sein können, um über Erfolg oder Misserfolg zu entscheiden.

Bei allen Bemühungen wird ein hoher Grad an Automatisierung angestrebt. Gegenüber stetig wiederkehrenden, manuell auszuführenden Aufgaben bietet Automatisierung die Möglichkeit, kontinuierlich wiederholbare Ergebnisse zu produzieren. Dabei liegt es allerdings in der Natur der Sache, dass einfache Tätigkeiten leichter zu automatisieren sind als komplexe Vorgänge. Hier gilt es, zuvor die Wirtschaftlichkeit der Vorhaben zu prüfen, sodass Entwickler nicht gänzlich ihrem natürlichen Spieltrieb frönen und auch unliebsame Tätigkeiten des Tagesgeschäfts abarbeiten.

Wer schreibt, der bleibt

Dokumentation, das leidige Thema, erstreckt sich über alle Phasen des Soft-



[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
eu.freeplace.infrastructure.application	7	7	0	0	21	0.0%	100.0%	0.0%	1
eu.freeplace.infrastructure.business	5	0	5	2	4	100.0%	67.0%	67.0%	1
eu.freeplace.infrastructure.domain	3	3	0	3	3	0.0%	50.0%	50.0%	1
eu.freeplace.infrastructure.exceptions	1	1	0	0	1	0.0%	100.0%	0.0%	1
eu.freeplace.infrastructure.service	1	1	0	0	4	0.0%	100.0%	0.0%	1
eu.freeplace.infrastructure.utils	6	6	0	2	12	0.0%	86.0%	14.0%	1

Packages

[summary] [packages] [cycles] [explanations]

eu.freeplace.infrastructure.application

Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
0	21	0.0%	100.0%	0.0%

Abstract Classes	Concrete Classes	Used by Packages	Uses Packages
None	eu.freeplace.infrastructure.application.ConfigurationDAOImpl eu.freeplace.infrastructure.application.DatabaseActionsImpl eu.freeplace.infrastructure.application.GenericDAOImpl	*None*	eu.freeplace.infrastructure.business eu.freeplace.infrastructure.domain eu.freeplace.infrastructure.utils

Abbildung 2: Test Coverage mit Cobertura

ware-Entwicklungsprozesses. Ob für API-Beschreibungen, das Benutzer-Handbuch, Planungsdokumente zur Architektur oder erlerntes Wissen über optimales Vorgehen – das Beschreiben zählt nicht zu den favorisierten Aufgaben aller beteiligten Protagonisten. Dabei lässt sich oft beobachten, dass anscheinend die landläufige Meinung vorherrscht, dicke Handbücher ständen für eine umfangreiche Funktionalität des Produkts. Lange Texte in einer Dokumentation sind jedoch eher ein Qualitätsmangel, der die Geduld des Lesers strapaziert, weil dieser eine präzise auf den Punkt kommende Anleitung erwartet. Stattdessen erhält er schwammige Floskeln mit trivialen Beispielen, die selten problemlösend sind.

Diese Erkenntnis lässt sich auch auf die Projekt-Dokumentation übertragen und wurde unter anderem von Johannes Sidersleben *[6]* unter der Metapher über viktorianische Novellen ausführlich dargelegt. Hochschulen haben diese Erkenntnisse bereits aufgegriffen. So hat beispielsweise die Hochschule Merseburg den Studiengang „Technische Redaktion" *[7]* etabliert. Es bleibt zu hoffen, zukünftig mehr Absolventen dieses Studiengangs in der Projekt-Landschaft anzutreffen.

Bei der Auswahl kollaborativer Werkzeuge als Wissensspeicher ist immer das große Ganze im Blick zu halten. Erfolgreiches Wissensmanagement lässt sich daran messen, wie effizient ein Mitarbeiter die gesuchte Information findet. Die unternehmensweite Verwendung ist aus diesem Grund eine Managemententscheidung und für alle Abteilungen verpflichtend.

Informationen haben ein unterschiedliches Naturell und variieren sowohl in ihrem Umfang als auch bei der Dauer ihrer Aktualität. Daraus ergeben sich verschiedene Darstellungsformen wie Wiki, Blog, Ticketsystem, Tweets, Foren oder Podcasts, um nur einige aufzuzählen. Foren bilden sehr optimal die Frage- und Antwort-Problematik ab. Ein Wiki eignet sich hervorragend für Fließtext, wie er in Dokumentationen und Beschreibungen vorkommt. Viele Webcasts werden als Video angeboten, ohne dass die visuelle Darstellung einen Mehrwert bringt. Meist genügt eine gut verständliche und ordentlich produzierte Audiospur, um Wissen zu verteilen. Mit einer gemeinsamen und normierten Datenbasis lassen sich abgewickelte Projekte effizient miteinander vergleichen. Die daraus resultierenden Erkenntnisse bieten einen hohen Mehrwert bei der Erstellung von Prognosen für zukünftige Vorhaben.

Test & Metriken – das Maß aller Dinge

Bereits beim Überfliegen des Quality Reports 2014 erfährt man schnell, dass der neue Trend „Software testen" ist. Unternehmen stellen vermehrt Kontingente dafür bereit, die ein ähnliches Volumen einnehmen wie die Aufwendungen für die Umsetzung des Projekts. Genau genommen löscht man an dieser Stelle Feuer mit Benzin. Bei tieferer Betrachtung wird bereits bei der Planung der Etat verdoppelt. Es liegt nicht selten im Geschick des Projektleiters, eine geeignete Deklarierung für zweckgebundene Projektmittel zu finden.

Nur deine konsequente Überprüfung der Testfall-Abdeckung durch geeignete Analyse-Werkzeuge stellt sicher, dass am Ende hinreichend getestet wurde. Auch wenn man es kaum glauben mag: In einer Zeit, in der Software-Tests so einfach wie noch nie erstellt werden können und verschiedene Paradigmen kombinierbar sind, ist eine umfangreiche und sinnvolle Testabdeckung eher die Ausnahme *(siehe Abbildung 2)*.

Es ist hinreichend bekannt, dass sich die Fehlerfreiheit einer Software nicht beweisen lässt. Anhand der Tests weist man einzig ein definiertes Verhalten für die erstellten Szenarien nach. Automatisierte Testfälle ersetzen in keinem Fall ein

manuelles Code-Review durch erfahrene Architekten. Ein einfaches Beispiel dafür sind in Java hin und wieder vorkommende verschachtelte „try catch"-Blöcke, die eine direkte Auswirkung auf den Programmfluss haben. Mitunter kann eine Verschachtelung durchaus gewollt und sinnvoll sein. In diesem Fall beschränkt sich die Fehlerbehandlung allerdings nicht einzig auf die Ausgabe des Stack-Trace in ein Logfile. Die Ursache dieses Programmierfehlers liegt in der Unerfahrenheit des Entwicklers und dem an dieser Stelle schlechten Ratschlag der IDE, für eine erwartete Fehlerbehandlung die Anweisung mit einem eigenen „try catch"-Block zu umschließen, anstatt die vorhandene Routine durch ein zusätzliches „catch"-Statement zu ergänzen. Diesen offensichtlichen Fehler durch Testfälle erkennen zu wollen, ist aus wirtschaftlicher Betrachtung ein infantiler Ansatz.

Typische Fehlermuster lassen sich durch statische Prüfverfahren kostengünstig und effizient aufdecken. Publikationen, die sich besonders mit Codequalität und Effizienz der Programmiersprache Java beschäftigen *[8, 9, 10]*, sind immer ein guter Ansatzpunkt, um eigene Standards zu erarbeiten.

Sehr aufschlussreich ist auch die Betrachtung von Fehlertypen. Beim Issue-Tracking und bei den Commit-Messages in SCM-Systemen der Open-Source-Projekte wie Liferay *[11]* oder GeoServer *[12]* stellt man fest, dass ein größerer Teil der Fehler das Grafische User Interface (GUI) betreffen. Dabei handelt es sich häufig um Korrekturen von Anzeigetexten in Schaltflächen und Ähnlichem. Die Meldung vornehmlicher Darstellungsfehler kann auch in der Wahrnehmung der Nutzer liegen. Für diese ist das Verhalten einer Anwendung meist eine Black Box, sodass sie entsprechend mit der Software umgehen. Es ist durchaus nicht verkehrt, bei hohen Nutzerzahlen davon auszugehen, dass die Anwendung wenig Fehler aufweist.

Das übliche Zahlenwerk der Informatik sind Software-Metriken, die dem Management ein Gefühl über die physische Größe eines Projekts geben können. Richtig angewendet, liefert eine solche Übersicht hilfreiche Argumente für Management-Entscheidungen. So lässt sich beispielsweise über die zyklische Komplexität nach McCabe *[13]* die Anzahl der benötigten Testfälle ableiten. Auch eine Statistik über die Lines of Code und die üblichen Zählungen der Packages, Klassen und Methoden zeigt das Wachstum eines Projekts und kann wertvolle Informationen liefern.

Eine sehr aufschlussreiche Verarbeitung dieser Informationen ist das Projekt CodeCity *[14]*, das eine solche Verteilung als Stadtplan visualisiert. Es ist eindrucksvoll zu erkennen, an welchen Stellen gefährliche Monolithe entstehen können und wo verwaiste Klassen beziehungsweise Packages auftreten.

Fazit

Im Tagesgeschäft begnügt man sich damit, hektische Betriebsamkeit zu verbreiten und eine gestresste Miene aufzusetzen. Durch das Produzieren unzähliger Meter Papier wird anschließend die persönliche Produktivität belegt. Die auf diese Art und Weise verbrauchte Energie ließe sich durch konsequent überlegtes Vorgehen erheblich sinnvoller einsetzen.

Frei nach Kants „Sapere Aude" sollten einfache Lösungen gefördert und gefordert werden. Mitarbeiter, die komplizierte Strukturen benötigen, um die eigene Genialität im Team zu unterstreichen, sind möglicherweise keine tragenden Pfeiler, auf denen sich gemeinsame Erfolge aufbauen lassen. Eine Zusammenarbeit mit unbelehrbaren Zeitgenossen ist schnell überdacht und gegebenenfalls korrigiert.

Viele Wege führen nach Rom – und Rom ist auch nicht an einem Tag erbaut worden. Es lässt sich aber nicht von der Hand weisen, dass irgendwann der Zeitpunkt gekommen ist, den ersten Spatenstich zu setzen. Auch die Auswahl der Wege ist kein unentscheidbares Problem. Es gibt siche-

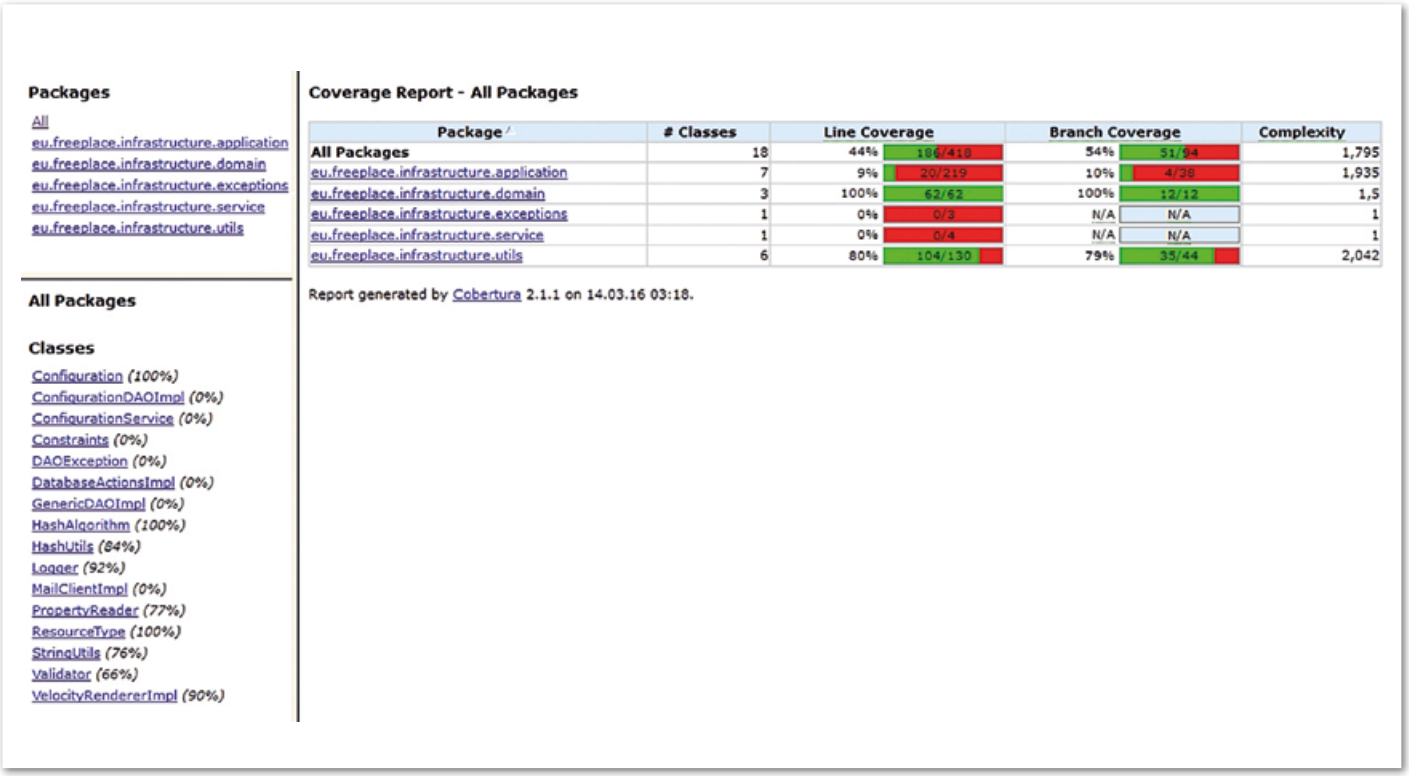


Abbildung 3: Maven JDepend Plugin – Zahlen mit wenig Aussagekraft

re Wege und gefährliche Pfade, auf denen auch erfahrene Wanderer ihre liebe Not haben, sicher das Ziel zu erreichen.

Für ein erfolgreiches Projektmanagement ist es unumgänglich, den Tross auf festem und stabilem Grund zu führen. Das schließt unkonventionelle Lösungen nicht grundsätzlich aus, sofern diese angebracht sind. Die Aussage in Entscheidungsgremien: „Was Sie da vortragen, hat alles seine Richtigkeit, aber es gibt in unserem Unternehmen Prozesse, auf die sich Ihre Darstellung nicht anwenden lässt", entkräftet man am besten mit dem Argument: „Das ist durchaus korrekt, deswegen ist es nun unsere Aufgabe, Möglichkeiten zu erarbeiten, wie wir die Unternehmensprozesse entsprechend bekannten Erfolgsstories adaptieren, anstatt unsere Zeit darauf zu verwenden, Gründe aufzuführen, damit alles beim Alten bleibt. Sie stimmen mir sicherlich zu, dass der Zweck unseres Treffens darin besteht, Probleme zu lösen, und nicht, sie zu ignorieren."

Referenzen

[1] Software Quality Report 2015 von CupGemini, URL: *https://www.de.capgemini.com/thought-leadership/world-quality-report-2014-15*

[2] Tom De Marco, Peopleware, 3rd. Ed., Pearson Education, 2013, ISBN 9780133440713

[3] Scrum: *https://www.scrum.org*

[4] Prince2: *www.prince2-deutschland.de*

[5] Kanban: *https://de.wikipedia.org/wiki/Kanban*

[6] Johannes Siderleben, Softwaretechnik – Praxiswissen für Softwareingenieure, Hanser, 2003, ISBN 3-446-21843-2

[7] HS Merseburg Beschreibung Studiengang Technische Redaktion, *https://www.hs-merseburg.de*

[8] Robert C. Martin, Clean Code - A Handbook of Agile. Software Craftsmanship, Pearson Education, 2009, ISBN 0-13-235088-2

[9] Joshua Bloch, Effective Java 2nd Ed, Addison Wesley, 2008, ISBN-13: 978-0-321-35668-0

[10] D. Boswell & T. Foucher, The Art of Readable Code, O'Reilly, 2012, ISBN: 978-0-596-80229-5

[11] Lifray Ticketsystem, *http://issues.liferay.com/browse/LPS/*

[12] GeoServer Projekt Management, *http://geoserver.org*

[13] T. J. McCabe, 1976, A Complexity Measure, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-2, NO.4

[14] CodeCity Webseite, *http://wettel.github.io/codecity.html*

Marco Schulz

marco.schulz@outlook.com

Marco Schulz studierte an der Hochschule Merseburg Diplom-Informatik. Sein persönlicher Schwerpunkt liegt in der Automatisierung von Build-Prozessen und dem Software-Konfigurationsmanagement. Seit mehr als zehn Jahren entwickelt er auf unterschiedlichen Plattformen Web-Applikationen. Derzeit arbeitet er als freier Consultant und ist Autor verschiedener Fachartikel.

Java Management Extensions

Philipp Buchholz, esentri AG

Innerhalb eines Enterprise-Software-Projekts gibt es immer wieder die Anforderung, eine Anwendung im laufenden Betrieb zu verwalten. Diese Aufgabe umfasst zumindest das Verändern von Konfigurationsparametern aufgrund geänderter Betriebssituationen zur Laufzeit, das proaktive Aussenden von Mitteilungen bei kritischen Situationen an registrierte Konsumenten und das periodische Aussenden von Reportingdaten für die Überwachung und das Health-Management.

Unter die Verwaltung einer Anwendung im laufenden Betrieb fallen diverse Szenarien. Ein einfaches Beispiel könnte das Ändern des Log-Levels durch einen Administrator sein, mit dem Ziel, das Verhalten zur Laufzeit besser zu analysieren. Mit einer entsprechenden Implementierung könnte man auch alle Konfigurationsparameter einer Anwendung zur Laufzeit anpassbar machen.

Innerhalb der Java-SE-Plattform beschreibt der Java-Management-Extensions-Standard (JMX) schon seit geraumer Zeit eine standardisierte Möglichkeit, um Anwendungen zur Laufzeit zu verwalten. JMX kann über sogenannte „Notifikationen" beim Eintreten kritischer Systemzustände Ereignisse aussenden. Diese sind auch für das periodische Reporting von Health-Informationen einer Anwendung einsetzbar. Sie werden von registrierten Listenern abgehört und weiterverarbeitet.

Ursprünglich wurde JMX im JSR 3 *[1]* des JCP *[2]* beschrieben. Über die Jahre sind mehrere Erweiterungen entstanden. So hat man etwa innerhalb des JSR 160 *[3]* „JMX Remote API" Möglichkeiten spezifiziert, wie JMX von entfernten Management-Clients verwendet werden kann. Der komplette JMX-Standard, der alle notwendigen Java Specification Requests beinhaltet, ist unter einer gemeinsamen JMX-Spezifikation zusammengefasst und veröffentlicht. Die aktuelle Version ist 1.4 *[4]*. Die Implementierung erfolgt mit Java SE 1.8.

Dieser Artikel zeigt, wie man einer Java-Anwendung auf Basis von JMX Management-Fähigkeiten hinzufügt. Dazu sind zunächst die Grundlagen von JMX erläutert und anschließend ein Anwendungsbeispiel skizziert und implementiert.

Grundlagen

Um effektiv mit JMX arbeiten zu können, wird als Erstes das grundlegende Schichtenmodell von JMX beschrieben. Wie *Abbildung 1*

zeigt, ist JMX in drei Schichten [5] eingeteilt. Die oberste Schicht deutet die Anwendung an, der Management-Fähigkeiten hinzugefügt werden sollen.

Der Instrumentation-Layer stellt die Schicht dar, die direkt an die zu verwaltenden Ressourcen angrenzt, die sogenannten „managed Resources". Darin liegen die Managed Beans (MBeans). Sie instrumentieren die zu verwaltenden Ressourcen und veröffentlichen zu diesem Zweck eine geeignete Management-Schnittstelle. „Instrumentieren" bedeutet in diesem Zusammenhang, dass ein bestehendes Java-Objekt die Fähigkeiten und Eigenschaften erhält, die für eine Verwaltung von außen notwendig sind.

Der Agent-Layer beinhaltet als wichtigste Komponente den MBeanServer. Dieser stellt eine Registry dar, in der alle MBeans registriert sind, die für die Verwendung von außen zur Verfügung stehen sollen.

Der Distributed-Layer dient als Zwischenschicht zwischen Anwendungen, die Management-Fähigkeiten nutzen wollen (Management-Clients), und dem Agent-Layer, der diese nach außen zur Verfügung stellt. Die Funktionalität des Distributed-Layers ist durch den JSR 160 „JMX Remote API" definiert und wird deshalb in der Literatur teilweise als Remote-Layer bezeichnet. Innerhalb der hier beschriebenen Spezifikation 1.4 von JMX wird aber weiterhin von Distributed-Layer gesprochen.

Innerhalb dieser Schicht gibt es zwei Arten von Komponenten – Konnektoren und Adaptoren. Konnektoren folgen dabei dem Client-Server-Prinzip und sind in eine client- und eine serverseitige Komponente aufgeteilt. Adaptoren sind einteilig und nur auf der Serverseite vorhanden. Sie können die benötigten Management-Informationen in bereits aufbereiteter Form, etwa als HTML, an einen Client, zum Beispiel einen Browser, zurückzuliefern. Im Diagramm wird bereits die Beziehung zwischen diesen Komponenten und dem MBeanServer angedeutet. Diese sind mit dem MBeanServer für den Zugriff auf Verwaltungs- und Steuerungs-Informationen verbunden. In diesem Artikel werden nur Konnektoren betrachtet, da diese in der Praxis am relevantesten sind. Adaptoren sind nur zur Vollständigkeit erwähnt und finden im weiteren Verlauf keine Beachtung mehr. Die Referenz-Implementierung, die innerhalb des JDK 8 enthalten ist, liefert weiterhin keinen Adaptor mit.

Beim JDK 8 ist standardmäßig der „javax.management.remote.rmi.RMIConnectorServer" [6] mitgeliefert. Dieser stellt die Funktionalität eines Konnektors unter Verwendung von „RemoteMethodInvocation" (RMI) [7] bereit. Er ist in der Abbildung als Schnittstelle nach außen angedeutet.

Managed Beans

Die sogenannten „Managed Beans" (MBeans) stellen die eigentliche Schnittstelle zur Verwaltung von Ressourcen bereit. Es gibt unterschiedliche Typen, die jeweils andere Konventionen erfüllen und für unterschiedliche Einsatzzwecke geeignet sind. Standard MBeans sind konkrete Java-Klassen, die ihre Verwaltungs-Schnittstelle durch das Implementieren eines Interface explizit definieren. Die angebotene Schnittstelle zur Verwaltung ist somit statisch. Das bedeutet, dass alle Operationen, die für die Verwaltung zur Verfügung stehen, direkt durch das Management-Interface definiert werden.

Beim Erstellen der Java-Klassen und des Management-Interface müssen Namenskonventionen eingehalten werden [8]. Das Management-Interface muss mit dem Namen der konkreten Klasse und dem Suffix „MBean" benannt werden. Nur Methoden, die innerhalb des Management-Interface definiert sind, werden für Verwaltungsaufrufe nach außen zur Verfügung gestellt. Attribute einer MBean werden über entsprechende Getter- und Setter-Definitionen innerhalb des MBean-Interface veröffentlicht.

Bei der Definition und Implementierung des Management-Interface und dessen Operationen sind die Vorgaben der JavaBeans-Specification einzuhalten. Details zu diesen Konventionen stehen entweder in der JavaBeans-Specification [9] oder in der JMX-Specification [10]. Durch das statische Management-Interface sind Standard MBeans ideal für Situationen, in denen die Verwaltungsschnittstelle bekannt ist und sich nicht oft ändert. Sollte diese einer erhöhten Änderungshäufigkeit unterliegen oder nicht genau bekannt sein, sind Dynamic MBeans flexibler und besser geeignet.

Dynamic MBeans

Dynamic MBeans definieren kein statisches Management-Interface, sondern implementieren „javax.management.DynamicMBean" [11]. Über die Methode „getMBeanInfo() : MbeanInfo" [12] kann ein Management-Client das dynamische Management-Interface einer Dynamic MBean abfragen. Die Methoden „getAttribute(attribute Name : String) : Object" beziehungsweise „getAttributes(attributeNames : String[]) : Object []" zeigen den Wert eines Attributs beziehungsweise die Werte mehrerer Attribute an. Dabei werden die Attribute mit den übergebenen Namen identifiziert. Entsprechende Setter-Methoden werden für das

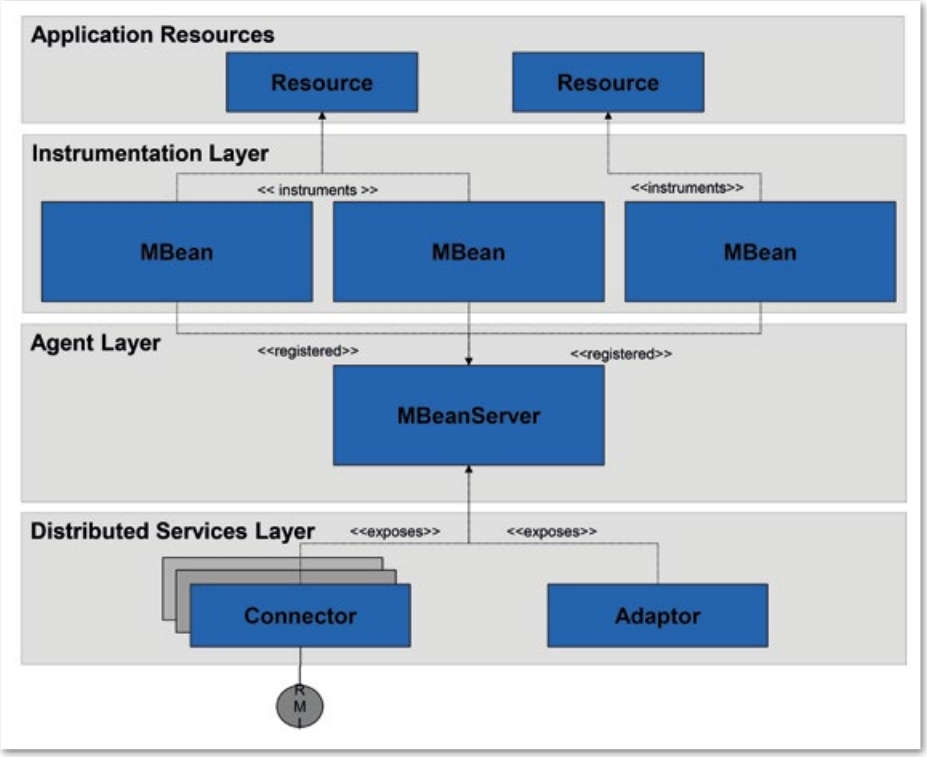


Abbildung 1: Das JMX-Schichtenmodell

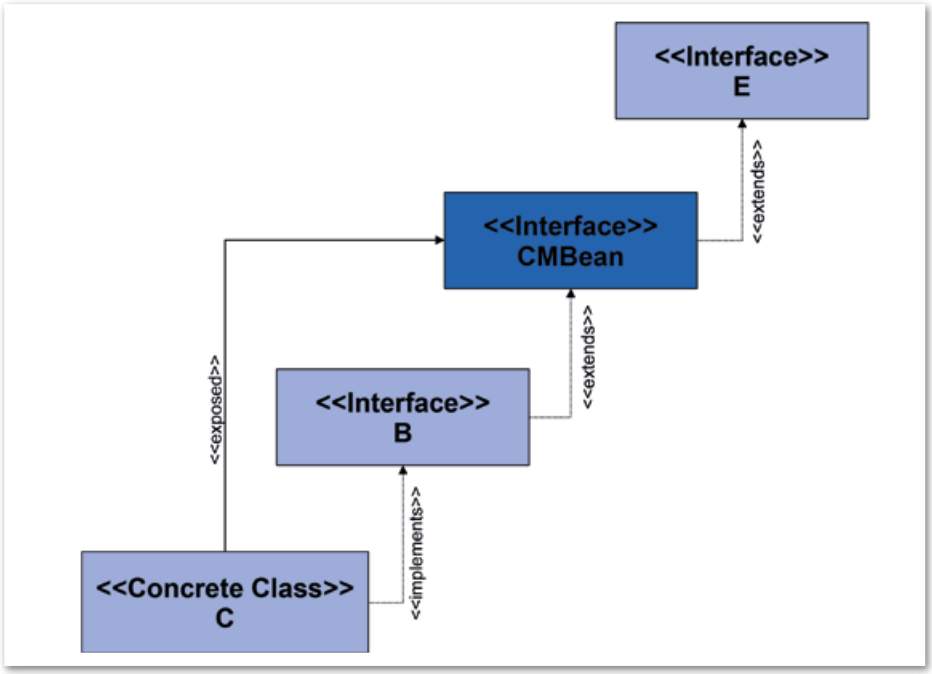


Abbildung 2: Ermittlung Management-Interface bei mehrstufiger Hierarchie

Setzen von Attribut-Werten angeboten. Die Methode „invoke(actionName : String, params : Object[], signature : String []) : Object" ermöglicht das Aufrufen einer Operation innerhalb einer Dynamic MBean.

Bei der Implementierung einer Dynamic MBean ist darauf zu achten, dass die beschriebenen Methoden exakt das Management-Interface veröffentlichen, das durch die verwaltete Ressource unterstützt ist. Bei diesem MBean-Typ hat man nicht die Sicherheit, die eine Standard MBean mit einer statischen Management-Schnittstelle bieten kann. Bei statischen MBeans wird bereits zur Entwicklungszeit gegen ein fest definiertes Interface entwickelt.

Der große Vorteil von Dynamic MBeans ist, dass sich die veröffentlichte Schnittstelle zur Laufzeit an die Umgebung anpassen und damit ändern kann *[13]*. Beim Eintreten bestimmter Ereignisse können beispielsweise weitere Operationen veröffentlicht oder bestimmte Operationen von der Veröffentlichung ausgenommen werden. So lassen sich Management-Schnittstellen mit großer Flexibilität entwickeln. Notifikationen *[14]* können in diesem Zusammenhang genutzt werden, um die dynamische Änderung eines Management-Interface bekanntzugeben.

Vererbungsregeln des statischen Management-Interface

Das Management-Interface einer MBean muss nicht direkt durch die oben beschriebene Vorgehensweise definiert und implementiert sein. In der Praxis kann es vorkommen, dass dieses innerhalb einer mehrstufigen Klassen- und Interface-Hierarchie definiert ist und innerhalb einer Superklasse implementiert wird.

Zur Laufzeit verwendet die JMX-Runtime Reflection *[15]*, um das Management-Interface zu ermitteln und zu veröffentlichen *(siehe Abbildung 2)*.

Innerhalb der gezeigten Vererbungshierarchie *[16]* liegt das Management-Interface CMBean auf der zweiten Ebene. Bei der Untersuchung der konkreten Klasse „C" evaluiert die JMX-Runtime die Vererbungshierarchie bis zum naheliegendsten Management-Interface, das die Namenskonventionen des gesuchten Interface erfüllt. Innerhalb dieses Beispiels wird das Interface „CMBean" als Verwaltungsschnittstelle veröffentlicht. Da „CMBean" vom Interface „E" erbt, werden auch alle Methoden aus der Schnittstelle „E" Teil des Management-Interface. Erbt eine Klasse von einer Klasse, die bereits ein Management-Interface veröffentlicht, dann übernimmt diese Klasse das Management-Interface, ohne es zu erweitern.

Beim Einsatz einer dynamischen MBean wird immer die nächste Implementierung von „javax.management.DynamicMBean" als Verwaltungsschnittstelle veröffentlicht. Es findet keine Zusammenstellung der Verwaltungsoperationen aus der Vererbungshierarchie statt. Im Beispiel sind auch die Operationen des Interface „E" in die Schnittstelle miteinbezogen.

