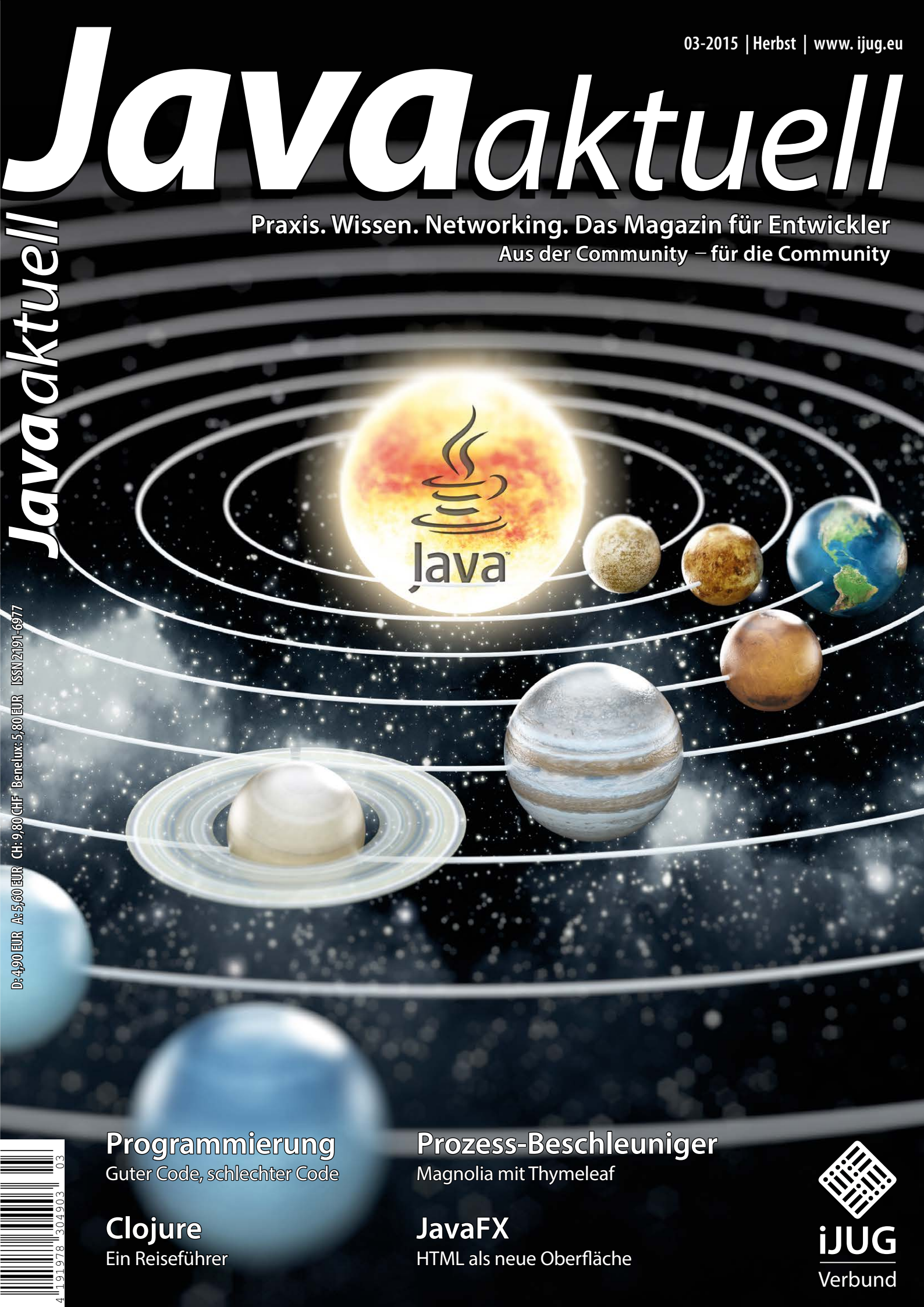
03-2015 | Herbst | www.ijug.eu

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

Programmierung
Guter Code, schlechter Code

Clojure
Ein Reiseführer

Prozess-Beschleuniger
Magnolia mit Thymeleaf

JavaFX
HTML als neue Oberfläche

iJUG Verbund

Wolfgang Taschner
Chefredakteur Java aktuell

http://ja.ijug.eu/15/3/1

Willkommen im Java-Universum

Auf der diesjährigen JavaLand ist mir mal wieder die riesige Bandbreite das Sprache Java bewusst geworden. Das Vortragsprogramm zeigte eine so große Vielfalt an Themen, dass viele Besucher Schwierigkeiten hatten, aus der Vielzahl an interessanten parallelen Vorträgen einen bestimmten auszuwählen. Auch der Gang durch die Community Area spiegelte das gesamte Java-Universum wider. „Coding Dojo", „Architektur Kata" oder „Tinkerforge" waren nur einige der vielen Aktivitäten, die die Community beschäftigte, während die „Early Adopters Area", der „Hackergarten" und das „Innovation Lab" einen Blick in die Zukunft boten.

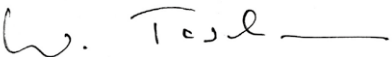
In seiner Keynote ließ Marcus Lagergren von Oracle 20 Jahre Java Revue passieren. Auf einer seiner ersten Folien war unter der Überschrift „Am Anfang war ein Homecomputer" ein Commodore 64 abgebildet. Hier schließt sich für mich ein ganz spezieller Kreis, da ich bereits im Jahr 1984 die Zeitschrift „HC – Mein Homecomputer" gegründet und als Chefredakteur geleitet habe. Auch darin waren, wie heute in der Java aktuell, Listings abgedruckt. Im Gegensatz zu Java war damals jedes Listing nur auf dem dafür vorgesehenen Rechner ablauffähig – das war dann ein Commodore, ein Atari oder ein Sinclair. Die Faszination von damals, ein Programm auf dem Rechner zum Laufen zu bekommen, ist bis heute geblieben. Jeder Entwickler kennt die Spannung, wenn das Programm zum ersten Mal gestartet wird, und die Glücksmomente, wenn es genau das macht, was es soll.

Wie geht es weiter mit Java? Hier ist auch die Community gefragt; jeder kann etwas dazu beitragen, und sei es nur ein kleines Stück. Es besteht die Möglichkeit zur Mitarbeit im JCP. Die „Adopt a JSR"-Initiative ist ein Weg dahin. Zudem bieten die mittlerweile 24 im Interessenverbund der Java User Groups e.V. (iJUG) organisierten Usergroups jede Menge Gelegenheiten, um sich bei einer Veranstaltung mit Gleichgesinnten zu unterhalten und Ideen zu entwickeln.

Übrigens – der Termin für die JavaLand 2016 steht bereits fest: Vom 8. bis 10. März 2016 steht das Phantasialand Brühl wieder im Mittelpunkt des Java-Universums.

In diesem Sinne wünsche ich Ihnen viel Spaß beim Lesen dieser Ausgabe und viel Erfolg bei Ihren Java-Projekten.

Ihr



Kunstprojekt im JavaLand 2015

Seit Java 1.5 erlaubt die Java Virtual Machine die Registrierung sogenannter „Java-Agenten"

Bei Mgnolia arbeiten Web-Entwickler und CMS-Experten mit ein und demselben Quellcode

Ein Heim-Automatisierungs-Projekt

Das Java-Tagebuch

Andreas Badelt, Leiter der DOAG SIG Java

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java – in komprimierter Form und chronologisch geordnet. Der vorliegende Teil widmet sich den Ereignissen im ersten Quartal 2015.

7. Januar 2015

SQL für NoSQL

Klingt paradox, ist aber seit einiger Zeit in Mode. SQL ist nicht totzukriegen. Warum auch? Als Query-Sprache hat es jahrzehntelange gute Dienste geleistet. Wie man Apache Drill zusammen mit HBase und Hive nutzt, um nicht nur JSON-Files, sondern „Big Data" mit SQL abzufragen, erklärt Carol McDonald in einem interessanten Blog-Eintrag.
https://www.mapr.com/blog/how-use-sql-hadoop-drill-rest-json-nosql-and-hbase-simple-rest-client

8. Januar 2015

JSF 2.3: Ersten Milestone ausprobieren und RI Mojarra selber bauen

Die Expertengruppe für Java Server Faces 2.3 hat sich formiert und inzwischen den ersten Meilenstein der Referenz-Implementierung Mojarra fertiggestellt. Ein Schwerpunkt der bisherigen Arbeit ist die CDI-Integration. Wer es schon mal ausprobieren möchte, kann sich Mojarra herunterladen.
https://javaserverfaces.java.net

13. Januar 2015

Reactive Jersey Client

REST-Services sind heute absoluter Standard nicht nur für Web-Frontends; Apache Jersey ist die Referenz-Implementierung und wohl die mit Abstand meistgenutzte Bibliothek für die Spezifikation JAX-RS. Nutzt man die asynchronen Möglichkeiten für REST-Calls und hat aufeinander aufbauende Calls, gerät man aber – nicht nur mit Jersey – schnell in die „Callback-Hölle". Schwer zu lesender Code und verstreute Fehlerbehandlung sind die Folge. Daher ist Jersey jetzt erweitert worden, um „Reactive Programming" zu unterstützen. Gleiche Effizienz wie beim normalen, asynchronen API, aber deutlich besser wartbarer Code sind das Ergebnis. Es werden bereits mehrere „Reactive"-Implementierungen unterstützt, nicht nur für Java 8.
https://blogs.oracle.com/theaquarium/entry/reactive_jersey_client

14. Januar 2015

JSON-P-1.1-Spezifikation gestartet

Wo wir gerade beim Thema „REST" sind: JSON-P(rocessing), die Unterstützung für JSON, das XML als Transport-Encoding verdrängt, wird auch für Java EE 8 aktualisiert. Die anvisierten Hauptthemen sind JSON-Pointer, um Werte innerhalb eines JSON-Dokuments zu referenzieren, und JSON-Patch, um eine Abfolge von modifizierenden Operationen auf einem JSON-Dokument zu definieren (als Unterstützung für die HTTP-Patch-Methode). Die Expertengruppe für JSON-P 1.1 sucht übrigens noch Mitglieder – aber auch externe Unterstützung, etwa im Rahmen von Adopt-a-JSR, wird natürlich begrüßt.
https://jcp.org/en/jsr/detail?id=374

21. Januar 2015

Updates für Java SE 7 und 8

Java SE 8 Update 31 und SE 7 Update 75/76 sind freigegeben worden. Damit sind laut Oracle 169 Sicherheitslücken, fünf davon mit der höchsten Gefahrenstufe nach dem „Common Vulnerability Scoring System", geschlossen worden. Eine der Änderungen: Das inzwischen als unsicher angesehene „SSL v3" ist standardmäßig deaktiviert (und sollte es auch besser bleiben). Oracle klopft sich bei diesem Release selbst auf die Schulter mit der Aussage, dass Java so sicher sei wie schon lange nicht mehr, was ein aktueller Report von Cisco zu bestätigen scheint. Parallel ist auch Java SE Embedded Version 33 freigegeben worden – allerdings ohne Unterstützung für JavaFX! Was werden die Raspberry-Fans wohl dazu sagen?
http://www.oracle.com/technetwork/java/javase/8u31-relnotes-2389094.html

22. Januar 2015

CDI für JPA-Entities?

Per CDI-Objekte in JPA-verwaltete Entities injizieren lassen? Das ist aktuell (aus guten Gründen) nicht möglich und würde wohl mancherorts auch Glaubenskriege auslösen. Für weniger religiöse Pragmatiker mit Lösungs-Bedarf gibt es einen Workaround mit JPA 2.1.
http://hantsy.blogspot.com/2013/12/jpa-21-cdi-support.html

24. Januar 2015

Vaadin, CDI and Java EE

Das Java-Web-Framework Vaadin unterstützt seit Kurzem offiziell Java EE 7 inklusive CDI. Mithilfe von CDI lassen sich Vaadin- und Java-EE-Applikationen leicht verknüpfen. Ein Beispiel dafür, wie die Standardisierung und hier insbesondere CDI das Java-Ökosystem zusammenhalten kann.
https://vaadin.com/javaee

28. Januar 2015

JavaFX-Embedded-Rückzug erläutert

Kevin Rushford und Dalibor Topic von Oracle haben ein bisschen Kontext zur Einstellung von JavaFX Embedded gegeben. In erster Linie war es wohl eine finanzielle Entscheidung, da es mangels Standardisierung schwer war, die große Menge an unterschiedlicher Hardware zu unterstützen. Der gesamte Source Code von JavaFX Embed-

ded ist aber an das OpenJFX-Projekt gegeben worden, ein Unterprojekt des OpenJDK. Damit liegt es nun an der Community – oder auch an Firmen mit speziellen Hardware-Interessen –, JavaFX Embedded für ihre Systeme wie den Raspberry weiterzutreiben. Inwieweit aber die Empfehlung ernst gemeint ist, dass als Alternativen ja ansonsten auch noch Swing und AWT zur Verfügung stehen, vermag ich nicht zu beurteilen.

https://jaxenter.de/ende-javafx-embedded-13340

29. Januar 2015

Der Enterprise Java Newscast

Reza Rahman zeigt in seinem Blog über den „Enterprise Java Newscast" eine zentrale Möglichkeit, um sich über alle möglichen News aus der Java-Welt auf dem Laufenden zu halten. Ich gebe das hier einfach mal weiter an alle, die es noch nicht kennen. Vom Akronym „JSF" nicht in die Irre führen lassen, das ist wohl eher historisch bedingt.

http://blogs.jsfcentral.com/JSFNewscast

10. Februar 2015

Java-EE-7-Maintenance-Release

An Java EE 7 werden nach knapp zwei Jahren einige Wartungsarbeiten durchgeführt. Java EE 7 Rev A dürfte aber nur kleinere Updates der Spezifikations-Dokumente enthalten, keine Änderungen der Referenz-Implementierungen oder TCKs.

https://blogs.oracle.com/theaquarium/entry/java_ee_7_maintenance_release

12. Februar 2015

Kein zweiter Teil für Avatar

Das Projekt Avatar, der Versuch, „Node.js" in die JVM zu bringen, ist (vorerst) beendet. Wie John Clingan auf „blogs.oracle.com" beschreibt, sollten die in Avatar 1 entdeckten Probleme (Kompatibilität mit Node.js, suboptimale Performance durch gleichzeitiges Tunen für Node und Java EE in derselben JVM) mit Avatar 2.0 gelöst werden, auch in der Hoffnung auf breiteres Interesse in der Community. Bei den Arbeiten an 2.0 stellte sich allerdings heraus, dass viele der Services, die man in „Node.js" über Avatar hätte verfügbar machen können, inzwischen RESTful-APIs anbieten, also direkt aus „Node.js" genutzt werden können. Ein Beispiel dafür ist der „Node.js"-Oracle-Treiber *(siehe „https://blogs.oracle.com/opal/entry/introducing_node_oracledb_a_node")*. Dadurch fällt eine Hauptmotivation weg. Aus diesem Grund hat Oracle sich entschieden, das Projekt zumindest vorerst ruhen zu lassen, um die weitere Entwicklung der Technologien abzuwarten. John Clingan betont jedoch, wie sehr die bisherigen Ergebnisse von Avatar bereits an vielen Stellen weitergenutzt werden, etwa in der WebSocket-Implementierung Tyrus oder für Performance-Verbesserungen der JavaScript Engine Nashorn. Oracle verzichtet aber auch ansonsten nicht auf das trendige Thema „Node.js" – es ist inzwischen ein eigener „Node Cloud Service" angekündigt, wie ihn andere Hersteller (RedHat, Microsoft etc.) bereits haben.