Notification-Mechanismus

Über den Notification-Mechanismus von JMX können MBeans Notifikationen aussenden, wenn sich der Status der verwalteten Ressourcen ändert. Die damit ausgesendeten Benachrichtigungen können für ein kontinuierliches Reporting – Stichwort „Health-Check" – oder für die Eskalation kritischer Systemzustände verwendet werden.

Die Klasse „javax.management.Notification" *[17]* bildet eine Notifikation ab. Sie kann entweder direkt verwendet oder es können Ableitungen und damit benutzerdefinierte Typen von Notifikationen erstellt werden. Die Implementierung benutzerdefinierter Notifikationen ist im Beispiel zu sehen.

Damit eine MBean solche Notifikationen aussenden kann, müssen das Interface „javax.management.NotificationBroadcaster" *[18]* oder „javax.management.NotificationEmitter" *[19]* implementiert sein. Sie bieten die Registrierung von Listenern, die diese Notifikationen empfangen können. Ein Listener muss das Interface „javax.management.NotificationListener" *[20]* implementieren. Implementierungen der Klasse „javax.management.NotificationFilter" *[21]* erlauben die Filterung von Notifikationen. Damit lässt sich definieren, auf welche Typen von Notifikationen gelauscht werden soll.

Handback Objects verwenden

Während der Registrierung und Deregistrierung von Listenern kommen sogenannte „Handback Objects" zum Einsatz. Diese werden von der MBean intern gespeichert und beim Aussenden von Notifikationen den Listenern wieder zur Verfügung gestellt. Sie müssen von der MBean transparent behandelt und dürfen nicht geändert werden. Auf diese Art lässt sich beispielsweise ein applikationsspezifischer Kontext beim Registrieren angeben. Dieser kann dann beim Empfangen der Notifikationen wiederverwendet werden.

Eine MBean kann unterschiedliche Typen von Notifikationen aussenden. Im Normalfall wird der Typ einer Notifikation als String-Attribut beim Instanziieren der Notification-Klasse definiert. Hier ist zu beachten, dass der Präfix „JMX." für Benachrichtigungen der JMX-Infrastruktur reserviert ist. Dieser Typ wird beispielsweise für das Senden von Benachrichtigungen beim Registrieren und Deregistrieren von MBeans verwendet. Ansonsten sind keine Konventionen erforderlich. Es empfiehlt sich jedoch, eine passende Namenskonvention einzuführen, um eine

spätere Verständlichkeit sicherzustellen. Ein Beispiel aus der Praxis für solch eine Namenskonvention findet sich im Abschnitt über das Anwendungsbeispiel.

Eine weitere Möglichkeit für die Definition von Notifikations-Typen ist das Ableiten von „javax.management.Notification". Dieser Ableitung lassen sich in der Folge Attribute hinzufügen, die das Ereignis näher beschreiben. Damit sind komplexere Anwendungsfälle realisierbar.

Notifikationen filtern und aussenden

Wie beschrieben, wird zum Filtern von Notifikationen das Interface „javax.management.NotificationFilter" implementiert. Der Filter-Methode wird die jeweilige Benachrichtigung übergeben. Anhand dieser kann entschieden werden, ob die Notifikation empfangen werden soll oder nicht. Zum Filtern finden meist die beiden beschriebenen Möglichkeiten zur Definition der Notifikations-Typen Anwendung. Diese Filtermöglichkeit kann sehr praktisch sein, wenn bestimmte Empfänger zum Beispiel nur auf als kritisch einzustufende Ereignisse reagieren und dann eine bestimmte Aktion auslösen sollen.

Innerhalb der kurzen Einführung zu Notifikationen wurde bereits auf die beiden Interfaces „javax.management.NotificationBroadcaster" und „javax.management.NotificationEmitter" hingewiesen *[22]*. Eines dieser Interfaces muss von einer MBean implementiert sein, wenn diese Benachrichtigungen aussenden möchte. Das Interface „javax.management.NotificationEmitter" wurde in der Version 1.2 von JMX hinzugefügt und erbt von „javax.management.NotificationBroadcaster". Es fügt eine weitere Methode zum Löschen von „NotificationListenern" hinzu.

Die Methode „removeNotificationListener(listener : NotificationListener, filter : NotificationFilter, handback : Object) : void" erlaubt das Entfernen eines bestimmten Listeners anhand der übergebenen Informationen. Neu entwickelte MBeans sollten immer das Interface „javax.management.NotificationEmitter" implementieren.

Anstatt die genannten Schnittstellen direkt zu implementieren, kann auch von der Klasse „javax.management.NotificationBroadcasterSupport" *[23]* geerbt werden. Diese Klasse enthält bereits grundlegende Logik, um Listener zu registrieren, zu entfernen und Notifikationen auszusenden. Sollte aufgrund der Klassenhierarchie ein Erben dieser Klasse nicht möglich sein, kann man an diese per Komposition delegieren. „AttributeChangeNotifications" sind spezielle, bereits vorhandene Notifikationen, die ausgesendet werden, wenn sich ein Attribut einer MBean ändert.

Das Aussenden von Benachrichtigungen, wenn sich ein Attribut einer MBean ändert, liegt in der Verantwortung einer MBean und ist entsprechend selbst zu implementieren. Hierbei wird auf die Klasse „javax.management.AttributeChangeNotification" *[24]* zurückgegriffen. Unter Nutzung der Klasse „javax.management.NotificationBroadcasterSupport" ist nur eine neue Instanz von „javax.management.AttributeChangeNotification" zu erstellen und zu versenden.

MBeans verwenden

Damit MBeans verwendet werden können, müssen sie innerhalb eines MBeanServer registriert sein. Dafür ist ein „javax.management.ObjectName" *[25]* erforderlich. Mit diesem lässt sich die MBean registrieren und für Management-Aufrufe nutzen.

Ein „ObjectName" hat folgenden Aufbau: „domain: key-property-list". Bei der Wahl der Domäne sollte immer der vollqualifizierte Domänenname revers vorangestellt sein. Diese Konvention ist analog der Konvention bei Package-Benennungen. Der Rest des „ObjectName" besteht aus Key-Value-Paaren, die durch Kommas getrennt sind. *Listing 1* zeigt einige Beispiele für valide „ObjectName". Ist ein passender „ObjectName" gewählt, kann damit eine MBean an einem MBeanServer registriert werden *(siehe Listing 2)*.

In diesem Beispiel wird, nachdem der „ObjectName" erstellt ist, der MBeanServer der aktuellen Plattform über die Klasse „java.lang.management.ManagementFactory" *[26]* ermittelt. Erfolgt dieser Aufruf innerhalb eines Java-EE-Containers, wird der MBeanServer dieses Containers zurückgeliefert. Die „ManagementFactory" bietet außerdem Zugriff auf die sogenannten „MXBeans". Dabei handelt es sich um Managed Beans, die Zugriff auf Eigenschaften und Operationen der aktuellen Java-Virtual-Machine erlauben. Die MemoryMXBean *[27]* bietet beispielsweise Zugriff auf das aktuelle Speicherverhalten. So kann die Speicherauslastung zur Laufzeit ausgewertet und bei einem erhöhten Speicherverbrauch reagiert werden. Es gibt noch zahlreiche weitere MXBeans. Details sind in der JavaDoc der Klasse „javax.management.ManagementFactory" zu finden.

Das Beispiel

Das nachfolgende Anwendungsbeispiel zeigt, wie mit den beschriebenen Möglichkeiten die Konfiguration einer Anwendung zur Laufzeit geändert und damit das Verhalten der Anwendung angepasst werden

```
de.example.Application: type=ManagementObject
de.example.Application: type=ManagementObject, name=ExampleName
```

Listing 1

```
/* Create ObjectName and register MBean in MBeanServer. */
ObjectName objectName = new
ObjectName("de.bu.javaee7.jmx.Application:type=ConfigurationServer,
subSystemName=ROOT");
MBeanServer platformMBeanServer = ManagementFactory.getPlatformMBeanSer-
ver();
     platformMBeanServer.registerMBean(new ManagementObject(), objectName);
```

Listing 2

```
public interface ConfigurationServerManagementMBean {
  <T> void writeConfigurationValue(String subSystemName, String configura-
tionElementName,
         T configurationElementValue);
  <T> T readConfigurationValue(String subSystemName, String configura-
tionElementName);

}
```

Listing 3

```
public class ConfigurationServerManagement extends NotificationBroadcaster-
Support
     implements ConfigurationServerManagementMBean {
  @Inject
  @ConfigurationServerType(configurationServerTypes = ConfigurationServer-
Types.ROOT)
  private ConfigurationServer rootConfigurationServer;
…
}
```

Listing 4

```
@Override
    public <T> void writeConfigurationValue(String subSystemName, String
configurationElementName,
      T configurationElementValue) {

  ...
  ...

  // Send notification about changes in ConfigurationServer.
  String configurationChangedNotificationType = this.buildNotificationType(
SubSystemAccessType.READ, subSystemName, configurationElementName);
  Notification configurationChangedNotification = new Notification(configur
ationChangedNotificationType, this, 0L);
  this.sendNotification(configurationChangedNotification);
}
```

Listing 5

```
...
JMXServiceURL jmxServiceURL =
    new JMXServiceURL("service:jmx:rmi:///jndi/rmi://:9999/jmxrmi");
JMXConnector jmxConnector = JMXConnectorFactory.connect(url, null);
…
```

Listing 6

kann. Dazu wird das Management-Interface „ConfigurationServerManagementMBean" angelegt *(siehe Listing 3)*.

Die Implementierung heißt „ConfigurationServerManagement", entsprechend den JMX-Namenskonventionen für Standard MBeans. *Listing 4* zeigt einen Ausschnitt mit den relevanten Codestellen aus der Implementierung.

Die Implementierung dient als Wrapper um die Klasse „ConfigurationServer". Diese ist im Beispiel die zentrale Komponente für den lesenden und schreibenden Zugriff auf Konfigurationseinstellungen. Sie sendet außerdem Notifikationen aus, wenn sich Konfigurationseinstellungen ändern. Das ist daran zu erkennen, dass die Klasse „NotificationBroadcasterSupport" abgeleitet wird. Der Code-Ausschnitt aus der „Configu-rationServerManagement"-Klasse zeigt in *Listing 5* das Aussenden von Notifikationen.

Wird ein Konfigurationswert geschrieben, sendet die MBean eine Notifikation aus, die von interessierten Listenern abgehört werden kann. Innerhalb der Methode „buildNotificationType(...)" wird der Typ der Notifikation zusammengestellt. Wie bereits oben beschrieben, ist hier eine durchgängige Konvention sehr wichtig, damit Management-Clients diese verlässlich auswerten und reagieren können.

In unserem Beispiel haben die Typen folgende Konvention:„ConfigurationServer.${NAME_SUBSYSTEM}.${OPERATION}.${CONFIGURATIONELEMENTNAME}". Wendet man diese Konvention an, ergibt sich beispielsweise „ConfigurationServer.DatabaseSubSystem.WRITE.TransactionTimeout".

Den Log-Level zur Laufzeit ändern

Für Administratoren einer Anwendung kann die Änderung des Log-Levels zur Laufzeit sehr nützlich sein, um beim Auftreten von Störungen eine detailliertere Analyse durchführen zu können. Zahlreiche Logging-Frameworks wie beispielsweise Log4j *[28]*, Log4j 2 *[29]* und Logback *[30]* bieten bereits MBean-Implementierungen an, die die Anpassung des Log-Levels für die unterschiedlichen Logger zur Laufzeit erlauben. Die Details zur Nutzung können in den jeweiligen Dokumentationen nachgelesen werden.

Verbindung zu einem JMX-Connector herstellen

Mit dem JDK 1.8 wird bereits ein JMX-Client ausgeliefert, die JConsole. Mit dieser kann die Verwaltungsschnittstelle von MBeans verwendet werden. Es lassen sich Getter und Setter für das Auslesen und Ändern von Eigenschaften und Operationen mit und ohne Parameter aufrufen. Für ein effizientes, automatisiertes und dem Anwendungsfall angepasstes Verwalten einer Anwendung ist oftmals ein JMX-Client zu entwickeln. Für das Herstellen einer Verbindung zu einem JMX-Connector kommen die Klassen aus dem Package „javax.management.remote" zum Einsatz.

Der Service-Access-Point

Die Klasse „javax.management.remote.JMXServiceUrl" *[31]* wird für die Angabe des Hosts, des Ports und des Protokolls verwendet. Hierbei folgt die Syntax dem Service-Schema, das in RFC2609 *[32]* beschrieben und in RFC3111 *[33]* ergänzt ist.

Der Aufbau einer URL für die Verbindung zu einem „JMXConnectorService" folgt grundsätzlich dem Aufbau „service:sap", wobei „sap" für Service-Access-Point steht und Angaben wie Host, Port und Protokoll aufnimmt. Details können dem JavaDoc der Klasse oder den RFCs selbst entnommen werden. Im Anwendungsbeispiel läuft der Server auf dem lokalen Rechner (localhost) auf Port 9999. Als Protokoll kommt RMI zum Einsatz *(siehe Listing 6)*. Bei einem Szenario aus der Praxis wären Client und Server physikalisch oder logisch *[34]* getrennt und befänden sich auf unterschiedlichen Rechnern.

Management-Interface von MBeans

Nachdem eine Verbindung mit dem entfernten „JMXConnectorServer" hergestellt ist, muss eine Verbindung zum MBeanServer erfolgen. Über diesen erfolgt der Zugriff auf die Verwaltungsschnittstelle von registrierten MBeans *(siehe Listing 7)*.

Über eine Instanz der „MbeanServer Connection"-Klasse *[35]* kann die Verwaltungsschnittstelle über die Methoden „invoke", „setAttribute" und „getAttribute" verwendet werden. Wie in der JavaDoc zu sehen ist, sind diese Methoden allerdings nicht wirklich komfortabel verwendbar. Der größte Nachteil ist, dass sie keine Typsicherheit zur Entwicklungszeit bieten. Die Parameter werden als Object-Array übergeben. Fehler fallen erst zur Laufzeit auf.

Komfortabler ist das dynamische Generieren von Proxy-Objekten über den „javax.management.JMX"-Singleton. Es generiert ein Proxy-Objekt für Management-Interfaces *(siehe Listing 8)*.

Im Beispiel wird für die Verwaltungsschnittstelle „ConfigurationServerMange-mentMBean" ein dynamischer Proxy generiert. Über diesen Ansatz erhält man Typsicherheit zur Entwicklungszeit, da gegen ein getyptes Stellvertreter-Objekt entwickelt wird.

Notifikationen abhören

Die Klasse MBeanServerConnection bietet über die Methode „addNotificationListener" einem Client die Möglichkeit, einen NotificationListener hinzuzufügen. Details zur Funktionsweise von Notifikationen sind oben beschrieben. In diesem Beispiel sendet das ConfigurationServer-Management-Interface einen regelmäßigen Heart-Beat. Dieser wird vom Client über eine Notifikation empfangen *(siehe Listing 9)*.

Fazit

Der Artikel zeigt, dass JMX weiterhin mächtige Möglichkeiten für die Verwaltung von Java-SE- und Java-EE-Anwendungen bietet. Gängige Applikationsserver implementieren den JMX-Standard vollständig und sind damit verwaltbar. Sehr oft lassen sich innerhalb eines Java-EE-Applikationsservers „NotificationListener" verwenden, um über Verwaltungsereignisse informiert zu sein. Verwaltungsereignisse können hier das Deployment und Undeployment von Java-EE-Anwendungen und mehr sein.

Aber auch Java-SE-Anwendungen lassen sich mit den beschriebenen Möglichkeiten leicht verwaltbar machen. Durch die Standardisierung über die JMX-Spezifikation arbeiten Applikationsserver und Anwendungen unterschiedlicher Hersteller ohne Probleme zusammen.

```
ObjectName rootConfigurationServerObjectName = new ObjectName(
     "de.bu.javaee7.jmx.Application: type=ConfigurationServer,
subSystemName=ROOT");
  MBeanServerConnection mbeanServerConnection = jmxc.getMBeanServerConnec-
tion();
```

Listing 7

```
ConfigurationServerManagementMBean configurationServerManagement = JMX.newM
BeanProxy(mbeanServerConnection,
     rootConfigurationServerObjectName, ConfigurationServerManagementMBean.
class);
  ConfigurationElement<String> configurationElement = configurationServer-
Management.readConfigurationValue("ROOT",
     „key1");
```

Listing 8

```
mbeanServerConnection.addNotificationListener(rootConfigurationServerObject
Name, new NotificationListener() {
      @Override
       public void handleNotification(Notification notification, Object
handback) {
     if (notification.getType().equals("configurationserver.heartbeat")) {
/* Handle heart beat notification here. */
     }
      }
  }, null, null);
```

Listing 9

Weitere Informationen

[1] *https://jcp.org/en/jsr/detail?id=3*
[2] Java Community Process, *https://www.jcp.org/en/home/index*
[3] *https://jcp.org/en/jsr/detail?id=160*
[4] *https://docs.oracle.com/javase/8/docs/technotes/guides/jmx/JMX_1_4_specification.pdf*
[5] siehe auch JMX 1.4 Specification, S. 25
[6] *https://docs.oracle.com/javase/8/docs/api/javax/management/remote/rmi/RMIConnectorServer.html*
[7] *https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/*
[8] siehe auch JMX 1.4 Specification, S. 42
[9] *http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/*
[10] siehe auch JMX 1.4 Specification, S. 44 bis 46, Abschnitt „Lexical Design Patterns"
[11] siehe auch JMX 1.4 Specification, S. 47
[12] *https://docs.oracle.com/javase/8/docs/api/javax/management/MBeanInfo.html*
[13] siehe auch JMX 1.4 Specification, S. 49, Abschnitt „Dynamics"
[14] siehe auch JMX 1.4 Specification, S. 54
[15] *http://docs.oracle.com/javase/7/docs/technotes/guides/reflection*
[16] siehe auch JMX1.4 Specification, S. 50, Abschnitt „Inheritance Patterns"
[17] *https://docs.oracle.com/javase/8/docs/api/javax/management/Notification.html*
[18] *https://docs.oracle.com/javase/8/docs/api/javax/management/NotificationBroadcaster.html*
[19] *https://docs.oracle.com/javase/8/docs/api/javax/management/NotificationEmitter.html*
[20] *https://docs.oracle.com/javase/8/docs/api/javax/management/NotificationListener.htm*
[21] *https://docs.oracle.com/javase/8/docs/api/javax/management/NotificationFilter.html*
[22] siehe auch JMX 1.4 Specification S. 56
[23] *https://docs.oracle.com/javase/8/docs/api/javax/management/NotificationBroadcasterSupport.html*
[24] *https://docs.oracle.com/javase/8/docs/api/javax/management/AttributeChangeNotification.html*
[25] *https://docs.oracle.com/javase/8/docs/api/javax/management/ObjectName.html*
[26] *https://docs.oracle.com/javase/8/docs/api/java/lang/management/ManagementFactory.html*
[27] *https://docs.oracle.com/javase/8/docs/api/java/lang/management/MemoryMXBean.html*
[28] *https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/jmx/LoggerDynamicMBean.html* und *https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/jmx/HierarchyDynamicMBean.html*
[29] *https://logging.apache.org/log4j/2.0/manual/jmx.html*
[30] *http://logback.qos.ch/manual/jmxConfig.html*
[31] *https://docs.oracle.com/javase/8/docs/api/javax/management/remote/JMXServiceURL.html*
[32] *http://www.ietf.org/rfc/rfc2609.txt*
[33] *http://www.ietf.org/rfc/rfc3111.txt*
[34] Eine logische Trennung wird beispielsweise durch den Einsatz von Virtualisierung erreicht
[35] *https://docs.oracle.com/javase/8/docs/api/javax/management/MBeanServerConnection.html*

Philipp Buchholz
philipp.buchholz@esentri.com

Philipp Buchholz ist Senior Consultant bei der esentri AG. Als Architekt konzipiert und implementiert er umfangreiche Enterprise-Software-Lösungen auf Basis von Java EE und Technologien aus dem Oracle-Middleware-Stack wie WebLogic und ADF. Bei der Modellierung komplexer Software setzt er auf die Techniken des Domain-driven-Design und der objektorientierten Analyse. Zusätzlich unterstützt er die Entwicklungsteams von Kunden als Scrum-Master.

Optional <Titel>

Dr. Frank Raiser, Konzept Informationssysteme GmbH

Eine der bedeutendsten Änderungen von Java 8 für den Entwickleralltag stellt die Klasse „java.util.Optional" dar. Damit lassen sich zahlreiche Fehlerquellen systematisch vermeiden, wobei gleichzeitig der Quellcode kürzer und verständlicher wird. Dieser Artikel zeigt die Historie der Klasse in anderen Sprachen, ihre Anwendungsmöglichkeiten sowie die Vor- und Nachteile auf. An einem durchgängigen Beispiel werden die verschiedenen Methoden von „Optional" und die daraus resultierende Umstellung auf eine Datenfluss-orientierte Entwicklung erläutert.

Blickt man über die Java-Welt hinaus, erkennt man, dass die Klasse „java.util.Optional" bei Weitem nichts Neues ist. Über lange Zeit war die bekannteste Variante die „Maybe"-Monade in Haskell. Die Programmierung mit Monaden geht auf Arbeiten von Moggi *[1]* Anfang der 1990er-Jahre zurück, sie ist jedoch viel weitreichender. Da man für ein Konzept wie „Optional" aber nicht auf deren Kenntnis angewiesen ist, sind ähnliche Umsetzungen bereits früher zu finden.

Im Laufe der Zeit haben immer mehr Sprachen die Idee eines Typs wie „Optional" aufgenommen. *Tabelle 1* zeigt, wie dieser Typ in verschiedenen Sprachen unter jeweils anderem Namen Verwendung findet. Man sieht auch, dass dies vor allem in statisch getypten Sprachen der Fall ist.

Nur wenige moderne Sprachen lassen einen entsprechenden Typ noch vermissen, darunter C# und JavaScript. Ein Blick auf die Quellen von „java.util.Optional" offenbart jedoch, dass man eine Implementierung mit wenig Aufwand selbst entwickeln kann. Daher gibt es in den entsprechenden Sprachen meist Ersatz in Form von Bibliotheken. Für Java 7 war die Verwendung bereits durch „com.google.guava.Optional" möglich. Die Implementierung in Java 8 ersetzt diese mittlerweile vollständig, da sie nicht nur Teil der JVM, sondern auch deutlich mächtiger ist.

Warum überhaupt „Optional"?

Wer sich mit „Optional" und dessen Einsatzmöglichkeiten noch nicht auseinandergesetzt hat, fragt sich vielleicht, warum wir diese Klasse als Java-Entwickler überhaupt benötigen. Die Frage ist durchaus berechtigt, konnten doch Java-Programme auch ohne diesen Typ schon seit zwei Jahrzehnten erfolgreich entwickelt werden.

Sprachen ohne Unterstützung eines „null"-Werts waren im Gegensatz zu Java dazu gezwungen. Beispielsweise gibt es sonst in Haskell keine Möglichkeit, eine Funktion zu implementieren, die aus einer Liste von Werten einen Wert mit einer bestimmten Eigenschaft heraussucht. Naturgemäß kann die Funktion so aufgerufen werden, dass der gefragte Wert in der Liste nicht vorkommt. Was soll dann aber das Ergebnis der Funktion sein? Sprachen wie Java haben zu diesem Zweck den speziellen Wert „null" eingeführt. Für Haskell wurde stattdessen mithilfe des „Maybe"-Typs ein Weg geschaffen, um auszudrücken, dass kein Ergebnis berechnet werden konnte.

Nun könnte man sich zurücklehnen mit der Erkenntnis, dass „null" das Problem bereits bestens löst. Wäre da nicht ein nagendes, schlechtes Gefühl, das bei jeder „NullPointerException" (NPE) stärker wird.

Die Idee, einen speziellen „null"-Wert in einer Sprache zu definieren, geht auf den mit dem Turing-Award ausgezeichneten Sir Charles Anthony Richard Hoare zurück, der einen solchen im Jahr 1965 erstmals für Algol W einführte. Knapp 40 Jahre später kommentierte er diese Idee als seinen „Milliarden Dollar Fehler" *[2]* und das nur, weil es „so einfach zu implementieren" war. Dass die Verwendung von „null" nicht der Weisheit letzter Schluss sein kann, weiß jeder erfahrene Entwickler. Was liegt also näher, als sich die Lösung anzusehen, die andere Sprachen, ohne „null"-Werte, verwenden?

Erste Verwendung von „Optional"

Um „Optional" als Ersatz für „null" zu verwenden, braucht es nicht viel. In diesem Artikel wird ein einfaches Programmstück bezüglich seiner Umstellung auf „Optional" untersucht. Es soll über die Kommandozeile beim Start einen Parameter „-f" und einen Dateinamen erhalten, um aus dieser Datei einen Konfigurationswert auszulesen. Die Methode in *Listing 1* zeigt, wie man mit klassischem „null"-Ansatz den Dateinamen aus den Kommandozeilen-Parametern ermitteln könnte. Selbstverständlich gibt es bessere Implementierungen und Bibliotheken zu diesem Zweck, aber uns interessiert hier lediglich der Umgang mit „null".

Bei der Verwendung einer solchen Methode muss der Aufrufer anschließend daran denken, das Ergebnis auf „null" zu prüfen. Dies kann leicht vergessen werden, was die Methode automatisch zu einer Fehlerquelle macht. Da „null" die Ursache dieses Problems ist, eliminiert der Einsatz von „Optional" die Fehlerquelle.

Sprache	Bezeichnung
Java, Swift	Optional
Haskell, Idris, Agda	Maybe
Coq, OCaml, Standard ML, F#	Option
Scala, Rust	Option
C++17	optional

Tabelle 1: „Optional"-Typen in verschiedenen Sprachen

```
public String getFilename(String [] arguments) {
  for (int i = 0; i < arguments.length; i++) {
    if ("-f".equals(arguments[i]) && i+1 < arguments.length) {
      return arguments[i+1];
    }
  }
  return null;
}
```

Listing 1: Klassische Methode mit „return null"

```
public Optional<String> findFilename(String [] arguments) {
  for (int i = 0; i < arguments.length; i++) {
    if ("-f".equals(arguments[i]) && i+1 < arguments.length) {
      return Optional.of(arguments[i+1]);
    }
  }
  return Optional.empty();
}
```

Listing 2: Die gleiche Methode mit „Optional"

Listing 2 zeigt, wie die gleiche Funktionalität mit Verwendung von „java.util.Optional" aussehen kann. Der Methodenname wurde geändert, um bereits deutlich zu machen, dass eine Suche nach einem Dateinamen stattfindet, die auch erfolglos bleiben kann. Der Rückgabetyp ist entsprechend auf „Optional<String>" gesetzt. Die „null"-Verwendung wurde durch Optional.empty() ersetzt und der eigentliche Ergebniswert mittels „Optional.of(...)" verpackt.

Für die Konstruktion eines „Optional"-Objekts gibt es neben diesen beiden Factory-Methoden noch „Optional.ofNullable", falls nicht bekannt ist, ob das übergebene Objekt „null" ist oder nicht. Nach der Erzeugung eines solchen „Optional"-Objekts kann man sich dieses als einen Wrapper vorstellen, der die Referenz auf einen Wert beinhaltet. Im Falle von „Optional.empty()" ist diese Referenz selbst „null"; bei „Optional.of(...)" findet sich dort eben der entsprechende Wert. Wer glaubt, dass es so einfach doch nicht sein kann, dem sei ein Blick auf die Quellen der Klasse „java.util.Optional" empfohlen. Tatsächlich findet sich dort wenig Überraschendes.

Die Vorteile

Nun liefert eine Methode „Optional<String>" anstelle eines einfachen „String" – was bringt uns das? Die „Optional"-Klasse bietet eine Methode „isPresent", mit der geprüft werden kann, ob sich in dem gekapselten Objekt ein gültiger Wert befindet. Damit lässt sich eine „null"-Prüfung umsetzen; eine Verwendung des „String"-Objekts ist anschließend mit „Optional#get" ebenfalls möglich. Bei dieser Umstellung wäre somit nur der Code ein wenig komplizierter geworden, aber Ablauf und Prüfungen blieben im Wesentlichen unverändert. Wer „Optional" nur so weit kennt oder die Klasse nur so verwendet, sollte unbedingt weiterlesen.

Wie bereits gesehen, ist eine Fehlerquelle vollständig ausgeschlossen: Ein Entwickler ist nun nicht mehr in der Lage zu vergessen, dass der Dateiname eventuell gar nicht erfolgreich ermittelt werden konnte. Wird etwa ein „null-String" an den „java.io.File"-Konstruktor übergeben, führt dies zur Ausführungszeit zu einer „NullPointerException". „Optional<String>" schließt dies aus, da der Compiler eine direkte Übergabe an den „File"-Konstruktor nicht akzeptiert.

Ein „Optional<T>" ist kein „T"-Objekt. Daher sind auch die NPEs, die von ei-

nem Methodenaufruf eines „null T"-Objekts wie „String#length" stammen, mit „Optional<T>" nicht möglich – mit Ausnahme der wenigen Methoden, die die Klasse „Optional" selbst anbietet.

Richtig interessant wird „Optional" aber erst, wenn man dessen Methoden mit ins Spiel bringt. Die Kurzfassung für funktionale Programmierer lautet: „java.util.Optional" ist eine Monade. Für alle anderen werden diese Vorteile in den folgenden Abschnitten am Beispiel Punkt für Punkt aufgezeigt.

Das Beispiel: Aus einem erfolgreich gelesenen Dateinamen soll ein „java.io.File"-Objekt entstehen. In einer klassischen Implementierung benötigt man dazu die „null"-Prüfung und kann dann den „File"-Konstruktor aufrufen. Das Ergebnis vom Typ „File" kann somit wiederum „null" sein. Anstatt dies analog mit „Optional<String>" nachzuprogrammieren, zeigt *Listing 3*, dass es mit „Optional#map" deutlich kürzer geht.

Die Methode „map" überprüft, ob ein Wert vorhanden ist. Falls ja, wird dieser an die übergebene Funktion – in diesem Fall an den Konstruktor von „File" – übergeben. Das Ergebnis von „map" ist somit entweder das – wieder in „Optional" verpackte – Ergebnis dieser Funktion oder „Optional.empty()".

Als Nächstes wird die Datei nur ausgelesen, falls sie auch wirklich existiert. Da man mit „File" auch nicht vorhandene Dateien repräsentieren kann, lässt sich dies mit „File#exists" prüfen. Im klassischen, „null"-basierten Java-Code stände hier eine weitere Fallunterscheidung an. *Listing 4* zeigt, wie „Optional#filter" das wiederum kürzer und im gleichen Kontrollfluss erledigen kann.

Wie genau der Konfigurationswert aus der Datei ausgelesen wird, ist hier nicht von Bedeutung. Nehmen wir daher an, dass eine Methode „readConfigValue(File)" existiert, die einen „Integer"-Wert als Ergebnis liefern kann. Sehen wir uns zuerst in *Listing 5* eine vollständige Implementierung der Beispielfunktion mithilfe von „null"-Rückgabewerten an. Sollte etwas schiefgehen, so soll dieser Konfigurationswert standardmäßig auf „0" gesetzt sein. Man sieht gut, dass sich selbst in einem so einfachen Beispiel bereits drei Fall-Unterscheidungen im Quellcode ergeben; das ist ein gutes Indiz für die Komplexität.

Selbstverständlich könnte der entsprechende Eintrag in der Datei selbst fehlen. Dementsprechend benennen wir die Methode „readConfigValue" in „findConfigValue" um und ändern den Rückgabetyp zu „Optional<Integer>". Somit entsteht ein „Optional<File>"-Objekt und mithilfe der „map"-Methode wäre das Ergebnis nach dem Aufruf von „findConfigValue" ein „Optional<Optional<Integer>>"-Objekt. Um eine derart ungewünschte Verschachtelung zu vermeiden, gibt es die Methode „Optional#flatMap". Sie funktioniert genauso wie „map", entfernt aber eine der Schachtelungen, sodass wir ein „Optional<Integer>"-Objekt erhalten.

Ferner gibt es die Methode „Optional#orElse" für den oft benötigten Fall, dass bei Abwesenheit eines Werts ein Standardwert herangezogen werden soll. Sollte in dem „Optional"-Objekt ein gültiger Wert enthalten sein, wird dieser zurückgeliefert, andernfalls der an „orElse" übergebene Wert.

Der Aufruf von „orElse" ist ein Methodenaufruf, dessen Argument fertig berechnet übergeben werden muss. Soll aber beispielsweise ein Standardwert aus einer langsamen Datenbank ermittelt werden, will man diese Auswertung möglichst einsparen können. Zu diesem Zweck kommt „Optional#orElseGet(Supplier<? extends T>)" zum Einsatz, um erst bei feststehender Abwesenheit des Werts den teuren Standardwert zu berechnen. Bringt man dies alles zusammen, erhält man die Methode aus *Listing 6*. Es bleibt lediglich eine einzelne „return"-Anweisung übrig. Diese ist verhältnismäßig leicht zu verstehen.

Die meisten Entwickler benötigen eine kurze Eingewöhnungszeit, aber dann sind insbesondere „map", „flatMap" und „filter" genauso natürlich zu lesen und zu verstehen wie klassische Methodenaufrufe oder Operatoren. Insbesondere, da auch das in Java 8 eingeführte Stream-API auf Methoden mit den gleichen Namen und einer ähnlichen Bedeutung basiert.

Interessant für Java-Entwickler ist auch die Erkenntnis, dass mit Java 8 ein Wechsel von Kontrollfluss- hin zu Datenfluss-orientierter Entwicklung stattfindet. Beim Vergleich der *Listings 5* und *6* sieht man gut, wie im ersten Fall die Kontroll-Struk-

```
Optional<File> file = findFilename(arguments).map(File::new);
```

Listing 3: Verwendung von „Optional#map"

```
Optional<File> existingFile = findFilename(arguments)
      .map(File::new)
      .filter(File::exists);
```

Listing 4: Verwendung von „Optional#filter"

```
private Integer getConfigValue(String[] arguments) {
  String filename = getFilename(arguments);
  if (filename != null) {
    File file = new File(filename);
    if (file.exists()) {
      Integer configValue = readConfigValue(file);
      if (configValue != null) {
        return configValue;
      }
    }
  }
  return 0;
}
```

Listing 5: Vollständige Implementierung mit „null"

```
private Integer getConfigurationValue(String[] arguments) {
  return findFilename(arguments)
          .map(File::new)
          .filter(File::exists)
          .flatMap(this::findConfigValue)
          .orElse(0);
}
```

Listing 6: Vollständige Implementierung mit „Optional"

turen zur Ablaufsteuerung dominieren. Im zweiten Listing hingegen kann man gut von oben nach unten lesend die Verarbeitungskette der Daten sehen:

- Dateiname aus Kommandozeilen-Argumenten heraussuchen
- File-Objekt daraus erstellen
- Sicherstellen, dass die Datei existiert
- Den Konfigurationswert aus der Datei lesen
- Im Fehlerfall den Standardwert „0" verwenden

Die funktionale Programmierung wirbt schon lange zu Recht damit, dass ein Entwickler mit Fokus auf den Datenfluss näher an der eigentlichen Problemstellung bleibt. Letzten Endes geht es bei Software primär um das Verarbeiten von Daten; Kontrollflüsse waren immer nur Mittel zu diesem Zweck. Auch das Stream-API verfolgt diese Philosophie und abstrahiert deswegen von Ablauf-Logiken und Parallelität, um dem Entwickler die Möglichkeit zu geben, sich auf den Datenfluss zu konzentrieren.

Die Nachteile

Nachdem nun zahlreiche Vorteile der Verwendung von „java.util.Optional" identifiziert sind, soll an dieser Stelle auch auf die Nachteile eingegangen werden. Zuallererst sind dies die offensichtlichen Nachteile der meisten Abstraktionen: Laufzeit- und Speicher-Kosten. Einen Wert in ein „Optional"-Objekt zu verpacken, kostet offensichtlich mehr Speicher, als nur für den Wert selbst benötigt wird. Der Overhead ist hier allerdings ähnlich gering wie etwa beim Boxing von „int" auf „Integer". Auch die Laufzeit ist geringfügig langsamer, da eine zusätzliche Indirektion über die „Optional"-Klasse stattfindet. In der Praxis sind diese Kosten jedoch in den allermeisten Fällen vernachlässigbar gering.

Interessanter ist der Nachteil, dass die Grundidee von „Optional", also die Vermeidung von „null" und NPEs, allein durch diese Klasse nicht verwirklicht werden kann. Es gibt den pathologischen Fall, dass eine Methode mit Rückgabetyp „Optional<T>" mithilfe eines „return null;" implementiert wird. Durch Reviews und das Wissen aus Artikeln wie diesem lässt sich einer derart bösartigen Verwendung von „Optional" jedoch effizient entgegenwirken. Nichtsdestotrotz ist und bleibt „null" auf absehbare Zeit ein Bestandteil von Java und somit wird auch die „NullPointerException" nicht vollständig verschwinden.

Insbesondere in neuen Projekten gibt es seit Java 8 kaum noch gute Gründe, um das „null"-Schlüsselwort zu verwenden. Ein Fall, in dem diese Verwendung beispielsweise noch immer unvermeidbar sein kann, ist die Ansteuerung von Bibliotheken. Dabei kann ein Methoden-Argument mit „null" übergeben werden müssen, um die gewünschte Funktionalität nutzen zu können.

Aber auch das JDK selbst hat in Version 8 seine Nachteile bezüglich „Optional". Aus Gründen der Abwärtskompatibilität wurde darauf verzichtet, bestehende APIs auf „Optional" umzuziehen. So kann etwa die Anfrage an eine „Map" mit einem darin nicht enthaltenen Schlüssel weiterhin „null" liefern. In modernen Sprachen wie Scala ist die „get"-Methode mit einem Rückgabetyp „Optional" definiert. In Java 8 muss man stattdessen daran denken, „Optional.ofNullable" für das Ergebnis von „Map#get" zu nutzen.

Wer einmal mit einem „Stream <Optional<T>>" hantiert hat, wird sich zudem fragen, warum es kein „Optional #stream" gibt, um eine Vereinfachung durch „Stream#flatMap" erreichen zu können. Dieser Defekt *[3]* wird leider erst mit Java 9 behoben sein.

Ähnlich wie beim Stream-API gestaltet sich das Debuggen von Lambda-Ausdrücken, wie sie „map", „flatMap" und „filter" verwenden, in den aktuellen IDEs häufig noch unnötig schwierig. Dies liegt daran, dass die IDEs in ihrer Entwicklung zum Teil noch nicht so weit sind, und sollte sich im Laufe der Zeit durch entsprechende Updates erübrigen.

Abschließend ist noch anzumerken, dass die Verwendung von „Optional" Fehlerursachen automatisch unterdrückt. Man sieht einem „Optional.empty()" nicht an, weswegen der Wert fehlt. Soll etwa in obigem Beispiel eine Meldung ausgegeben werden, falls die angegebene Datei nicht existiert, ist die Datenverarbeitungskette zu diesem Zweck zu unterbrechen. Wer oft solche Anforderungen hat, kann sich von fortgeschritteneren Techniken wie zum Beispiel der „Validation"-Monade von scalaz *[4]* inspirieren lassen.

Fazit

Die Vorteile des „Optional"-Einsatzes überwiegen bei Weitem die Nachteile. Nicht umsonst ist dieses Konzept in derart vielen Programmiersprachen zu finden. Seit der Einführung von Lambda-Ausdrücken und Streams mit Java 8 ist nun auch eine elegante monadische Verwendung möglich. Dadurch gelingt es in zunehmendem Maße, in Java 8 Datenfluss-orientiert zu entwickeln, was kürzere, verständlichere und weniger komplexe Programme ergibt.

Der Autor setzt „java.util.Optional", dessen Vorgänger „com.google.guava.Optional" und Varianten in anderen Sprachen seit Jahren in verschiedenen Projekten mit großem Erfolg ein. Die Klasse ist für Team-Mitglieder schnell zu erlernen und einfach zu verwenden; die Vorteile sind in der Praxis schon nach kurzer Zeit deutlich zu spüren. Aufgrund dieser Erfahrungen hat der Autor ein Checkstyle-Plug-in *[5]* geschrieben, das Verwendungen des „null"-Schlüsselworts im Quellcode findet und beanstandet. Bei bestehenden Projekten lässt sich somit die Zahl der Verwendungen stetig reduzieren. Seit etwa einem Jahr entwickelt der Autor zudem mit einem kleinen Team ein Java-Projekt, in dessen Quellcode kein einziges „null" mehr zu finden ist. „NullPointerExceptions" sind dementsprechend selten.

Quellen

[1] Moggi, Eugenio (1991), „Notions of computation and monads". Information and Computation 93 (1)

[2] Hoare, Tony (25 August 2009). „Null References: The Billion Dollar Mistake". *InfoQ.com*.

[3] *https://bugs.openjdk.java.net/browse/JDK-8050820*

[4] *https://github.com/scalaz/scalaz*

[5] *https://github.com/FrankRaiser/checkstyle-null-literal*

Dr. Frank Raiser
frank.raiser@konzept-is.de

Dr. Frank Raiser arbeitet als Software-Entwicklungsingenieur bei der Konzept Informationssysteme GmbH in Ulm. Er berät und unterstützt Projekte sowie Kunden bei technischen Fragestellungen und führt Schulungen und Coachings zu Themen wie „Java", „Clean Code" oder „Agile Entwicklung" durch. Seine Schwerpunkte sind moderne Ansätze in Programmiersprachen und Software-Entwicklungsprozessen.

Oracle BLOB-ZIP-Funktion für die Datenbank

Frank Hoffmann, Cologne Data

Mit kleinen Java-Programmen kann man sehr nützliche Funktionen einfach in die Datenbank integrieren. Im nachfolgenden Beispiel lassen sich Binary-Large-Objects-Files (BLOB) bis 4 GB Volumen aus der Datenbank komprimieren und in eine Tabelle schreiben oder in Prozeduren weiterverarbeiten.

Der Java-Code ist nur ein paar Zeilen lang und baut auf bestehenden Bibliotheken auf. Wir streamen die BLOBs, schreiben das Ergebnis in einen weiteren BLOB und vergeben einen Namen. Der Code ist getestet worden auf den ORACLE-Versionen 9*i*, 10*g*, 11*g* und 12*c*. Die Java-Klasse *(siehe Listing 1)* kann über PL/SQL angesprochen werden *(siehe Listing 2)*.

Für ein Praxisbeispiel wird zuerst mit „create table meintest (meinblob blob);" eine Tabelle angelegt. Dann generiert man mit ein paar Zeilen CSV-Format einen gezippten BLOB *(siehe Listing 3)*.

Ein Blick in die Tabelle liefert das binäre Ergebnis *(siehe Abbildung 1)*. Wir speichern es unter „beispiel.zip" ab *(siehe Abbildung 2)* und sehen dann unser komprimiertes CSV-File *(siehe Abbildung 3)*. Ein Code-Beispiel steht unter „www.cologne-data.com".

```
CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED "OracleBlob"
AS
//
// ORACLEBlob: Zippen einer BLOBDATEI
//
import oracle.sql.BLOB;
import java.util.zip.*;
import java.io.InputStream;

public class OracleBlob
{
 public static void compressBlob
(BLOB unzippedBlob,
 BLOB zippedBlob ,
 java.lang.String zip_entry_name)
 throws Exception
 {
  {
  InputStream in=unzippedBlob.getBinaryStream();
  ZipOutputStream z= new ZipOutputStream
   (zippedBlob.getBinaryOutputStream());
   z.putNextEntry(new ZipEntry(zip_entry_name));
   byte[] buffer=new byte [zippedBlob.getBufferSize()];
   int cnt;
   while ((cnt=in.read(buffer)) > 0)
   {
    z.write(buffer,0,cnt);
   }
   in.close();
   z.close();
  }
 }
}
```

Listing 1

```
create or replace procedure
  P_GET_ZIPPED_FILE
  (
   p_unzipped_blob BLOB,
   p_zipped_blob BLOB,
   p_zipped_entry_name VARCHAR2)
as
language java name
'OracleBlob.compressBlob(oracle.sql.BLOB, oracle.sql.BLOB, java.lang.String)';
```

Listing 2

```
begin
declare
 l_blob_zipped     BLOB;
 l_blob_unzipped   BLOB;
 begin
 -- Erstellung von Beispielblobs
 DBMS_LOB.createtemporary (l_blob_unzipped, TRUE);
 DBMS_LOB.append (l_blob_unzipped, UTL_RAW.cast_to_raw ('a; b; c;'|| chr(10)));
 DBMS_LOB.append (l_blob_unzipped, UTL_RAW.cast_to_raw ('e; f; g;'|| chr(10)));
 DBMS_LOB.createtemporary (l_blob_zipped, TRUE);

 -- JAVA Procedure wird aufgerufen mit den Blobs sowie dem gewünschten Name der ZIP Datei
 p_get_zipped_file (l_blob_unzipped, l_blob_zipped, 'Beispiel.CSV');
 insert into meintest values (l_blob_zipped);
 commit;
 end;
end;
```

Listing 3

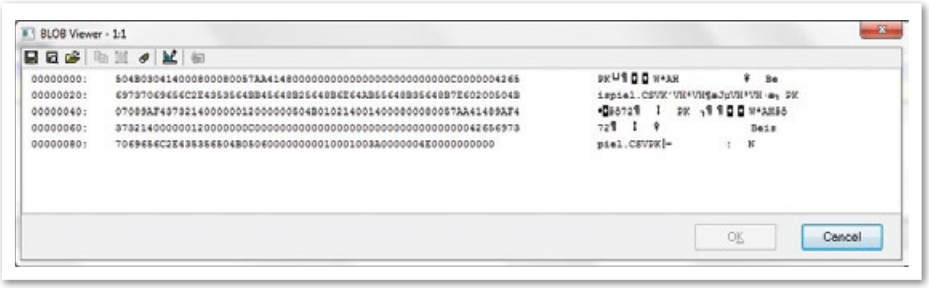


Abbildung 1: Das binäre Ergebnis

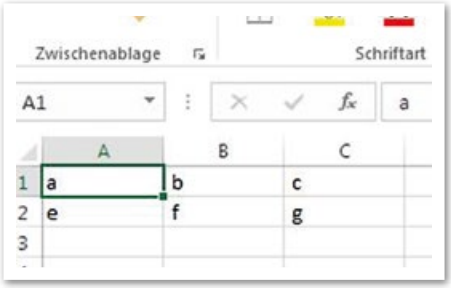


Abbildung 3: Das komprimierte CSV-File

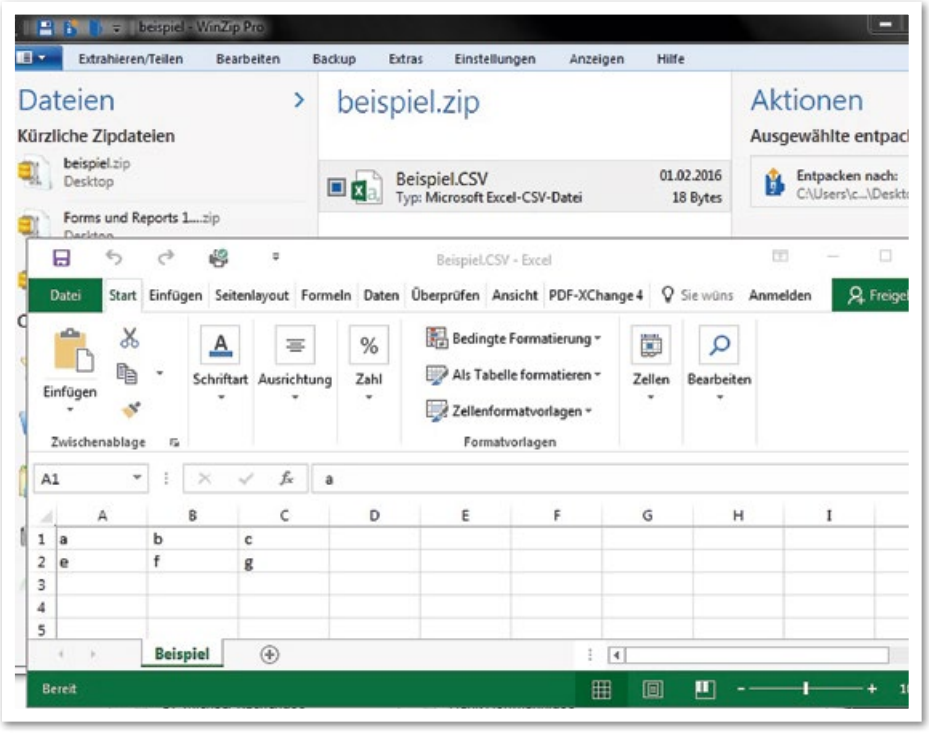


Abbildung 2: Unter „beispiel.zip" abspeichern

Frank Hoffmann
frank.hoffmann@cologne-data.de

Frank Christian Hoffmann ist Dipl.-Ing. Informationsverarbeitung an der GH Paderborn. Seine Schwerpunktthemen sind Oracle Forms Entwicklung (4GL), Migration (beispielsweise zu Forms12c), Modernisierung und Java-Integration. Er war von 1994 bis 1999 Berater beziehungsweise Seniorberater bei Opitz Consulting und Oracle Deutschland und ist seit dem Jahr 1999 Geschäftsführer der Cologne Data GmbH.

Ansible – warum Konfigurationsmanagement auch für Entwickler interessant sein kann

Sandra Parsick

Das automatisierte Konfigurieren von Servern ist dank Orchestrierungswerkzeugen wie Puppet und Chef heute kein Problem mehr. Doch diese eignen sich wenig für die regelmäßige Verteilung von typischen Java-Webapplikationen. Ansible hat dieses Problem erkannt und liefert Lösungen für das Konfigurationsmanagement und die Software-Verteilung aus einer Hand.

Der Artikel zeigt am Beispiel einer Infrastruktur für eine Java-Webapplikation die Funktionsweise von Ansible. Zusätzlich wird darauf eingegangen, warum Ansible auch für Entwickler interessant sein kann. Dabei ist zu erkennen, wie Continuous Deployment auch in einer klassischen Unternehmensstruktur umsetzbar ist.

Ansible beschreibt sich selbst als ein Werkzeug für das Konfigurationsmanagement, für die Verteilung von Software und für die Ausführung von Ad-hoc-Kommandos. Der Begriff „Konfigurationsmanagement" kennt in der IT mehrere Aspekte. Eine Definition lautet: „Das Konfigurationsmanagement umfasst alle technischen, organisatorischen und beschlussfassenden Maßnahmen und Strukturen, die sich mit der Konfiguration (Spezifikation) eines Produkts befassen" *[1]*. In der IT kann sich das auf die Konfiguration der Software beziehen, etwa im Hinblick darauf, welche Datenbank die Anwendung wie benutzen soll. Außerdem kann damit die Konfiguration der Hardware gemeint sein („Wie viel CPU-Kerne soll der Server erhalten?") oder auch die Konfiguration eines Systems („Welche Software soll wie installiert sein?").

Bei Ansible geht es um die Konfiguration von Systemen. Die Idee ist, dass die Konfigurationen per Skript beschrieben sind („Infrastructure as Code"); mithilfe dieser Skripte konfiguriert Ansible automatisiert die jeweiligen Zielmaschinen *(siehe Abbildung 1)*.

Das Thema klingt bisher eher typisch für System-Administratoren als für Entwickler. Daher ein kurzer Einblick, wie die Zusammenarbeit zwischen System-Administratoren und Entwickler in einigen Unternehmen aussieht.

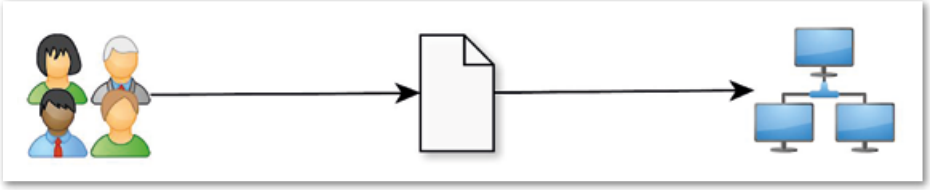


Abbildung 1: „Infrastructure as Code"

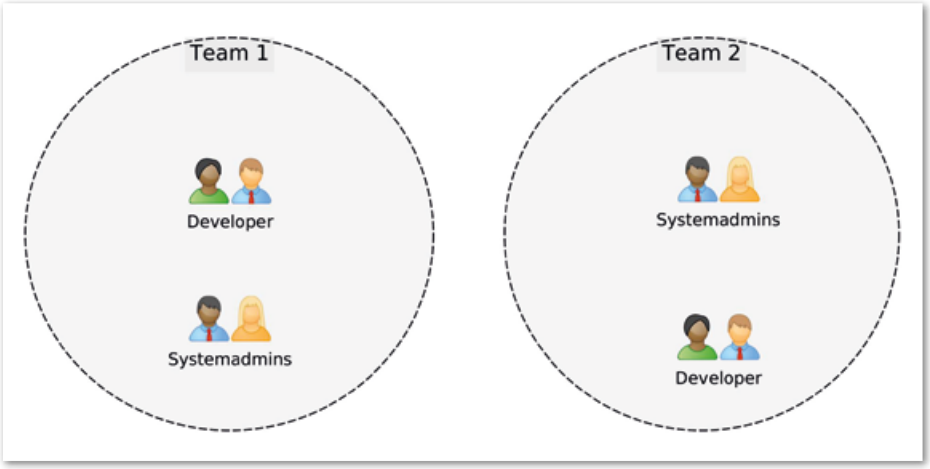


Abbildung 2: Cross-funktionale Teams

Motivation für Entwickler

Wenn man der Fachpresse Glauben schenken kann, wird in den meisten Unternehmen eine DevOps-Kultur gelebt. Es existieren kleine, cross-funktionale Teams, in denen Entwickler und System-Administratoren Hand in Hand zusammenarbeiten und in denen die Teams die Verantwortlichkeiten tragen – angefangen von der Entwicklung bis hin zum Betrieb der Software *(siehe Abbildung 2)*.

Das mag vor allem für Startup-Unternehmen zutreffen, doch in den meisten Unternehmen herrscht eher das klassische Bild; Entwicklung und Betrieb sind in zwei Abteilungen aufgeteilt, die unter Umständen unterschiedliche Ziele verfolgen *(siehe Abbildung 3)*.

Wie die Kommunikationswege zwischen den Abteilungen aussehen können, wird anhand einer Bestellung und Konfiguration eines neuen Testservers beschrieben, der auf Linux basiert *(siehe Abbildung 4)*. Meist erfolgt die Bestellung über ein Ticketsystem oder per E-Mail, in der beschrieben ist, welche Software (wie Oracle Java, Tomcat etc.) das Entwicklungsteam auf dem Server haben möchte und wie sie konfiguriert sein soll.

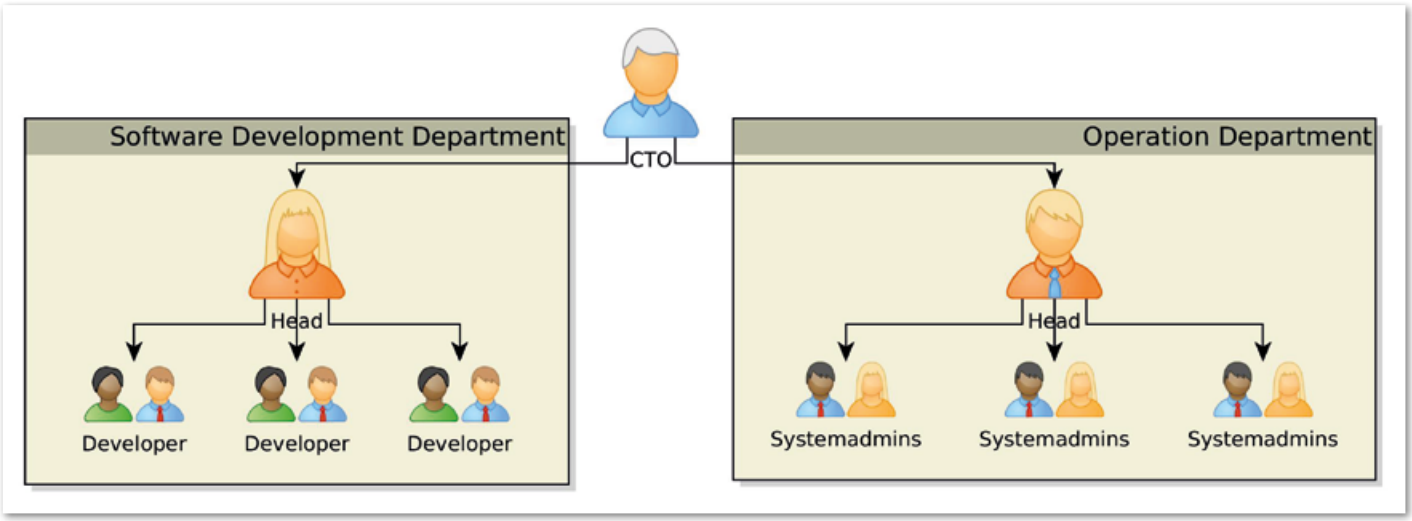


Abbildung 3: Klassische Trennung von Entwicklung und Betrieb

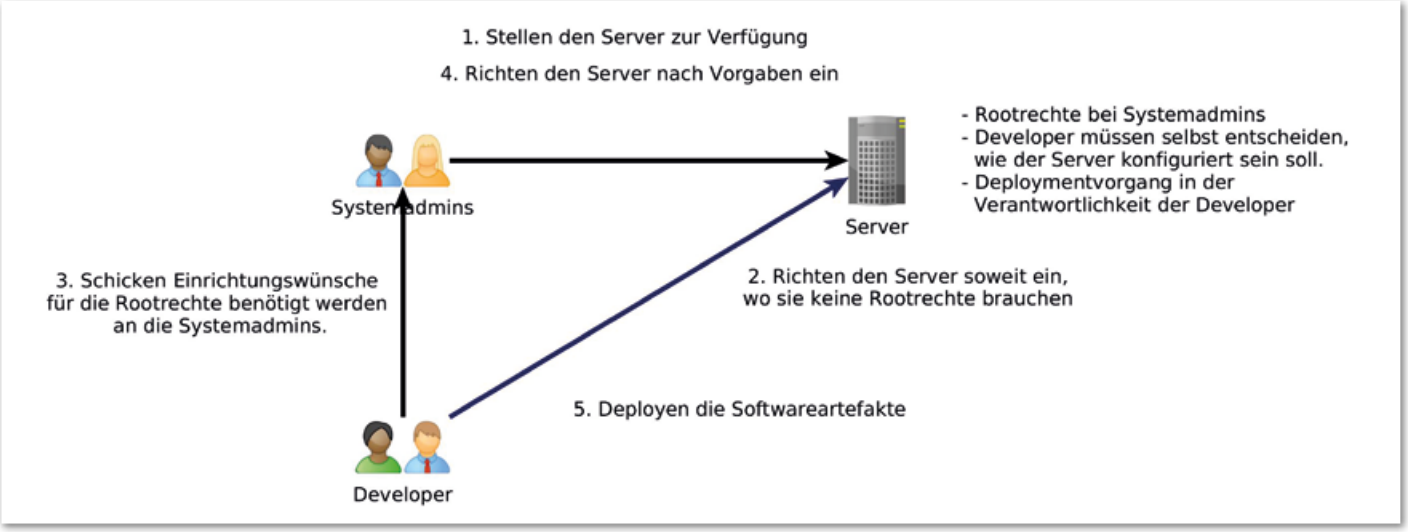


Abbildung 4: Prozess zwischen Entwicklung und Betrieb

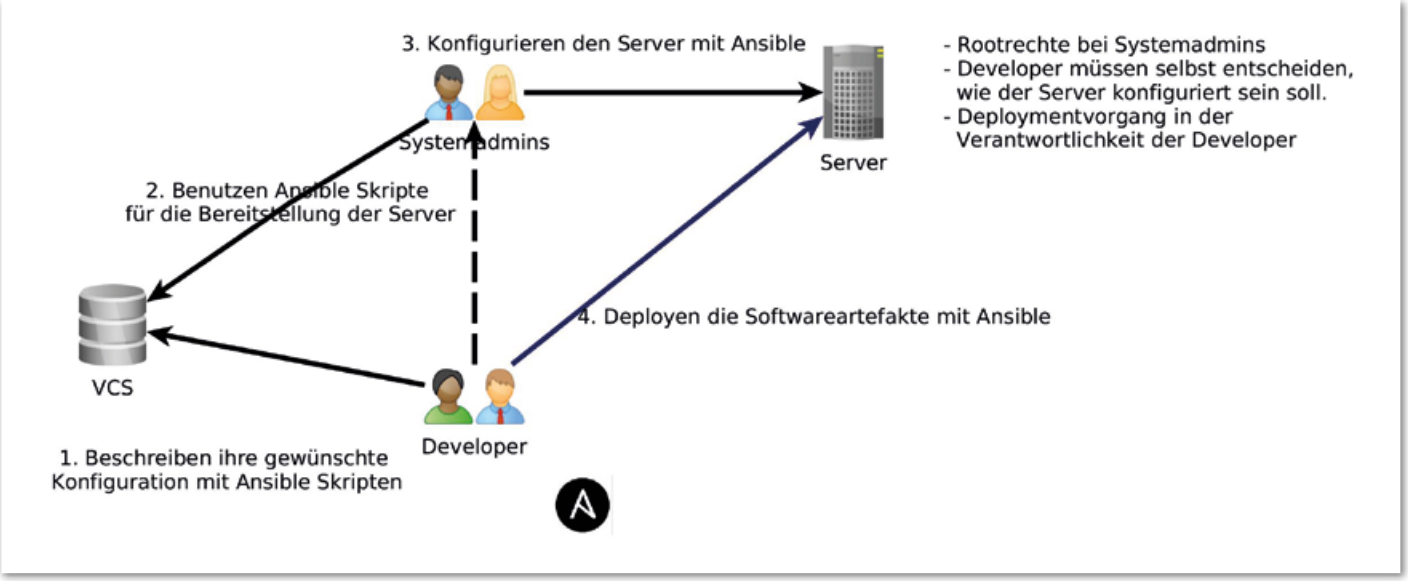


Abbildung 5: Lösungsidee mit Ansible

Oft werden nicht die aktuellsten Versionen installiert, da nur ältere Versionen in den Package-Repositories vorhanden sind. Hinzu kommt, dass die Distributionshersteller die Pakete „Linux-konform" bereitstellen, also aus Entwicklersicht verteilt auf das ganze System. Im besten Fall einigen sich die Betriebs- und die Entwicklungsabteilung darauf, dass die Entwickler die Java-Anwendungen selbst betreuen. Dann installieren und konfigurieren die Entwickler die Anwendungen manuell. Doch sobald der zweite oder auch dritte Test-Server angeschafft und bereitgestellt wird, kommt es zu Abweichungen, da der Mensch einfach nicht dazu geschaffen ist, monoton Sachen zu wiederholen.