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

16. Februar 2015

Byteman, die Allzweckwaffe für harte Probleme

Elegantes Reproduzieren von „Concurrency Issues" anstelle von verzweifelten Brute-Force-Tests? Schnelles Testen der Auswirkungen von Exceptions, die an unerwarteten Stellen geworfen werden? Das alles und noch viel mehr kann Byteman, ein Tool zur Manipulation von Bytecode. Byteman erlaubt die Anwendung von Regeln auf Klassen und Methoden anhand einer einfach zu nutzenden DSL, die zur Laufzeit für die entsprechenden Verhaltensänderungen sorgen, etwa um Threads zu synchronisieren, sodass sie sich garantiert in die Quere kommen. Vorgestellt von Guest-Blogger Jochen Mader im Markus-Eisele-Blog.

http://blog.eisele.net/2015/02/byteman-swiss-army-knife-for-byte-code.html

17. Februar 2015

„Endorsed"-Verzeichnisse für JDK und JRE sollen abgeschafft werden

In Java 8 Update 40 sollen zwei Erweiterungsmöglichkeiten als „deprecated" markiert werden, mit dem Ziel, sie in Java 9 komplett zu entfernen: der „endorsed standard override mechanism" (Platzieren von Libraries unter „<java-home>/lib/endorsed") und der normale „extension mechanism" (Nutzung von „.../jre/lib/ext"). Die Maßnahme steht unter dem Motto „Abbau technischer Schuld", und es wird argumentiert, dass spezifische Libraries heute sowieso eher direkt mit der Applikation gebündelt werden.

http://www.oracle.com/technetwork/java/javase/8u40-relnotes-2389089.html

19. Februar 2015

HTTP/2-Spezifikation praktisch fertig

HTTP/2 ist von der „Internet Engineering Steering Group" (IESG, ein Teil der IETF) akzeptiert worden. Damit sind nur noch ein paar kleine formale Schritte nötig, bevor es als RFC veröffentlicht wird. Dies ist eine wichtige Nachricht für den laufenden Servlet 4.0-JSR, der HTTP/2-Unterstützung als Hauptthema hat.

https://www.mnot.net/blog/2015/02/18/http2

25. Februar 2015

NetBeans, der Allrounder

NetBeans wird immer weiter aufgerüstet. Für PHP- und reine Web-Entwickler wird ja inzwischen allerhand geboten, so auch AngularJS-Unterstützung, was unter anderem Geertjan Wielenga in seinem Blog immer wieder beschreibt. Auch IoT-Entwickler kommen auf ihre Kosten, wie beim jüngsten „Oracle Virtual Technology Summit" präsentiert wurde. Neu ist jetzt auch ein Plug-in für JBoss Forge.

https://blogs.oracle.com/geertjan

3. März 2015

JDK 8 Update 40

Java SE 8 Update 40 ist da. Im Gepäck ist unter anderem eine verbesserte Reaktion auf „Memory Pressure" (sorry, „Speicherdruck" klingt auch nicht viel besser ...), um „Out of Memory"-Fehler auf Systemen mit mehreren laufenden JVMs zu vermeiden. Außerdem ist der JavaFX-Media-Stack für MacOS modernisiert worden und basiert jetzt auf dem neuen AVFoundation-Framework. Damit werden entsprechende Apps nun auch im App Store akzeptiert. Hier ebenfalls hilfreich: Verbesserungen am „Native Packaging", um Applikationen mit plattformspezifischem „Look & Feel" auszuliefern, die kein installiertes JRE benötigen.

http://www.oracle.com/technetwork/java/javase/8u40-relnotes-2389089.html

3. März 2015

Expertengruppe für Java EE Security API legt los

Die Expertengruppe für das neue Java-EE-Securit-API ist praktisch komplett, nur ein Beitrag von IBM ist noch nicht bestätigt. Dann kann es ja jetzt richtig losgehen. Neben der Expertengruppe gibt es natürlich noch weitere Möglichkeiten, an dieser zentralen Spezifikation mitzuwirken, etwa im Rahmen von Adopt-a-JSR.

java.net/projects/adoptajsr

13. März 2015

Neues vom MVC JSR

Die Expertengruppe für MVC hatte vor einiger Zeit entschieden, dass MVC auf der RESTful-Services-Spezifikation JAX-RS basieren soll, da viele Gemeinsamkeiten bestehen und so Redundanzen vermieden werden können. Ebenso soll keine weitere Template-Lösung eingeführt werden, sondern JSPs und Facelets (als offizielle Java-EE-Bestandteile) genutzt werden können. Es sollen aber auch andere Template-Lösungen („View Engines") über ein Service-Provider-Interface ermöglicht werden, wobei die Unterstützung nicht Bestandteil der Referenz-Implementierung Ozark ist. Spec Lead Manfred Riem macht in seinem Blog klar, dass es nur so gehen kann, da es viel zu viel Maintenance-Aufwand bedeuten würde, wenn weitere View Engines Teil der Referenz-Implementierung wären. Durch Community-Beiträge können aber offenbar bereits jetzt FreeMarker, Velocity, Mustache und einige andere genutzt werden.

https://www.java.net/blogs/mriem

18. März 2015

Java-9-Status

Oracle Java Chief Architect Mark Reinhold hat auf der EclipseCon über den Stand von Java SE 9 berichtet. Zentraler Bestandteil ist die Modularisierung (Jigsaw), aufgeschlüsselt sind die neue modulare Struktur für das JDK (Jigsaw) sowie die entsprechende Modularisierung des Source Codes und die Unterstützung in den „Run Time Images". Letztlich soll eine besser skalierbare und sicherere (Reduzierung auf die jeweils benötigten Module = weniger Angriffsfläche) Java-Plattform erreicht werden. Die entsprechenden JDK Enhancement Proposals (JEP) im OpenJDK-Projekt laufen parallel zur Spezifikation JSR-376 innerhalb des Java Community Process.

http://mreinhold.org

19. März 2015

Java EE Management API 2.0 - Experten gesucht

Ein weiterer JSR für Java EE 8, das Management API 2.0, ist zwar schon gestartet, sucht aber noch nach weiteren Mitgliedern für die bislang eher kleine Expert Group. Es geht um eine vollständige Überarbeitung des alten Management-API (JSR-77), basierend auf neueren Protokollen und Patterns wie REST, SSE und möglicherweise WebSockets. Die alte Spezifikation hatte viele Details den jeweiligen Implementierungen in den Containern überlassen, sodass der Nutzen am Ende begrenzt war – was auch verbessert werden soll. Darüber hinaus geht es nicht mehr nur um Management und Monitoring, sondern auch das Deployment von Artefakten soll im Rahmen der Spezifikation behandelt werden.

https://jcp.org/en/jsr/detail?id=373

25. März 2015

Das JavaLand ist voller Leben

Die Konferenz „von der Community für die Community" im Phantasialand bei Köln geht in den zweiten Tag und es ist wunderbar anzuschauen, wie viele Aktivitäten es hier gibt und dass mit tausend Besuchern noch mehr gekommen sind als letztes Jahr. In der Community Hall vermischen sich neugierige Java-Anfänger, alte Hasen und die Experten, die man sonst meist nur weit weg auf der Bühne einer der großen Konferenzen sieht. Die Gadget-Freunde kommen im Innovation Lab und in der Tinkerforge-Ecke auf ihre Kosten; wer einen Blick in die Zukunft des OpenJDK und der diversen Java-Spezifikationen werfen möchte, der kommt in die „Early Adopters Area" oder den Hackergarten und diskutiert oder programmiert dort mit den Experten. Das alles hat einiges an Vorbereitung erfordert – aber es hat sich gelohnt. Die Zukunft von Java liegt (für zwei Tage) im Phantasialand. Der Termin für das kommende Jahr steht auch schon fest: 8. bis 10. März 2016.

http://www.javaland.eu/javaland-2015

26. März 2015

Java-7-Support läuft aus

Der öffentliche Support für Java 7 läuft im April aus. Wer keinen kommerziellen Supportvertrag mit Oracle hat, sollte also für kritische Applikationen schnellstmöglich auf Java 8 migrieren, wenn nicht schon geschehen. Details zu den Zeitplänen hat Oracle gerade bekanntgegeben.

https://blogs.oracle.com/java-platform-group/entry/future_updates_of_java_7

2. April 2015

Erster MVC-Entwurf schon fertig

Die MVC-Spezifikation befindet sich jetzt schon im „Early Draft Review", als erste Java-EE-8-Spezifikation. Das bedeutet hoffentlich jede Menge frühzeitiges Feedback aus der Community.

https://jaxenter.de/mvc-1-0-early-draft-review-phase-18037

Andreas Badelt
Leiter der DOAG SIG Java

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

http://ja.ijug.eu/15/3/2

Write once – App anywhere

Axel Marx, Tricept Informationssysteme AG

Die Entwicklung mobiler Anwendungen ist ein stetig an Bedeutung zunehmender Bereich. Vor allem Cross-Platform-Ansätze sind für die Entwicklergemeinde interessant, um mit (hoffentlich) geringem Aufwand Anwendungen für ein möglichst breites Gerätespektrum anbieten zu können. Dieser Artikel zeigt Möglichkeiten, mobile Anwendungen soweit möglich mit Java-Know-how zu entwickeln, ohne sich tiefer mit JavaScript- oder HTML5-Programmierung auseinandersetzen zu müssen, und richtet sich an Java-Entwickler, für die das Schlagwort „mobile Entwicklung" mehr bedeutet als die bloße Gestaltung von GUIs oder die Erstellung mobiler Websites.

Der Autor ist seit vielen Jahren in der Entwicklung von Client-Server-Systemen tätig und beschäftigt sich dabei mit dem Entwurf und der Implementierung sowohl der Oberflächen als auch der Fachlogik mithilfe von Java-Technologien. Ausgehend von diesem Hintergrund sowie der wachsenden Bedeutung mobiler Anwendungen und der Frage „Nativer oder Cross-Platform-Ansatz bei der Entwicklung?" mit ihren unterschiedlichen Möglichkeiten der Umsetzung, stellte sich die Frage, inwieweit (und ob) Plattformunabhängige Entwicklung mit Java überhaupt möglich ist.

Wer sich als Java-Entwickler mit dem Gedanken trägt, seine Java-Kenntnisse zur Entwicklung mobiler Anwendungen einzusetzen und dabei möglichst ohne tiefere Kenntnisse in den Standard-Techniken der mobilen Entwicklung wie HTML5, CSS oder JavaScript auszukommen zu wollen, der sollte sich zuerst ein-

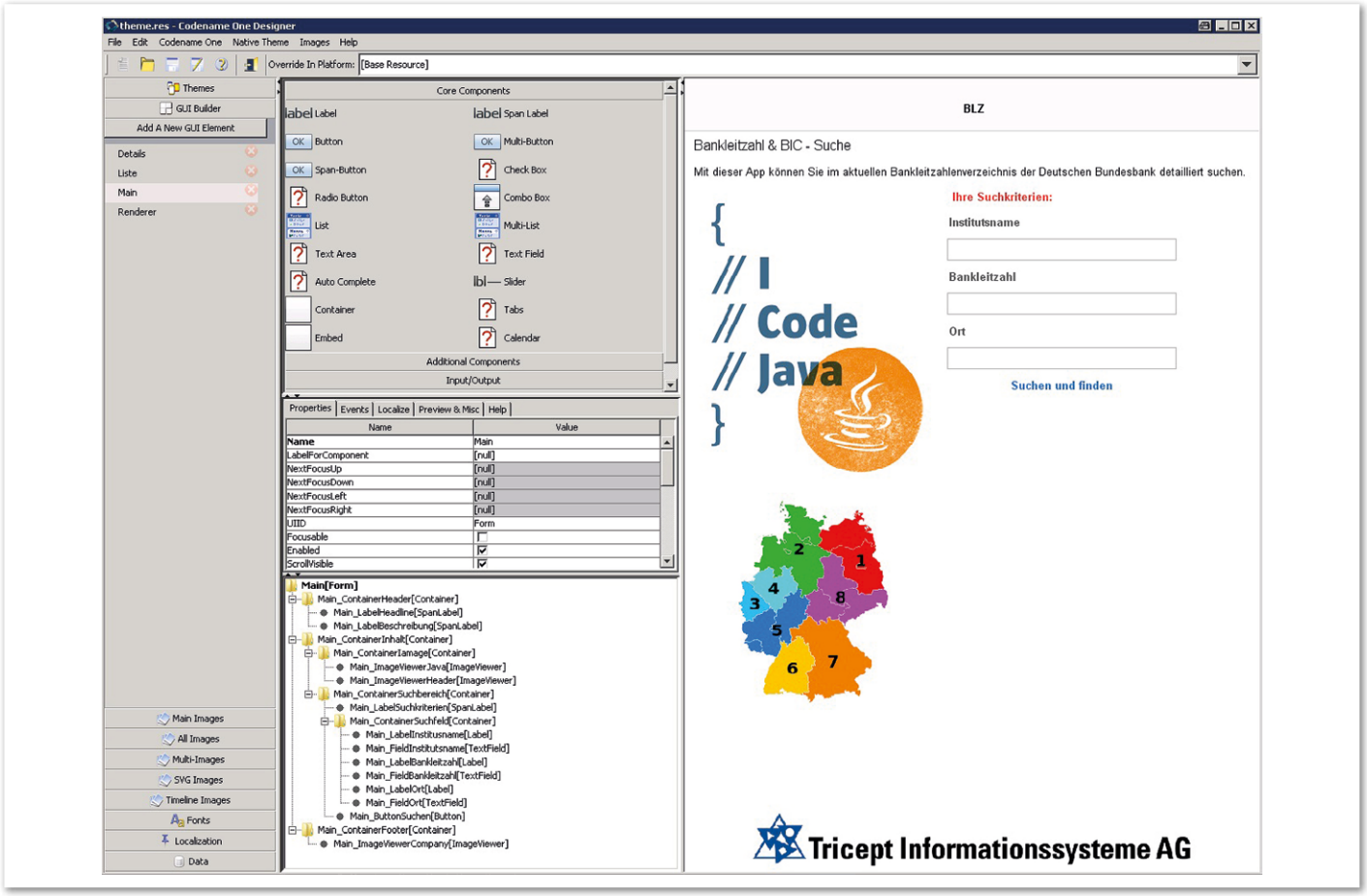


Abbildung 1: Die App in der Entwicklungsumgebung

```
140510001Sparkasse Mecklenburg-Nordwest   23951Wismar, Meckl      Spk Mecklenburg-Nordwest   51499NOLADE21WIS20048818U000000000
140510002Sparkasse Mecklenburg-Nordwest   19217Rehna             Spk Mecklenburg-Nordwest   51499      20048244U000000000
140510002Sparkasse Mecklenburg-Nordwest   23996Bad Kleinen       Spk Mecklenburg-Nordwest   51499      20048824U000000000
140510002Sparkasse Mecklenburg-Nordwest   23946Ostseebad Boltenhagen Spk Mecklenburg-Nordwest 51499      20048825U000000000
140510002Sparkasse Mecklenburg-Nordwest   23942Dassow            Spk Mecklenburg-Nordwest   51499      20048826U000000000
140510002Sparkasse Mecklenburg-Nordwest   23972Dorf Mecklenburg  Spk Mecklenburg-Nordwest   51499      20048827U000000000
140510002Sparkasse Mecklenburg-Nordwest   19205Gadebusch         Spk Mecklenburg-Nordwest   51499      20048828U000000000
```

Abbildung 2: Auszug aus der XML-Datei mit den Bank-Informationen

mal einen Überblick über den aktuellen Stand der Technik und die am Markt verfügbaren Werkzeuge und ihre Eigenarten verschaffen.