Dasselbe geschieht bei der manuellen Verteilung der Anwendungen in Form von WAR-Dateien. Immer wieder hakt irgendetwas, weil eine Kleinigkeit vergessen wurde. Hier kann Ansible eine große Hilfe sein, da die Konfigurations- und Verteilungsschritte mithilfe verständlicher Skripte beschrieben werden können und für die Ausführung der Skripte derselbe Mechanismus gebraucht wird wie bei dem manuellen Vorgang, nämlich ein SSH-Zugang *(siehe Abbildung 5)*.

Die Funktionsweise

Ansible ist in Python geschrieben. Um es zu benutzen, ist Python nicht zwangsläufig erforderlich, da die Ansible-Skripte in YAML geschrieben sind. YAML ist eine Abstraktion von JSON mit dem Unterschied, dass sie besser lesbar ist *[2]*. Die Ansible-Skripte werden „Playbooks" genannt.

Damit Ansible eingesetzt werden kann, muss auf den Zielmaschinen Python 2.x installiert und ein Zugriff auf diese Maschinen über SSH möglich sein. Ansible ist nur auf der Maschine zu installieren, von der die Playbooks ausgeführt werden („Host Maschine"); sie muss ein Linux-System sein. Für Windows auf den Zielmaschinen bietet Ansible nur eine rudimentäre Unterstützung an. Da Python von System-Administratoren gerne für ihre eigenen Skripte benutzt wird, ist die Wahrscheinlichkeit recht hoch, dass Python standardmäßig auf den Linux-Servern installiert ist.

Wenn die in YAML geschriebenen Playbooks ausgeführt werden, übersetzt Ansible diese in Python-Skripte und führt sie auf den Zielmaschinen aus. Anschließend werden sie wieder von den Zielmaschinen entfernt.

Ansible wird in den nächsten Abschnitten anhand einer Infrastruktur für eine klassische Java-Web-Anwendung vorgestellt. Dabei werden ein Tomcat 8 und ein OpenJDK

installiert. Die Anwendung wird als WAR-Datei auf dem Tomcat eingesetzt.

Playbook für das Setup

Das in *Listing 1* gelistete Playbook „setup-app.yml" beschreibt die Konfiguration für die Installation von Tomcat 8 und OpenJDK. Jedes Playbook fängt mit der Beschreibung an, für welche Server („hosts") dieses Playbook gilt. Das kann eine Auflistung einzelner Hostnamen sein oder eine Bezeichnung für eine Gruppe von Servern. Diese Gruppe wird in einer eigenen Datei beschrieben, dem sogenannten „inventory". Dazu später mehr.

Optional lassen sich Variablen definieren („vars"), die später im Playbook vorkommen („{{ variabel_name }}"). Danach werden die Schritte beschrieben, wie der Server konfiguriert werden muss („tasks"). Ansible arbeitet diese Tasks von oben nach unten ab. Sobald ein Task fehlschlägt, ist die komplette Ausführung beendet – ohne die schon ausgeführten Tasks rückgängig zu machen.

Jeder Task beschreibt einen Aufruf eines sogenannten „module". Das sind fertige Skripte, die den Server konfigurieren. Sie müssen mindesten zwei Bedingungen erfüllen: Idempotent sein, ein mehrmaliges Ausführen muss also immer dasselbe Ergebnis liefern, und ihre Rückgabewerte müssen im JSON-Format sein. Die Skripte werden mithilfe von Key-Value-Paaren parametrisiert.

Die Skriptsprache der Module ist Ansible egal. Einzige Bedingung ist, dass ein Interpreter für die gewählte Sprache auf den Zielmaschinen installiert ist. Man wird aber sehr selten in die Lage kommen, eigene Module zu schreiben *[3]*, da Ansible von Haus aus eine beträchtliche Anzahl fertiger Module mitbringt *[4]*.

Im *Listing 1* wird als erster Task das Modul „apt" aufgerufen. Es soll über den Paket-Manager „apt" dafür sorgen, dass das Paket „openjdk-7-jdk" („name=openjdk-7-jdk") auf der Zielmaschine installiert ist („state=present"). Wenn also das angeforderte Paket schon installiert ist, dann macht das Modul nichts, ansonsten installiert es das Paket über den Paket-Manager.

Da für die Installation meist „sudo"-Rechte erforderlich sind, bekommt dieser Task zwei weitere Anweisungen („become: yes" und „become_method: sudo") mit, damit Ansible ihn mit „sudo" ausführt. Jeder Task kann mit einer Beschreibung versehen werden („name"). Sie vereinfacht es später bei der Ausführung festzustellen, welcher Task gerade ausgeführt wird.

Manchmal soll ein Task auf der Host-Maschine ausgeführt werden und nicht auf der Ziel-Maschine. Dafür gibt es „local_action", die als Eingabeparameter den Modulnamen und seine Parameter bekommt. Im *Listing 1* ist es der Task „Download Tomcat 8", der ein Tomcat-8-Archiv auf der Host-Maschine herunterlädt. Tomcat soll als Service laufen, dafür wird ein „init.d"-Skript benötigt.

Ansible bietet ein Modul „copy" an, das Dateien vom Host zur Ziel-Maschine kopiert (im *Listing 1* Task „install init.d script for tomcat"). Manchmal muss man Dateien beim Kopieren noch auf die jeweilige Ziel-Maschine anpassen, dafür gibt es das Modul „template". Dabei werden Dateien vom Host zur Ziel-Maschine kopiert und mithilfe einer Template-Engine entsprechend angepasst. Ansible benutzt dafür Jinja2 *[5]*, eine Template-Engine für Python.

In *Listing 1* soll anhand der Template-Datei „setenv.sh.j2" im Task „setup setenv.sh" mit „CATALINA_OPTS="{{ catalina_opts }}"" eine „setenv.sh"-Datei erzeugt werden. Die Besonderheit an diesem Task ist, dass er nur ausgeführt werden soll, wenn die Variable „catalina_opts" definiert ist („when: catalina_opts is defined").

Damit Tomcat auch gestartet und gestoppt werden kann, müssen alle Tomcat-Shell-Skripte ausführbar sein. Dies stellt das Modul „file" sicher (im *Listing 1* die Task „ensure tomcat scripts are executable"). Normalerweise muss dieser Task für jede Datei einzeln definiert sein. Ansible bietet aber eine Möglichkeit an, diesem Task eine Liste an Werten mitzugeben, für die ein Task wiederholt werden soll („with_items"). Im Beispiel wird diese Liste mithilfe eines vorherigen Task („shell: ls /opt/{{ tomcat_base_name }}/bin/*.sh") er-

```
- hosts: application-server
  vars:
    tomcat_version: 8.0.24
    tomcat_base_name: apache-tomcat-{{ tomcat_version }}
    #catalina_opts: "-Dkey=value"

  tasks:
    - name: install java
      apt: name=openjdk-7-jdk state=present
      become: yes
      become_method: sudo

    - name: Download Tomcat 8
      local_action: get_url url="http://archive.apache.org/dist/tomcat/
tomcat-8/v{{ tomcat_version }}/bin/{{ tomcat_base_name }}.tar.gz" dest=/tmp

    - name:
      file: name=/opt mode=777
      become: yes
      become_method: sudo

    - name: Install Tomcat 8
      unarchive: src=/tmp/{{ tomcat_base_name }}.tar.gz dest=/opt creates=/
opt/{{ tomcat_base_name }} owner=vagrant group=vagrant

    - name: Set link to tomcat 8
      file: src=/opt/{{ tomcat_base_name }} dest=/opt/tomcat  state=link
force=yes

    - name: setup setenv.sh
      template: dest="/opt/{{ tomcat_base_name }}/bin/setenv.sh"
src="roles/tomcat8/templates/setenv.sh.j2" mode=755
      when: catalina_opts is defined

    - shell: ls /opt/{{ tomcat_base_name }}/bin/*.sh
      register: tomcat_scripts
      ignore_errors: yes

    - name: ensure tomcat scripts are executable
      file: name={{item}} mode=755
      with_items: tomcat_scripts.stdout_lines

    - name: install init.d script for tomcat
      copy: src=roles/tomcat8/files/init.d/tomcat dest=/etc/init.d/tomcat
owner=vagrant group=vagrant mode=755
      become: yes
      become_method: sudo
```

Listing 1

mittelt und in der Variable „tomcat_scripts“ gespeichert („register: tomcat_scripts“), durch die dann im nächsten Task iteriert wird.

Um das Playbook auszuführen, muss in der Konsole „ansible-playbook setup-app.yml -u remote-user -i inventories/test“ aufgerufen werden. Der Parameter „-u“ definiert, mit welchem Benutzer das Playbook auf der Zielmaschine ausgeführt werden soll. Er ist gleichzeitig auch der SSH-Login-Benutzer. Der Parameter „-i“ definiert, welches Inventory für das Playbook benutzt wird.

Inventories

Wie im letzten Abschnitt beschrieben, fängt jedes Playbook mit einer Auflistung an, für welche Server dieses Playbook gelten soll. Die Auflistung kann auch Gruppen von Servern beinhalten. Diese werden in einem sogenannten „inventory“ genauer spezifiziert *(siehe Listing 2)*. Als Format wird INI benutzt *[6]*. Gruppen werden mit „[Gruppenname]“ definiert und dann folgt die Auflistung der Server (als IP-Adresse, Hostname etc.), die zu dieser Gruppe gehören. Zusätzlich lassen sich Gruppen von Gruppen bilden („[Gruppe_von_Gruppen_Name:children]“), die dann als Auflistung Gruppen haben. Zusätzlich zu den Servernamen können Variable als Key-Value-Paare definiert werden, die dann für den speziellen Server gelten. Sollen Variable für eine ganze Gruppe gelten, werden sie nach „[Gruppenname:vars]“ als eine Auflistung von Key-Value-Paaren definiert.

Mithilfe mehrerer Inventory-Dateien lassen sich Test- und Produktionsumgebung voneinander getrennt verwalten. Die Inventories haben beispielsweise dieselben Gruppen definiert, es werden dort aber andere Server aufgelistet. So können dieselben Playbooks für beide Umgebungen benutzt werden und der Parameter „-i“ beim Aufruf der Playbooks steuert, ob sie Richtung „Produktion“ oder „Test“ ausgeführt werden sollen.

Role

Aus Entwicklersicht lassen sich die Tasks für die Tomcat-Installation aus *Listing 1* gut zu einer Einheit gruppieren, da die Tasks allein betrachtet nicht viel Sinn ergeben. In Ansible können solche Einheiten mithilfe sogenannter „roles“ gebildet werden *[7]*. Dafür werden die Roles anhand einer festen Projektstruktur definiert *(siehe Listing 3)*. Dabei bildet jeder Unterordner von „roles“ eine Role ab; es werden zwei Roles definiert, „jdk“ und „tomcat8“.

```
[application-server]
192.168.33.10
ubuntu-server db_host=mysql01

[mysql-db-server]
mysql1
mysql2

[oracle-db-server]
db-a.oracle.company.com
db-b.oracle.company.com

[database-server:children]
mysql-db-server
oracle-db-server

[application-server:vars]
message="Welcome"

[database-server:vars]
message="Hello World!"
```

Listing 2

```
.
├── inventories
│   ├── production
│   └── test
├── roles
│   ├── jdk
│   │   └── tasks
│   │       └── main.yml
│   └── tomcat8
│       ├── defaults
│       │   └── main.yml
│       ├── files
│       │   └── init.d
│       │       └── tomcat
│       ├── tasks
│       │   └── main.yml
│       └── templates
│           └── setenv.sh.j2
└── setup-app.yml
```

Listing 3

```
- hosts: application-server
  roles:
    - jdk
    - { role: tomcat8, tomcat_version: 8.0.30 }
```

Listing 4

```
- hosts: application-server
  roles:
    - {role: deploy-on-tomcat,
webapp_source_path: ./demo-app-
ansible-deploy-1.0-SNAPSHOT.war,
webapp_target_name: demo }
```

Listing 5

```
.
├── inventories
│   ├── production
│   └── test
├── roles
│   ├── deploy-on-tomcat
│   │   ├── defaults
│   │   │   └── main.yml
│   │   └── tasks
│   │       ├── cleanup-webapp.yml
│   │       ├── deploy-webapp.yml
│   │       ├── main.yml
│   │       ├── start-tomcat.yml
│   │       └── stop-tomcat.yml
└── deploy-demo.yml
```

Listing 6

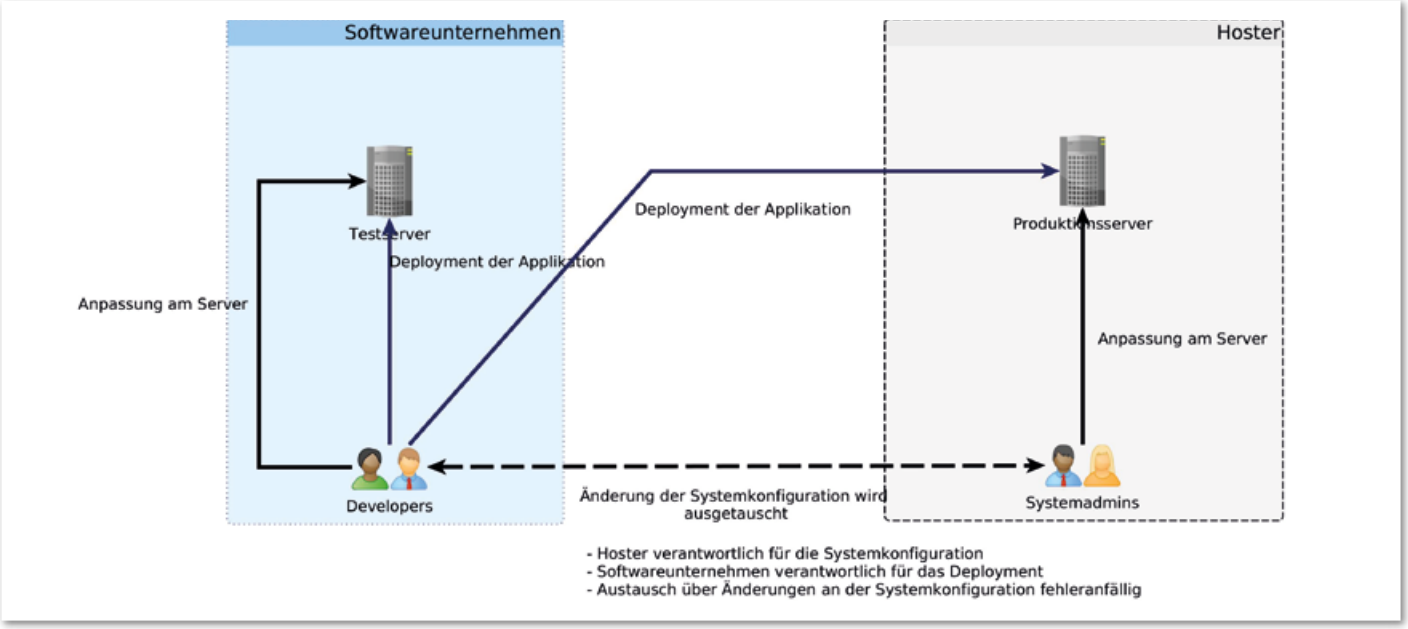


Abbildung 6: Prozess zwischen Entwicklung und externem Hoster

Alle Tasks einer Role sind in „tasks/main.yml" definiert. Dateien, die unverändert auf die Zielmaschine kopiert werden sollen, werden unter „files" abgelegt; Template-Dateien unter „template". In „default/main.yml" sind Defaultwerte für Variablen definiert. Das Playbook aus *Listing 1* kann dann, wie in *Listing 4* gezeigt, vereinfacht werden.

Deployment der Webapplikation

Nachdem Tomcat und Java installiert sind, wird jetzt die Webapplikation in Form einer WAR-Datei eingerichtet. Dazu definiert man ein separates Playbook „deploy-demo.yml" *(siehe Listing 5)*. Es ruft die Role „deploy-on-tomcat" auf *(siehe Listing 6)*. Deren „main.yml"-Datei definiert die Tasks *(siehe Listing 7)* für das Deployment. Hier sind die Tasks noch weiter in einzelne Dateien gruppiert, die durch die „include"-Anweisungen zusammengeführt werden. Dann sieht der Ablauf für das Deployment folgendermaßen aus:

1. Tomcat herunterfahren („stop-tomcat.yml", *siehe Listing 8)*
2. Alte Webapplikation löschen („cleanup-webapp.yml", *siehe Listing 9)*
3. Neue Webapplikation hochladen („deploy-webapp.yml", *siehe Listing 10)*
4. Tomcat hochfahren („start-tomcat.yml", *siehe Listing 11)*

Auch dieses Playbook wird mit „ansible-playbook" ausgeführt („ansible-playbook deploy-demo.yml -u remote-user -i inventories/test").

```
- include: stop-tomcat.yml
- include: cleanup-webapp.yml
- include: deploy-webapp.yml
- include: start-tomcat.yml
```

Listing 7

```
- name: stop tomcat
  command: service tomcat stop

- name: wait tomcat shutdown
  wait_for: port=8080 state=stopped timeout=60
```

Listing 8

```
- name: cleanup {{ webapp_target_name }}
  file: name={{tomcat_app_base}}/{{ webapp_target_name }} state=absent
```

Listing 9

```
- name: delete previous backup
  file: path={{ tomcat_app_base }}/{{ webapp_target_name }}.war.previous
state=absent

- name: create new backup
  command: mv {{ tomcat_app_base }}/{{ webapp_target_name }}.war {{ tom-
cat_app_base }}/{{ webapp_target_name }}.war.previous
  ignore_errors: yes

- name: copy webapp {{ webapp_source_path }} to {{ webapp_target_name }}
  copy: src={{ webapp_source_path }} dest={{ tomcat_app_base }}/{{ webapp_
target_name }}.war mode=660
```

Listing 10

```
- name: start tomcat
  command: service tomcat start

- name: wait for tomcat to start
  wait_for: port=8080 timeout=60
```

Listing 11

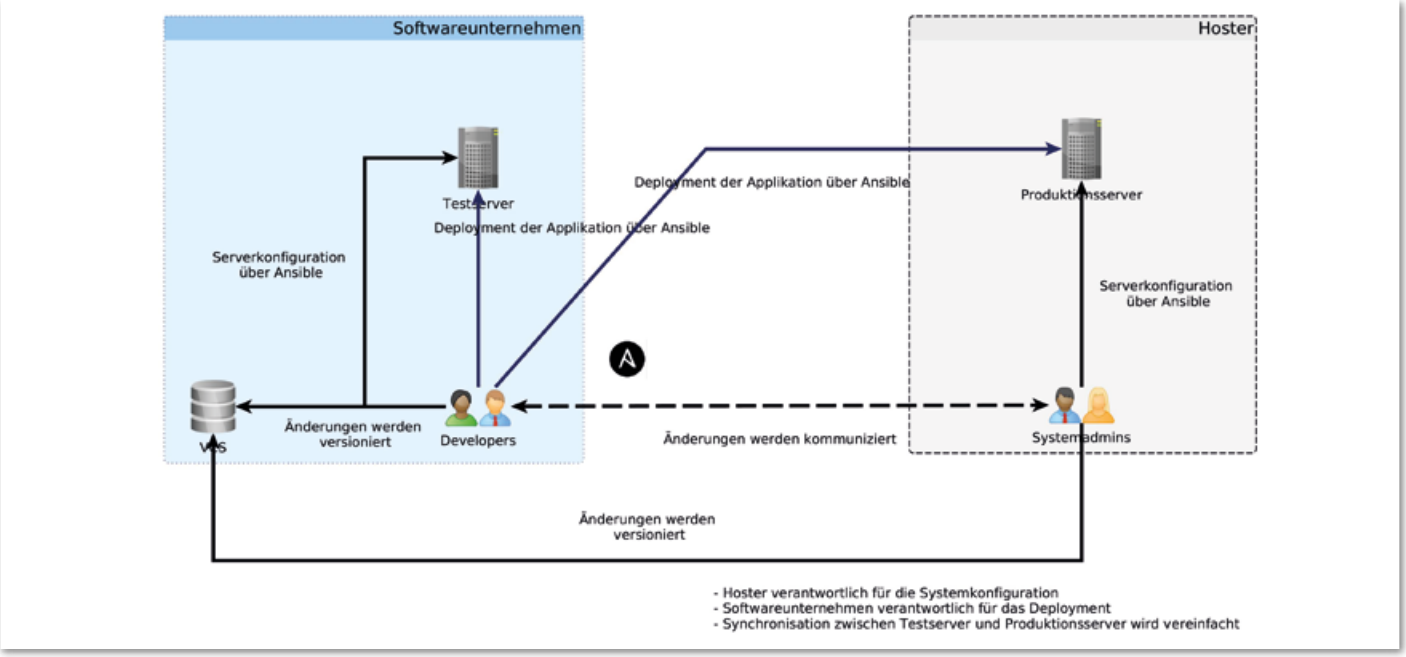


Abbildung 7: Lösungsidee für die Kooperation mit Hoster

Weitere Einsatzszenarien aus Entwicklersicht

Die in der Motivation vorgestellte Ausgangssituation *(siehe Abbildung 4)* kann auch in einer leicht abgewandelten Variante existieren. In dieser Variante sind die Kommunikationswege zwischen Entwicklung und Betriebsabteilung genauso wie in der vorgestellten Ausgangssituation. Der Unterschied liegt darin, dass die Betriebsabteilung ein Konfigurationsmanagement-Werkzeug wie zum Beispiel Puppet im Einsatz hat. Dennoch gibt es dieselben Probleme. Eine Lösung wäre, dass die Entwicklung Zugang zu den Puppet-Skripten bekommt und ihre Änderungswünsche selbst implementiert. Die Betriebsabteilung kontrolliert die Änderungen und führt sie aus. Dieses Vorgehen beschleunigt die Zeit zwischen Anforderung und Umsetzung. Die Automatisierung des Deployments kann wiederum in Ansible erfolgen.

Eine weitere Ausgangssituation kann sein, dass die Produktionsserver bei einem externen Hoster betrieben werden, aber die Testserver im eigenen Unternehmen *(siehe Abbildung 6)*.

Damit die Tests in einer produktionsnahen Umgebung ausgeführt werden können, ist die Konfiguration der Server zwischen Hoster und eigenem Unternehmen zu kommunizieren. Meist erfolgt dies per E-Mail und/oder über ein Ticketsystem. In diesem Vorgehen sind jedoch einige Fallstricke versteckt. Beide Seiten müssen immer daran denken, die andere Seite zu informieren, wenn Änderungen an der Konfiguration erfolgen. Das wird gerne im Eifer des Gefechts vergessen. Dann kann die Umsetzung sich unterscheiden.

Je nachdem, wer das Deployment auf den Produktionsserver machen darf, kann es auch hier Unterschiede in der Umsetzung geben. Auch wenn die Verantwortlichkeit des Deployments bei den Entwicklern liegt, wird auch hier meist nur ein SSH-Login bereitgestellt und das Deployment erfolgt manuell. Das alles zusammengenommen führt oft dazu, dass mit der Zeit die Umsetzung der Konfiguration und des Deployments auf dem Test- und auf dem Produktionsserver auseinanderlaufen.

Eine Lösung wäre, dass der Hoster und die eigene Entwicklung Zugriffe auf gemeinsame Ansible-Playbooks haben, die die Konfiguration der Server sowie das Deployment für beide Umgebungen beschreiben. Mit deren Hilfe ist sichergestellt, dass die Test- und die Produktionsserver gleich konfiguriert sind und dass das Deployment in der Produktion genauso abläuft wie in den Tests *(siehe Abbildung 7)*.

Fazit

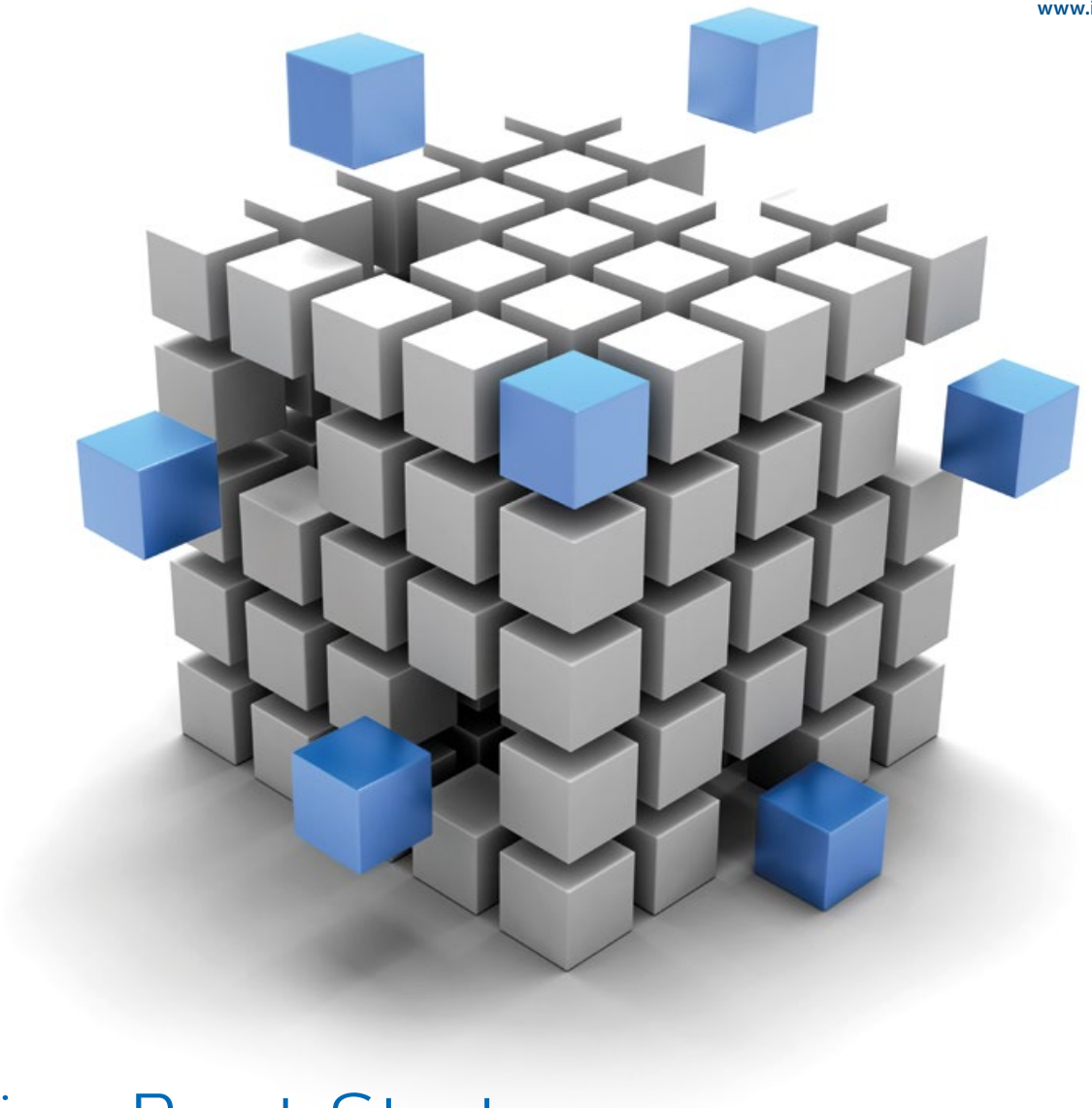
Dieser Artikel gibt einen kleinen Überblick über Ansible und zeigt auf, dass es auch für Entwickler lohnt, sich mit Konfigurationsmanagement-Werkzeugen auseinanderzusetzen. Wer tiefer in Ansible einsteigen möchte, dem sei die sehr gute Dokumentation von Ansible *[8]* und *[9]* empfohlen. Alle hier vorgestellten Playbooks findet man auch auf GitHub *[10]*.

Quellen

[1] *https://www.projektmagazin.de/glossarterm/konfigurationsmanagement*
[2] *https://en.wikipedia.org/wiki/YAML*
[3] *http://docs.ansible.com/ansible/developing_modules.html*
[4] *http://docs.ansible.com/ansible/list_of_all_modules.html*
[5] *http://jinja.pocoo.org/docs/dev*
[6] *https://en.wikipedia.org/wiki/INI_file*
[7] *http://docs.ansible.com/ansible/playbooks_roles.html#roles*
[8] *http://docs.ansible.com*
[9] *Lorin Hochstein: Ansible: Up and Running. O'Reilly Media (2014)*
[10] *https://github.com/sparsick/ansible-java-aktuell*

Sandra Parsick
info@sandra-parsick.de

Sandra Parsick, geb. Kosmalla, ist als freiberufliche Software-Entwicklerin und Consultant im Java-Umfeld tätig. Seit dem Jahr 2008 beschäftigt sie sich mit agiler Software-Entwicklung in verschiedenen Rollen. Ihre Schwerpunkte liegen im Bereich der Enterprise-Anwendungen, agilen Methoden, Software Craftsmanship und in der Automatisierung von Softwareentwicklungs-Prozessen. In ihrer Freizeit engagiert sie sich in der Softwerkskammer Dortmund.



Spring Boot Starter – komfortable Modularisierung und Konfiguration

Michael Simons, ENERKO Informatik GmbH

Dieser Artikel zeigt anhand eines konkreten Beispiels, der Optimierung von Webressourcen, wie eigene Spring Boot Starter dabei helfen können, Anwendungen sauber zu modularisieren.

Auch in Zeiten von Microservices ist Architektur und sinnvolle Modularisierung wichtig. Ein Microservice ohne innere Struktur ist ebenso wenig wartbar wie ein Monolith. Auch wenn Architekturen immer feiner zergliedert werden, gibt es dennoch Module, die in mehreren Microservices genutzt werden, ohne selber wiederum ein Microservice zu sein. Konkrete Beispiele sind:

- Datenstrukturen (Entitäten) und zugehörige Repositories für wiederkehrende Aufgaben
- Logging, Status (Healthchecks) und ähnliche querschnittliche Funktionen, die gerade zum Management der immer mehr werdenden Anwendungen unerlässlich sind
- Integration (Caches, Datenbanken und vieles mehr)

Natürlich müssen diese Module nicht nur bereitgestellt werden, sie sind auch zu konfigurieren. In vielen Fällen kann dabei allerdings eine Vorgabe mit sinnvollen Standardwerten manuellen Mehraufwand erheblich verringern. Eine Möglichkeit ist es, eigene Module zur Bereitstellung von Funktionen und zur Autokonfiguration für Spring-Boot-basierte Anwendungen zu erstellen.

Spring Boot

Spring Boot *[1]* kann im weitesten Sinne als Schirmprojekt im Spring-Eco-System betrachtet werden, das mit folgenden Zielen geschaffen wurde:

- Einen schnelleren und fehlerfreieren Zugang zur Anwendungsentwicklung mit dem Spring Framework *[2]* zu ermöglichen
- Sinnvolle Vorbelegung der am häufigsten benutzen Funktionen, die aber ohne Klimmzüge überschrieben werden können, zu bieten
- Eine große Auswahl nicht-funktionaler Eigenschaften, die in annähernd jedem Projekt benötigt werden, zur Verfügung zu stellen

Während des Bootstrapping von Anwendungen und deren Autokonfiguration wird zu keinem Zeitpunkt Code generiert. Alle Mittel, die von Spring Boot genutzt werden, um festzustellen, welche Bibliotheken verfügbar sind, in welchem Umfeld eine Anwendung läuft und wie sie konfiguriert ist, stehen auch außerhalb des Frameworks zur Entwicklung eigener Starter zur Verfügung.

JavaScript im Griff mit Maven, WebJars und Wro4j

Auch im Jahr 2016 soll es den Bedarf geben, Webanwendungen mit serverseitig gerendertem HTML-Code, JavaScript als progressive Verbesserung und einfacher Verwaltung von JavaScript zu erstellen, ohne das „Universal Install Script" benutzen zu müssen *(siehe Abbildung 1)*.