Abhängig von den zur Wahl stehenden Werkzeugen gibt es verschiedene Ansätze, sich dem Thema zu nähern. Während mittlerweile fast jeder Anbieter die gängigen Entwicklungsumgebungen wie Eclipse, NetBeans und IntelliJ IDEA mehr oder weniger gut unterstützt, liegen die Unterschiede dabei vor allem im Aufbau und in der Struktur der erzeugten App (kompilierter Binärcode zur Installation auf dem Zielgerät oder auf einen Server ausgelagerte Fach- und Darstellungslogik mit einem Viewer auf dem Mobilgerät für das Rendering der GUI) sowie in den verwendeten Techniken zur Umsetzung von UI und Fachlogik (ohne oder mit unterschiedlich hohen Anteilen an HTML5, CSS und JavaScript). Zunächst ein Überblick über die betrachteten Entwicklungswerkzeuge.

Codename One

Das Tool erlaubt die Entwicklung von Standalone-Clients in Java, ohne dass weitere Arbeiten mit HTML5, CSS oder JavaScript erfolgen müssen, und ist als Plug-in für alle gängigen Entwicklungsumgebungen wie Eclipse, NetBeans oder IntelliJ IDEA erhältlich. Sowohl Fachlogik als auch Oberflächenelemente werden in Java entwickelt, wobei für die Erstellung der UI neben der Verwendung von Java 5 SE auch ein eigenes GUI-Builder-Tool zur Verfügung steht. Ein gesonderter Build-Server generiert dann je nach Ziel-Betriebssystem eine native App, die direkt auf dem Endgerät installiert oder über den Apple App Store sowie den Google Play Store bereitgestellt werden kann. Der Java-Code wird dabei über den Device-Simulator von Codename One überwacht.

Tabris

Im Gegensatz zu Codename One bekommt man bei Tabris keine für ein bestimmtes Betriebssystem kompilierte native App. Stattdessen entwickelt man eine Server-Anwendung in Java, die auf dem Device durch einen von Tabris mitgelieferten Viewer generisch visualisiert wird. Die komplette Oberflächensteuerung liegt dabei auf dem Server. Tabris erweitert die Ecplise Remote Application Platform (RAP) und lässt die in Java implementierte Fachlogik in einem Servlet ausführen.

Auf dem Endgerät erfolgt lediglich das Rendering des mithilfe des Java SWT erstellten GUI durch den Viewer. Device und Server kommunizieren hierbei via HTTPS

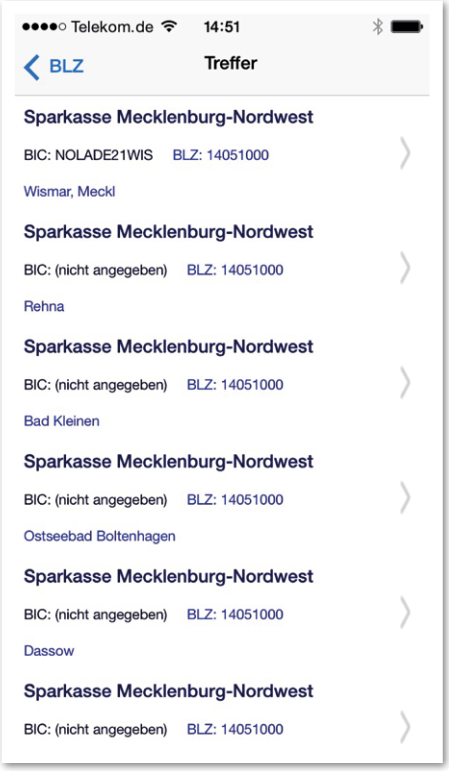


Abbildung 3: Suchergebnis, das dem Suchkriterium „Institutsname=Nord" entspricht

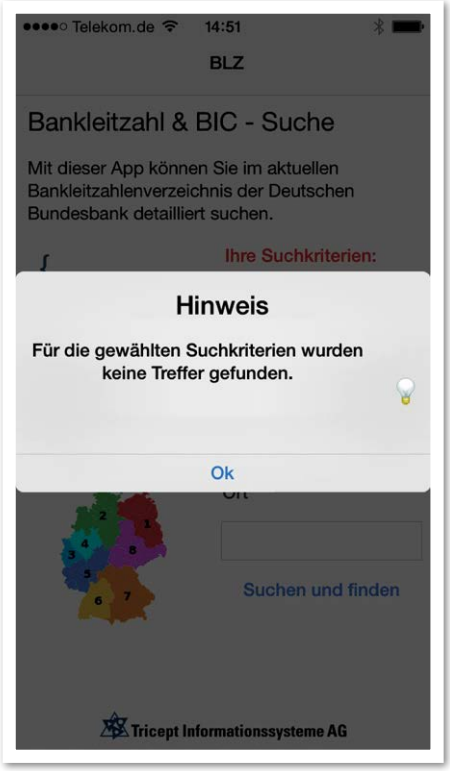


Abbildung 4: Meldung bei fehlgeschlagener Suche

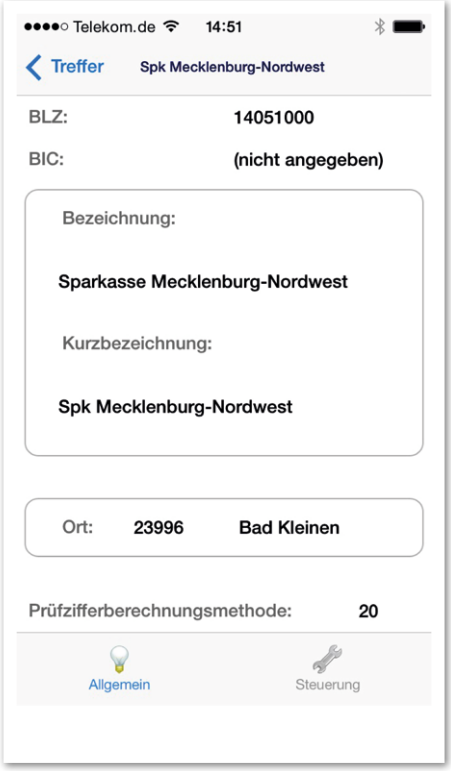


Abbildung 5: Seite „Allgemein"

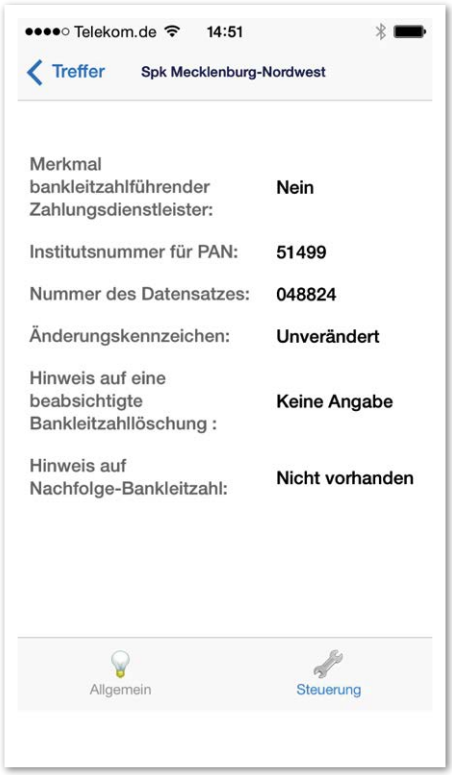


Abbildung 6: Seite „Steuerung"

Abbildung 7: Lage der „BLZ.xml" im Ressourcen-Verzeichnis für iOS

```
// Android, Simulator
com.codename1.io.BufferedInputStream stream =
  new com.codename1.io.BufferedInputStream(
    com.codename1.io.FileSystemStorage.getInstance().
      openInputStream("/BLZ.xml"));

// iOS
java.io.InputStream stream =
  com.codename1.ui.Display.getInstance().
    getResourceAsStream(getClass(), "/BLZ.xml");

java.io.InputStreamReader streamReader =  new java.io.InputStreamReader(stream);

com.codename1.xml.XMLParser  parser = new com.codename1.xml.XMLParser();

this.start(parser.parse(streamReader));      // public void start(com.codename1.xml.Element e) { ... }
```

Abbildung 8: Java-Code für Android und iOS zur Verwendung externer Ressourcen

```
protected void onAction_ButtonSuchen(com.codename1.ui.Component newCompo-
nent, com.codename1.ui.events.ActionEvent newEvent) {
if (this.find_FieldInstitutsname() != null) {
this.setData_institutsname(
this.find_FieldInstitutsname().getText());
}

// Code für BLZ hier analog
  // Code für Ort hier analog

     int treffer = 0;

for (int i = 0; i < this.getInstitutListe().size(); i++) {
de.tricept.codenameone.blz.Institut institut = this.getInstitutListe().
get(i);

            if (this.getData_institutsname() != null) {
if (institut.getBezeichnung().toUpperCase()
.indexOf(this.getData_institutsname()
.toUpperCase()) < 0) {
                        continue;
                }
        }

            // Code für BLZ und Ort auch hier analog zum
// Institutsnamen

treffer++;
} //for

if (treffer == 0) {
// Keine Treffer
            com.codename1.ui.Image image = null;
try {
com.codename1.ui.util.Resources res = com.codename1.ui.util.Resources.
open("/theme.res");
                image = res.getImage("lightbulb_48.png");
} catch (java.io.IOException ex) {
            // do nothing
     }

com.codename1.ui.Dialog.show(
this.convert("Hinweis"),
            this.convert("Für die gewählten Suchkriterien wurden keine
Treffer gefunden."),
com.codename1.ui.Dialog.TYPE_INFO, image, this.convert("Ok"),
null);
} else {
// Liste anzeigen
this.showForm("Liste", null);
}
}
```

Listing 1

und JSON miteinander, wobei der Viewer Benutzer-Eingaben an den Server sendet und von diesem im Gegenzug Anweisungen zum Rendering der Oberflächen-Elemente erhält und diese umsetzt.

Oracle ADF mobile und Oracle MAF

Eine weitere Möglichkeit stellen das Framework „ADF mobile" beziehungsweise dessen Nachfolger Mobile Application Framework (MAF) dar. Nachdem ADF mobile nur für den JDeveloper 11 verfügbar war und lediglich Unterstützung für Java 1.4 ME bot, ist es mit dem MAF nun möglich, neben dem JDeveloper ab Version 12.1.3 auch die Eclipse IDE zusammen mit dem Oracle Enterprise Pack For Eclipse 12.1.3.1 zu verwenden und Unterstützung für Java 8 zu erhalten.

ADF mobile und MAF unterstützen sowohl Browser-basierte Anwendungen als auch die Entwicklung von Apps, bei denen die Fachlogik in einer eigenen mitinstallierten JVM direkt auf dem Endgerät ausgeführt wird. Die Daten werden zur Visualisierung an die Darstellungsschicht übermittelt, die dann das Rendering mithilfe von HTML5 und JavaScript übernimmt. Die einzelnen Anwendungskomponenten lassen sich dabei auf unterschiedliche Weise realisieren:

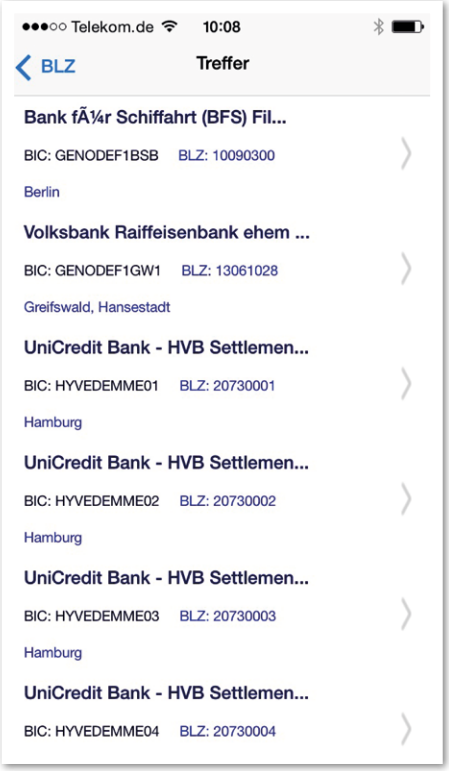


Abbildung 9: Fehlerhafte Darstellung der Umlaute im Namen des ersten Such-Ergebnisses

- Deklarativ unter Verwendung von HTML und JavaScript
- Lokales HTML
- Remote URL (Server-HTML)

Die App „Bankleitzahlensuche" in Java

Um ein Gefühl für die App-Entwicklung mit Java zu bekommen, hat der Autor als Beispiel eine Anwendung mit Codename „One" zur Suche von Bankdaten entwickelt. Die Entscheidung für Codename „One" als Entwicklungswerkzeug war hierbei hauptsächlich dadurch bedingt, dass sich uns hier die Möglichkeit bot, sowohl GUI als auch Fachlogik wie gewünscht komplett in Java zu entwickeln. Zum Installationsumfang der App gehört eine aktuelle Bankleitzahlen-Datei der Deutschen Bundesbank im XML-Format, die die Bank-Informationen zur Suche bereitstellt. *Abbildung 1 und 2* zeigen den Startbildschirm der App in der Codename-One-IDE sowie einen formatierten Auszug aus der BLZ-Datei mit den in der App angezeigten Informationen.

Die Funktionsweise der App ist denkbar einfach und intuitiv. Abhängig von den eingegebenen Such-Parametern liefert die Suche ein für den Benutzer aufbereitetes Ergebnis zurück *(siehe Abbildung 3)*, das in Form einer Übersicht angezeigt wird.

```
protected boolean onMainCommand1() {
    this.formSettings =
    (com.codename1.ui.Form) createContainer(fetchResourceFile(), "Settings");

    com.codename1.ui.Command[] cmds =
    new com.codename1.ui.Command[] {
      // Die Zeichenkette "Übernehmen" wird falsch dargestellt.
      new com.codename1.ui.Command( ("Übernehmen"),
        userclasses.Const.COMMAND_ID_OK),
      new com.codename1.ui.Command( ("Abbrechen"),
        userclasses.Const.COMMAND_ID_CANCEL) };

    formSettings.setDefaultCommand(cmds[0]);
    ...
}
```

Abbildung 10: Umlaute korrekt (grün) beziehungsweise fehlerhaft dargestellt (rot)

Schlägt die Suche fehl, wird dem Benutzer eine entsprechende Meldung ausgegeben *(siehe Abbildung 4)*.

Nach Auswahl eines Suchergebnisses wird eine Seite mit allgemeinen Angaben zum gewählten Kreditinstitut angezeigt. Eine weitere Seite enthält Detail-Informationen zur Steuerung des Zahlungsverkehrs (siehe *Abbildung 5 und 6)*. Hierbei wandelt die Fachlogik der App die in der Datei enthaltenen und teilweise kryptischen Werte in für den Anwender lesbaren Klartext um.

Der Java-Code

Listing 1 zeigt den Java-Code, der die Suche durchführt. Dabei wird jeder Eintrag der XML-Datei auf jedes angegebene Suchkriterium (Name, BLZ, Ort) geprüft und dem Suchergebnis nur im Falle einer entsprechenden Übereinstimmung hinzugefügt. Bei einer erfolglosen Suche wird die Nachricht aus *Abbildung 4* erzeugt und angezeigt.

Auffälligkeiten während der Entwicklung

Wie zu erwarten, läuft bei der Entwicklung natürlich nicht immer alles sofort nach Plan. Abhängig von der Ziel-Plattform muss an verschiedenen Stellen eingegriffen werden, um externe Ressourcen wie etwa die XML-Datei in unserem Beispiel einzubinden. Hier sind manuelle Anpassungen im Java-Code notwendig, um den Unterschieden zwischen den Betriebssystemen Android und iOS Rechnung zu tragen.

Während die Datei für Android an beliebiger Stelle im Dateisystem liegen kann, um mit Java eingebunden und verwendet zu werden, so ist für eine iOS-App zu beachten, dass verschachtelte Verzeichnis-Strukturen hier nicht zulässig sind. In diesem Fall müssen alle Ressourcen direkt im Root-Verzeichnis der App liegen, damit sie für die App verfügbar sind *(siehe Abbildung 7)*. *Abbildung 8* zeigt den unterschiedlichen Java-Code, mit

```
// Wird vom XML-Parser aufgerufen
private java.lang.String getValue(com.codename1.xml.Element element) {
    return this.convert(element.getText());
}


private java.lang.String convert(java.lang.String text) {
    try {
        // Konvertierung in Unicode
        return new java.lang.String(text.getBytes(), "UTF-8");
    } catch (java.io.UnsupportedEncodingException ex) {
        return text;
    }
}
```

Abbildung 11: Konvertierungsroutine für den XML-Parser

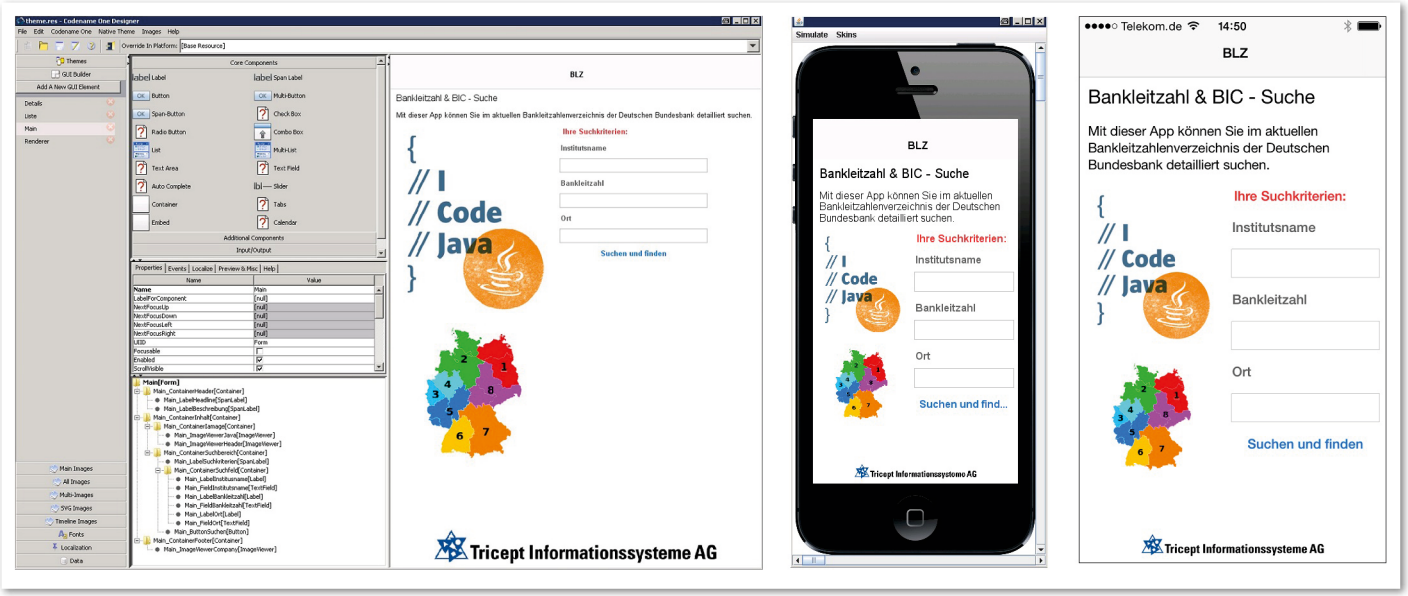


Abbildung 12: App im GUI-Builder, im Simulator und auf dem Gerät

dem die Bankleitzahlen-Datei unter Android beziehungsweise iOS angesprochen wird.

Ein weiteres Problem bei der Entwicklung ist das unterschiedliche Verhalten im Umgang mit Umlauten. Während eine iOS-App in Java-Strings verwendete Umlaute und solche, die über den GUI-Editor direkt in die Properties geschrieben werden, korrekt anzeigt, werden Umlaute aus der XML-Datei fehlerhaft dargestellt *(siehe Abbildung 9 und 10)*.

Abhilfe schafft hier die explizite Konvertierung der Texte in UTF-8. Die in *Abbildung 11* vorgestellte Konvertierungsroutine wird auch in *Listing 1* verwendet, um am Ende den Text des Nachrichtenfensters korrekt auszugeben. Neben den gerade erwähnten Punkten ist es genauso unerlässlich, während des gesamten Entwicklungsprozesses immer wieder das „Look&Feel" der App zu überprüfen. So variiert das Erscheinungsbild mitunter recht stark, wenn man sich die App in der Entwicklungsumgebung, im Simulator oder als fertige Version auf dem physischen Gerät selbst anschaut. Auch zwischen Android und iOS sind Unterschiede zu beobachten und entsprechend zu berücksichtigen.

Auftretende Abweichungen zwischen dem gewünschten Endlayout und der Ansicht im GUI-Builder darf man nicht aus dem Auge verlieren und muss diese immer wieder manuell in zum Teil recht anstrengender Kleinarbeit korrigieren. *Abbildung 12* zeigt, wie sich das Layout der Beispiel-Anwendung von Fall zu Fall unterscheidet.

Fazit

Abschließend lässt sich sagen, dass es bereits gute Ansätze und Möglichkeiten gibt, mobile Anwendungen in Java zu entwickeln. Man sollte dabei aber immer im Hinterkopf behalten, dass man immer noch Unzulänglichkeiten und Hindernisse in Kauf nehmen muss. Da der gesamte Bereich einem stetigen Wandel und schnell fortschreitender Weiterentwicklung unterworfen ist, verbessern sich die Werkzeuge allerdings permanent. Dies stimmt zuversichtlich im Hinblick darauf, dass auch künftig der Frustrationsfaktor sinken und der Motivationsfaktor steigen wird und die bequeme Entwicklung mobiler Anwendungen mit Java sich hier einen festen Platz erarbeitet. Bis es jedoch soweit ist, heißt es weiterhin: Write once – test everywhere!

Axel Marx
axel.marx@tricept.de

Axel Marx studierte Mathematik und Informatik an der TU Braunschweig und ist seit mehr als fünfzehn Jahren in der Anwendungsentwicklung tätig. Dabei sammelte er Erfahrungen auf dem Gebiet klassischer Client-Server-Systeme und in der Koordination und Betreuung von Software-Migrationsprojekten im In- und Ausland. Zurzeit betreut er für die Tricept Informationssysteme AG Kunden im Java-Umfeld, hauptsächlich aus den Bereichen „Banking" und „Automotive". Seine besonderen Interessen liegen in der Datenbank- und Mobile-Entwicklung.

http://ja.ijug.eu/15/3/3

Java für Mac: Ärger über Adware und wie man sie umgeht

Laut Berichten von heise und zdnet reichert Oracle Java für den Mac inzwischen mit einer Adware an. User und Community sind verärgert. Allerdings kann man dies am Oracle JDK 8 Update 40 umgehen. Dafür muss der „Java Control Panel" aufgemacht werden.

Unter „Advanced" und „Miscellaneous" ist die Einstellung „Suppress sponsor offers when installing or updating Java" zu finden. Damit kann mögliche Werbung reduziert werden.

Mach mit: partizipatives Kunstprojekt im JavaLand 2015

Wolf Nkole Helzle

10.000 Bilder wurden es nicht, allerdings sind die Macher stolz, dass es der vielbeschäftigen Community neben allen anderen Aktivitäten auf der diesjährigen JavaLand dann doch gelungen ist, rund 2.000 ihrer Eindrücke über Smartphones auf den myMatrix-Server hochzuladen – trotz oft fehlender Internet-Verbindung und schlechten WLANs.

Das fertige 3D-Mosaik *(siehe nächste Seite)* wurde ab Mittwoch-Nachmittag im Großformat auf die Wand in der Ausstellerhalle projiziert und jeder konnte sowohl seine eigenen Bilder erkennen, als auch das Gesamtbild, das sich daraus ergibt. Es zeigte sich, dass eine künstlerische Intervention bei derlei Veranstaltungen eine Wahrnehmungsebene einzieht, die sonst nicht unbedingt vorkommt und die als Bereicherung wahrgenommen wird. Manch einer meinte gar, durch die Beanspruchung der anderen Gehirnhälfte revitalisiert zu werden …

Interessant in diesem Zusammenhang ist auch, wie dieses Projekt überhaupt zur JavaLand eingeladen wurde: Als die Inter-Face AG in Unterhaching bei München den Medienkünstler Wolf Nkole Helzle im Jahr 2013 beauftragte, ein Medien-Kunstwerk zu entwickeln und dieses anlässlich ihres 30-jährigen Firmenjubiläums im Jahre 2014 uraufzuführen, ahnte noch niemand eine Erfolgsgeschichte dahinter. Frank Schütz und die Studenten der TU München, die er im IF-Lab betreut, waren zusammen mit dem Künstler ein Jahr mit der Entwicklung der Software beschäftigt.

Die technische Grundlage für die Verwaltung und Erstellung des Kunstwerks bildet eine JEE-Web-Anwendung auf GlassFish mit PostgreSQL. Die Admin-Anwendung sollte sowohl lokal unter Windows lauffähig sein als auch über Internetzugang für externe Uploads des Publikums bereitstehen. Durch die Generierung eines Einzelprojekts entstand eine HTML5-Anwendung mit webGL für die Darstellung.

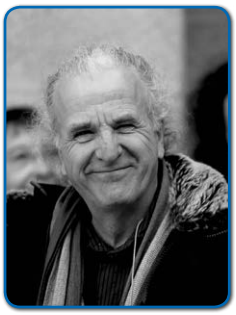
Zum Firmenjubiläum wurden lokal Tausende Bilder aus der Firmengeschichte hochgeladen, hinzu kamen Bilder von der Veranstaltung im Stadion des FC Unterhaching. Die TU München stellte ihren riesigen Touchscreen zur Verfügung, sodass Kunden, Mitarbeiter und Gäste durch die Geschichte des Software-Unternehmens surfen konnten.

Eingeladen, das Projekt beim Innovation LAB in Nürnberg vorzustellen, wurde gleich vor Ort mit André Sept von der DOAG die Frage diskutiert, ob sich das nicht auch für das Innovation LAB auf der JavaLand anbieten würde. Nach Abstimmung mit den Veranstaltern wurden die Umgebungs-Bedingungen gecheckt und nachdem alles gut passte, ging es auf der JavaLand 2015 los.

Der Social-Media-Künstler Wolf Nkole Helzle beschäftigt sich hauptsächlich mit dem Thema „das Eine und die Vielen". Er hat in der Vergangenheit bereits einige partizipative Arbeiten entwickelt, in denen Besucher, Gäste, Mitarbeiter oder Passanten Teil eines Kunstwerks werden.

MyMatrix *(siehe „http://my-matrix.org")* ist sein neuestes Werk, das nach der JavaLand anlässlich eines Festivals in Luxemburg im Mai aufgeführt und im September am College of Art and Design der TU Beijing in China Premiere feiern wird. Auch dort wird es eine Zusammenarbeit mit Studenten geben; die Idee ist, dass jeder der rund 1.600 Studenten 10 Fotos aus seiner Heimat (Familie etc.) einbringt *(siehe „www.helzle.com")*.

Wolf Nkole Helzle
mail@helzle.com



Wolf Nkole Helzle (geb. 1950) studierte Malerei an der Freien Kunstschule in Stuttgart und Malerei/Plastik an der Hochschule für bildende Künste in Kassel bei Prof. Harry Kramer. Danach arbeitete er zwanzig Jahre in der Hard- und Software-Industrie. Seit 1996 ist er freischaffender Medienkünstler. Im Jahr 2000 folgte die Dozentur für Medienkunst an der Fachhochschule für Gestaltung Schwäbisch Hall und 2006/2007 erhielt er ein Atelierstipendium Künstlerhaus Stuttgart, Lehrauftrag Videokunst, Hochschule der Medien, Stuttgart. Im Jahr 2012 kam die Förderung durch den Digital Content Fund der Medien- und Filmgesellschaft Baden-Württemberg. Wolf Nkole Helzle ist Mitglied im Deutschen Künstlerbund.

http://ja.ijug.eu/15/3/4

Die JavaLand-Community war gefragt

Wir freuen uns sehr, dass der bekannte Social-Media-Artist Wolf Nkole Helzle dieses Jahr zu Gast im JavaLand war. Er brachte uns sein (mit Java entwickeltes) neuestes Kunstprojekt mit, das auf den Namen „myMatrix" hört: ein interaktives, Community- und Server-basiertes 3D-Mosaik. So war jeder der 1.000 Teilnehmer aufgefordert, sich ein Bild von der diesjährigen JavaLand zu machen und deshalb alle persönlich wichtigen

10 bis 20 Erlebnisse und Eindrücke (wie beispielsweise Räume, Menschen, Situationen, Neuigkeiten, Überraschungen etc.) am ersten Tag per Smartphone aufzunehmen und auf den myMatrix-Server hochzuladen. Daraus entstand ein Mosaik, dessen Entwicklung im Bereich des Innovation LAB verfolgt werden konnte; ab 12 Uhr des zweiten Tages gab es eine große Beamer-Projektion des Ergebnisses samt eines Touchscreens zum individuellen Navigieren in der „JavaLandCommunityBilderwelt".

Aspektorientiertes Programmieren mit Java-Agenten

Rafael Winterhalter, Bouvet ASA

Aspektorientiertes Programmieren ist ein mächtiges Werkzeug, das jedoch nur wenige Programmierer nutzen. Seit Java 1.5 erlaubt die Java Virtual Machine die Registrierung sogenannter „Java-Agenten", die einen natürlichen Zugang zur aspektorientierten Programmierung mit Java bieten. Der Artikel wirft einen genaueren Blick auf solche Agenten und zeigt, wie sich Aspekte durch Bytecode-Manipulation einfach in ein Java-Programm integrieren lassen.

Obwohl den meisten Software-Entwicklern aspektorientierte Programmierung ein Begriff ist, spielt dieser Lösungsansatz im Java-Ökosystem eine eher untergeordnete Rolle – nicht zuletzt, weil viele Entwickler Sprach-Erweiterungen zur aspektorientierten Programmierung als zu umständlich empfinden. Denn selbst für triviale Anwendungsfälle erfordern diese Toolkits häufig Plug-ins für Entwicklungsumgebung und Build-Tool, da der Java-Compiler selbst keinen Aufsatzpunkt zur Implementierung von Aspekten bietet. Noch dazu ist die Sprache der „Pointcuts" und „Join Points", die in der aspektorientierten Programmierung üblich sind, vergleichsweise abstrakt und erfordert einige Einarbeitungszeit.

Im Kontrast zum Java-Compiler bietet die Java-Laufzeit-Umgebung, die Java Virtual Machine, allerdings einen natürlichen Ansatz zur Umsetzung von Aspekten in einer Java-Anwendung. Bereits seit Java 1.5 erlauben sogenannte „Java-Agenten" das dynamische Umschreiben von Java-Klassen und -Methoden sogar während der Laufzeit einer Java-Anwendung. Agenten werden dabei beim Start einer solchen Anwendung auf der Kommandolinie angegeben und sind ähnlich zu einer Bibliothek anschließend auf dem Klassenpfad verfügbar. Jeder Agent definiert dabei aber zusätzlich noch eine „premain"-Methode, die – wie der Methoden-Name bereits nahelegt – vor dem Start der „main"-Methode, der eigentlichen Java-Anwendung, aufgerufen wird. *Listing 1* zeigt, wie ein einfacher Agent damit implementiert werden könnte.

Würde die oben stehende Java-Klasse mit einer entsprechenden Manifest-Datei in eine „.jar"-Datei verpackt und mithilfe des Parameters „-javaagent" zu einer Anwendung hinzugefügt, dann wäre der Konsolenauswurf aus der „premain"-Methode also noch vor dem Start der eigentlichen Anwendung zu lesen.