Wir haben alle schon über Maven geflucht, aber in der Regel funktioniert Maven unauffällig und ist ziemlich gut darin, Abhängigkeiten zu verwalten, aufzulösen und widersprüchliche Versionen (wie mit dem „enforcer-plugin") zu entdecken.

WebJars *[3]* ist ein Projekt von James Ward *[4]* und ermöglicht die explizite und einfache Verwaltung von JavaScript, CSS und ähnlichen Abhängigkeiten mit Maven, SBT, Gradle, ivy oder anderen Build-Tools. Spring Boot unterstützt WebJars unmittelbar und Abhängigkeiten können direkt genutzt werden *(siehe Listing 1)*.

Wro4j *[5]* ist ein Werkzeug zur Analyse und Optimierung von Webressourcen, das viele moderne Werkzeuge aus der Webentwicklung zusammenbringt. Ziel ist es, Abhängigkeiten im pom.xml zu definieren *(Listing 2)*.

Daraus entstehen dann logische Einheiten. In diesem kleinen Beispiel soll eine Library mit allen fremden Ressourcen zusammen mit den eigenen JavaScript-Programmen ein gemeinsames Modul bilden. Dabei wird die Wro4j-XML-Notation genutzt, es steht darüber hinaus auch eine Groovy-DSL zur Verfügung *(siehe Listing 3)*.

Diese Konfiguration sollte mit dem zu erstellenden Starter ausreichen, um jQuery, AngularJS und eigene Funktionen in eine Webseite über ein einziges Element einzubinden.

Spring Boot Starter

Zu einem Starter gehören folgende Komponenten:

- Ein „autoconfigure"-Modul, das die automatische Konfiguration mit ihren entsprechenden Standardwerten beinhaltet
- Das eigentliche „starter"-Modul, das vom „autoconfigure"-Modul abhängig ist und gegebenenfalls weitere Abhängigkeiten mitbringt

In der eigentlichen Anwendung ist das „starter"-Modul eingebunden. Natürlich können beide Module in einem Modul kombiniert werden.

Pivotals Spring Developer Advocate Josh Long *[6]* spricht sich explizit dagegen aus, die Autokonfiguration durch Spring Boot als Magie zu bezeichnen, und er hat Recht damit. Die automatische Konfiguration der Umgebung basiert auf zwei Kernkomponenten

- Profilen („@Profile", verfügbar seit Spring 3.1)
- Bedingungen („@Conditional" im Zusammenspiel mit „org.springframework.context.annotation.Condition", verfügbar seit Spring 4)

Profile ermöglichen die Bereitstellungen von Beans auf Basis aktivierter Umgebungen („spring.profiles.active"), Bedingungen ermöglichen einfache bis komplexe Bedingungen, unter denen Beans instanziiert und konfiguriert werden:

- „@Conditional" aus dem Kernframework selber ermöglicht mit der Implementierung eigener Bedingungen beliebige Abfragen

```
<link rel='stylesheet' href='/webjars/bootstrap/3.1.0/css/bootstrap.min.css'>
```

Listing 1

Abbildung 1: The Universal Install Script [18]

```
<properties>
     <jquery.version>1.11.3</jquery.version>
     <angularjs.version>1.2.29</angularjs.version>
</properties>
<!-- ... -->
<dependencies>
     <dependency>
          <groupId>org.webjars</groupId>
          <artifactId>jquery</artifactId>
          <version>${jquery.version}</version>
     </dependency>
     <dependency>
          <groupId>org.webjars</groupId>
          <artifactId>angularjs</artifactId>
          <version>${angularjs.version}</version>
     </dependency>
</dependencies>
```

Listing 2

```
<groups xmlns="http://www.isdc.ro/wro">
     <group name="lib">
          <js minimize="false">/webjars/jquery/@jquery.version@/jquery.min. js</js>
          <js minimize="false">/webjars/angularjs/@angularjs.version@/angular.min.js</js>
          <js minimize="false">/webjars/angularjs/@angularjs.version@/angular-route.min.js</js>
     </group>

     <group name="biking2">
          <group-ref>lib</group-ref>
          <css>/css/stylesheet.css</css>
          <js>/js/app.js</js>
          <js>/js/controllers.js</js>
          <js>/js/directives.js</js>
     </group>
</group>
```

Listing 3

Aus Spring Boot selbst einige Highlights:

- „@ConditionalOnBean / @ConditionalOnMissingBean": Konfiguration oder Beans werden nur erzeugt, wenn bestimmte andere Beans vorliegen oder fehlen
- „@ConditionalOnClass / @ConditionalOnMissingClass": Konfiguration oder Beans werden nur erzeugt, wenn bestimmte Klassen auf dem Klassenpfad verfügbar sind
- „@ConditionalOnResource": Tritt nur in Kraft, wenn spezifische Ressourcen verfügbar sind
- „@ConditionalOnProperty": Eine mächtige Bedingung zur Abfrage auf konkrete Eigenschaften in Konfigurationsdateien

So sehr die automatische Umgebung magisch wirkt, sie ist es nicht. Spring Boot stellt mit dem ConditionEvaluationReport *[7]* sogar eine Reporting-Lösung zur Verfügung, mit der genau nachvollzogen werden kann, warum, wie und in welcher Reihenfolge automatische Konfiguration durchgeführt wurde. Übrigens nutzt auch Spring seine eigenen Features intensiv: Die „@Profile"-Annotation wurde im Rahmen von Spring 4 neu geschrieben und basiert nun selber auf „Conditions".

Erstes Beispiel: Bereitstellung des Wro4j-Servlet-Filters

Der Kern des Wro4j-Pakets ist der Servlet-Filter, der unter einem bestimmten Mapping Anfragen für CSS- und JavaScript-Dateien entgegennimmt und entsprechend seiner Konfiguration verarbeitet. Durch Hinzufügen des wro4j-spring-boot-starter soll dieser Filter ohne weitere Nacharbeit bereitgestellt werden *(siehe Listing 4)*.

Die „Wro4jAutoConfiguration" ist eine gewöhnliche Spring-„@Configuration"-Klasse, in der eine Bean programmatisch über Java-

```
@Configuration
@ConditionalOnClass(WroFilter.class)
@ConditionalOnMissingBean(WroFilter.class)
@EnableConfigurationProperties(Wro4jProperties.class)
@AutoConfigureAfter(CacheAutoConfiguration.class)
public class Wro4jAutoConfiguration {
}

@ConfigurationProperties("wro4j")
public class Wro4jProperties {
}
```

Listing 4

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration = ac.simons.
spring.boot.wro4j.Wro4jAutoConfiguration
```

Listing 5

```
@Bean
ConfigurableWroFilter wroFilter(WroManagerFactory wroManagerFactory,
Wro4jProperties wro4jProperties) {
     ConfigurableWroFilter wroFilter = new ConfigurableWroFilter();
     wroFilter.setProperties(wroFilterProperties(wro4jProperties));
     wroFilter.setWroManagerFactory(wroManagerFactory);
return wroFilter;
}
```

Listing 6

Code instanziiert und konfiguriert werden kann. Die in ihr vorgenommene Konfiguration soll dann ausgeführt werden, wenn die „WroFilter"-Klasse verfügbar ist und es noch keine Bean entsprechenden Typs gibt. Sie soll nach der automatischen Cache-Konfiguration durchgeführt werden, da diese benötigt wird. Ferner stellt sie alle Eigenschaften, die mit „wro4j.*" beginnen, als Objekt mit entsprechenden Attributen zur Verfügung.

Entsteht der eigene Spring Boot Starter innerhalb des Projektes, in dem er zuerst benutzt wird, übersieht man sehr schnell „META-INF/spring.factories", da alle „@Configuration"-Klassen mit ihren Bedingungen in einem Spring-Boot-Projekt automatisch ausgewertet werden. „spring.factories" ähnelt nicht ganz zufällig dem eingebauten Java Service Provider Interface. Die „Factories"-Definition für den Wro4j-Starter sieht ganz einfach aus *(siehe Listing 5)*.

Diese Aufzählung von Konfigurationsklassen führt beim Laden der Bibliothek zu ihrer Auswertung und vermeidet dadurch sowohl ein manuelles Aufzählen als auch einen vollständigen Scan des Klassenpfads nach „@Component"-Beans. Doch zurück zum Beispiel. Der eigentliche Filter ist nichts anderes als eine normale Spring Bean und wird wie folgt in „Wro4jAutoConfiguration" erzeugt *(siehe Listing 6)*.

Gut erkennbar: Der Filter hängt von einer WroManagerFactory und zusätzlichen Eigenschaften ab, die in diese Methode injiziert werden. Am Beispiel der automatischen Konfiguration zeigt sich die Mäch-

```
@Bean
@ConditionalOnBean(CacheManager.class)
@ConditionalOnProperty("wro4j.cacheName")
@ConditionalOnMissingBean(CacheStrategy.class)
@Order(-100)
<K, V> CacheStrategy<K, V> springCacheStrategy(CacheManager cacheManager, Wro4jProperties wro4jProperties) {
     return new SpringCacheStrategy<K, V>(cacheManager, wro4jProperties.getCacheName());
}

@Bean
@ConditionalOnMissingBean(CacheStrategy.class)
@Order(-90) <K, V> CacheStrategy<K, V> defaultCacheStrategy() {
     return new LruMemoryCacheStrategy<K, V>();
}
```

Listing 7

tigkeit der Java-basierten Konfiguration von Spring Beans: Komplexe Bedingungen lassen sich in Quelltext ausdrücken und mit Werkzeugunterstützung aller aktuellen IDEs nachvollziehen.

Zweites Beispiel: Unterschiedliche Laufzeitumgebung

Wro4j unterstützt Caching von optimierten Ressourcen in Form einer einfachen, speicherbasierten „Least Recently Used Strategy". Eine Implementierung dieser „CacheStrategy" ist schnell geschrieben und mit wenig Code lässt sich „org.springframework.cache.CacheManager" nutzen, sofern zur Laufzeit eine Instanz vorliegt. Das ist zum Beispiel immer dann der Fall, wenn Spring „@EnableCaching" genutzt wird; Spring unterstützt eine Vielzahl von Caches, unter anderem „ehcache" und „Redis". Ist für die jeweilige Applikation kein Cache konfiguriert, wird die standardmäßige Wro4j-Implementierung genutzt *(siehe Listing 7)*.

Die „springCacheStrategy"-Methode wird nur dann ausgeführt, wenn ein „CacheManager" aktiv, der Name des zu nutzenden Cache konfiguriert und noch keine anderweitig konfigurierte Strategie vorhanden ist. Durch „@Order"-Annotation wird sichergestellt, dass die Methode vor „defaultCacheStrategy" ausgewertet wird, die die Default-Implementierung von Wro4j nutzt.

Es gibt sicherlich Stimmen, die dieses Beispiel gerne auf „@Annotatiomania" *[8]* sehen würden, aber gerade zur Konfiguration außerhalb von Geschäftslogik und fachlichen Objekten hält der Autor Annotationen für ein probates und gut lesbares Werkzeug, um Bedingungen zu definieren.

Das Deployment

Das Beispiel ist ein normales Maven-Projekt, in dem ein gemeinsames „autoconfigure"- und „starter"-Modul implementiert wurde. Es bringt daher die Abhängigkeiten zu Wro4j direkt mit *(siehe Listing 8)*.

```
<dependencies>
     <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter</artifactId>
     </dependency>

     <dependency>
          <groupId>ro.isdc.wro4j</groupId>
          <artifactId>wro4j-core</artifactId>
          <version>${wro4j.version}</version>
     </dependency>

     <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <scope>test</scope>
     </dependency>
</dependencies>
```

Listing 8

```
<resources>
     <resource>
          <directory>src/main/resources</directory>
          <filtering>true</filtering>
          <includes>
               <include>wro.xml</include>
          </includes>
     </resource>
</resources>
```

Listing 9

```
wro4j.filterUrl = /owr
wro4j.managerFactory.preProcessors = removeSourceMaps, cssUrlRewriting, cssImport, cssMinJawr, semicolonAppender, jsMin
wro4j.cacheName = wro4j
```

Listing 10

Bei zwei separaten Projekten werden die Abhängigkeiten des „autoconfigure"-Moduls in der Regel als „optional" gekennzeichnet und erst im „starter"-Modul konkretisiert. Die Besonderheit des hier erstellten Starters liegt darin, dass er als Child-Artefakt vom offiziellen „spring-boot-starters"-Projekt erbt, weil der Autor sich bei diesem Projekt an die Kodierungsrichtlinien des Spring-Teams halten wollte. Erleichternd dabei war die Tatsache, dass NetBeans *[9]* Checkstyle-Regeln ebenso direkt unterstützt wie JaCoCo. Der vollständige Quelltext des Projekts ist auf GitHub *[10]* verfügbar.

Fazit

Der hier beschriebene Spring Boot Starter wird in mehreren Projekten produktiv ein-

gesetzt, unter anderem bei der Euregio JUG *[11]*. Durch Einbinden des Starters entfallen das manuelle Einbinden von Abhängigkeiten, zusätzliche Konfiguration und vieles mehr.

Es gibt wenige Alternativen zu Spring Boot Startern im Spring-Umfeld, wenn gemeinsame Module für Anwendungen erstellt werden sollen. Wird Annotations-basierte Konfiguration in diesen Modulen eingesetzt (zum Beispiel „@Service" und Ähnliche), so muss Sorge dafür getragen werden, dass diese Komponenten auch im Klassenpfad gefunden werden. XML-basierte Konfiguration ist aufwändig und oft fehlerträchtig.

Das „spring-boot-starters"-Repository *[12]* listet zahlreiche Starter auf, von Datenbank-Anbindungen über Template-Sprachen bis hin zu Remote Shells ist viel Nützliches dabei. Der Autor nutzt weitere Starter, zum Beispiel zur Bereitstellung von Entitäten und Repositories zur Speicherung von Oauth-Credentials.

Das Ziel des „wro4j-starter" wurde vollständig erfüllt. In *Listing 3* sieht man, dass die „wro.xml"-Konfigurationsdatei Maven-Platzhalter wie zum Beispiel „@jquery.version@" enthält, die während des Build-Vorgangs durch die entsprechenden Maven-Properties ersetzt werden *(siehe Listing 9)*.

Die automatische Konfiguration liest „application.properties" aus *(siehe Listing 10)*. Das Ergebnis ist jeweils eine minimierte CSS- und JavaScript-Datei, die mit einem Statement eingebunden werden kann. Diese Optimierung funktioniert auch problemlos mit AngularJS-SPA-Anwendungen.

Ausblick

Der Starter lässt sich dahingehend erweitern, dass je nach Umgebung die URL der optimierten Ressourcen oder die Informationen aus dem Modell in die Seite generiert werden. Das Vorgehen wäre identisch: Feststellen, ob Unterstützung für Template-Sprachen wie Thymeleaf *[13]* oder andere auf dem Klassenpfad sind, und entsprechende zusätzliche Tags registrieren.

Quellen und Links

[1] Spring Boot: *http://projects.spring.io/spring-boot*
[2] Spring Framework: *https://projects.spring.io/spring-framework*
[3] WebJars: *http://www.webjars.org*
[4] James Ward: *https://twitter.com/_JamesWard*
[5] wro4j: http://alexo.github.io/wro4j
[6] Josh Long: *https://twitter.com/starbuxman/status/708579187087511552*
[7] ConditionEvaluationReport: *http://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/condition/ConditionEvaluationReport.html*
[8] @Annotatiomania: *http://www.annotatiomania.com*
[9] NetBeans: *http://www.netbeans.org*
[10] GitHub: *https://github.com/michael-simons/wro4j-spring-boot-starter*
[11] Euregio JUG: *http://www.euregjug.eu*
[12] spring-boot-starters: *https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters*
[13] Thymeleaf: *http://www.thymeleaf.org*
[14] ENERKO Informatik: *http://www.enerko-informatik.de*
[15] info.michael-simons.eu: *http://info.michael-simons.eu*
[16] The primary goals of Spring Boot: *https://spring.io/blog/2013/08/06/spring-boot-simplifying-spring-for-everyone*
[17] Creating your own starter: *http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-starter*
[18] Universal install script: *http://xkcd.com/1654*
[19] Maven Enforcer Plugin - The Loving Iron Fist of Maven: *http://maven.apache.org/enforcer/maven-enforcer-plugin*

Michael Simons
michael@simons.ac

Michael Simons ist Software-Architekt (CPSA-F) bei ENERKO Informatik [14] in Aachen und entwickelt dort GIS-, EDM- und Vertriebsmanagement-Systeme für Stromnetzbetreiber und Energielieferanten. Michael ist Mitgründer der Euregio JUG [11] und schreibt über ihre Entwicklung und andere Dinge unter info.michael-simons.eu [15].

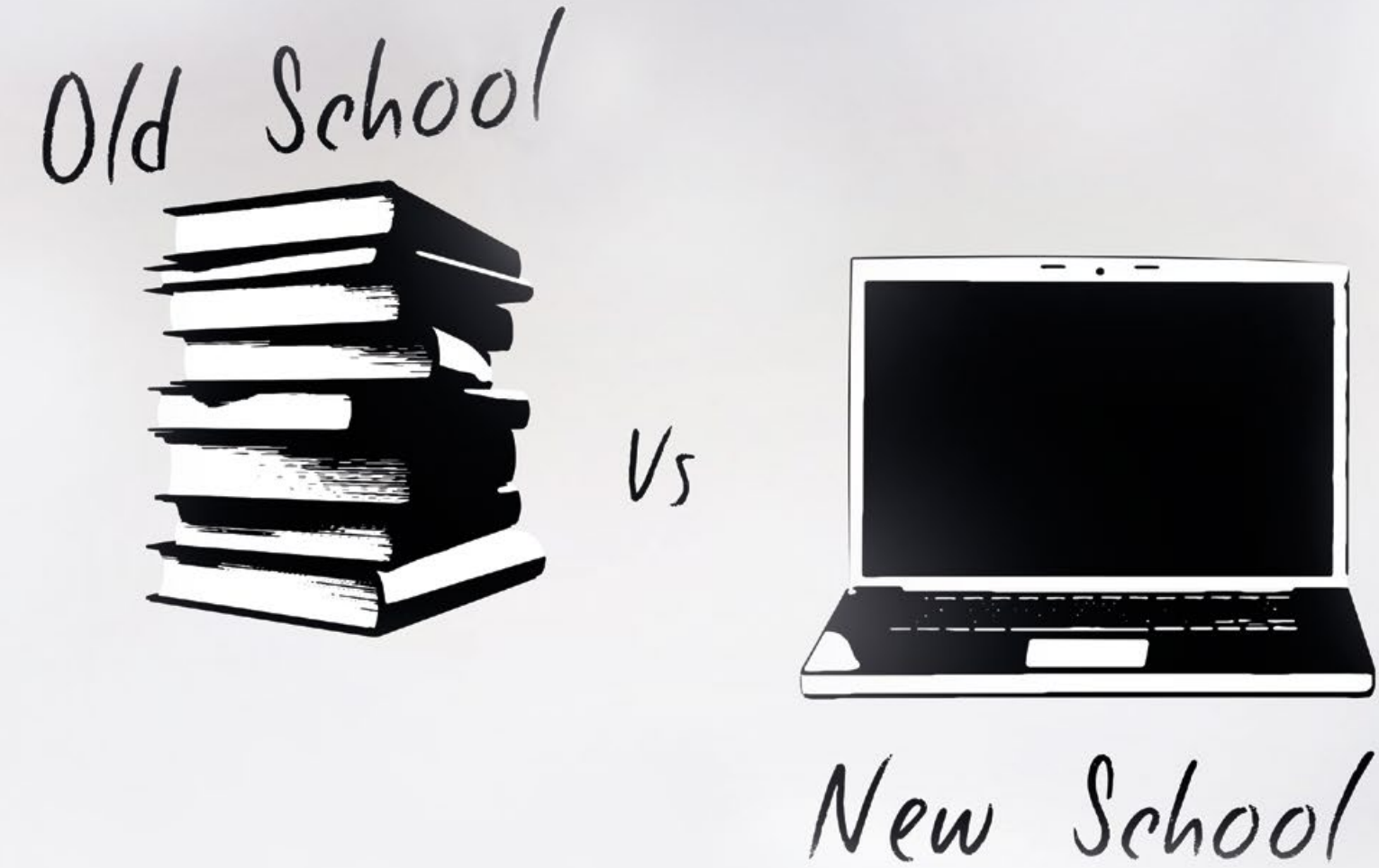
Veranstaltungen der im iJUG organisierten JUGs auf Erfolgskurs

Von Java User Groups organisierte Fachkonferenzen sind voll im Trend und erfreuen sich stark steigender Besucherzahlen. Mit Java Forum Stuttgart, Berlin Expert Days, Java Forum Nord und JavaLand sind die führenden deutschen Java-Konferenzen von der Community organisiert.

Mit einer Rekordbeteiligung von rund 1.700 Teilnehmern fand zum 19. Mal das Java Forum in Stuttgart statt. Die Java User Group Stuttgart hatte 49 Vorträge in sieben parallelen Tracks organisiert. Zudem waren 34 Aussteller vor Ort, darunter auch der Interessenverbund der Java User Groups e.V. (iJUG).

Am 15. und 16. September 2016 waren in Berlin die Berlin Expert Days. Der Verein Berlin Expert Days e.V. wurde im Jahr 2010 mit dem Ziel gegründet, eine Plattform zum Informationsaustausch anzubieten. Als offen geführter Verein steht eine solide Basis bereit, um die Unabhängigkeit von Herstellern und Dienstleistern zu gewährleisten. Auf der diesjährigen Veranstaltung waren mehr als 500 Teilnehmer auf den 44 Vorträgen.

Das Java Forum Nord öffnet am 20. Oktober 2016 seine Pforten. Die eintägige, nicht-kommerzielle Konferenz in Norddeutschland mit Themenschwerpunkt Java ist für Entwickler und Entscheider. Mit mehr als 25 Vorträgen in parallelen Tracks und einer Keynote wird ein vielfältiges Programm zu einem unschlagbaren Preis geboten, der regionale Bezug bietet zudem interessante Networking-Möglichkeiten. Auch das Datum für die JavaLand 2017 steht bereits fest. Sie findet vom 28. bis 30. März 2017 an gewohnter Stätte im Phantasialand Brühl statt. Mit mehr als 1.200 Teilnehmer gehört die JavaLand zu den größten Java-Konferenzen Europas. Für drei Tage wird der Freizeitpark zum Zentrum der Java-Community.



Old school meets hype: Java Swing und MongoDB

Marc Smeets, Technidoo Solutions UG

NoSQL ist im Kommen. Was anfangs noch stellenweise belächelt wurde, hat sich zu einem ernstzunehmenden Konkurrenten des altbekannten Relational Database Management System (RDBMS) gemausert. Häufigere Artikel in Fachmagazinen und die elitären Kundenlisten der Anbieter sprechen eine deutliche Sprache. „NoSQL-Hype" heißt es manchmal, als handele es sich um eine möglicherweise vergängliche Mode-Erscheinung. Dabei warten NoSQL-Datenbanken mit erheblichen Vorteilen auf, etwa mit einer erstaunlichen Performance gegenüber teils schwergewichtigen relationalen Systemen.

Um den Hype einmal auszuprobieren, kommt für diesen Artikel MongoDB zum Einsatz. Um zu sehen, ob auch alte Männer wie der Autor (natives RDBMS-Volk) damit klarkommen, direkt zum Abmildern in Tateinheit mit einer Wohlfühl-„old school"-Technologie das sicherlich den meisten Lesern wohlbekannte Java SE Swing.

MongoDB ist in der Grundversion eine NoSQL-Open-Source-Datenbank und speichert Daten als BSON-Dokumente (binary JSON) innerhalb von Collections. Diese entsprechen in der relationalen Datenbank-Welt Tabellen, die Dokumente den Zeilen darin. MongoDB ist allerdings komplett ohne Schema, und zwar im wahrsten Sinne des Wortes. Es gibt kein Schema/Data Model und man kann jeder Collection x-beliebig strukturierte Dokumente hinzufügen.

Während jede relationale Datenbank das Einfügen verweigert, wenn die Felder nicht übereinstimmen, geht mit MongoDB tatsächlich alles. Kundendaten mit PLZ und Telefonnummer lassen sich wunderbar in eine Rezepte-Sammlung einfügen. Daher ist Vorsicht geboten. Jedes eingefügte Dokument wird seitens MongoDB automatisch um eine alphanumerische ObjectId („_id") erweitert, die keinesfalls Ersatz für eine eigene ID ist oder sein sollte.

Darüber hinaus wird MongoDB komplett objektorientiert angesprochen, was natürlich bedeutet, dass sich der Code hervorragend in Java-Code einbetten lässt. Es gibt keine SQL-Skripte mehr im Code, die schon seit jeher Fremdkörper waren und über „execute"-Methoden ausgeführt werden mussten.

Die Shell

Den ersten, prinzipiellen NoSQL-Einstieg schafft man nach erfolgreicher Installation *[1]* und Start der Datenbank einfach über die Shell. Um alle Dokumente einer Collection, in diesem Falle der Beispiel-Collection „restaurants", zurückzuliefern, bedient man sich der „find()"-Methode. Wohlgemerkt, der Methode. MongoDB funktioniert im Gegensatz zu SQL objektorientiert. Das entspricht dem altbekannten „SELECT * FROM table_name" der relationalen (Skript-)Welt. Um eine vernünftig lesbare Ausgabe statt Spaghetti-JSONs zu bekommen, wird das noch um „pretty()" erweitert *(siehe Listing 1)*.

Hinweis: *Listing 1 und 2 stehen unter http://www.doag.org/go/java_aktuell/201604/listings.*

Die Methode „findOne()" liefert eins, und zwar das erste Dokument in der natürlichen Reihenfolge, also auf der Festplatte, das den Kriterien entspricht. Die Methode „pretty()" wird nicht extra benötigt *(siehe Listing 2)*.

Selbstverständlich kann das „SELECT" eingeschränkt werden, indem die „find()"-Methode um Parameter erweitert wird. Alle Restaurants im Stadtteil Manhattan bekommt man per entsprechenden Parameter *(siehe Listing 3)*.

Eine Einschränkung des Returns wird durch Benennung der Attribute, definiert durch eine „1" für „anzeigen" und eine „0" für „ausschließen", erreicht. „_id" wird grundsätzlich zurückgeliefert, wenn es nicht explizit ausgeschaltet ist *(siehe Listing 4)*.

MongoDB und Java

Nach dieser Grundlagen-Erläuterung bietet sich NetBeans für das Java-Beispiel an. Das MongoDB-Plug-in „NBMongo" lässt sich per Mausklick hinzufügen, die Applikation wird um die Library „mongo-java-driver-3.2.2.jar" erweitert. Selbstverständlich funktioniert alles mit anderen IDEs wie beispielsweise Eclipse („Java MongoDB Plug-in for Eclipse") genauso einfach. Als erster Schritt wird die Verbindung zur Datenquelle, der Collection „restaurants", in der Datenbank „test" eingerichtet *(siehe Listing 5)*. Um ein erstes Dokument abzufragen, kommt die Methode „find()", erweitert um „first()", zum Einsatz, um nur das erste Ergebnis zurückgeliefert zu bekommen *(siehe Listing 6)*. *Listing 7* zeigt die Ausgabe.

```
db.restaurants.find( { "borough": "Manhattan" } )
```

Listing 3

```
> db.restaurants.find( { "borough": "Manhattan" }, { "name": 1 })
{ "_id" : ObjectId("57162828116edd69a12c83c1"), "name" : "Dj Reynolds Pub
And Restaurant" }
{ "_id" : ObjectId("57162828116edd69a12c83cc"), "name" : "1 East 66Th
Street Kitchen" }
{ "_id" : ObjectId("57162828116edd69a12c83d1"), "name" : "Glorious Food" }
{ "_id" : ObjectId("57162828116edd69a12c83d4"), "name" : "Bully'S Deli" }
{ "_id" : ObjectId("57162828116edd69a12c83d6"), "name" : "Harriet'S Kitch-
en" }
{ "_id" : ObjectId("57162828116edd69a12c83d7"), "name" : "P & S Deli Gro-
cery" }
...
```

Listing 4

```
MongoClient client = new MongoClient();
MongoDatabase db = client.getDatabase("test");
MongoCollection<Document> collection = db.getCollection("restaurants");
```

Listing 5

```
Document firstRestaurant = collection.find().first();
System.out.println(firstRestaurant);
```

Listing 6

```
Document{{_id=57162828116edd69a12c83bf, address=Document{{building=1007, coord=[-73.856077, 40.848447],
street=Morris Park Ave, zipcode=10462}}, borough=Bronx, cuisine=Bakery, grades=[Document{{date=Mon Mar 03 01:00:00
CET 2014, grade=A, score=2}}, Document{{date=Wed Sep 11 02:00:00 CEST 2013, grade=A, score=6}}, Document{{date=Thu
Jan 24 01:00:00 CET 2013, grade=A, score=10}}, Document{{date=Wed Nov 23 01:00:00 CET 2011, grade=A, score=9}},
Document{{date=Thu Mar 10 01:00:00 CET 2011, grade=B, score=14}}], restaurant_id=30075445, name=Morris Park Bake
Shop}}
```

Listing 7

Mit Java werden Einschränkungen wie in der Shell nicht über zweite „{}“ erreicht, sondern über entsprechende Methoden. Im folgenden Beispiel wird das Restaurant „The Movable Feast“ gesucht und es sollen nur Cuisine und der Name retourniert werden *(siehe Listing 8)*. *Listing 9* zeigt die Ausgabe.

Eine Einschränkung der Attribute erfolgt über „projection()“, es stehen außerdem auch „sort()“ und weitere zur Verfügung. *Listing 10* zeigt ein Grundgerüst für eine Abfrage.

Über „find()“ werden wie bereits besprochen die Such- und über „projection()“ die Attribut-Parameter mitgegeben. „Sort()“ sortiert auf- oder absteigend nach Attribut, „skip(20)“ übergeht die ersten zwanzig Treffer und „limit(50)“ begrenzt die Ausgabe auf fünfzig Treffer.

```
Bson search = new Document("name", "The Movable Feast");
Bson showNameAndCuisineNotID = new Document("name", 1)
.append("cuisine", 1)
.append("_id", 0);
Document searchResult = collection.find(search)
.projection(showNameAndCuisineNotID)
.first();
System.out.println(searchResult);
```

Listing 8

```
Document{{cuisine=American , name=The Movable Feast}}
```

Listing 9

```
List<Document> results = collection.find()
                .projection()
                .sort()
                .skip()
                .limit()
                .into(new ArrayList<Document>());
```

Listing 10

MongoDB und Java Swing

Java Swing gibt es noch. Tatsächlich. Allem EE und der neumodischen Anforderung zum Trotz, dass sich alles zwingend im Browser abspielen muss, selbst auf Desktop-PCs ausgeführte ERP-Software, die niemals außerhalb des eigenen Netzwerks und sogar Gebäudes genutzt wird. Und es funktioniert wie eh und je, mit den guten alten J...

Um MongoDB mit old school Java Swing zu testen, dient eine altmodische Eingabemaske. „Good ol', good ol'...“. Ein klassischer „Jframe“ mit „JtextField“, „Jlabel“ und „Jbutton“ hilft, um der Restaurant-Collection neue Datensätze hinzuzufügen.

Wie in der Collection üblich, werden die Adressdaten als Sub-Dokument angelegt, daher die Verarbeitung der Daten in zwei Dokumenten, wobei das Adress-Dokument („newRestaurantJSONsubAddress“) in das Restaurant-Dokument („newRestaurantJSON“) eingefügt wird. Statt eines „INSERT INTO“-SQL-Skripts bedient man sich der „insertOne()“-Methode der Collection. *Listing 11* zeigt den erforderlichen Code, *Abbildung 1* die Eingabemaske und die Abfrage des neuen Beispiel-Datensatzes in der Datenbank, zur Kontrolle per Shell nach der „restaurant_id“-Abfrage.

Der möglicherweise besseren Lesbarkeit wegen zeigt *Listing 12* die Abfrage des neu über die Swing-Maske hinzugefügten Datensatzes per Shell. Wie zu sehen ist, hat Mongo-DB wieder zusätzlich zur „restaurant_id“ die „ObjectId“ („_id“) hinzugefügt. Gibt man eine eigene „_id“ mit, wird diese nicht überschrieben. Des gerade eingefüg-

```
private void b_SaveActionPerformed(java.awt.event.ActionEvent evt) {

MongoClient client = new MongoClient();
MongoDatabase db = client.getDatabase("test");
MongoCollection<Document> collection = db.getCollection("restaurants");

String inputRestaurantID = tf_Restaurant_ID.getText();
String inputName = tf_Name.getText();
     String inputCuisine = tf_Cuisine.getText();
     String inputBorough = tf_Borough.getText();
     String inputBuilding = tf_Building.getText();
     String inputStreet = tf_Street.getText();
     String inputZipcode = tf_Zipcode.getText();

Document newRestaurantJSON = new Document("restaurant_id", inputRestaurantID)
.append("name", inputName)
     .append("cuisine", inputCuisine)
     .append("borough", inputBorough);

Document newRestaurantJSONsubAddress = new Document("building", inputBuilding)
    .append("street", inputStreet)
    .append("zipcode", inputZipcode);

newRestaurantJSON.put("address",
newRestaurantJSONsubAddress);

     collection.insertOne(newRestaurantJSON);
```

Listing 11

ten Test-Datensatzes entledigt man sich einfach mittels „remove()"-Methode der Collection *(siehe Listing 13)*.

Fazit

NoSQL-Datenbanken, hier beispielhaft durch MongoDB vorgestellt, sind eine ernsthafte Konkurrenz zu altbekannten RDBMS und warten vor allem mit Performance-Vorteilen auf, allerdings auch mit Redundanz durch die Speicherung in Dokumenten. Hinsichtlich der Einsatzfähigkeit gibt es keine Einschränkung, NoSQL funktioniert im IoT-Umfeld wie auch im Rahmen von Java EE oder, wie hier aufgezeigt, in Swing-Applikationen.

Weitere Informationen

[1] *https://docs.mongodb.com/manual/installation*

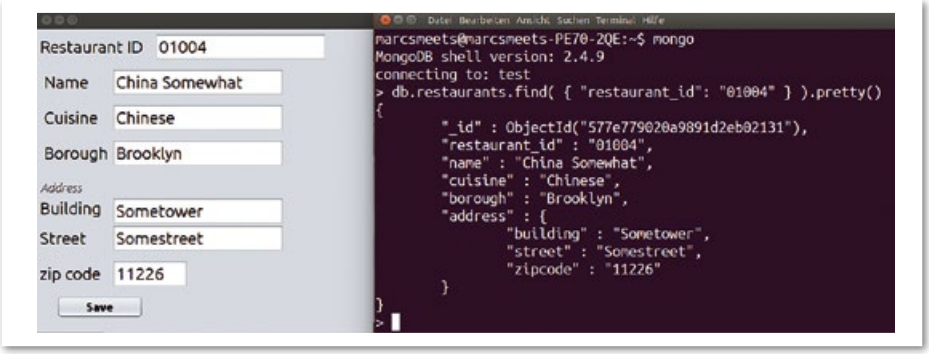


Abbildung 1: Eingabemaske und Shell-Test

```
> db.restaurants.find( { "restaurant_id": "01004" } ).pretty()
{
  "_id" : ObjectId("577e779020a9891d2eb02131"),
  "restaurant_id" : "01004",
  "name" : "China Somewhat",
  "cuisine" : "Chinese",
  "borough" : "Brooklyn",
  "address" : {
     "building" : "Sometower",
     "street" : "Somestreet",
     "zipcode" : "11226"
  }
}
```

Listing 12

```
collection.remove("restaurant_id", "01004");
```

Listing 13

Marc Smeets
marc.smeets.java@gmail.com

Marc Smeets ist Consultant/Solution-Architect mit den Schwerpunkten Oracle ADF, Java, Android und MongoDB. Zusätzlich kann er Projekte als zertifizierter Scrum-Master (scrum.org) begleiten.

iJUG vereint bereits 30 Java User Groups

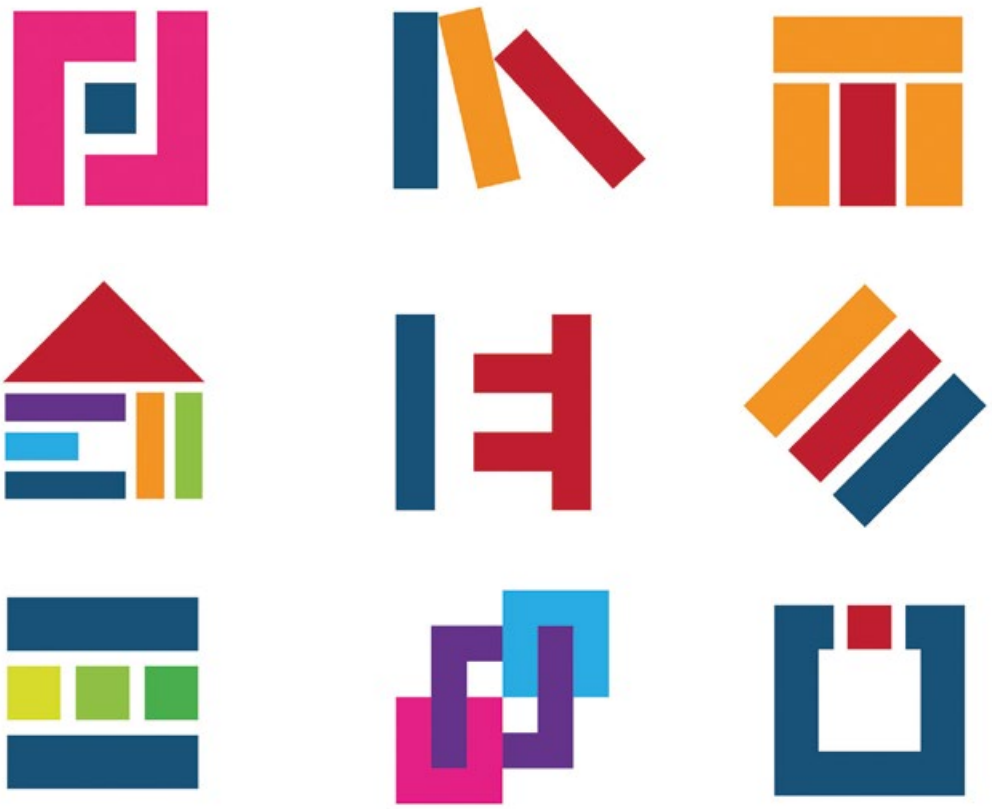
Nach der Übernahme von Sun durch Oracle im Jahr 2009 haben sieben Anwendergruppen den iJUG gegründet. Der Interessenverbund ist auf mittlerweile 30 Mitglieder (siehe Seite 66) gewachsen und vertritt damit mehr als 20.000 Entwickler aus Deutschland, Österreich und der Schweiz. Ziel ist die umfassende Vertretung der gemeinsamen Interessen der Java User Groups sowie der Java-Anwender im deutschsprachigen Raum.

Ein großer Erfolg des iJUG ist die Durchführung der JavaLand-Konferenz, die bereits im dritten Jahr im Phantasialand in Brühl stattfindet und mittlerweile zu den größten Java-Konferenzen in Europa zählt. Hinzu kommt die Herausgabe der Zeitschrift Java aktuell, die bereits nach einem Jahr einen führenden Platz unter den deutschsprachigen Java-Magazinen erreicht hat.

Durch regelmäßige Öffentlichkeitsarbeit und in zahlreichen Gesprächen mit Vertretern von Oracle konnte der iJUG etliche Verbesserungen für die Java-Community erzielen. Themen waren unter anderem Sicherheitslücken in Java, Probleme beim Java-Update sowie Aufforderungen an Oracle hinsichtlich der Weiterentwicklung von Java FX und Java EE.

Ein weiteres Angebot des iJUG besteht darin, Interessenten bei der Gründung einer Java User Group zu unterstützen sowie bestehenden Gruppen Referenten für Veranstaltungen zu vermitteln.

„Der Erfolg zeigt, dass sich der Zusammenschluss der User Groups im iJUG gelohnt hat", sagt Fried Saacke, Vorstandsvorsitzender des iJUG. „In diesem Sinne werden wir unsere Aktivitäten in den kommenden Jahren weiter fortführen und ausweiten."



REST-Architekturen erstellen und dokumentieren

Martin Walter

REST-Architekturen sind seit Jahren auf dem Vormarsch und nicht zuletzt das Thema „Microservices" hat dies noch einmal verstärkt. Doch nicht nur das Erstellen von REST-basierten Diensten bietet eine breite Diskussionsgrundlage, auch deren Spezifikation und Dokumentation. Im immer ähnlichen Aufbau der Dienste steckt hierfür enorm viel Potenzial. Genau hier setzen diverse Tools und Frameworks an, die dieser Artikel näher beleuchtet.

Schaut man sich REST-Services von früher an, läuft es einem oft kalt den Rücken herunter. In URLs werden oft Substantive und Verben gemischt, mal sind Ressourcen im Singular, mal im Plural definiert. Eine Dokumentation sucht man vergebens – es war ja nur wenig Funktionalität umzusetzen, die sich fast von selbst aus der XML-Antwort des Servers erschließt. Die Spezifikation kann man im besten Fall zerstückelt in Anforderungspaketen aus dem Ticket-System wiederherstellen. Doch wie schafft man es, Spezifikation und Dokumentation in ausreichender Qualität zu erstellen?

Der Schlüssel zum Erfolg liegt im prinzipiell immer gleichen Aufbau von REST-Services. Es werden Ressourcen angeboten, von denen Informations-Entitäten abgefragt oder auf denen Änderungen durchgeführt werden können, letztlich das klassische CRUD (create read update delete). Diese recht einfache Arbeitsweise, mit der sich wiederum aber sehr komplexe Abläufe steuern lassen, ermöglicht uns einheitliche Spezifikationen und auch einheitliche Dokumentationen. Die Frage, die einen Entwickler umtreibt, ist nun, wie man den größten nachhaltigen Nutzen bei geringstem Aufwand erhält.

Als Beispiel soll ein minimaler REST-Service dienen, der Planeten und Monde in einer Galaxie verwaltet. Die Planeten besitzen jeweils eine ID, einen Namen, einen (maximalen) Durchmesser und eine Liste von Hauptmonden. Die Monde wiederum besitzen Ids, Namen und Umkreisungszeiten. *Listing 1* zeigt vorab die fertigen URLs inklusive Parametern.

Zu dokumentieren gibt es die Übersicht über alle Ressourcen (in diesem Fall nur

eine), deren Methoden inklusive Parametern und Rückgabewerten, die Konsequenzen, die aus einem Aufruf entstehen, und letztlich optional auch Beispiel-Requests und -Responses. Im Idealfall bietet man darüber hinaus noch die Möglichkeit an, den Service direkt auszuprobieren, oder verweist auf eine „Spielwiese", auf der man ihn ohne Gefahr für Live-Systeme testen kann. Im Folgenden werden verschiedene Klassen von Werkzeugen vorgestellt, mit denen sich unser Beispiel dokumentieren (beziehungsweise erstellen) ließe.

Code-Generatoren

Die erste Klasse von Werkzeugen, mit denen der Service erstellt werden kann, sind Code- und Dokumentations-Generatoren. Die Funktionsweise ist denkbar einfach. In einer abstrakten Notation wird der REST-Service beschrieben. Diese Beschreibung wird vom Generator eingelesen und verarbeitet, je nach Fähigkeiten kann daraus anschließend Java- oder beliebiger anderer Code generiert werden. Ein Beispiel für einen solchen Generator bietet das Projekt „apiblueprint.org", das Wiki-Syntax verwendet und auf dem weitere Tools und Frameworks aufsetzen. Die Definition, um einen Planeten abzurufen, könnte wie in *Listing 2* aussehen.

Für geübte Wiki-Schreiber ist die Syntax vermutlich schnell erlernt, für alle anderen erscheint sie eher kontra-intuitiv. Gute Generatoren geben beim Schreiben bereits Live-Previews der Dokumentation, ermöglichen gemeinsames Arbeiten über das Netz und erzeugen aus den Beispieldaten einen Mock-Server. Für grundlegend neue Services mag dies praktisch sein, in der Realität wird man jedoch schnell das Problem haben, bestehenden Code importieren zu wollen, was leider nicht möglich ist. Alte Projekte bleiben also in ihrem Quellcode gefangen und müssten in Wiki-Syntax neu spezifiziert werden. Ein weiterer Kritikpunkt ist, dass es kein Meta-Modell gibt, in das zunächst transformiert wird und in das man beispielsweise Legacy-Code portieren könnte.

Doclets

Legt man nicht so viel Wert auf die Unterstützung bei der Erstellung von REST-Services, helfen dem geneigten Java-Entwickler Doclets weiter. Diese Javadoc-Erweiterungen generieren aus bestehendem Quellcode heraus ohne zusätzliche Annotationen oder spezielle Kommentare zusätzliche Dokumentation. Das SpringDoclet *(siehe „http://scottfrederick.github.io/springdoclet")* erzeugt zum Beispiel aus Spring-Annotationen eine Übersicht über die Ressourcen und Methoden mit den kurzen Beschreibungen aus Quellcode-Kommentaren und verweist dann auf die entsprechende Javadoc. Für unser Beispiel reicht es also völlig, wenn man den sowieso erforderlichen Code mit Annotationen schreibt und mit Kommentaren versieht *(siehe Listing 3)*.

```
POST /planet/ erstellt einen neuen Planeten,
GET /planet/{id} ruft einen Planeten ab
GET /planet/{id}/moons liefert die Monde eines Planeten
PUT /planet/{id} verändert einen Planeten
DELETE /planet/{id} löscht einen Planeten
```

Listing 1

```
# Planet Service
This is the planet service...

## Planet [/planet/{id}]
+ Parameters
    + id: 1 (required, number) - ID of the planet
+ Response 200 (application/json)
{"name":"Merkur", ...}
```

Listing 2

```
/** The planet resource that provides... **/
@Controller
public class PlanetController {

/** Serves a planet with a given id... **/
@RequestMapping("/planet/{id}")
```

Listing 3

Man muss nur ein kleines Tool beim Bauen der Anwendung automatisch mitlaufen lassen und erhält eine sehr reduzierte Übersicht, die jedoch gerade bei größeren Anwendungen an ihre Grenzen stößt. Diese Helferlein sind eher für Legacy-Produkte brauchbar, in die nicht mehr viel Arbeit gesteckt werden soll. Voraussetzung ist, dass diese im Quellcode zumindest auf Klassen- und Methodenebene ausreichend dokumentiert sind.

Inline-Kommentar-Analyse

Parallel zu den vorangegangen beschriebenen Doclets, die mit Javadoc mitlaufen, gibt es eigenständige Tools, die von der Funktionsweise her sehr ähnlich sind. Es werden spezielle Tags in Quellcode-Kommentare eingebaut, die vom Tool ausgewertet und in eine hübsche HTML-Dokumentation verpackt werden. Der Vorteil liegt darin, dass man in Kommentaren grundsätzlich frei in der Wahl der Syntax ist und nicht invasiv in den normalen Quellcode eingreifen muss. Der wohl größte Vertreter ist „apidoc.js" *(siehe „http://apidocjs.com")*, der hier beispielhaft näher beschrieben werden soll.

Über die Tags hinaus bietet „apidoc.js" noch weitere Features. Legt man Wert auf einheitliches Layout, so kann man eigene HTML-Templates für die Erzeugung der Dokumentation verwenden und somit das eigene Corporate Design gerade bei öffentlichen Schnittstellen einfach wahren. Unser Beispiel würden wir initial wie in *Listing 4* kommentieren.

Mit „@api" wird die Methode beschrieben, mit „@apiName" der Name festgelegt. Über „@apiGroup" lassen sich mehrere Methoden gruppieren, etwa alle, die zu einer Ressource gehören. „@apiParam" beschreibt Parameter mit Typ und Name. Darüber hinaus lassen sich HTTP-Fehlerantworten, Zugriffsrechte, Beispiel-Requests und deren Antworten einrichten. Um sich etwa bei Fehler-Antworten nicht überall wiederholen zu müssen, bietet „apidoc.js" die Tags „@apiDefine" und „@apiUse" an. „@apiDefine" beschreibt einen festen Block *(siehe Listing 5)*, der anschließend an verschiedenen Stellen wiederverwendet werden kann *(siehe Listing 6)*.

Wer Wert auf gut versionierte Schnittstellen legt, kann dies mit „apidoc.js" auch angemessen bewerkstelligen. Das Tool legt beim Erstellen eine History-Datei an, sodass

```
/**
* @api {get} /planet/:id
* @apiName getPlanet
* @apiGroup Planet
* @apiParam {Number} id The id of the planet.
**/
@RequestMapping("/planet/{id}")
```

Listing 4

```
/**
* @apiDefine NotFoundError
* @apiError PlanetNotFound The <code>id</code> of the planet could not be
found.
**/
```

Listing 5

sich bei Nutzung von „@apiVersion" mit Major, Minor und Patch-Version erfassen lässt, was sich von Version zu Version verändert hat. In der erzeugten Dokumentation kann man nun zwei beliebige Versionen einer Methode vergleichen und sehen, welche Parameter beispielsweise hinzugekommen oder weggefallen sind. Alle diese Informationen liefert „apidoc.js" von sich aus mit, wenn das Versions-Tag immer mit gepflegt wurde. Sind Front- und Backend-Entwicklung im Unternehmen voneinander getrennt, bietet diese Funktionalität einen enormen Vorteil und erspart aufwändig geführte, separate Change-Logs und Support-Anfragen.

Reicht einem die Funktionalität nicht aus, kann man das Tool um eigene Tags erweitern. Es lassen sich eigene Parser hinzufügen, die die Auftrennung der Daten vornehmen. Ebenso Worker, die letztlich die Tags auswerten, und Filter, die für das Reduzieren auf die notwendigen Informationen verantwortlich sind.

Besonders die Sprach-Unabhängigkeit durch Kommentar-Analyse und das gut nutzbare HTML-Erzeugnis inklusive Historie sind die Vorteile von „apidoc.js".

Der Vorteil der Kommentare ist allerdings gleichzeitig auch ein Nachteil. Ändert sich nämlich der Code, ist der Entwickler auch für das Updaten der Kommentare verantwortlich. Ebenfalls können Code-Formatter je nach Vorlage die Kommentare so ruinieren, dass „apidoc.js" sie nicht mehr versteht. Nutzt man andere Tools in Kombination, beispielsweise Checkstyle, kommt man dort um spezielle Regeln nicht herum. Denn etwa das Prüfen auf Vorhandensein von Javadoc führt sonst oftmals zu doppelter Dokumentation, die nicht nötig wäre, und bedeutet für den Entwickler, dass er zwei Dokumentationen parallel pflegen muss. Das Chaos, wenn die Javadoc-Beschreibung von der für „apidoc.js" abweicht, ist bei guter Arbeitsauslastung vorprogrammiert.

Meta-Modell als Grundlage

Eine Möglichkeit besteht darin, die Annotationen als Tags in Kommentaren zu verstecken. Andere Tools verfolgen einen invasiveren Ansatz. Das bekannteste Werkzeug ist Swagger *(siehe „http://swagger.io")*, das ein kleines Ökosystem für sich darstellt. Ziel von Swagger war zunächst die Definition einer abstrakten REST-Repräsentation.

Ein Service lässt sich in JSON-Notation unabhängig von der Programmiersprache definieren, in der er entwickelt wird. Auf Basis dieses sprachneutralen Meta-Modells lassen sich einerseits Tools entwickeln, die für die Übersetzung beispielsweise von Java-Code hin zu diesem Modell verantwortlich sind, andererseits auch einheitliche Tools, die mit diesem Meta-Modell arbeiten und etwa die Dokumentation erzeugen. Für Letzteres wird Swagger-UI angeboten, ein Paket bestehend aus HTML, CSS und JavaScript, das auf jedem Webserver läuft.

In der Oberfläche kann man den Pfad zum Meta-Modell JSON angeben, das zerteilt und als interaktive HTML-Dokumentation angezeigt wird. „Interaktiv" bedeutet hier auch tatsächlich, dass man direkt aus der Dokumentation heraus die beschriebene Schnittstelle komfortabel mit eigenen Daten befeuern kann und reale Antworten erhält. Es gilt lediglich zu beachten, dass das Bereitstellen des Meta-Modells nicht auf Produktions-Servern geschehen sollte *(siehe Abbildung 1)*.

Für Spring-REST-Services und viele weitere gibt es bereits Anbindungen zum Generieren des Meta-Modells. Über entsprechende Annotationen lässt sich zur Laufzeit das Meta-Modell unter einer konfigurierbaren URL abrufen. Im Code ist dazu das Annotieren der Methoden notwendig *(siehe Listing 7)*.

Mit Version 2 bietet Swagger nun auch einen Spezifikations-Editor, der aus YAML ein entsprechendes Code-Gerüst für Server und auch Client generieren kann. Nachteilig ist hier wie bei „apidoc.js", dass man gegebenenfalls wieder doppelt dokumentieren oder Tools wie Checkstyle entsprechend umkonfigurieren muss und auch hier Dokumentation und Quellcode auseinanderlaufen können.

Die Do-it-yourself-Methode

Ist man früh auf den REST-Zug aufgesprungen, hat man sicherlich einen großen Legacy-Bestand. Ist dieser in einem Format, für das es kein passendes Tool gibt, so bleibt einem nur die händische Dokumentation oder die Do-it-