Klassen durch Java-Agenten zur Laufzeit verändern

An sich ist der genannte Ansatz noch nicht sonderlich bemerkenswert, denn ein ähnliches Ergebnis ließe sich auch durch einfaches Aneinanderreihen von Methoden-Aufrufen erzielen. Ihren eigentlichen Wert entfalten Java-Agenten erst nach Definition eines zweiten, optionalen Methoden-Parameters vom Typ „Instrumentation". Instanzen dieses Interface erlauben es, Einfluss auf das Klassen-Laden in einer Java-Anwendung zu nehmen. So ist es beispielsweise möglich, den Klassenpfad auch noch nach Start einer Anwendung mit zusätzlichen „.jar"-Dateien zu erweitern. Sogar der Bootstrapklassen-Lader, der eigentlich nur Java-Basisklassen wie „String" oder „Object" lädt, kann auf diesem Weg mit weiteren Einträgen versehen werden.

Das wohl mächtigste Werkzeug, das durch das „Instrumentation"-Interface zugänglich gemacht wird, ist allerdings die Möglichkeit, einen „ClassFileTransformer" zu registrieren. Instanzen dieses Interface erlauben es, jegliche Klasse, die während der Ausführung eines Java-Programms geladen wird, beliebig zu verändern, noch bevor diese Klasse in das Programm geladen wird. Auch AspectJ und andere Sprach-Erweiterungen zur aspektorientierten Programmierung haben diese Möglichkeit für sich entdeckt, um sogenanntes „Load Time Weaving" als eine Alternative zum traditionellen „Build Time Weaving" umzusetzen. Zumindest das Umschreiben der Klassen kann damit zur Laufzeit erfolgen.

Um Nutzern eine Möglichkeit einzuräumen, eine bereits kompilierte Java-Klasse zur Laufzeit eines Java-Programms zu verändern, wird ein „ClassFileTransformer" jedes Mal aktiviert, wenn eine Klasse in einer Java-Anwendung durch einen „ClassLoader" geladen wird. Dies geschieht in der Regel bei der ersten Verwendung der Klasse, also wenn eine Zeile eines Java-Programms ausgeführt wird, in der diese Klasse zum ersten Mal genutzt wird. Daraus folgt auch, dass eine Klasse, die niemals benutzt wird, auch niemals geladen oder durch einen „ClassFileTransformer" verändert wird. Das hat den Vorteil, dass ein Java-Agent den Start eines Programms nicht erheblich verzögert, da unter anderem das Umschreiben einer Klasse nur bei Bedarf ausgeführt wird.

Die zu verändernde Java-Klasse wird einem „ClassFileTransformer" dabei in kompilierter Form als Byte-Array als ein Argument der einzigen Methode des Interface übergeben. Als Rückgabewert selbiger Methode erwartet der Agent dieselbe Klasse in der gewünschten angepassten Form, ebenfalls in Form eines Byte-Arrays, das die kompilier-

```
public class MeinAgent {
  public static void premain(String args) {
     System.out.println("Hallo! Ich bin ein Agent.");
  }
}
```

Listing 1

```
class NoOpClassFileTransformer implements ClassFileTransformer {
  @Override
  public byte[] transform(ClassLoader classLoader,
            String className,
            Class<?> classBeingRetransformed,
            ProtectionDomain protectionDomain,
            byte[] classFileBuffer) {
     return classFileBuffer;
  }
}
```

Listing 2

te Ausgabeklasse repräsentieren soll. Jede kompilierte Klasse wird dabei durch sogenannten „Java Bytecode" repräsentiert und nicht mehr durch den vielen Entwicklern gut bekannten Java-Quelltext.

Tatsächlich kann die Java Virtual Machine, trotz ihres Namens, nichts mit Java-Quelltextdateien anfangen. Die Programmiersprache Java ist durch den Java-Compiler vollständig vor der virtuellen Maschine verborgen, die einzig und allein Java-Bytecode ausführen kann. Nicht zuletzt deswegen ist es möglich, dass auch andere Programmiersprachen wie Scala oder Clojure nach Übersetzung in das Bytecode-Zwischenformat von der JVM ausgeführt werden können und auch mit in Java geschriebenen Bibliotheken interagieren. Im einfachsten Fall lässt sich ein „ClassFileTransformer" damit wie in *Listing 2* implementieren.

Wie der Name des „ClassFileTransformer" bereits andeutet, wird durch selbigen Transformator nichts bewirkt, da jede als „classFileBuffer" überreichte Klasse lediglich in ihrer bestehenden Form zurückgegeben wird. Alternativ dazu könnte der Transformator auch „null" zurückgeben, was der virtuellen Maschine zu verstehen gibt, dass eine zu ladende Klasse nicht verändert werden soll.

Zusätzlich zu der kompilierten Klasse erhält ein „ClassFileTransformer" auch den Namen der zu ladenden Klasse überreicht, sowie den „ClassLoader", der diese Klasse gerade zu laden versucht. Darüber hinaus wird der „ProtectionDomain" der Klasse mitgeteilt, welche etwaigen Sicherheitsbeschränkungen ein „SecurityManager" mit sich bringt, die auch ein „ClassFileTransformer" berücksichtigen soll. Für den Fall, dass eine bereits geladene Klasse durch die Java-HotSwap-Funktion neu definiert werden soll, würde noch dazu die zuvor geladene Klasse übergeben werden. Für erstmalig geladene Klassen ist dieser dritte Parameter allerdings immer „null".

```
MethodVisitor methodVisitor = ...
methodVisitor.visitIns(Opcodes.ICONST_1);
methodVisitor.visitIns(Opcodes.ICONST_2);
methodVisitor.visitIns(Opcodes.IADD);
```

Listing 3

```
class MeineKlasse {
  String hallo() { return null; }
}
```

Listing 4

```
TypePool typePool = TypePool.Default.ofClassPath();
new ByteBuddy()
  .redefine(typePool.describe("MeineKlasse"),
            ClassFileLocator.ForClassLoader.ofClassPath())
  .method(ElementMatchers.named("hallo"))
  .instrument(FixedValue.value("Hallo Welt!"))
  .make()
  .load(ClassLoader.getSystemClassLoader(),
        ClassLoadingStrategy.Default.INJECTION);
System.out.println(new MeineKlasse().hallo()); // Hallo Welt!
```

Listing 5

Java-Bytecode mit ASM direkt manipulieren

Als größtes Problem verbleibt allerdings der Umgang mit Java-Bytecode. Denn auch wenn dieser sich relativ nahe an der Programmiersprache Java orientiert, verbleibt dennoch eine Vielzahl an Unterschieden. Java-Bytecode ist als Annäherung an eine Maschinensprache konzipiert und führt seine Operationen ausschließlich auf einem (ebenso virtuellen) Stapel-Speicher aus. Dabei werden zu verarbeitende Werte auf dem genannten Stapel zunächst abgelegt und dann von anschließenden Operationen wieder ausgelesen.

Möchte man in Java beispielsweise die Zahlen „eins" und „zwei" addieren, werden beide Zahlen zunächst in Java-Bytecode auf den Stapelspeicher geschrieben. Nach diesem Ablegen der beiden Zahlen werden bei einer Addition beide Werte ausgelesen, um anschließend das Ergebnis zurück auf den Stapel zu schreiben. Auf gleiche Weise wird auch eine Methode aufgerufen, deren Argumente zunächst auf dem Stapel abgelegt werden müssen, bevor die Methode diese bei ihrer Ausführung auslesen kann.

Als Vorteil, eine virtuelle Maschine durch einen Stapelspeicher zu organisieren, ergibt sich eine geringere Länge des Bytecodes für diese virtuelle Maschinensprache. Zum Vergleich ergeben sich bei einer Maschinensprache, in der Werte in Registern abgelegt werden, häufig größere Programme, da Instruktionen sich nicht implizit auf den Wert an der Spitze des Stapels beziehen, sondern immer eine Registeradresse benötigen. Da Java als „Sprache des Internets" konzipiert ist, wurde es als Vorteil verstanden, dass bei geringem Platzverbrauch eine Java-Klasse schneller über ein Netzwerk übertragen würde, beispielsweise beim Laden eines Applets durch einen Browser. Doch gleichzeitig kann diese eher unübliche Abstraktion heute das direkte Arbeiten mit Java-Bytecode erschweren, auch wenn das natürlich möglich ist.

Eine beliebte Möglichkeit zur direkten Verarbeitung von Java-Bytecode ist die freie Bibliothek ASM, die unter anderem auch innerhalb des OpenJDK verwendet wird. Dabei wird jede Operation durch einen auch in der offiziellen JVM-Spezifikation üblichen Opcode zum Ausdruck gebracht. Dieser beschreibt dabei eine der virtuellen Maschine bekannte Stapel-Operation. Als Beispiel würden die folgenden Operationen eine Methode erzeugen, die die Zahlen „eins" und „zwei" addiert *(siehe Listing 3)*.

Wie zuvor beschrieben, müssen hierzu zunächst die Zahlen „eins" und „zwei" auf dem Stapelspeicher abgelegt werden, sodass die „IADD"-Operation sie anschließend auslesen kann, um dann die Summe der beiden Zahlen zurück auf den Stapel zu schreiben. Auf ähnliche Art und Weise kann auch eine kompilierte Methode aus einer „class"-Datei interpretiert werden. Dazu muss ein eigenes Objekt „MethodVisitor" übergeben werden, dem dann beim Lesen der Methode alle OpCodes in der vorgefundenen Reihenfolge als Argument übergeben werden. Auf diese Art könnte auch die durch einen „ClassFileTransformer" übergebene „class"-Datei interpretiert und umgeschrieben werden. Doch wie bereits zu erahnen, ist dieses Vorgehen häufig unnötigerweise umständlich und fehleranfällig. Direktes Manipulieren von Bytecode sollte deswegen für aspektorientierte Programmierung nur dann eingesetzt werden, wenn sehr spezielle Klassen-Veränderungen benötigt werden, die nicht durch ein weniger abstraktes API umgesetzt werden kann.

Bytecode-Manipulation mit Byte Buddy

Byte Buddy ist eine Bibliothek, die Bytecode-Manipulation auch ohne ein Verständnis des Java-Bytecode-Formats zugänglich macht. Byte Buddy basiert hierzu auf einer domänenspezifischen Sprache, mithilfe derer sich die Redefinition einer Klasse durch gewöhnlichen Java-Code ausdrücken lässt.

Als einfacher Anwendungsfall lässt sich zum Beispiel der Rückgabewert einer Methode zur Laufzeit auf einen festen Wert verändern. Für ein Beispiel soll dazu die einzige Methode der folgenden Klasse umgeschrieben werden *(siehe Listing 4)*.

Anstelle der „null"-Referenz soll die Methode mit Namen „hallo" aus „MeineKlasse" überschrieben werden, um einen festen Wert „Hallo Welt!" anstatt von „null" zurückzugeben. Mithilfe von Byte Buddy ist dies mit nur wenigen Linien Code möglich *(siehe Listing 5)*.

Da bisher kein Java-Agent zum Einsatz kommt, der vor dem Laden einer Klasse automatisch benachrichtigt wird, ist es dazu wichtig, dass „MeineKlasse" nicht bereits durch einen „ClassLoader" geladen wird, bevor die Klasse umgeschrieben wurde. Denn nach dem Laden einer Klasse erlaubt die Java Virtual Machine nur noch sehr eingeschränkte Änderungen an deren Funktionalität oder Struktur.

Um ein Laden der Klasse vor deren Umschreiben zu vermeiden, kann ein „TypePool" zum Einsatz kommen, der es erlaubt, eine Java-Klasse ähnlich wie durch das Java-Reflection-API zu beschreiben. Diese Beschreibung kann dann von Byte Buddy verarbeitet werden, um das Verhalten der Klasse zu verändern. Dabei werden zunächst eine oder mehrere Methoden durch einen „ElementMatcher" identifiziert. In oben stehendem Beispiel wird die „toString"-Methode anhand ihres Namens erkannt. Anschließend lässt sich das Verhalten dieser Methoden mithilfe einer Byte-Buddy-„Instrumentation" anpassen. Im Beispiel ist lediglich ein fester Wert durch einen „FixedValue" definiert, eine der vielen Implementierungen des Interface.

Die Rückgabe eines festen Werts wie „Hallo Welt!" ist dabei ein einfacher Anwendungsfall. In der aspektorientierten Programmierung soll das Verhalten einer Methode in der Regel aber dynamischer angepasst werden. Auch dies ist mithilfe von Byte Buddy einfach zu lösen, beispielsweise durch die Verwendung einer „MethodDelegation". Über eine solche Delegation erlaubt Byte Buddy, einen Methoden-Aufruf durch eine alternative Methode zu ersetzen. Dabei werden die Parameter-Werte der aufzurufenden Methode durch Annotationen bestimmt. Zum Beispiel ermöglicht die „@Origin"-Annotation, die umgeschriebene Methode abzufragen, wie der Interceptor im Beispiel vom *Listing 6* demonstriert.

Mithilfe des genannten Interceptor kann „MeineKlasse" nun entsprechend umgeschrieben werden, sodass anstelle von „methode" die „intercept"-Methode von „MeinInterceptor" aufgerufen wird. Letztere Methode wird dabei als Argument eine Referenz zu der ursprünglich aufgerufenen „methode" übergeben, wie das Beispiel in *Listing 7* verdeutlicht.

Ein einfacher Agent mit Byte Buddy

In Kombination mit der Möglichkeit, einen Java-Agenten zu registrieren, wird es plötzlich einfach, Aspekte auch ohne eine Sprach-Erweiterung wie AspectJ zu implementieren. Als Beispiel soll der Aufruf jeder Methode, die mit der folgenden „Loggen"-Annotation annotiert ist, auf der Konsole angezeigt werden *(siehe Listing 8)*.

Um dies mit Byte Buddy zu implementieren, muss zunächst ein entsprechender Interceptor für das Loggen definiert sein. Dieser empfängt als Argument eine Referenz der annotierten

Methode und schreibt deren Namen anschließend auf die Standard-Ausgabe *(siehe Listing 9)*.

Mithilfe des Interceptor kann, wie im vorherigen Abschnitt gezeigt, eine Methode umgeschrieben werden, sodass die Methode des „LogInterceptor" anstelle der eigentlichen Methode aufgerufen wird. Dies entspricht allerdings noch nicht der Anforderung, da der Interceptor zusätzlich zu und nicht anstatt der umgeschriebenen Methode aufgerufen werden soll. Auch dies ist mit Byte Buddy möglich, da viele der mitgelieferten Instrumentierungen miteinander verknüpfbar sind. Um nach Aufruf des Interceptor die ursprüngliche Methode aufzurufen, kann die Delegation mit einem solchen Aufruf durch „MethodDelegation.to(LogInterceptor.class).andThen(SuperMethodCall.INSTANCE)" verbunden werden. Die verknüpfte Instrumentierung ruft dabei lediglich die ursprüngliche Methode auf.

Um die Definition eines Agenten zu erleichtern, bietet Byte Buddy die Nutzung eines „AgentBuilder". Dieser erlaubt es zunächst, Klassen mithilfe eines „ElementMatcher" zu identifizieren, die vor dem Laden umgeschrieben werden sollen. Anschließend wird ein „Transformator" für diese Klassen registriert, der es ermöglicht, jede der zuvor identifizierten Klassen mithilfe der bereits bekannten domänenspezifischen Sprache umzuschreiben. Über einen solchen „AgentBuilder" kann ein Aspekt zum Loggen eines annotierten Methodenaufrufs nun einfach implementiert werden, wie das Beispiel in *Listing 10* zeigt.

Der oben stehende Aspekt zum Loggen eines Methodenaufrufs soll für sämtliche Klassen implementiert werden, aber nur für solche Methoden, die mit „Loggen" annotiert sind. Für diese Methoden soll dann zunächst der „LogInterceptor" und dann die ursprüngliche Methode aufgerufen werden.

Da Annotationen, die nicht auf dem Klassenpfad aufzufinden sind, von der Java-Laufzeit-Umgebung ignoriert werden und keine „ClassNotFoundException" verursachen, können der Agent, die Annotation und der „LogInterceptor" entfernt werden, ohne die Funktionalität der Anwendung zu stören. Genauso wenig muss die Anwendung vor dem Entfernen oder Hinzufügen des Agenten neu kompiliert werden. Die „Loggen"-Annotation kann aus diesem Grund sogar als „drop-in Feature" verwendet werden.

Natürlich bieten viele Rahmenwerke wie beispielsweise Spring bereits eine Möglichkeit, Aspekte zu implementieren. Außerdem ist es möglich, Aspekte mit Sprach-Erweiterungen wie zum Beispiel AspectJ zu implementieren. Durch Definition eines eigenen Agenten lassen sich Aspekte allerdings ohne viele Codezeilen und unabhängig von einer tiefer greifenden Infrastruktur umsetzen. Gerade Querschnittsanforderungen wie Logging, Sicherheit oder Caching sind dadurch auf einfache Art und Weise zu realisieren.

```
public class MeinInterceptor {
  public static String intercept(@Origin Method methode) {
     return "Hallo von " + methode.getName() + "!";
  }
}
```

Listing 6

```
TypePool typePool = TypePool.Default.ofClassPath();
new ByteBuddy()
  .redefine(typePool.describe("MeineKlasse"),
            ClassFileLocator.ForClassLoader.ofClassPath())
  .method(ElementMatchers.named("hallo"))
  .instrument(MethodDelegation.to(MeinInterceptor.class))
  .make()
  .load(ClassLoader.getSystemClassLoader(),
        ClassLoadingStrategy.Default.INJECTION);
System.out.println(new MeineKlasse().hallo()); // Hallo von methode!
```

Listing 7

```
@Retention(RetentionPolicy.RUNTIME)
@interface Loggen { }
```

Listing 8

```
public class LogInterceptor {
  public static void intercept(@Origin Method methode) {
     System.out.println(methode.getName() + " wurde aufgerufen!");
  }
}
```

Listing 9

```
public class LogAgent {
  public static void premain(String args,
                     Instrumentation instrumentation) {
     new AgentBuilder.Default()
        .rebase(ElementMatchers.any())
        .transform(
           (builder, typeDescription) -> return builder
              .method(ElementMatchers.isAnnotatedWith(Loggen.class))
              .intercept(MethodDelegation.to(LogInterceptor.class)
                    .andThen(SuperMethodCall.INSTANCE));
     ).installOn(instrumentation);
  }
}
```

Listing 10

Rafael Winterhalter
rafael.wth@gmail.com

Rafael Winterhalter lebt in Oslo und arbeitet dort als Softwareentwickler. Er beschäftigt sich intensiv mit der JVM und entwickelt aktiv das Open-Source-Projekt Byte Buddy (http://bytebuddy.net).

http://ja.ijug.eu/15/3/5

Guter Code, schlechter Code

Markus Kiss und Christian Kumpe, Netpioneer GmbH

Was ist eigentlich guter Code? Wahrscheinlich haben sich viele Entwickler diese Frage schon gestellt und werden sie auch in Zukunft stellen. Bücher wie „Clean Code" von Robert C. Martin [1] enthalten eine Vielzahl an Regeln, die Code befolgen sollte. Aber was ist eigentlich die Essenz des Ganzen? Die Autoren sagen einfach: „Guter Code muss lesbar sein".

Im Unternehmen der Autoren wird Bewerbern beim Vorstellungsgespräch eine kleine zweistündige Programmieraufgabe gestellt. Für ein gegebenes Interface müssen einige Methoden implementiert werden, wahlweise in Java oder C#. Weitreichende technische Kenntnisse sind dafür nicht erforderlich. Das Wissen über das Java-Standard-API, hauptsächlich das Collections Framework *[2]* beziehungsweise deren C#-Äquivalente reichen vollkommen aus. Außerdem ist das Internet und Google während der Aufgabe verfügbar. Im Anschluss soll der Bewerber seinen Code kurz erklären und seine Design-Entscheidungen begründen.

```
public void printSortedByAge() {
  Iterator<Customer> iter = customers.iterator();
  ArrayList<String> list = new ArrayList<String>();
  while (iter.hasNext()) {
     Customer customer = iter.next();
     list.add(customer.getAge() + " " + customer.getLoginName());
  }
  Collections.sort(list);
  Iterator<String> iter2 = list.iterator();
  while (iter2.hasNext()) {
     try {
        String str = iter2.next();
        System.out.println(findCustomerByName(str.substring(str.
lastIndexOf(" ") + 1)));
     } catch (CustomerNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
     }
  }
}
```

Listing 1

Als Betreuer für diese Aufgabe ist den Autoren bald aufgefallen, dass mancher Code schon beim ersten Durchsehen zu verstehen ist, sogar während man den Erklärungen des Bewerbers folgt. Kleine Fehler, soweit vorhanden, fallen einem sofort auf und man kann schnell in die Diskussion einsteigen. Manchmal ist der Code aber auch bei genauem Hinsehen einfach nicht verständlich und man braucht mehrere Minuten, um den Ablauf einzelner Methoden zu verstehen und dem Bewerber Feedback über mögliche Fehler zu geben.

Das Fünfsekunden-Experiment

Aus den Bewerber-Gesprächen ist bei den Autoren die Idee des Fünfsekunden-Experiments entstanden: *Listing 1* zeigt ein Stück Code, das sie so oder ähnlich einmal wäh-

rend einer Programmieraufgabe präsentiert bekamen. Man betrachtet das Beispiel, legt nach etwa fünf Sekunden das Heft kurz beiseite und überlegt, was der Code macht, machen sollte und wo sich mögliche Fehler verstecken.

Was macht der Code?

Am aussagekräftigsten ist wahrscheinlich noch der Methoden-Name. Kennt man auch das zu implementierende Interface, ist klar: Hier sollen Kunden nach Alter sortiert ausgegeben werden. Aber wie macht das der Code und funktioniert die Implementierung? Wo liegen ihre Grenzen? Das kann wohl kaum jemand innerhalb weniger Sekunden beurteilen.

Ohne jetzt zu sehr ins Detail gehen zu wollen: Die Kunden werden sortiert, indem Strings der Form „Alter" + „ " + „Loginname" gebildet werden. Diese Strings werden anschließend in sortierter Reihenfolge abgearbeitet, wobei der Login-Name aus dem String extrahiert und anschließend der entsprechende Kunde anhand seines Loginnamens geholt und ausgegeben wird. Weitere Details finden sich unter *[3]*.

Dieser Lösungsweg ist sicher nicht einfach zu erfassen und auch mögliche Fehler sind nicht direkt offensichtlich. Aber soweit die Kunden zwischen 10 und 99 Jahre alt sind und im Login-Namen keine Leerzeichen erlaubt sind, funktioniert es.

Ein zweites Beispiel

Nun wird das *Listing 2* etwa fünf Sekunden lang betrachtet. Dem Methoden-Namen nach sollte das Programm genau das Gleiche tun, aber auch ohne diese Hilfe sind die drei zentralen Schritte eigentlich sehr schnell zu erfassen: Die Liste der Kunden wird kopiert, die Kopie mit Hilfe des „AgeComparators" nach dem Alter sortiert und anschließend in einer Schleife auf der Konsole ausgegeben.

Der Code ist aus vielen Gründen deutlich einfacher zu lesen und schneller zu erfassen: Er ist kürzer, enthält keinen unnötigen Anweisungen und geht so ziemlich den direktesten Weg. Durch sprechende Variablen-Namen und gute Strukturierung ist die Methode auch ohne jeden Kommentar schnell zu verstehen. Daraus folgen einige ausgewählte Regeln, mit denen sich gut lesbarer Code schreiben lässt.

Zu viele Kommentare

Gut gemeintes Kommentieren von Code kann schnell neue Probleme aufwerfen, was in *Listing 3* verdeutlicht wird.

Die Methode „deleteCustomer" enthält zwar nur wenig Logik, doch über jeder Code-Zeile steht ein Kommentar, der nochmals in Prosa beschreibt, was der darauffolgende Code macht. Dies bläht den Code nicht nur optisch auf, sondern zwingt andere Entwickler dazu, neben dem Code auch die Kommentare zu lesen und zu verstehen – ein zeitaufwändiger Prozess. Passt dann noch aufgrund von Refactorings der Code nicht mehr zu den Kommentaren, sind Verwirrungen vorprogrammiert. Man fragt sich dann zum Beispiel, ob der Code etwa fehlerhaft ist, weil der Kommentar etwas anderes ausdrückt, oder ist der Kommentar selbst falsch?

Diese Irritationen lassen sich bereits von vorneherein vermeiden, indem man den Code bereits so klar strukturiert, dass keine Kommentare zur Erläuterung notwendig sind. Dies kann etwa durch aussagekräftige Klassen-, Variablen- und Methoden-Namen erreicht werden. Guter Code erzählt von alleine, was er macht.

Im Beispiel ist auch ein JavaDoc-Block aufgeführt, der den Methodenzweck inklusive der Ein- und Ausgabewerte beschreibt. Diese Art von Kommentaren ist wiederum ganz hilfreich, um die öffentliche Schnittstelle zu dokumentieren und insbesondere auf semantische Sachverhalte hinzuweisen, wie zum Beispiel wann „true" und wann „false" zurückgeliefert wird.

Code-Formatierung und -Strukturierung

Neben aussagekräftigen Bezeichnern spielt auch die Strukturierung und Formatierung

```
public void printSortedByAge() {
  List<Customer> customersToSort = new ArrayList<>(customers);
  Collections.sort(customersToSort, new AgeComparator());
  for (Customer customer : customersToSort) {
     System.out.println(customer);
  }
}
```

Listing 2

```
/**
 * Deletes the given {@link Customer}.
 * @param argCustomer customer to delete
 * @return true if the customer has been deleted, false otherwise.
 */
@Override
public boolean deleteCustomer(Customer argCustomer) {
  // Check if we have a customer in our list.
  // If we don't have one, we return false as
  // the customer hasn't been deleted actually.
  if (containsCustomer(argCustomer)){
     // Remove the customer.
     // We could also directly return the result of remove()
     customers.remove(argCustomer);
     // Return true to communicate the result
     return true;
  }
  // Return false if the customer could not be found
  return false;
}
```

Listing 3

```
public List<Customer> findMaleAdultCustomers() {
  List<Customer> custFound = new ArrayList<>();
  for (int i = 0; i < custList.size(); i++) {
  if (custList.get(i).getAge() < 18)
  continue;
  else
     {
        if (custList.get(i).getGender() != Gender.FEMALE)
     custFound.add(custList.get(i)); }
     }
  return custFound;
}
```

Listing 4

des Codes eine wesentliche Rolle. In *Listing 4* ist die Implementierung der Methode „findMaleAdultCustomers" zu sehen, die aus einer Liste von Kunden diejenigen heraussucht und zurückgibt, die männlich und erwachsen sind.

Beim ersten Blick fällt auf, dass die Struktur des Codes und der Programmfluss nicht sofort ersichtlich ist: Es ist unklar, welchen Block die „for"-Schleife umfasst und wo die „if"- und „else"-Blöcke hingehören. Sinnvolle Einrückungen und Formatierungen nach allgemein bekannten Konventionen schaffen hier Abhilfe. In Entwicklungsumgebungen wie Eclipse, IntelliJ IDEA und NetBeans ist die Auto-Formatierung für Java-Code bereits mitgeliefert.

Darüber hinaus fällt auf, dass die Ausdrücke in den „if"-Abfragen jeweils den Negativfall prüfen: Es wird geprüft, ob das Alter des Kunden kleiner als 18 ist und der Kunde keine Frau ist. Insbesondere die Prüfung des Alters macht den weiteren Ablauf komplex, da der aktuelle Schleifendurchlauf per „continue" übersprungen wird. Die positive Formulierung des Ausdrucks „Alter des Kunden ist größer-gleich 18" würde viel klarer ausdrücken, dass im weiteren Ablauf mit volljährigen Kunden gearbeitet wird. Ebenso würde die Prüfung „Kunde ist männlich" direkt beschreiben, dass man nur diese Kunden im Fokus hat.

In *Listing 5* ist dieselbe Funktionalität implementiert, jedoch auf deutlich übersichtlichere Art und Weise: Zum einen wird eine „for each"-Schleife verwendet, um deutlicher zu machen, dass über eine Collection iteriert wird. Zum anderen wurden die „if"-Ausdrücke in eigene boolesche Methoden extrahiert, die durch ihren Namen ausdrücken, was sie prüfen. Dadurch liest sich der „if"-Ausdruck wie Prosa: „Wenn der Kunde männlich und erwachsen ist, dann …". Bereits diese kleinen Refactorings führen dazu, dass der Kontrollfluss der Methode auf einen Blick erfasst werden kann.

```
public List<Customer> findMaleAdultCustomers() {
  List<Customer> maleAdultCustomers = new ArrayList<>();
  for (Customer customer : customers) {
     if (isMale(customer) && isAdult(customer)) {
        maleAdultCustomers.add(customer);
     }
  }
  return maleAdultCustomers;
}
private boolean isMale(Customer customer) {
  return customer.getGender() == Gender.MALE;
}
private boolean isAdult(Customer customer) {
  return customer.getAge() >= 18;
}
```

Listing 5

```
public CustomerServiceImpl() {
  super();
}
public boolean containsCustomer(final Customer argCustomer) {
  if (customers.contains(argCustomer) == true) {
     return true;
  } else {
     return false;
  }
}
```

Listing 6

```
public void printSortedByLastName() {
  SortedSet<Customer> sortedCustomers = new TreeSet<Customer>(new LastNam-
eComparator());
  sortedCustomers.addAll(customers);
  for (Customer customer : sortedCustomers) {
     System.out.println(customer);
  }
}
```

Listing 7

Unnötiger Code

Ein weiteres Problem ist die Verwendung von überflüssigem Code. Beispiel 6 zeigt gleich mehrere dieser Fälle auf, etwa den parameterlosen Konstruktor: Dieser ruft nur „super()" auf, ohne weitere Initialisierungen vorzunehmen. In diesem Fall ist die Deklaration des Konstruktors vollkommen überflüssig, da ein solcher laut Java Language Specification ([4] (Abschnitt 8.8.9) sowieso automatisch generiert wird.

Ebenfalls unnötig ist die explizite Prüfung auf die booleschen Werte „true" oder „false" in „if"-Anweisungen. Mit ein bisschen Refactoring lässt sich die komplette Methode „containsCustomer" auf eine einzige Anweisung reduzieren: „return customers.contains(argCustomer)".

Am schönsten ist es immer, wenn man Code löschen kann, ohne etwas am Verhalten zu ändern. Der Vorteil liegt auf der Hand: Man muss keinen Code mehr lesen, der sowieso automatisch im Hintergrund durch den Compiler erzeugt wird, und kann sich auf die wesentlichen Abläufe konzentrieren *(siehe Listing 6)*.

Keine Überraschungen

Das „Principle of Least Astonishment" ist ein Prinzip aus Clean Code [1], das im Kern besagt, dass geschriebener Code keine Überraschungen jeglicher Art mit sich bringen sollte. Eine solche Überraschung versteckt sich in *Listing 7:* Hier wird die Collection „customers" nach dem Nachnamen des Kunden sortiert auf der Konsole ausgegeben.