```
/**
* @api {get} /planet/:id
* @apiUse NotFoundError
* …
**/
```

Listing 6

```
@ApiOperation(value = "Retrieves a planet",
    response = Planet.class,
    notes = "Retrieves a planet with a given id.")
@ApiResponses({
    @ApiResponse(code = 200, message = "Returned planet.",
        response = Planet.class),
    @ApiResponse(code = 404, message = "Planet not found.",
        response = RestError.class)
    })
@RequestMapping(value = "/planet/{id}")
```

Listing 7

Abbildung 1: Die Swagger-UI

yourself-Methode. Mittels Java Parser *(siehe „https://github.com/javaparser/javaparser")* lässt sich der eigene Quellcode analysieren. So kann man beispielsweise die Kommentare zu Methoden mit bestimmten Annotationen auslesen und ins Swagger-Format übersetzen oder mit „apidoc.js"-Tags anreichern.

Der Aufwand für einen solchen Spezial-Konverter liegt ungefähr bei einer Woche und man kann sich überlegen, ob die manuelle Methode schneller oder langsamer ist. Bei mehreren Legacy-Projekten bietet der Do-it-yourself-Ansatz durchaus eine gangbare Alternative. Allerdings sollte die Konverter-Logik im Vorhinein genau betrachtet werden.

Prototyping oder Vorab-Spezifikation?

Einfluss auf die Wahl des besten Tools hat natürlich auch der Entwicklungs-Ansatz. Beginnt man mit einer Spezifikation und programmiert erst los, wenn diese abgenommen ist, oder verfolgt man den Prototyping-Ansatz und spezifiziert/dokumentiert on-the-fly? Gerade im Bereich der REST-Architekturen haben Entwickler oft ein Gespür dafür, ob ein Service aus einem Guss ist oder konzeptionelle Ungereimtheiten aufweist.

Die Frage, ob sich ein Service gut anfühlt, lässt sich beim Prototyping deutlich einfacher beantworten – besonders wenn mehrere Parteien beteiligt sind. Hier sollte man jedoch klare Regeln für die Benennung von Ressourcen, Methoden und Eigenschaften aufstellen, die sonst zwischen den Code-Artefakten uneinheitlich werden können.

Der Umfang des Projekts spielt natürlich auch eine Rolle. Als Faustregel kann man sich merken: Je kleiner ein Service ist, desto besser funktioniert der Prototyping-Ansatz. Für größere Projekte lässt sich jedoch auch eine Kombination finden, bei der Durchstiche prototypisch entwickelt und die Rückschlüsse dann in einer Vorab-Spezifikation niedergeschrieben werden. Unsere Beobachtung bei der Deutschen Welle ist, dass bei einer Vorab-Spezifikation meist eine etwas höhere Dokumentationsqualität entsteht, was aber auch beim gemischten Ansatz der Fall ist.

Fazit

Zum Erstellen der REST-Services gibt es zahlreiche Frameworks und Tools. Betrachtet man jedoch gute Dokumentation als gewünschtes Feature, dünnt sich der Markt sehr schnell aus – Dokumentation hat nun mal bedauerlicherweise sehr oft einen schlechteren Stand als spannende neue Features. Je nach Arbeitsweise und Workflow lässt sich mit abstrakten Spezifikationen arbeiten, aus denen später ganz einfach der initiale Quellcode generiert wird. Existiert bereits Legacy-Code, führt oftmals kein Weg daran vorbei, sich mit spärlicher Dokumentation zufrieden zu geben oder mit nicht unerheblichem Aufwand Tool-spezifische Anweisungen in den Code zu integrieren.

Für schnelles Dokumentieren ohne viel Einarbeitung und Konfiguration ist „apidoc.js" sicher ein guter Kandidat, auch wenn es vom Feature-Umfang her mit Swagger nicht mithalten kann. Letzteres ist mit seiner Interaktivität besonders interessant, wenn man Entwicklern eine Dokumentation „zum Anfassen" zur Verfügung stellen möchte. Der seit Version 2 angebotene Editor ermöglicht wahlweise auch bereits eine Vorab-Spezifikation.

Hat man bereits Legacy-Code, kommt man momentan meist um manuelles Nachtragen der Dokumentation nicht herum. Es gibt kein einfaches und ausgereiftes Tool, das den vorhandenen Code analysiert und daraus eine Dokumentation oder abstrakte Spezifikation erstellen kann.

Kommerzielle Anbieter im REST-Bereich versprechen viel, jedoch sind die Ergebnisse oftmals weit unter dem, was Swagger & Co. liefern. Um das passende Tool für sich und das eigene Unternehmen zu finden, sollte man sich die eigenen Kriterien nochmal vor Augen halten, die entsprechende Tool-Klasse wählen und diese idealerweise an einem kleinen Beispielprojekt ausprobieren.

Hinweis: Die Source-Codes für diesen Artikel sind unter *http://www.doag.org/go/java_aktuell/Martin_Walter/Walter_Source* hochgeladen.

Martin Walter
martin.walter@martoeng.com

Nach dem Studium der Medieninformatik an der Universität Ulm und der freiberuflichen Tätigkeit als Web-Entwickler und Consultant war Martin Walter Software-Architekt bei der Deutschen Welle, einem der großen internationalen Broadcaster.

BPM macht Spaß!

Bernd Rücker, Camunda Services GmbH

Viele Entwicklungsprojekte setzen fachliche Anforderungen rund um Workflows oder Geschäftsregeln um. Dabei scheuen Entwickler gerne BPM-Werkzeuge, da sie als Fremdkörper und Störfaktor wahrgenommen werden (was leider für viele Produkte auch zutrifft). Das ist sehr schade, da es heute entwicklerfreundliche Open-Source-Alternativen gibt, die keinen Schmerz bereiten, sondern viel Spaß machen.

Standards wie BPMN, CMMN und DMN verbreiten sich weltweit rasant und stiften einen Nutzen, den sich Java-Entwickler nicht entgehen lassen sollten. Dieser Artikel beschreibt ein Anwendungsbeispiel, das mithilfe der drei genannten Standards auf der Open-Source-Plattform Camunda BPM umgesetzt wird.

Das fachliche Beispiel ist ein Versicherungsneuantrags-Prozess. Die *Tabelle 1* weiter unten listet weitere mögliche Anwendungsfälle auf. Die Steuerung des Prozesses wird über eine Process Engine automatisiert, die dann Services aufrufen und den Menschen über Aufgabenlisten einbeziehen kann. *Abbildung 1* zeigt einen vereinfachten Versicherungsneuantrag in Business Process Model and Notation 2.0 (BPMN), dem weltweiten ISO-Standard zur Prozessmodellierung und Automatisierung.

Nach Eingang des Antrags wird zuerst ein Regelwerk aufgerufen, das eine Risikobewertung durchführt. Auf dieser Grundlage wird entschieden, ob direkt vollautomatisiert eine Police ausgestellt oder der Antrag abgelehnt wird. Im Zweifelsfall wird ein Mensch in die Entscheidung involviert. Sollte der Mensch dieser Aufgabe nicht nachkommen, wird er alle zwei Tage daran von seinem Vorgesetzten erinnert, was ebenfalls modelliert ist. Auch die Aufrufe an das Bestandssystem sowie die Schreiben an Kunden sind zu sehen.

Geschäftsregeln mit DMN

Decision Model and Notation (DMN) ist ein sehr junger Standard für das „Decision Management". Es gibt bereits seit längerer Zeit diverse „Business Rule"-Lösungen, die aber

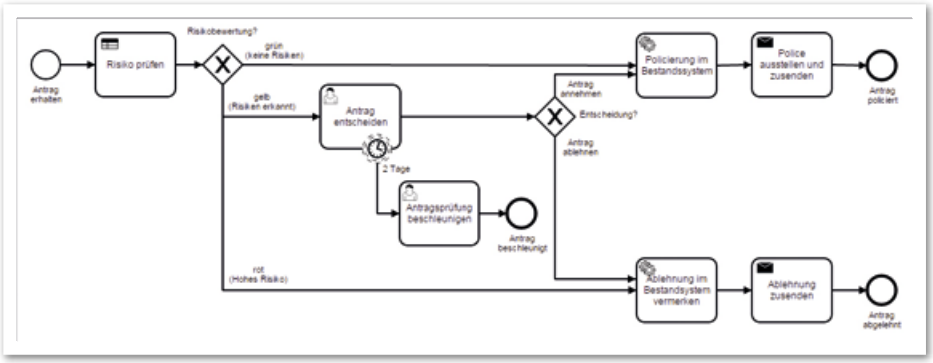


Abbildung 1: Der Versicherungsneuantrags-Prozess in BPMN

Risikoprüfung

risikopruefung

-	Input +			Output +		
	Alter	Fahrzeughersteller	Fahrzeugtyp	Risiko	Risikobewertung	
	neuantrag.antrags...	neuantrag.fahrzeu...	neuantrag.fahrzeu...	risiko	risikobewertung	
	integer	string	string	string	(gelb, rot)	Annotation
1	<= 21	-	-	Anfänger	gelb	-
2	<= 30	BMW	-	Jung und schnell	gelb	-
3	-	Porsche	911	Sinnlose Raserei	gelb	-
4	-	BMW	X3	Hochwertfahrzeug	gelb	-
5	<= 25	Porsche	911	Jung und zu schnell	rot	-

Abbildung 2: Entscheidungstabelle in DMN

alle proprietäre Regelwerke verwenden. Dies wurde mit DMN standardisiert. Für die Automatisierung sind vor allem Entscheidungstabellen wie die in *Abbildung 2* gezeigte Risikoprüfung relevant. Die grauen Zellen enthalten das technische Binding, damit die Tabelle ausführbar ist. Wenn die Tabelle wie hier aus einem Prozess heraus aufgerufen wird, können in Camunda alle Prozess-Variablen verwendet werden. Die Ergebnisse des Regelwerks können wiederum im Prozess zum Routing benutzt werden.

Alternativ kann man die Decision Engine natürlich auch komplett unabhängig von einem BPMN-Prozess verwenden. Dazu übergibt man einfach die Eingabedaten und bekommt ein Ergebnis zurück. Die DMN Engine kann auch ohne Container oder Datenbank verwendet werden. Dann ist es eine schlanke Library, die in jede beliebige Anwendung integriert werden kann.

Werden die grauen Zellen ausgeblendet, hat man wie in BPMN eine wunderbar fachbereichstaugliche Sicht. Oft werden die Regelinhalte (= Zeilen) durch Fachbereiche selbst gepflegt, nachdem die IT die Struktur (= Spalten + Binding) erstellt hat.

Case Management mit CMMN

Der zuvor gezeigte Prozess hat eine klare Reihenfolge. In BPMN muss man zwingend eine Reihenfolge vergeben. Es gibt allerdings Situationen, in denen diese Reihenfolge fachlich nicht sinnvoll vorab definiert werden kann, weil erst zur Laufzeit entschieden werden soll, welche Aktivitäten nötig sind. Solche Szenarien können nur begrenzt mit BPMN abgebildet werden, denn es entsteht

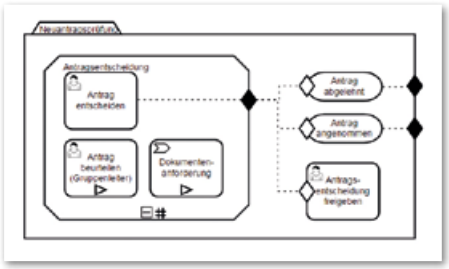


Abbildung 3: Case-Definition in CMMN

schnell ein sogenanntes „Schnittmuster", bei dem jede Aktivität mit jeder anderen verbunden wird. Daher gibt es einen weiteren Standard im Bunde: Case Management Model and Notation (CMMN).

In unserem Beispiel könnte die manuelle Antragsentscheidung auch ein (zugegeben sehr einfacher) Case sein, wie in *Abbildung 3* dargestellt. CMMN kennt ähnliche Aktivitäten wie BPMN, definiert jedoch keine Reihenfolge, stattdessen können bestimmte Attribute und Bedingungen konfiguriert werden:

- „Antrag beurteilen" hat ein kleines Play-Symbol, welches ausdrückt, dass der Mensch diese Aufgabe erst selbst starten muss („Play" drücken). Die Aufgabe landet also erst in der Aufgabenliste des Gruppenleiters, wenn der Sachbearbeiter entscheidet, dass er ihn hinzuziehen möchte. Um dies besser zu verstehen, lohnt ein Blick auf das Formular in *Abbildung 4*.

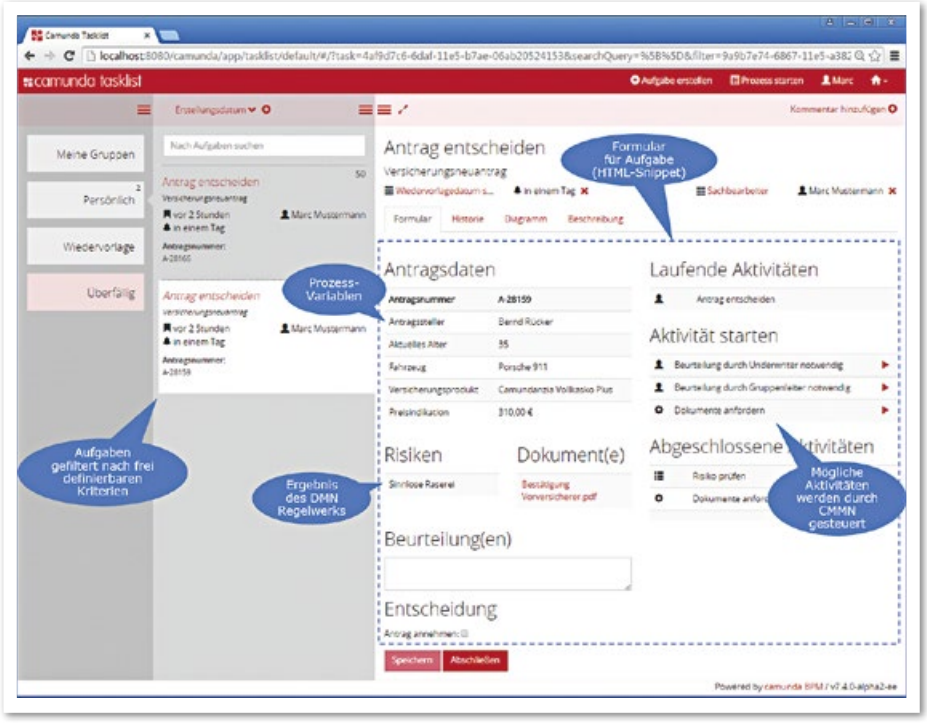


Abbildung 4: Aufgabenliste inklusive Formular für die Antragsentscheidung mit möglichen Aktivitäten aus CMMN

	Business Process Model and Notation (BPMN)	Case Management Model and Notation (CMMN)	Decision Model and Notation (DMN)
Worum geht's?	Workflows/strukturierte Geschäftsprozesse	Fallbearbeitung/unstrukturierte Prozesse	Regelbasierte Entscheidungen/ Business Rules
Zielgruppe	Fachanwender, Business-Analysten, Software-Entwickler	Fachanwender, Business-Analysten, Software-Entwickler	Fachanwender, Business-Analysten, Software-Entwickler
Wann nehmen?	Für eher strukturierte (Teil-)Prozesse mit potenziell hohem Automatisierungsgrad	Für eher unstrukturierte (Teil-)Prozesse mit relativ geringem Automatisierungsgrad	Für die automatisierte Entscheidungsfindung auf Basis strukturierter Regelwerke
Aktuelle Version	2.0	1.1	1.1
Standard seit (OMG/ISO)	2005 / 2014	2014 / -	2015 / -
Grafische Modellierung	✓	✓	✓
XML-Austauschformat	✓	✓	✓
Ausführbar	✓	✓	✓
Typische Beispiele	Bestellung oder Antragseingang (wie DSL-Neuanschluss, KFZ-Versicherung, Kreditkarte, Kleider); Bearbeitung einer Schadensmeldung (Haftpflicht, Autounfall); Lieferung (etwa Paket) etc.	Manuelle Prüfung eines Antrags (wie private Krankenversicherung, gewerbliche Gebäudeversicherung, Patentantrag); Bearbeitung von Ausnahmefällen (wie technische Störungen bei DSL-Schaltung, Verlust des Pakets während des Transports) etc.	Automatische Prüfung eines Antrags (wie Risikoprüfung bei KFZ-Versicherung); automatische Entscheidung (wie Bonitätsbewertung, Ablehnungsgründe für Bestellung); Preis und Rabattfindung etc.

Tabelle 1

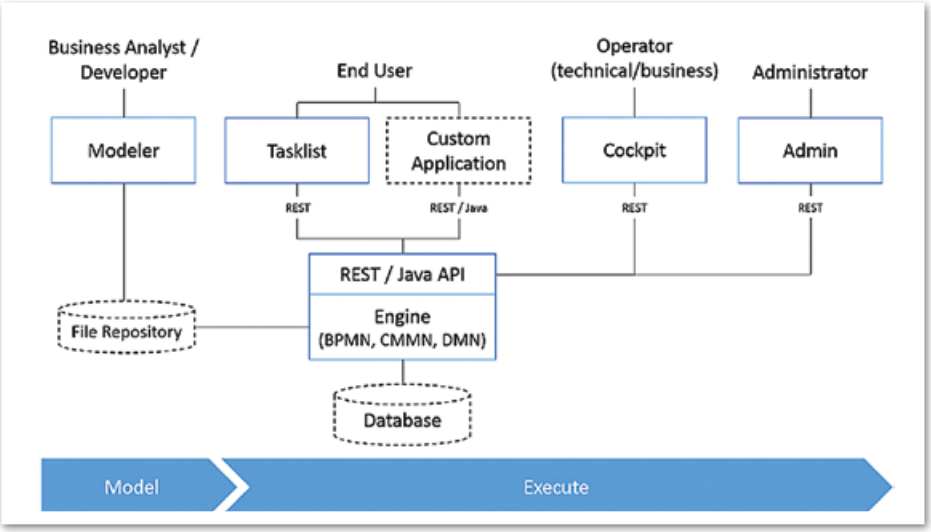


Abbildung 5: Komponenten von Camunda BPM

- „Antragsentscheidung freigeben" hat einen Wächter (die kleine Raute), über den Bedingungen dafür definiert werden, wann die Aktivität überhaupt möglich ist. So müssen im Beispiel die „Antragsentscheidung" abgeschlossen sein und außerdem Gründe für ein Vier-Augen-Prinzip vorliegen. Letzteres kann in Camunda durch Expression Language oder aber auch wieder durch DMN ausgedrückt werden. Der Meilenstein „Antrag angenommen" wird dementsprechend nur dann erreicht, wenn eben kein Vier-Augen-Prinzip notwendig ist oder dies bereits erfolgreich durchlaufen wurde. Letzteres wird nicht durch eine Reihenfolge ausgedrückt, sondern auf der Datenebene.
- Bei „Antragsentscheidung freigeben" kann der Mensch entscheiden, dass die „Antragsentscheidung" erneut durchlaufen werden muss. Dies drückt das „#" an der sogenannten „Stage" aus.

Das soll für einen ersten Eindruck genügen. CMMN ist in der Praxis schwieriger verständlich als BPMN, kann unstrukturierte Probleme aber sehr gut ausdrücken. Da Camunda beide Standards unterstützt, kann man sich das richtige Tool für den richtigen Job aussuchen.

Tabelle 1 zeigt nochmal alle drei Standards im Überblick. Es gibt zu jedem dieser Standards übrigens ein Online-Tutorial *(siehe „https://camunda.org/bpmn/tutorial")*. Das hier vorgestellte Fachbeispiel ist auch auf einem kostenfreien Poster erhältlich *(siehe „http://www.bpm-guide.de/2015/11/09/new-bpm-poster-available-now")*.

Camunda BPM

Camunda BPM *(siehe „www.camunda.org")* ist eine Open-Source-BPM-Plattform unter Apache-Lizenz, die alle drei Standards unterstützt. Es existiert eine Enterprise Edition, die neben Support und zusätzlichen Features vor allem stetig Patches für bereits ausgelieferte Versionen bereitstellt. Dadurch erhält man Bugfixes, ohne auf das nächste Open-Source-Release warten zu müssen, das dann wiederum schon neue Features mitbringt.

Abbildung 5 zeigt die Komponenten der Plattform: im Herzen die Execution Engine mit Java-API, darauf aufbauend das REST-API. Daneben gibt es einen grafischen Modeler für BPMN und DMN (CMMN wird zurzeit fertiggestellt). Zusätzlich existiert eine Web-Anwendung, die eine Aufgabenliste Richtung Endbenutzer *(siehe Abbildung 4)* sowie ein Cockpit für den Betrieb *(siehe Abbildung 9)* bereitstellt. Persistenz erfolgt über eine relationale Datenbank, wobei alle gängigen Produkte unterstützt werden.

Neben der Core-BPM-Plattform gibt es zahlreiche Community Extensions wie Integration mit Apache Camel, ElasticSearch, Grails, PHP oder auch Persistenz auf Cassandra. Einen guten Einstieg bieten die Dokumentation *(siehe „http://docs.camunda.org")*

Abbildung 7: Entscheidung

oder das kostenlos verfügbare Online-Training *(siehe „https://network.camunda.org/training/camunda/1“)*.

Ausführbarkeit

Alle hier gezeigten Modelle sind nicht nur schöne Bilder, sie liegen vielmehr als XML-Dateien vor, die direkt ausgeführt werden können. Dazu müssen noch weitere Attribute hinzugefügt werden, die für die Ausführung wichtig sind, zum Beispiel:

- *Formulare für Aufgaben*
  Es gibt verschiedene Möglichkeiten, Formulare zu hinterlegen, von generischen oder generierten Formularen bis hin zu HTML5-Snippets. Das in *Abbildung 4* dargestellte HTML5-Formular verwendet JavaScript inklusive AngularJS, um beliebige Funktionalität einzubinden. Die Mächtigkeit ist also nicht eingeschränkt.

- *Service-Aufrufe*
  Diese werden primär über Java-Code umgesetzt, der im Modell referenziert wird *(siehe Abbildung 6)*. Hierbei können auch Spring oder CDI Beans aufgerufen werden.

- *Entscheidungslogik*
  Jede laufende Prozess- oder Case-Instanz kann Daten beinhalten, man spricht hierbei von Prozess- oder Case-Variablen. An BPMN-Entscheidungspunkten kann per Expression Language auf diese Daten zugegriffen werden, um den weiteren Pfad auszuwählen, wie in *Abbildung 7* gezeigt. Ähnliches ist auch in CMMN möglich.

Architektur und Deployment

Die Camunda Engine ist eine sogenannte „Embeddable Engine“, also in der Hauptsache eine Library, die in jeder beliebigen Umgebung eingesetzt werden kann. So

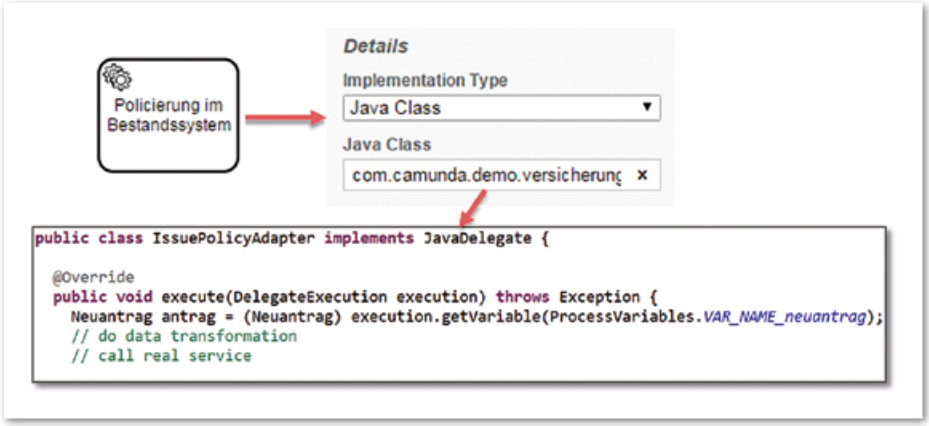


Abbildung 6: Serviceaufruf

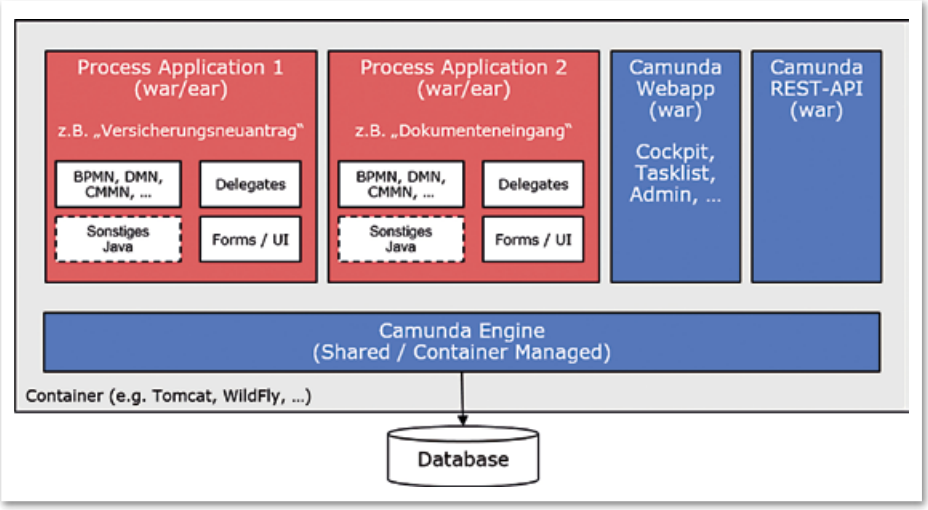


Abbildung 8: Deployment und Prozessanwendungen

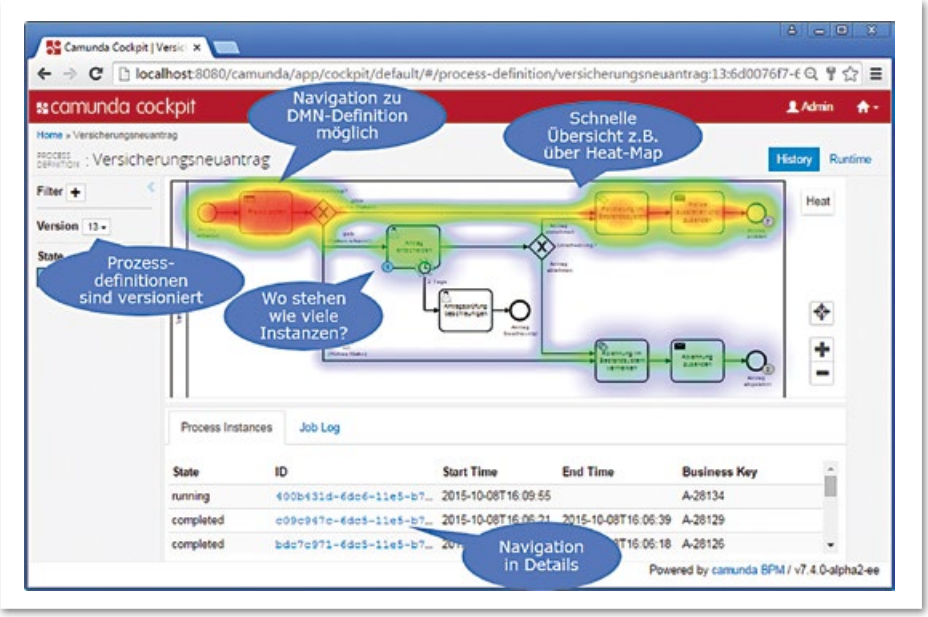


Abbildung 9: Prozess-Transparenz in Cockpit

```
ProcessEngine processEngine = new StandaloneProcessEngineConfiguration().buildProcessEngine();
processEngine.getRepositoryService().createDeployment().addClasspathResource("...").deploy();
processEngine.getRuntimeService().startProcessInstanceByKey("versicherungsneuantrag", variables);
```

Listing 1

```
@Test
@Deployment(resources = {"Versicherungsneuantrag.bpmn", "Risikopruefung.dmn"})
public void testManuelleVerarbeitung() {
  // TODO: Mock initialisieren

  // Testdaten: Police mit Risiko "Sinnlose Raserei"
  Neuantrag neuantrag = DemoData.createNeuantrag(40, false, "Porsche", "911");

   // Objekte als Prozessvariable sind möglich, sollten dann aber als JSON oder XML in der DB serialisiert werden:
  VariableMap variables = Variables.createVariables().putValue(
      ProcessVariables.VAR_NAME_neuantrag,
      Variables.objectValue(neuantrag).serializationDataFormat(SerializationDataFormats.JSON).create());

  ProcessInstance processInstance = runtimeService().startProcessInstanceByKey("versicherungsneuantrag", vari-
ables);

  assertThat(processInstance).task()
    .hasDefinitionKey("userTaskAntragEntscheiden")
    .hasCandidateGroup("sachbearbeiter");
  // Benutzereingabe simulieren
  complete(task(), withVariables("approved", Boolean.TRUE));

  // Prozess ist beendet und die Police wurde ausgestellt
  assertThat(processInstance).isEnded()
    .hasPassed("serviceTaskPoliceAusstellen", "sendTaskPoliceZusenden");
}
```

Listing 2

lässt sich mit wenigen Zeilen Code eine Engine hochfahren, auf der dann die Modelle (BPMN, DMN und CMMN) eingerichtet werden, um dann eine neue Prozess-Instanz zu starten. Dabei kann eine Map von Daten mitgegeben werden *(siehe Listing 1)*.

Die Engine kann jetzt beispielsweise in eine Spring-Anwendung eingebettet oder auch per Spring Boot oder Dropwizard gestartet werden. Ein alternativer Modus ist die „Container Managed Engine". Dabei wird die Camunda Engine in dem verwendeten Container installiert (im Tomcat als „Shared Library", in WildFly als „Module" und in anderen Java-EE-Containern wie WebSphere als „EAR"). Danach können die Modelle (= XML-Dateien) normalen Java-Deployments („WAR" oder „EAR") hinzugefügt werden. Diese werden dann automatisch auf der Engine eingerichtet. In diesem Szenario merkt sich die Engine, in welchem Deployment verknüpfter Java-Code oder Oberflächen zu suchen sind. Dies erlaubt es, verschiedene solcher Prozess-Anwendungen nebeneinander auf der gleichen Engine einzurichten, man spricht daher auch von der „Shared Engine" *(siehe Abbildung 8)*. In diesem Szenario können auf dem Container zusätzlich Web-Anwendungen ausgebracht werden, die ebenfalls auf die Shared Engine zugreifen und Tasklist, Cockpit sowie REST-API bereitstellen.

Unit-Tests

Für automatisierte Unit-Tests wird die Engine im Test-Case mit einer In-Memory-H2-Datenbank hochgefahren. Dabei gibt es Assertions für möglichst lesbare Test-Cases *(siehe Listing 2)*. Natürlich können hier auch Mocking-Frameworks eingesetzt werden, um die echten Serviceaufrufe (z.B. Richtung Bestandssystem) in den Tests nicht auszuführen. Im online verfügbaren Quellcode ist dies beispielhaft mit Mockito gezeigt.

Fazit

Es hat immer gleich angefangen: „Für diese kleine Workflow-Anforderung ist eine Workflow-Engine zu aufwändig". Am Schluss ist dann manchmal ein ganzes Team dauerhaft mit der Wartung der eigenen Engine beschäftigt, die natürlich trotzdem dem Markt hinterherhinkt. Jeder noch so kleine Zustandsautomat ist am Ende komplexer als die Einführung eines leichtgewichtigen Frameworks.

Entwicklerfreundliche Engines der neuen Generation sind keine proprietären Black-Box-BPM-Suiten mehr, sondern flexible Frameworks, die in beliebige Architekturen eingebettet werden können. Das reicht von Java EE über Java SE mit Spring & Co. bis hin zu polyglotten Architekturen, bei denen HTTP gesprochen wird.

Hinweis: Der komplette Quellcode für das hier vorgestellte Beispiel ist auf GitHub verfügbar *(siehe „https://github.com/camunda/camunda-consulting/tree/master/showcases/de/versicherungsneuantrag")* und kann direkt auf der Tomcat- oder WildFly-Distribution einer Camunda BPM 7.4 eingerichtet werden *(siehe „http://camunda.org/download")*.

Bernd Rücker
bernd.ruecker@camunda.com

Bernd Rücker ist Mitgründer und Technology Evangelist bei Camunda. Vor Camunda BPM hat er aktiv an der Entwicklung der Open-Source-Workflow-Engines JBoss jBPM 3 und Activiti mitgearbeitet. Er begleitet seit mehr als zehn Jahren Kundenprojekte rund um BPM, Process Engines und Java Enterprise. Bernd Rücker ist Autor mehrerer Fachbücher, zahlreicher Zeitschriftenartikel und regelmäßiger Sprecher auf Konferenzen.

Der will doch nur spielen

Jens Stündel, Goodgame Studios

Im September veranstaltete die Java User Group Hamburg zusammen mit dem Gastgeber Goodgame Studios einen Special Day zum Thema „Gaming". In vier hochkarätig besetzten Talks kam die Spiele-Entwicklung unter die Lupe. Der Artikel wirft einen Blick hinter die Kulissen eines Spiele-Unternehmens und zeigt, wie sich die Software-Entwicklung in der Spielebranche von der in anderen Bereichen unterscheidet.

Software-Entwickler haben immer noch viele Vorurteile gegenüber der Spielebranche: Es werde keine ernstzunehmende Software entwickelt, dort arbeiteten nur Freaks, die nicht erwachsen werden wollen, und die Gehälter seien alle unterdurchschnittlich. Wenn dann dazu eventuell noch Eltern mit Sprüchen kommen wie „Spielen kannst du in deiner Freizeit, mach lieber was Richtiges", entsteht ein Bild, das Entwickler eher davon abhält, sich für diese Branche zu entscheiden. Dabei hat sich in der Spiele-Entwicklung viel getan: Online- und Mobile-Games gehören zu den am schnellsten wachsenden Märkten, es werden weltweit erfolgreiche Produkte geschaffen und hochperformante Software geschrieben, und auch die Gehälter sind mittlerweile konkurrenzfähig, sodass auch erfahrene Entwickler angezogen werden.

Wenn man als Führungskraft in einem Job-Interview erfahrene Bewerber fragt, was ihnen an einer Position wichtig ist, werden in fast allen Fällen herausfordern-

de Aufgaben genannt. In der Tat stellt sich dem Entwickler in der Spielebranche eine Reihe spezieller Herausforderungen, die von außen häufig unterschätzt werden. Die Spiele-Industrie ist im Bereich der Programmierung ein besonders schnelllebiges und innovatives Gebiet. Insbesondere im „Free to play"-Segment gilt es, stets die neuesten Trends im Auge zu behalten und den Spielern in kurzen Zyklen neue, spannende Inhalte zu liefern.

Ein Spiel ist keine praktische Anwendung, die einen bestimmten Zweck erfüllt, wie beispielsweise eine Online-Banking-Software. Es soll einfach nur Spaß machen und ablenken; wenn es nicht (mehr) gefällt, kommt das nächste dran. Trends kommen und gehen – in der Unterhaltungsbranche besonders schnell. Ein erfolgreicher Spiele-Entwickler muss mit diesen Trends mithalten und idealerweise selbst Trends setzen. Dazu gehört natürlich auch die technische Basis. Neben der Marktbeobachtung, um neue Tendenzen zu erkennen, sind ständig neue Technologien zu evaluieren, Prototypen zu bauen und zu verwerfen.

Um welche Herausforderungen es in der Spieleentwicklung sonst noch geht, wird am Beispiel von Goodgame Studios dargestellt werden. Das Hamburger Unternehmen betreibt mehr als zehn Spiele weltweit, die insgesamt 300 Millionen registrierte Nutzer verzeichnen, und hat sich mit bekannten Titeln wie der App „Empire: Four Kingdoms", deren Browser-Version „Goodgame Empire" sowie mit „Goodgame Big Farm" bereits international einen Namen gemacht. Mit mehr als 1.100 Mitarbeitern aus über 60 Nationen ist Goodgame Studios Deutschlands mitarbeiterstärkster Entwickler von Spiele-Software.

Server-Programmierung mit Java

Neben dem Game-Client macht einen großen Teil der Spiele-Entwicklung die Entwicklung der Gameserver aus. Deren größte Herausforderung ist die Concurrency. Mehrere 10.000 Spieler sind in der Regel mit einem Gameserver verbunden und wollen gegen- oder miteinander spielen. Alle Aktionen der Spieler beeinflussen sich gegenseitig, dadurch ist die richtige Reihenfolge der Abarbeitung entscheidend. Merkbare Latenzen werden nicht lange toleriert und schnell mit dem Verlassen des Spiels quittiert.

Wird beispielsweise mit Locks gearbeitet, sind die richtige Locking-Strategie und ihr konsequenter Einsatz entscheidend, da Fehler in der Umsetzung sonst in teuren Deadlocks enden. Es gibt aber auch Spiele-Entwicklungsteams, die mit anderen Ansätzen wie zum Beispiel Lock-Free Programming oder dem Actor Model gute Erfahrungen gemacht haben.

Neben der Parallelität ist auch die Skalierbarkeit eines der Top-Themen. Aktuell wird bei Goodgame dabei verstärkt auf das Mesos/Marathon-Framework gesetzt. Für das Business-Intelligence-Team werden zudem große Mengen an aggregierten Daten zum Spielerverhalten gesammelt, die dazu dienen, die Spiele optimal zu justieren, um einen möglichst großen Spielspaß zu ermöglichen.

Warum wird bei Goodgame Studios überhaupt Java für die Serverseite der Spiele eingesetzt, obwohl andere Sprachen aus Sicht der Problemstellung vielleicht die naheliegende Wahl wären? Zum einen, weil sich Java als vielseitige Allzweckwaffe erwiesen hat. Alle Stärken und Schwächen sind bekannt, alle serverseitigen Anwendungsfälle sind damit umsetzbar. Außerdem lässt sich die Performance sehr gut direkt beeinflussen. Nicht zuletzt gibt es sehr gute, erfahrene Java-Entwickler – bei exotischeren Sprachen wie zum Beispiel Erlang sind diese schon schwieriger zu finden.

Frameworks richtig eingesetzt

Die Tendenz in der allgemeinen Software-Entwicklung geht aktuell dahin, immer weiter zu abstrahieren. Man will möglichst viele Frameworks einsetzen, um die Entwicklungsgeschwindigkeit zu erhöhen und die Fehlerquote zu verringern. Bei Goodgame Studios hat man ebenfalls versucht, die Arbeit durch den Einsatz von Frameworks zu erleichtern, doch die Erfahrung hat gezeigt, dass diese oft zu sehr einschränken, zu unflexibel sind oder die Performance beeinträchtigen. So kommen beispielsweise Spring oder Hibernate nur selten zum Einsatz. Dadurch erhöht sich das Fehlerpotenzial zwar, aber da die Performance der Spiele stark profitiert, wird dieses Risiko gerne in Kauf genommen.

Hier zeigt sich dann auch das Problem auf der Suche nach geeigneten Entwicklern. Viele Bewerber sind heutzutage zwar Experten im Einsatz von Frameworks, haben aber das reine Programmieren nahezu verlernt. Dadurch scheitern sie schon an grundlegenden Programmieraufgaben. Eine ausgewogene Mischung aus Erfahrung mit Frameworks und reiner Programmierung ist optimal.

Lernen zu failen – failen, um zu lernen

Um Erfolge zu produzieren, ist es wichtig, Misserfolge oder Fehlschläge schnell zu erkennen. Es ist eher unwahrscheinlich, mehrere erfolgreiche Spiele direkt hintereinander zu produzieren. Zwischen zwei Erfolgen liegen immer einige Fehlversuche. Das lässt sich in einem kreativen Bereich wie der Spielebranche nicht verhindern. Umso wichtiger ist es, im Unternehmen die richtige Kultur im Umgang mit Misserfolgen zu etablieren. Je früher ein Fehlversuch erkannt wird, desto schneller kann der nächste potenzielle Hit produziert werden. Die Herausforderung ist, das Gelernte ins nächste Projekt mitzunehmen und somit die Entwicklung stets zu optimieren.

Spieler sind die besten Tester

Im Bereich „Testing" zeichnet sich die Software-Entwicklung in der Spielebranche durch ihre besondere Flexibilität und Agilität aus. Entgegen dem aktuellen Trend, auf Software-Entwicklungsebene zuerst zu testen und dann zu entwickeln, ist die Programmierarbeit bei Spielen weniger testgetrieben. Unit-Tests und Systemtests sind bei Backend-Services, die von allen Spielen genutzt werden, absolut sinnvoll. Doch gerade bei der Feature-Entwicklung im eigentlichen Game-Code sind sie oft hinderlich. Bei Goodgame Studios haben die erfolgreichsten Spiele beispielsweise die geringste Testabdeckung. Das ist historisch gewachsen, doch auch heutzutage können die Entwickler sowohl im Entstehungs- als auch im Instandhaltungs-Prozess eines Spiels sehr flexibel agieren, ohne stark von einem testbasierten Konzept eingeschränkt zu werden. Natürlich werden vor der Produktion eines neuen Titels die speziellen Anforderungen an das Spiel festgelegt, auf deren Basis dann ein Produkt entwickelt wird, das diese minimalen Ansprüche erfüllt. Ob es das tut, wird in mehreren Testzyklen geprüft, die jedoch die Entwicklerarbeit nicht bremsen, sondern hilfreiches Feedback für die weitere Programmierung liefern.

Nachdem die Quality-Assurance-Abteilung das Spiel gründlich getestet hat, das Produkt optimiert wurde und es marktfä-

hig ist, wird es in ausgewählten Märkten im Rahmen eines Technical Launches veröffentlicht, um die Spiele-Mechanik und technische Funktionalität zu prüfen. Nach erneuter Optimierung tritt es dann in den Soft Launch ein und wird erst dann global verfügbar, wenn die Erkenntnisse aus diesen Testphasen bestmöglich umgesetzt sind.

Auf Entwicklungs-Ebene geht es in allen Phasen darum, Fehler zu beheben, die weiterhin im Spiel auftreten können. Sobald das Spiel in einen Markt eintritt, ergeben sich Impulse aus der Spielerschaft, die ins Produkt eingearbeitet werden. Von daher ist eine schnelle Fehlererkennung und -behebung effektiver, als alle Probleme, die wahrscheinlich nie auftreten werden, schon im Vorfeld ausschließen zu wollen.

Wöchentliche Updates

Eine Besonderheit von Online-Games sind die sehr kurzen Release-Zyklen. Bei Goodgame Studios gibt es das Ziel, für jedes aktive Spiel wöchentlich ein Update herauszubringen. Das bedeutet entweder sequenzielle Ein-Wochen-Entwicklungszyklen oder parallele, zeitversetzte Zwei-Wochen-Zyklen. Solche Parallel-Zyklen sind natürlich besonders spannend, da sie sich immer gegenseitig beeinflussen. Da muss schon der ein oder andere Konflikt gelöst werden. Agile Methoden haben sich hier als eine sinnvolle Lösung etabliert. Jedes Team findet die für sich passende Methode und entwickelt sie individuell weiter. So entstehen maßgeschneiderte Varianten von Scrum und Kanban oder auch Mischungen aus beiden – immer mit dem Ziel, die effektivste Teamorganisation zu finden.

Die Grundprinzipien sind in jedem Team vertreten: So beginnt ein Sprint mit einem Planning-Meeting. Während eines Sprints gibt es tägliche Stand-ups und am Ende werden die Ergebnisse in einem Review präsentiert und abgenommen. Besonderen Wert wird auf die Retrospektive gelegt. Die Herausforderung ist es, durch offenes Feedback und daraus folgende Verbesserungen ein optimales Umfeld für das Team zu schaffen. Bei Goodgame werden die Entwicklungsteams durch dedizierte Agile Coaches unterstützt.

Knowledge Transfer

Es ist wichtig, immer einen regen Austausch der Entwickler zu fördern, sodass optimale Lösungen entstehen. Dadurch verbessern sich sowohl die Code-Qualität als auch die fachliche Kompetenz der Entwickler. Die effektivste Weiterbildungsmaßnahme ist hier der Austausch untereinander, um Wissens-Silos zu vermeiden. Pair Programming oder auch Mob Programming sind gern gesehen. Code-Reviews sind fester Bestandteil des Entwicklungsprozesses. Die meisten Teams setzen auf Two-Face-Commits, es wird also kein Code freigegeben, den nicht mindestens zwei Entwickler gesehen haben.

Wenn man in kleineren, separaten Teams arbeitet, kann der Wissensaustausch über die Teamgrenzen hinweg zur Herausforderung werden. Dieser Wissensaustausch ist aber ungemein wichtig für ein effizientes Arbeiten in einem Unternehmen, etwa um den gleichen Fehler nicht mehrmals zu begehen.

Bei Goodgame Studios gibt es verschiedene Maßnahmen, um den Knowledge-Transfer zu fördern. Dazu gehören ein firmenweites Wiki ohne Lesebeschränkung, Zugang zu den Repositories aller Projekte, Communities of Practice (CoP) zu relevanten Themen sowie dedizierte Slack-Channels für den schnellen direkten Austausch. Zudem gibt es einen Pool von Architektur-Experten, der allen Teams zur Verfügung steht. Besteht in einem Team eine anspruchsvolle Architekturaufgabe, bei der es Unterstützung braucht, kann es sich einen Experten für eine begrenzte Zeit aus dem Pool holen. Das hat den Vorteil, dass der Experte sein Wissen ins Team bringt und andererseits auch das Wissen aus dem Team in andere Teams mitnehmen kann. Die Experten tauschen sich wiederum untereinander aus.

Ein weiteres beliebtes Angebot zum Wissensaustausch sind die Tech Talks. In dieser Reihe werden regelmäßig interne Vorträge auf Konferenzniveau zu aktuellen Themen veranstaltet, präsentiert von erfahrenen Mitarbeitern.

Fazit

Die Gaming-Branche ist besser als ihr Ruf. Entwickler haben die Möglichkeit, sich mit den aktuellsten technologischen Entwicklungen zu beschäftigen und daran selbst zu wachsen. Interessante Herausforderungen gibt es genug. „Out of the box"-Denken wie auch Kreativität werden hier gefördert und gelebt, die eigenen Produkte sind weltweit eingesetzt und machen den Menschen Freude. Wer also gerne flexibel, in schnellen Zyklen und kreativ arbeitet, ist als Software-Entwickler in einer Gaming-Company genau richtig.

Die Zukunft hält sicherlich noch einige spannende Herausforderungen für die Spielebranche bereit. Aufgrund der Schnelllebigkeit des Markts müssen Spiele-Entwickler stets neue Technologien im Auge behalten. Wird dabei etwas Interessantes entdeckt, sollte der Einsatz evaluiert und eventuell mit einem Prototyp verifiziert werden. Zum Beispiel wird schon lange mit Augmented Reality (AR) experimentiert und der Erfolg von Spielen wie Pokémon Go bestärkt diesen Ansatz. Auch Virtual Reality (VR) wird in Zukunft eine große Rolle spielen, nicht nur in der Gaming-Branche. Eins ist sicher: Langeweile wird bei der Entwicklung nicht aufkommen.

Jens Stündel
jstuendel@goodgamestudios.com

Jens Stündel hat mehr als fünfzehn Jahre Erfahrung in der Entwicklung von Online- und Mobile-Games und dabei überwiegend Java als Server-Technologie eingesetzt. Seit fünf Jahren ist er bei Goodgame Studios als Head of Game Technology verantwortlich für Technologie-Empfehlungen, die Bereitstellung von Basismodulen und gemeinsam genutzten Services für alle Spiele, die Evaluation neuer Technologien und Methoden sowie Knowledge-Transfer über Teamgrenzen hinweg.

Der Einspritzmotor

Sven Ruppert

Wenn man beginnt, sich mit dem Thema „Dependency Injection“ zu beschäftigen, trifft man auf eine Vielzahl von Frameworks. Wie soll man sich endscheiden? Was sind grundlegende Unterschiede, die auf das Projekt Einfluss nehmen?

Beginnen wir ganz von vorne. In Java gibt es verschiedene Möglichkeiten, eine Referenz mit einem Objekt zu verbinden. Die einfachste Art besteht in der Verwendung der Set-Methoden. Sicherlich, das ist trivial und in der OO-Welt einer der Kernpunkte *(siehe Listing 1)*.

Ein weiterer Weg besteht darin, eine Referenz als Konstruktor-Parameter zu übergeben. Dabei muss entschieden werden, ob es einen weiteren Konstruktor ohne diesen Parameter geben wird, oder ob eine Referenz zwanghaft gesetzt werden muss *(siehe Listing 2)*.

Mittels „Reflection" kann man ebenfalls den Inhalt eines Klassenattributs setzen. Hier ist ein wenig mehr Aufwand erforderlich, allerdings muss man hier weder einen Konstruktor-Parameter vorsehen, noch eine Methode, mittels der eine Referenz gesetzt werden kann *(siehe Listing 3)*.