Eine Frage: Wo findet das Sortieren eigentlich statt? Java-Kenner werden wissen, dass dies in der „TreeSet"-Implementierung intern beim Aufruf von „addAll()" erfolgt – unter Verwendung des „LastNameComparator". Wichtig ist, welche Auswirkungen der Einsatz eines „TreeSet" hat: Da ein Set per Definition keine Duplikate erlaubt, würde nur ein Teil der Kunden auf der Konsole ausgegeben werden. Hätte man drei Kunden mit Nachnamen „Meier", würde nur einer davon ausgegeben werden – selbst wenn alle drei Kunden unterschiedliche Vornamen haben. Zweifelsohne eine Überraschung, die vielleicht erst im Live-Betrieb aufgefallen wäre, wenn Kunden-Datensätze fehlen. Sinnvoller wäre es, das Sortieren explizit mittels „Coll-

ections.sort()" durchführen zu lassen *(analog zu Beispiel 1)*, um solche unangenehmen Überraschungen zu vermeiden und gleichzeitig die Lesbarkeit des Codes zu verbessern.

Fazit

Dieser Artikel zeigt Beispiele für guten und schlechten Code, den die Autoren so oder in ähnlicher Form immer wieder gesehen haben. Unter *[3]* haben sie den Quelltext und weitere Code-Beispiele mit Erläuterungen bereitgestellt.

Wenn sich die Autoren abschließend die Frage stellen: „Was ist guter Code, was ist schlechter Code?", dann können sie zwar viele Regeln und Prinzipien aufzählen, doch alle führen zu derselben einfachen wie wirksamen Kernaussage: „Guter Code ist lesbar, schlechter Code ist es nicht."

Als Entwickler verbringen wir im Schnitt 80 Prozent unserer Zeit damit, bestehenden Code zu lesen und nur 20 Prozent mit dem tatsächlichen Schreiben von neuem Code. Schon deshalb lohnt es sich, beim Schreiben von Code darauf zu achten, dass andere Entwickler später möglichst wenig Zeit benötigen, ihn zu verstehen.

Sollte man sich manchmal unsicher sein, ob sein Code wirklich lesbar ist, dann bietet sich das Fünfsekunden-Experiment als Mini-Metrik an: Lässt sich der Zweck und die Funktionalität des Codes innerhalb von wenigen Sekunden erfassen? Wenn nicht, woran hat es gelegen? Gab es zu komplexe Verschachtelungen? Ist man über überflüssigen Code oder Kommentare gestolpert? Waren die Methoden- oder Variablennamen unklar?

Diese und viele weitere Fragen, die sich aus dem Fünfsekunden-Experiment ergeben, können Anlass für Refactorings sein und den Code schrittweise lesbarer machen. Andere Entwickler werden es einem danken – auch man selbst, wenn man seinen Code Monate oder Jahre später weiterentwickeln möchte.

Weiterführende Informationen

[1] Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall International, ISBN 978-0132350884

[2] Das Java Collections Framework: http://docs.oracle.com/javase/8/docs/technotes/guides/collections

[3] Code-Beispiele zum Artikel: https://github.com/kumpe/gcsc

[4] Java Language Specification: http://docs.oracle.com/javase/specs

Markus Kiss
markus.kiss@netpioneer.de

Markus Kiss, M. Sc., hat in Karlsruhe und Mannheim Informatik studiert und arbeitet als Senior Software-Entwickler bei der Netpioneer GmbH in Karlsruhe. Er interessiert sich seit mehreren Jahren für Clean Code und saubere Software-Architekturen. Code Smells und Antipatterns bringen ihn zwar immer wieder zum Schmunzeln, doch er findet es spannend, wie man mit einfachen Techniken und Praktiken genau dem entgegenwirken kann.

Christian Kumpe
christian.kumpe@netpioneer.de

Christian Kumpe studierte Informatik am Karlsruher Institut für Technologie (KIT) und sammelte bereits während seines Studiums als Freelancer Erfahrung in diversen Java-Projekten. Seit dem Jahr 2011 arbeitet er als Software-Entwickler bei der Netpioneer GmbH in Karlsruhe. Seine aktuellen Themenschwerpunkte sind Java-basierte Portale und Internet-Plattformen. Es ist ihm ein wichtiges Anliegen die Wartbarkeit von Software-Systemen durch guten Code zu erhalten und zu verbessern.

http://ja.ijug.eu/15/3/6

Oracle stopft 98 kritische Sicherheitslücken

Oracle hat mit der Veröffentlichung seines vierteljährlichen Critical Patch Update (CPU) insgesamt 98 Sicherheitsupdates herausgebracht. Davon betroffen sind unter anderem die Oracle-Datenbank, Java, Fusion Middleware, MySQL sowie die Geschäftsapplikationen. Um böswilligen Angriffen vorzubeugen, sollten die Updates zeitnah eingespielt werden.

Allein 14 der 98 Schwachstellen entfallen auf Java. Viele davon können über das Netzwerk ausgenutzt werden, weshalb man die Patches so schnell wie möglich installieren sollte. Drei der Sicherheitslücken sind mit der höchsten Gefahrenstufe 10.0 nach dem Common Vulnerability Scoring System (CVSS) als hochkritisch bewertet. Das Update schließt zudem letztmalig die Lücken der älteren Java-Version 7. Nach April 2015 stellt Oracle hierfür keine Updates mehr bereit. Der Hersteller empfiehlt daher die Umstellung auf Java 8.

Mit dem Update werden auch vier Schwachstellen in der Oracle-Datenbank geschlossen. Keine von ihnen ermöglicht jedoch einen unbefugten Zugriff über das Netzwerk. Der höchste CVSS-Score beträgt 9.0.

In MySQL stopft Oracle ganze 26 Lücken, von denen vier ohne Anmeldung über das Netzwerk ausgenutzt werden können und damit den höchstkritischen CVSS-Score von 10.0 erhalten.

Von 17 Schwachstellen in Oracles Fusion-Middleware-Produkten erlauben zwölf das Ausnutzen ohne Authentifizierung aus der Ferne. Der CVSS-Score der am höchsten eingestuften Sicherheitslücke beträgt 10.0.

Auch für seine Applications stellt Oracle Patches bereit: In der E-Business Suite schließt der Hersteller vier Lücken, in JD Edwards und Siebel CRM jeweils eine Schwachstelle.

Das nächste Critical Patch Update folgt im Juli 2015.

HTML als neue Oberfläche für JavaFX

Wolfgang Nast, MT AG

In Java hat es schon mehrere Oberflächen gegeben. Angefangen mit dem Abstract Window Toolkit (AWT), gefolgt von Swing und JavaFX. Bedeutend ist noch das Standard Widget Toolkit (SWT). Was sich aber die ganze Zeit über gehalten hat, war HTML mit JavaScript.

JavaFX bringt als Anzeige-Element die Klasse „WebScene" mit „WebEngine". Diese ist zwar dafür gedacht, fremde Web-Inhalte anzuzeigen; es ist aber auch praktisch, damit seine eigenen Inhalte darzustellen. Deshalb die Frage: Warum immer das Rad neu erfinden? Lieber etwas Beständiges nehmen, das sich weiterentwickelt.

Die Idee, HTML zu nutzen, ist nicht neu. In Swing kann man den Text von Elementen wie JLabel mit Farben und Schriften dekorieren. In JavaFX können Teile von CSS genutzt werden, um die Anzeigeelemente einheitlich zu dekorieren. Aber warum nicht einen Schritt weiter gehen und die Oberfläche komplett in HTML schreiben? *Listing 1* zeigt ein einfaches Beispiel, bei dem die Java-Dokumentation in einem Fenster angezeigt wird.

Beim Ausführen ist zu beachten, dass hier kein Proxy berücksichtigt wird. Das ist meist der Grund für eine leere Anzeige. Jetzt wird das Java-API angezeigt und man kann darauf ganz normal navigieren *(siehe Abbildung 1)*.

Im nächsten Schritt wird das Programm umgestellt, sodass es lokal die Daten holt. Dafür legt man ein Verzeichnis für den Webinhalt an, hier „HTML" genannt. Der Name ist jedoch frei wählbar. In das Verzeichnis wird jetzt die eigene HTML-Seite abgelegt oder einfach das Java-API kopiert. Die Hauptseite der Anwendung heißt hier „Main.html". Für das Java-API wird „index.html" verwendet.

Jetzt muss noch das Programm angepasst werden, um die lokale HTML-Seite zu laden. Die Zeile „engine.load(„http://doc.oracle.com/javase/docs/api/");" ist durch *Listing 2* zu ersetzen.

Der Aufruf der „ExternalForm" für den Path hat den Vorteil, ziemlich unabhängig vom Ausführungsort der Applikation und vom Betriebssystem zu sein. So wird die passende URL für die Datei „Main.html" abgerufen und nicht vom Programmierer vorgegeben.

Die Webseite läuft jetzt lokal mit HTML, CSS und JavaScript. Dabei ist zu beachten, dass PDF-, XML- und Office-Dokumente nicht angezeigt werden können. JQuery und andere JavaScript-Bibliotheken lassen sich lokal einsetzen. Damit von der HTML-Oberfläche aus auch Java-Klassen angesprochen werden können, verwendet man die Anbindung der WebEngine. *Listing 3* zeigt eine einfache Java-Klasse, die man dafür verwenden kann.

```
package org.anzeige;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.web.WebEngine;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class HtmlAnzeige extends Application {

  @Override
  public void start(Stage stage) {
     stage.setTitel("Anzeige");
     stage.setWidth(800);
     stage.setHeight(600);
     Scene scene = new Scene(new Group);

     WebView browser = new WebView();
     WebEngine engine = browser.getEngine();

     engine.load("http://doc.oracle.com/javase/docs/api/");

     scene.setRoot(browser);
     stage.setScene(scene);
     stage.show();
  }

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

Listing 1

```
  Path inhaltPath = Paths.get("HTML", "Main.html");
  try {
        engine.load(inhaltPath.toUri().toURL().toExternalForm());
     } catch (MaleformedURLException e) {
     System.out.println("Fehler beim Erzeugen der URL: " + e.getMessage());
  }
```

Listing 2

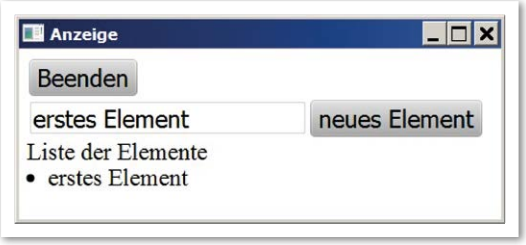


Abbildung 2: Beispiel-HTML mit Element

```
package org.anzeige;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class JavaScriptObj {
  private final Document doc;

  public JavaScriptObj(Document document) {
     doc = document;
  }

  public void exit() {
     Platform.exit();
  }

  public void add(String text) {
     Element liElem = doc.createElement("li");
     liElem.appendChild(doc.createTextNode(text));
     doc.getElementById("top").appendChild(liElem);
  }
}
```

Listing 3

```
  engine.getLoadWorker().stateProperty().addListener(
     new ChangeListener<State>() {
        public void changed(ObservableValue ov, State oldState, State newState) {
           if (newState == State.SUCCEEDED) {
              Document doc = engine.getDocument();
              JSObject window = (JSObject) engine.executeScript("window");
              Window.setMember("app", new JavaScriptObj(doc));
           }
        }
     });
```

Listing 4

```
<button id="bt1" onClick="app.exit()">Beenden</button><br>
<input id="text"></input>
<button id="bt2" onClick="var x=document.getElementById('text').value; app.
add(x)">neues Element</button><br>
<div id="top">Liste der Elemente</li>
```

Listing 5

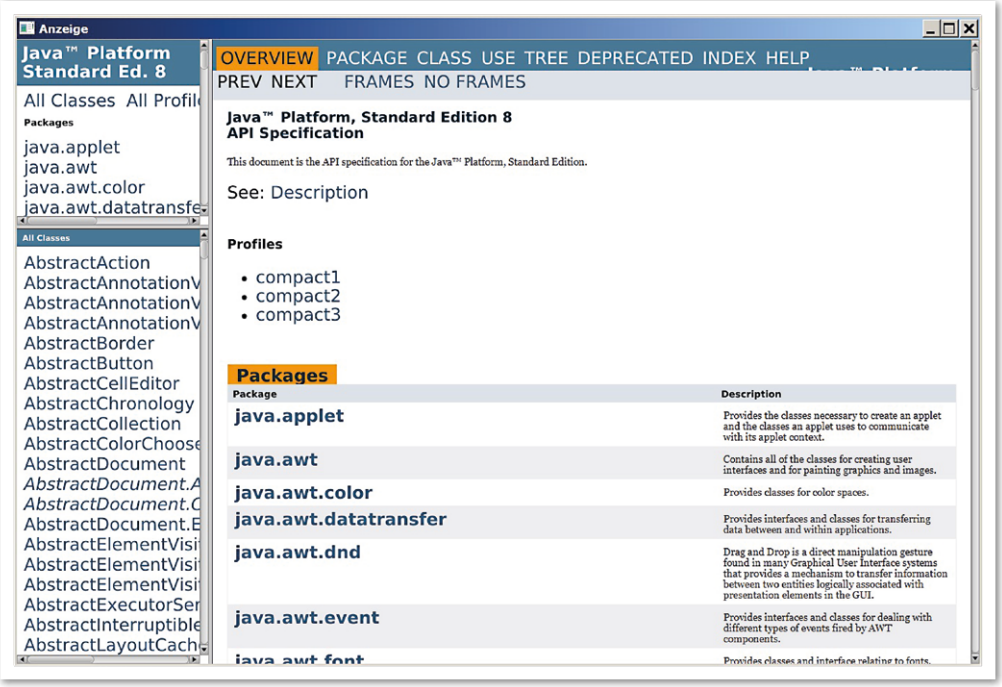


Abbildung 1: HTML mit dem Java-API

Die Methode „exit" beendet das Programm, die Methode „add" erstellt ein neues Listen-Element mit dem übergebenen Text und fügt es unter dem Element mit der Id „top" ein. Nun ist die Klasse nur noch passend einzubinden, damit sie wie JavaScript verwendet werden kann. *Listing 4* zeigt einen Code, der vor der geänderten Stelle einzufügen ist, damit er bei jedem Laden einer Webseite auch ausgeführt wird.

Jetzt wird bei jedem Laden einer Seite eine neue Instanz des „JavaScriptObj" als JavaScript-Variable „app" an die Seite gebunden. Auf der HTML-Seite lässt sich die Anwendung mit „app.exit()" beenden. Mit „app.add(text)" wird ein neues Listen-Element mit dem Text von „text" erstellt und unter dem Element mit der Id „top" eingefügt *(siehe Listing 5)*. *Abbildung 2* zeigt das Beispiel auf einer HTML-Seite.

Fazit

Die wichtigsten Grundlagen sind gegeben, um eine lokale HTML-Seite anzusprechen und diese mit den Java-Klassen zu verbinden. Es ist auch möglich, gemischte Anwendungen zu implementieren, die auch Menüs oder andere Anzeige-Elemente von JavaFX nutzen. Auch Online-Inhalte können eingebunden werden.

Wolfgang Nast
wolfgang.nast@mt-ag.com

Wolfgang Nast (Dipl. Ing.) ist Senior-Berater bei der MT AG und seit dem Jahr 1998 mit Java SE sowie seit dem Jahr 2006 mit Java EE in den Bereichen „Web-Services" und „Architektur" beratend und umsetzend tätig.

http://ja.ijug.eu/15/3/7

JavaFX – beyond „Hello World“

Jan Zarnikov, Irian Solutions Softwareentwicklungs- und Beratungsgesellschaft mbH

JavaFX ist der längst überfällige Nachfolger von Swing. Endlich lassen sich wieder Desktop-Anwendungen mit Java entwickeln, die man ohne Plattform-spezifische Library einfach einsetzen kann und die auch gut aussehen. Mit dem Java-Release „8u40“ wurden viele Bugs behoben und fehlende Features wie Accessibility und Dialog-Fenster nachgeholt. Mittlerweile ist JavaFX so stabil, dass man es problemlos auch für große Projekte produktiv einsetzen kann.

Einführungsartikel und triviale Beispiele zu JavaFX gibt es im Internet und in Fachzeitschriften mittlerweile genug. Dieser Artikel beschreibt die größten Pitfalls, die dem Autor bei der Entwicklung einer großen JavaFX-Anwendung aufgefallen sind. Er ist für Entwickler bestimmt, die JavaFX vielleicht schon mal ausprobiert haben und gerade überlegen, es bei einer größeren Anwendung einzusetzen.