```
public interface Service {
    void doWork();
  }

  public class BusinessModule {
    private Service service;

    public void setService(final Service service) {
      this.service = service;
    }
  }
```

Listing 1

```
  public interface Service {
    void doWork();
  }
  public static class BusinessModule {
    private Service service;
    public BusinessModule(final Service service) {
      this.service = service;
    }
  }
```

Listing 2

```
  public interface Service {
    void doWork();
  }
  public static class BusinessModule {
    private Service service;
  }
  public static void main(String[] args) {
    Service service; // instance is comming somewhere
    BusinessModule businessModule; //instance from somewhere

    try {
      Class<? extends BusinessModule> businessModuleClass
            = businessModule.getClass();
      Field declaredField
            = businessModuleClass.getDeclaredField("service");
      boolean accessible = declaredField.isAccessible();
      declaredField.setAccessible(true);
      declaredField.set(businessModule, service);
      declaredField.setAccessible(accessible);
    } catch (NoSuchFieldException e) {
      e.printStackTrace();
    }
  }
```

Listing 3

Die Lebenslinien der Referenzen

Wenn man nun alle drei Wege betrachtet, stellt sich sofort die Frage, welcher Weg Vor- beziehungsweise Nachteile mit sich bringt. Um das für den jeweiligen Fall bewerten zu können, sehen wir uns die Lebenslinien der jeweiligen Varianten an.

Bei der Set-Methode lassen sich die beiden beteiligten Instanzen zu unterschiedlichen Zeitpunkten erzeugen (t1). Zu einem beiden Ereignissen nachfolgenden Zeitpunkt können diese beiden Referenzen miteinander verbunden werden (t2). Ab diesem Zeitpunkt t2 kann dann zu einem beliebigen Zeitpunkt t3 ein Methodenaufruf erfolgen, der die benötigte Referenz in der Ausführung benötigt. Hieraus ergibt sich implizit die Möglichkeit, erst dann die Referenzen setzen zu müssen, wenn diese auch tatsächlich benötigt werden. Meistens wird dieses Verhalten durch einen Virtual-Proxy realisiert.

Kommt der Konstruktor zum Einsatz, muss zum Zeitpunkt der Erzeugung der haltenden Instanz die benötigte Instanz bereits erzeugt worden sein. Daraus ergeben sich die Zeitpunkte t1 (Erzeugung der benötigten Instanz) und t2 (Erzeugung der haltenden Instanz). Nun kann zu einem beliebigen weiteren Zeitpunkt der Aufruf der Methoden an dem haltenden Objekt erfolgen. Auch hier besteht wieder die Möglichkeit, mit einem Virtual-Proxy die Erzeugung der benötigten Instanz auf einen Zeitpunkt kurz vor der ersten Verwendung zu verlegen.

Wenn man sich für den Einsatz von Reflection entschieden hat, können wie bei der Verwendung einer Set-Methode die beteiligten Instanzen zu den beliebigen Zeitpunkten t1 und t2 erzeugt werden. Erst nach dem Zeitpunkt t3 (Setzen der benötigten Instanz in den Holder) kann zu einem weiteren Zeitpunkt t4 der Aufruf der Methoden am haltenden Objekt durchgeführt werden.

Die Unterschiede der Methoden

Im Nachfolgenden vernachlässigen wir den Einsatz eines Virtual-Proxy und beschränken uns auf den Ablauf bei Verwendung nicht umhüllter Instanzen. Bei der Betrachtung der unterschiedlichen Methoden fällt als erstes auf, das bei der Verwendung der Konstruktor-Methode nur zwei Erzeugungs-Zeitpunkte (t1 und t2) vor Verwendung der Ziel-Referenz vorhanden sind. Ebenfalls ist es von Bedeutung, das die zu setzenden Instanz immer vor Erzeugung der haltenden Instanz erstellt werden muss. Demnach muss die Endscheidung, welches die richtige Implementierung ist, vorher getroffen werden.

Wenn man nun die Methoden „Set" und „Reflection" vergleicht, kann man bei der

```
public class Service { }
@Inject Service service;
```

Listing 4

```
public interface Service { }
 public class ServiceImpl implements Service { }
 @Inject Service service;  // how to find the impl
```

Listing 5

```
@Inject Sevice service;
public interface Service {}
public class ServiceImpl implements Service {}
```

Listing 6

zeitlichen Abfolge keine Unterschiede erkennen. Lediglich, dass bei Verwendung von Reflection keine Methode in der Zielklasse vorhanden ist, die ein späteres gefahrloses Setzen suggeriert. Natürlich ist das Fehlen einer Set-Methode kein Schutz vor dem Setzen eines Attributs an sich, jedoch ist es einem Entwickler eher bewusst, dass es nicht den vorgesehenen Weg darstellt. Ähnliches kann man natürlich auch durch das Vorschalten von Interfaces realisieren, bei denen die Set-Methoden nicht vorhanden sind.

Grundsätzlich kann man bei allen drei Methoden mehrere Zeitpunkte für Entscheidungen verwenden. Daraus resultierend, dass zu dem Zeitpunkt, zu dem man sich für eine bestimmte Implementierung der benötigten Instanz endscheidet, der Kontext der haltenden Instanz für die Endscheidung mit herangezogen werden kann. Der Einsatz von Reflection benötigt ein wenig Laufzeit-Overhead, verhindert aber auch, dass eine Methode angeboten werden muss (Set-Methode) die eventuell nicht zur allgemeinen Verwendung vorgesehen ist.

Welche Fragen sich ergeben

Aber zurück zu den Dependency-Injection-Frameworks. Welche Fragen kann ein Dependency-Injection-Framework nun beantworten und was ist notwendig, um Antworten zu den jeweiligen Zeitpunkten zu liefern? Die erste Frage ist diejenige nach der richtigen Implementierung in dem jeweiligen Einzelfall. Gibt es nur eine Implementierung, ist diese Frage trivial, existieren allerdings mehrere, so muss es eine Lösungsstrategie geben, mit der ein Framework diese Entscheindung treffen kann.

Die zweite Frage hat temporale Aspekte und lautet: „Wann muss diese Endscheindung getroffen werden?" Sie ist durch die Verwendung der jeweiligen Methode zum Setzen der Referenz deutlich beeinflusst.

Zum Schluss ergibt sich sofort die Fragestellung: „Wie oft muss diese Endscheidung für dieselbe Instanz erneut getroffen werden?" Hier beginnen wir eine Diskussion, die recht lange geführt werden kann. Aber ein Schritt nach dem anderen. Die wichtigste Frage ist immer noch die erste nach der Auswahl der richtigen Implementierung.

Endscheidung für eine Implementierung

Wen es nur eine Implementierung ohne Interface gibt, ist die Endscheidung trivial *(siehe Listing 4)*. Aber schon bei der Kombination Interface und eine Implementierung ist es nicht mehr so einfach *(siehe Listing 5)*. Sicherlich gibt es nur eine Lösung, aber wie gelangt man zu dieser?

Hier haben die Frameworks unterschiedliche Wege beschritten. Eine Möglichkeit besteht in der direkten Angabe der korrespondierenden Implementierung. Das kann an verschiedenen Stellen erfolgen.

Beschreibende Files und Annotationen

Die Information, welche Implementierung zu welchem Interface gehört, lässt sich einfach in eine Datei schreiben. Diese wird zum Beispiel zu Start der Anwendung ausgelesen und ergibt dann das Reflection-Modell. Anhand dieses Modells werden dann die zu instanziierenden Klassen ausgesucht. In unserem Fall ist dort die Information enthalten, dass Interface-Service zur Klasse „ServiceImpl" gehört. Nachteil hier ist, dass diese Information nicht so einfach zu den realen Klassen synchron gehalten werden kann. Aber das hat so ziemlich jeder wohl schon mit „.xml"-Files erlebt.

Die Informationen können natürlich auch im Sourcecode mit angegeben sein. Die derzeit gebräuchlichste Form ist die Annotation. Daraus ergeben sich verschiedene Zeitpunkte, zu denen dieses ausgewertet wird. Zum einen kann das bei der Übersetzung der Sourcen erfolgen. Seit JDK1.6 gibt es das „AnnotationProcessing". Die benötigten Informationen werden aus den Annotationen ausgelesen und führen dann zu generiertem Quelltext, der die Endscheidung darstellt. Zur Laufzeit kann man ebenfalls diese Informationen auslesen und verwerten. Der Weg geht dann über das Classpath-Scanning. Je nachdem, in welcher Umgebung man sich bewegt, kann das ein etwas umständliches Unterfangen darstellen.

Man kann sich aber auch vorstellen, das es diese Informationen auch im Quelltext selbst beziehungsweise als Quelltext (ja, Annotationen sind auch Quelltext-Bestandteil) vorliegen. Sehr oft führen diese Ansätze dazu, das der Entwickler selber den Binding-Code schreibt. Oft kommen das Design-Pattern „Builder" oder „Factory" zum Einsatz. Die Verdrahtung ist demnach an anderer Stelle explizit vorhanden.

Auch zur Laufzeit kann mittels Reflection das Klassenmodell extrahiert werden. Ist diese Information vorhanden, stellt sich die Frage der Endscheidung. Wie wird sich zwischen zwei Implementierungen eines gemeinsamen Interface endschieden? Ein Ansatz kann sein, dass in einem solchen Fall das Framework nach einem Resolver sucht. Dieser muss für genau diese Frage zuständig sein, in unserem Fall: Was die zu verwendende Implementierung ist. Ähnliches Verhalten kann man beispielsweise auch bei CDI erreichen, wenn man als statische Endscheidung einen Producer gewählt hat und in diesem dann dynamisch die Endscheidungen trifft.

Der Zeitpunkt der Erzeugung

Wenn wir nun in der Lage sind, die Endscheidung zu treffen, welches die richtige Implementierung sein wird, stellt sich die Frage nach dem Zeitpunkt, wann diese Fra-

ge beantwortet werden soll. Nun haben wir ein klassisches Henne-Ei-Problem. Je näher wir an dem Zeitpunkt der Verwendung sind, desto mehr Informationen können theoretisch Einfluss auf die Endscheidung nehmen, was die richtige Implementierung ist. Mit anderen Worten, der Kontext kann immer dominanter im Endscheidungsprozess werden.

Wenn wir zu einem Interface zwei Implementierungen haben, unterscheiden sich diese lediglich darin, dass die Implementierungen an einen Mandanten gebunden sind (Kunde A und Kunde B), Wenn wir ein Session-getriebenes Szenario haben, reicht es aus, diese Endscheidung zu dem Beginn der Session einmal zu treffen (unter der Annahme, dass der Mandant in der Session nicht gewechselt werden kann). Der Zeitpunkt ist klar definiert und zumeist statischer Natur.

Aber nehmen wir nun an, dass die Implementierungen sich darin unterscheiden, dass Implementierung A sehr schnell ist und viel Speicher benötigt; Implementierung B hingegen braucht kaum Speicher, ist aber leider nicht performant. Jetzt könnte man zu jedem Request die Endscheidung treffen, welche Implementierung nun die Bessere für das Gesamtsystem ist, basierend auf den technischen Metriken zu dem Zeitpunkt der Verwendung. Dafür lassen sich selbst konfigurierende Regelsysteme aufbauen.

Auf die Frage „Was gehört alles zu einem DI-Framework?" bekommt man sehr unterschiedliche Antworten. Wenn man sich allerdings einige Frameworks ansieht, so kann man das als Grundlage nehmen. Zentraler Bestandteil ist auf jeden Fall die Möglichkeit, die Endscheidung für eine Implementierung zu separieren. Allerdings kommen dann auch schnell Elemente wie Scopes, Events, Interceptoren und Lifecycles (PostContruct, PreDestroy) ins Spiel. Bei der Betrachtung der Frameworks selbst ist auch immer festzustellen, welche Elemente vorhanden sind.

```
@Inject Sevice service;              //ServiceImplA
@Inject @CustomerX Service service;  //ServiceImplB

public interface Service {}
public class ServiceImplA implements Service {}

@CustomerX
public class ServiceImplB implements Service {}
```

Listing 7

```
@Inject Sevice service;              //ServiceImplA
@Inject @CustomerX Service service;  //result from producer
public interface Service {}
public class ServiceImplA implements Service {}

@CustomerX @Produces
public Service produceService() {
        return new Service(){}; // could be a dynamic dec.
}
```

Listing 8

Beispiele für die jeweiligen Lösungsansätze

Nun einige Beispiele, wie verschiedene Frameworks die Frage nach der richtigen Implementierung gelöst haben. Dabei handelt es sich nicht um eine vollständige Auflistung aller Möglichkeiten in einem der genannten Frameworks, es soll aber ein Gefühl dafür vermitteln, wie sich das in einem Projekt auswirken könnte. Sicherlich ist die Auswahl sehr davon geprägt, welche Anforderungen von außen ebenfalls eine Rolle spielen.

Bei **Weld** *(siehe „http://weld.cdi-spec.org")* handelt es sich um eine CDI-Implementierung. Hier können die Antworten auf die Frage nach der richtigen Implementierung in der „beans.xml" oder per Annotations definiert werden. Zum Startzeitpunkt des Containers sind alle Endscheidungen fix.

Bei Multiplizietäten müssen einzelne Implementierungen in andere Namensräume verlegt werden. Zum Beispiel, indem die jeweilige Implementierung mit einem Qualifier versehen wird und damit nicht mehr in dem Default-Scope vorhanden ist. Ebenfalls kann man an der Zielstelle mit Qualifiern auf vorhandene Producer verweisen. Hier kann die Endscheidung einfach getroffen werden, da es nur eine Implementierung gibt *(siehe Listing 6)*.

Hier allerdings sieht es anders aus. Um Weld die Möglichkeit zu geben, sich zu endscheiden, muss man die Lösung etwa mit Qualifiern eindeutig beschreiben. Hier wird „ServiceImplA" genommen, da die andere Implementierung im Namensraum „@CustomerX" liegt *(siehe Listing 7)*. Es kann auch mithilfe von Producern gearbeitet werden, was einem eine größere Flexibilität zur Laufzeit gibt *(siehe Listing 8)*.

BoonDI *(siehe „https://github.com/boon-project/boon")* ist ein Vertreter, der rein auf

```
final Module module = DependencyInjection
                        .classes(
                          Service.class,SubService.class);

public class ServiceModule {  public Service provideService(){ return new Service(); }
 public SubService provideSubService() { return new SubService(); } }

final Module module = DependencyInjection
                        .module(new ServiceModule());
final Context context = DependencyInjection.context(module);
final Service service = context.get(Service.class);
```

Listing 9

```
@Module
public class BusinessModule {
      @Provides MainService
      provideMainService(MainServiceImpl mainService) {
         return mainService;
      }
}

@Component(modules = BusinessModule.class)
public interface BusinessService {
  MainService makeMainService(); // FactoryMethod for MainService
}

//usage
final BusinessService businessService = DaggerBusinessService.builder()
    .businessModule(new BusinessModule())
    .build();

final MainService mainService = businessService.makeMainService();
```

Listing 10

```
@Inject Sevice service;
public interface Service {}
public class ServiceImpl implements Service {}
```

Listing 11

```
public interface Service {}
  public class ServiceImplA implements Service {}
  public class ServiceImplB implements Service {}

  @ResponsibleFor(Service.class)
  public static class ServiceClassResolver
                      implements ClassResolver<Service> {
    @Override
    public Class<? extends Service>
    resolve( Class<Service> interf) {
      return ServiceImplB.class;
    }
  }
```

Listing 12

Reflection basiert. Hier sind alle Abhängigkeiten beziehungsweise der für den jeweiligen Graphen gültige Wertevorrat an Klassen und Interfaces manuell zur Verfügung gestellt. Hierzu kann man entweder die Elemente direkt angeben oder auf Provider (Factories) verweisen. Vorteilhaft ist sicherlich, dass man zur Laufzeit mehrere Versionen von Dependency-Graphen anlegen kann. Objekte, die zwischen diesen Graphen migrieren, können nicht erkannt werden. Bei kleineren Graphen oder Graphen, die eine geringe Ebenentiefe haben, ist das recht handlich.

Um einen Graphen aufzubauen muss man den Wertevorrat explizit angeben. Das kann zum einen direkt erfolgen, indem man die Klassen und Interfaces der Methode „DependencyInjection.classes(..)" direkt übergibt, oder Module definiert *(siehe Listing 9)*.

Dagger 2 *(siehe „http://google.github.io/dagger")* basiert auf Annotation-Processing. Hier werden die jeweils benötigten Factories/Builder generiert und müssen dann von dem Entwickler selbst verwendet werden. Die Basis für die Generierung ist von dem Entwickler zu programmieren (Module und Factory-Interfaces). Das Ziel liegt hier in der Unterstützung von Java und Android, demnach sind Techniken wie Reflection zu meiden. Zur Laufzeit sind die Aufwände für das Framework selbst recht klein, da es sich lediglich um Methoden-Aufrufe handelt *(siehe Listing 10)*.

Bei RapidPM Dynamic DI *(siehe „http://www.dynamic-dependency-injection.org")* liegt der Fokus auf der Dynamik zur Laufzeit. Hier wird das ReflectionModel zur Laufzeit extrahiert und auf Basis von Konventionen der Anteil der Binding-Elemente reduziert, die der Entwickler schreiben muss. Multiplizietäten werden durch Resolver aufgelöst, somit sind zur Laufzeit dem Context angepasste Endscheidungen möglich. Das Reflection-Modell kann zur Laufzeit verändert, genauso wie es auf bestimmte Packages begrenzt werden kann *(siehe Listing 11)*. Prinzipiell ist immer dann, wenn es eine Mulitiplizietät gibt, ein Resolver im Einsatz. Dieser kann für seine Endscheidungen alle Möglichkeiten nutzen, die im System gegeben sind *(siehe Listing 12)*. Mit diesem Ansatz kann man dann auch bei mehr als einem Producer endscheiden, welcher gerade aktiv ist.

Fazit

Alles zusammen ergibt ein Bild, bei dem man sehr darauf achten sollte, welche der Strategien zu dem jeweiligen Projekt passen. Nicht zu vernachlässigen ist der Aufwand bei den jeweiligen Frameworks, wenn es um die Definition größerer Object-Graphen geht. Hier und im Bereich des Refactoring trennen sich sehr schnell die Wege zwischen den Frameworks. Ebenfalls genauer sollte man die Möglichkeit die sich mit dem Framework im Bereich des TDD ergeben ansehen.

Sven Ruppert
sven.ruppert@gmail.com

Sven Ruppert programmiert bereit seit dem Jahr 1996 in Java. Er leitet die Forschung und Entwicklung bei Reply in München. In seiner Freizeit spricht er auf internationalen und nationalen Konferenzen und schreibt für IT-Magazine sowie für Tech-Portale.

Community-Konferenz organisiert von Java User Groups aus dem Norden

http://javaforumnord.de

@JavaForumNord

JAVA FORUM NORD

Das Java Forum Nord ist eine eintägige, nicht-kommerzielle Konferenz in Norddeutschland mit Themenschwerpunkt Java für Entwickler und Entscheider. Mit 28 Vorträgen in bis zu fünf parallelen Tracks und einer Keynote wird ein vielfältiges Programm geboten. Der regionale Bezug bietet zudem interessante Networkingmöglichkeiten.

Mit Vorträgen von Sabine Bernecker-Bendixen, Dirk Dittert, Frank Düsterbeck, Christoph Engelbert, Oliver Fischer, Oliver Gierke, Roger Gilliar, Rabea Gransberger, Mark Heckler, Stefan Hildebrandt, Oliver Hock, Lutz Hühnken, Dierk König, Philipp Krenn, Roland Kuhn, Dr. Carola Lilienthal, Gerrit Meier, Bernd Müller, Robert Panzer, Nicolai Parlog, Lars Röwekamp, Dominik Schadow, Jens Schauder, Bennet Schulz, Falk Sippach, Anatole Tresch, Dirk Weil, Stefan Zörner und einer Keynote von Uwe Friedrichsen.

Hannover, Donnerstag 20. Oktober 2016

Die iJUG-Mitglieder auf einen Blick

Java User Group Deutschland e.V.
www.java.de

DOAG Deutsche ORACLE-Anwendergruppe e.V.
www.doag.org

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

Java User Group Köln
www.jugcologne.eu

Java User Group Darmstadt
http://jug-da.de

Java User Group München (JUGM)
www.jugm.de

Java User Group Metropolregion Nürnberg
www.source-knights.com

Java User Group Ostfalen
www.jug-ostfalen.de

Java User Group Saxony
www.jugsaxony.org

Sun User Group Deutschland e.V.
www.sugd.de

Swiss Oracle User Group (SOUG)
www.soug.ch

Berlin Expert Days e.V.
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
www.jug-h.de

Java User Group Augsburg
www.jug-augsburg.de

Java User Group Bremen
www.jugbremen.de

Java User Group Münster
www.jug-muenster.de

Java User Group Hessen
www.jugh.de

Java User Group Dortmund
www.jugdo.de

Java User Group Hamburg
www.jughh.de

Java User Group Berlin-Brandenburg
www.jug-berlin-brandenburg.de

Java User Group Kaiserslautern
www.jug-kl.de

Java User Group Switzerland
www.jug.ch

Java User Group Euregio Maas-Rhine
www.euregjug.eu

Java User Group Görlitz
www.jug-gr.de

Java User Group Mannheim
www.majug.de

Lightweight Java User Group München
www.meetup.com/de/lightweight-java-user-group-munchen

Java User Group Düsseldorf rheinjug
www.rheinjug.de

Java User Group Goldstadt
https://gitlab.com/groups/jugpf

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

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, *www.ijug.eu*) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH,Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Dr. Dietmar Neugebauer. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH, (Anschrift siehe oben)
Chefredakteur (ViSdP): Wolfgang Taschner
Kontakt: redaktion@doag.org

Redaktionsbeirat:
Ronny Kröhne, IBM-Architekt; Daniel van Ross, FIZ Karlsruhe; André Sept, InterFace AG; Jan Diller, Triestram und Partner

Titel, Gestaltung und Satz:
Caroline Sengpiel,
DOAG Dienstleistungen GmbH

Fotonachweis:
Titel: © rawpixel/123RF
Foto S. 8 © vasabii/Fotolia
Foto S. 14 © kues1/Fotolia
Foto S. 20 © Maxim Evseev/123 RF
Foto S. 26 © icetray/123 RF
Foto S. 30 © modella/123 RF
Foto S. 32 © Andrius Repsys/123 RF
Foto S. 39 © Tonis Pan/Fotolia
Foto S. 44 © Wavebreak Media Ltd/123 RF
Foto S. 48 © Marko Radunovic/123 RF
Foto S. 52 © peshkov/Fotolia
Foto S. 60 © Christian Delbert/123 RF

Anzeigen:
Simone Fischer, DOAG Dienstleistungen GmbH
Kontakt: anzeigen@doag.org

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

Inserentenverzeichnis

www.ijug.eu

JETZT ABO BESTELLEN

Sichern Sie sich 4 Ausgaben für 18 EUR

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/

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

Java aktuell

+++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN

☐ **Ja**, ich bestelle das Abo Java aktuell – das IJUG-Magazin: 4 Ausgaben zu 18 EUR/Jahr
☐ **Ja**, ich bestelle den kostenfreien Newsletter: Java aktuell – der iJUG-Newsletter

ANSCHRIFT

Name, Vorname

Firma

Abteilung

Straße, Hausnummer

PLZ, Ort

GGF. ABWEICHENDE RECHNUNGSANSCHRIFT

Straße, Hausnummer

PLZ, Ort

E-Mail

Telefonnummer

X

Die allgemeinen Geschäftsbedingungen* erkenne ich an, Datum, Unterschrift

*Allgemeine Geschäftsbedingungen:
Zum Preis von 18 Euro (inkl. MwSt.) pro Kalenderjahr erhalten Sie vier Ausgaben der Zeitschrift "Java aktuell - das iJUG-Magazin" direkt nach Erscheinen per Post zugeschickt. Die Abonnementgebühr wird jeweils im Januar für ein Jahr fällig. Sie erhalten eine entsprechende Rechung. Abonnementverträge, die während eines Jahres beginnen, werden mit 4,90 Euro (inkl. MwSt.) je volles Quartal berechnet. Das Abonnement verlängert sich automatisch um ein weiteres Jahr, wenn es nicht bis zum 31. Oktober eines Jahres schriftlich gekündigt wird. Die Wiederrufsfrist beträgt 14 Tage ab Vertragserklärung in Textform ohne Angabe von Gründen.


JavaLand
28.-30. März 2017 in Brühl
JavaLand 2017
www.javaland.eu