```
public class Employee {
  private final StringProperty name = new SimpleStringProperty();

  public String getName() {
     return name.get();
  }

  public StringProperty nameProperty() {
     return name;
  }

  public void setName(String name) {
     this.name.set(name);
  }
}

public class EmployeePane extends BorderPane {
  public EmployeePane(Employee employee) {
     Label nameLabel = new Label();
     nameLabel.textProperty().bind(employee.nameProperty());
     setCenter(nameLabel);
  }
}
```

Listing 1

```
TextField nameField = new TextField();
employee.nameProperty().bind(nameField.textProperty());
```

Listing 2

```
<GridPane xmlns:fx="http://javafx.com/fxml" fx:controller="at.irian.javafx.demo.fxml.ControllerExample">
  <Label text="First name" GridPane.rowIndex="0" GridPane.columnIndex="0"/>
  <TextField fx:id="firstName" GridPane.rowIndex="0" GridPane.columnIndex="1"/>

  <Label text="Last name" GridPane.rowIndex="1" GridPane.columnIndex="0"/>
  <TextField fx:id="lastName" GridPane.rowIndex="1" GridPane.columnIndex="1"/>

  <Button text="Submit" onAction="#submit" GridPane.rowIndex="2" GridPane.columnIndex="0"/>
  <Button text="Reset" onAction="#reset" GridPane.rowIndex="2" GridPane.columnIndex="1"/>
</GridPane>
```

Listing 3

JavaFX-Property

Das wohl wichtigste Prinzip in JavaFX sind die Properties. Ein JavaFX-Property ist ein typsicherer Wrapper, der einen Wert speichern kann. Zudem besteht die Möglichkeit, „Listener“ darauf zu registrieren, die bei einer Änderung des Werts aufgerufen werden. Es ist außerdem möglich, ein Property an ein anderes Property zu binden. Ersteres wird dann automatisch aktualisiert, wenn sich letzteres ändert

So gut wie alle Eigenschaften von UI-Elementen in JavaFX sind Properties. Dementsprechend wichtig ist es, mit den Properties richtig umzugehen. Wenn man die Entität-Klassen auch aus JavaFX-Properties aufbaut, kann man das Property der UI-Elemente an die Entitäten direkt binden, wie das Beispiel in *Listing 1* zeigt.

Der große Vorteil dieses Ansatzes ist, dass man sich manuelle Updates von UI-Elementen erspart. Der von „Label“ dargestellte Text wird automatisch jedes Mal aktualisiert, wenn sich der Wert von „Employee.name“ ändert. Natürlich kann man beliebig viele UI-Elemente an

„nameProperty()“ binden und auch die Richtung des Bindings umdrehen, indem man „nameProperty()“ beispielsweise an den aktuellen Wert eines „TextField“ bindet *(siehe Listing 2)*.

Damit wird die Entität automatisch aktualisiert, wenn der Benutzer etwas in „TextField“ schreibt. Auch eine bidirektionale Bindung ist möglich: „employee.nameProperty().bindBidirectional(nameField.textProperty());“. Die automatische Synchronisation von Entitäten und UI ist zwar sehr schön, allerdings bringt sie auch einige Nachteile:

- Die Entität-Klasse, die ja normalerweise Technologie-unabhängig sein sollte, hat jetzt eine Abhängigkeit auf „javafx.property.*“. Das Konzept von Properties und Bindings ist an sich UI-unabhängig, leider hat man es aber versäumt, die entsprechenden Klassen in ein geeignetes Package zu geben, sodass man sie auch außerhalb von JavaFX verwenden kann.
- Employee ist nicht mehr serialisierbar, weil „Employee.name“ – eine „SimpleStringProperty“ – nicht serialisierbar ist.
- Da es sich bei „Employee“ nicht mehr um ein einfaches POJO handelt, kann es bei Integration anderer Frameworks wie JPA zu Problemen kommen.

Es ist also nicht möglich, dass sich beispielsweise der JavaFX- und der Server-Teil einer großen Anwendung das Modell teilen. Um die Vorteile der JavaFX-Properties voll nutzen zu können, braucht man also ein eigenes Modell für das UI, das aus JavaFX-Properties aufgebaut ist, was zu Boilerplate-Code führen kann, weil man eventuell zwischen dem JavaFX-Modell und dem Nicht-JavaFX-Modell übersetzen muss. Meistens lohnt es sich dennoch, diese kleinen Nachteile in Kauf zu nehmen, denn ein Modell aus Properties macht die Entwicklung mit JavaFX wesentlich einfacher.

UI im FXML definieren

Imperative Programmiersprachen waren nie wirklich dazu geeignet, UIs zu definieren. Jeder, der schon mal ein komplexeres UI in reinem Java schreiben wollte, kennt dieses Problem. Diverse UI-Editoren bieten nur begrenzt Abhilfe, weil der von ihnen generierte Code oft unnötig kompliziert und schwer anpassbar ist. JavaFX versucht, dieses Problem mit FXML zu lösen, und das durchaus mit Erfolg. Man definiert seine UIs in XML und vergibt IDs an die einzelnen UI-Elemente *(siehe Listing 3)*. Die UI-Elemente werden dann in den Controller injiziert *(siehe Listing 4)*.

Eine neue Instanz unseres UI kann sehr einfach mit dem „FXMLLoader“ erstellt werden: „Parent view = FXMLLoader.load(ControllerExampleApplication.class.getResource(„/at/irian/javafx/demo/fxml/FxmlExample.fxml“));“ *(siehe Abbildung 1)*. Die Verwendung von FXML ist zwar optional (man kann seine UIs natürlich auch im Code definieren), aber sehr empfehlenswert. Es ermöglicht eine saubere Trennung von Aussehen und Verhalten. Mit FXML kann man das Layout und das Styling des UI ändern, ohne eine einzige Java-Klasse anfassen zu müssen, solange sich die Typen der vom Controller referenzierten UI-Elemente und deren „fx:id“ nicht ändern.

FXML und Property

Im folgenden Beispiel gibt es ein optionales „TextField“, das nur dann sichtbar sein soll, wenn eine „CheckBox“ angekreuzt ist *(siehe Abbildung 2)*. Dank des Property lässt sich diese Funktionalität sehr leicht umsetzen *(siehe Listing 5)*. Die „initialize()“-Methode wird vom „FXMLLoader“ aufgerufen, nachdem die mit „@FXML“ annotierten Variablen injiziert wurden. Dies ist die geeignete Stelle, um solche Bindings anzulegen.

Abbildung 1: Das im „FxmlExample.fxml“ definierte UI

Abbildung 2: Optional sichtbares „TextField“

```
public class ControllerExample {

  @FXML private TextField firstName;
  @FXML private TextField lastName;

  public void submit() {
     // submit the data;
  }

  public void reset() {
     firstName.setText("");
     lastName.setText("");
  }
}
```

Listing 4

```
<HBox xmlns:fx="http://javafx.com/fxml" fx:controller="at.irian.javafx.demo.binding.OptionalField">
  <CheckBox fx:id="useCustomValue" text="Use optional value"/>
  <TextField fx:id="customValue"/>
</HBox>

public class OptionalField {

  @FXML private CheckBox useCustomValue;
  @FXML private TextField customValue;

  public void initialize(){
     customValue.visibleProperty().bind(useCustomValue.selectedProperty());
  }
}
```

Listing 5

Ein relativ wenig bekanntes Feature von JavaFX ist die Möglichkeit, Property-Bindings auch in FXML zu erstellen. Statt der Methode „initialize()“ im Controller lässt sich das Binding zwischen dem „visibleProperty()“ des „TextField“ und dem „selectedProperty()“ der „CheckBox“ auch im FXML definieren als: „<TextField fx:id=“customValue“ visible=“${useCustomValue.selected}“/>“.

Obwohl es auf den ersten Blick vielleicht so aussieht, handelt es sich bei „${useCustomValue.selected}“ um keinen klassischen EL-Ausdruck, wie wir es aus JSP oder JSF kennen. Der „$“-Operator ermöglicht es, UI-Elemente anhand ihrer „fx:id“ und ihres Property aufzulösen, aber mehr nicht. Komplexere Ausdrücke sind nicht möglich und meistens auch nicht erwünscht, denn schließlich soll das FXML nur das Aussehen und nicht das Verhalten definieren.

```
public class Employee {
  private final StringProperty name = new SimpleStringProperty();
  private final ObjectProperty<Company> company = new SimpleObjectProperty<>();

  public StringProperty nameProperty() {
     return name;
  }

  public ObjectProperty<Company> companyProperty() {
     return company;
  }

  // getters and setter omitted.
}

public class Company {
  private final StringProperty name = new SimpleStringProperty();

  public StringProperty nameProperty() {
     return name;
  }

  // getters and setter omitted.
}
```

Listing 6

```
public class EmployeePane extends BorderPane {
  public EmployeePane(Employee employee) {
     Label companyNameLabel = new Label(); companyNameLabel.textProperty().
bindBidirectional(employee.getCompany().nameProperty());
     setCenter(companyNameLabel);
  }
}
```

Listing 7

```
public class EmployeePane extends BorderPane {
  public EmployeePane(Employee employee) {
     Label companyNameLabel = new Label();
     companyNameLabel.textProperty().bind(Bindings.select(employee.
companyProperty(), „name“));
     setCenter(companyNameLabel);
  }
}
```

Listing 8

```
Bindings.select():
companyNameLabel.textProperty().bind(EasyBind.select(employee.
companyProperty()).selectObject(Company::nameProperty));
```

Listing 9

```
public void loadEmployeeCompany(Employee employee) {
  ExecutorService.execute(() -> {
     Company company = loadCompanyFromDB(employee);
     Platform.runLater(() -> employee.setCompany(company));
  });
}
```

Listing 10

Verkettung

Das „Employee“-Beispiel wird nun um eine neue Entität namens „Company“ sowie eine Beziehung zwischen „Employee“ und „Company“ erweitert *(siehe Listing 6)*. Danach wird „EmployeePane“ so geändert, dass nicht der Name des „Employee“ anzeigt wird, sondern der Name der „Company“, für die er arbeitet. Natürlich erfolgt das wieder mit einem Binding, doch das ist nicht mehr so einfach. *Listing 7* zeigt einen naiven Ansatz, der allerdings ein großes Problem hat.

„Label“ wird nur dann aktualisiert, wenn sich „nameProperty()“ der „Company“ ändert. Wenn „Employee“ allerdings zu einer neuen „Company“ wechselt, erscheint weiterhin der alte Wert. Dieses Problem kommt relativ häufig vor. Es besteht hier eine Verkettung von Properties: „Employee.company.name“. Es soll immer der aktuelle Wert am Ende der Kette angezeigt werden. Dabei kann jedes Property entlang der Kette seinen Wert ändern. Natürlich kann der Wert von „Employee.company“ auch „null“ sein. Es gibt zwei Möglichkeiten, dieses Problem zu lösen. *Listing 8* zeigt, wie man mit „Bindings.select()“ eine Verkettung von Properties binden kann.

Natürlich könnte die Kette auch länger sein: „Bindings.select(employee.companyProperty(), „address“, „country“, ...))“. Dieser Lösungsansatz hat jedoch einen entscheidenden Nachteil: Properties werden als Strings übergeben. Das ist nicht typsicher und es wird erst zur Laufzeit überprüft, ob es das Property auch wirklich gibt. Das macht das Refactoring des Modells schwierig und riskant. Das Projekt „EasyBind“ bietet eine interessante Alternative dazu *(siehe Listing 9)*. Dort werden die Properties entlang der Kette nicht als Strings, sondern als Method-References übergeben. Das Ergebnis ist ein typsicheres Binding, das dank Java 8 genauso kompakt wie „Bindings.select()“ ist.

Property und Thread

Viele Properties von verschiedenen UI-Elementen dürfen nur aus dem FX-Application-Thread gesetzt werden (etwa „ChoiceBox.valueProperty()“). Wenn man also ein solches Property an das Datenmodel bindet, muss man sicherstellen, dass das Modell ausschließlich aus dem FX-Application-Thread aktualisiert wird. Andererseits sollte man teure Operationen wie I/O im FX-Application-Thread vermeiden, damit das UI schön „responsive“ bleibt. Um diesen beiden Anforderungen gerecht zu werden, muss man also die teuren Operationen außerhalb von FX-Thread ausführen, das Modell dann aber wieder mit „Platform.runLater()“ im FX-Thread aktualisieren *(siehe Listing 10)*.

Memory Leaks

Wie schon erwähnt, kann man bei jedem Property beliebig viele Listener registrieren, die aufgerufen werden, wenn sich der Wert ändert. Jetzt wird „EmployeePane“ so erweitert, dass ein Sound abgespielt wird, wenn sich „Employee.company“ ändert. Dazu fügt man der „companyProperty()“ einen „InvalidationListener“ hinzu *(siehe Listing 11)*.

Diese Implementierung funktioniert zwar, führt jedoch zu einem Memory-Leak. Die „EmployeePane“ wird nie vom Garbage Collector erfasst, auch wenn sie längst nicht mehr sichtbar ist, weil „Employee.companyProperty()“ jetzt intern eine Referenz auf den Listener und somit auch auf „EmployeePane“ hat. Die „EmployeePane“ wird also erst dann aus dem Speicher gelöscht, wenn auch die „Employee“-Entität gelöscht wird. Man sollte deshalb nie einen harten Listener eines kurzlebigen Objekts (wie „UI“) auf ein Property eines langlebigen Objekts (z.B. Entität) registrieren. Für dieses Problem gibt es zwei Lösungen. Erstens, man entfernt den Listener explizit, wenn die „EmployeePane“ geschlossen wird *(siehe Listing 12)*.

Nachteil dieser Lösung ist, dass man „EmployeePane.destroy()“ explizit aufrufen muss, wenn das UI nicht mehr verwendet wird. Die zweite Lösung sieht einen „WeakInvalidationListener“ vor, der den eigentlichen Listener mit einer „WeakReference“ wrappt: „employee.companyProperty().addListener(new WeakInvalidationListener(observable -> playSound()));“. Ein explizites Entfernen des Listeners ist in diesem Fall nicht notwendig.

Observable Collections

Neben den Properties bietet JavaFX auch eine erweiterte Version der klassischen Collections „ObservableList“, „ObservableMap“ und „ObservableSet“. Zusätzlich zu der normalen Funktionalität kann man bei den „observable“ Collections ähnlich wie bei Properties einen Listener registrieren, der aufgerufen wird, wenn sich der Inhalt der Collection ändert *(siehe Listing 13)*.

Die Utility-Klasse „FXCollections“ bietet einige nützlichen Methoden für die „observable“ Collections und ist stark an „java.util.

```
public class EmployeePane extends BorderPane {
  public EmployeePane(Employee employee) {
     Label employeeName = new Label();
     employeeName.textProperty().bind(employee.nameProperty());
     employee.companyProperty().addListener(observable -> playSound());
     setCenter(employeeName);
  }
}
```

Listing 11

```
public class EmployeePane extends BorderPane {
  private final Employee employee;

  public EmployeePane(Employee employee) {
     this.employee = employee;
     Label employeeName = new Label();
     employeeName.textProperty().bind(employee.nameProperty());
     employee.companyProperty().addListener(observable -> playSound());
     employee.companyProperty().addListener(this::companyChanged);
     setCenter(employeeName);
  }

  public void companyChanged(Observable observable) {
     playSound();
  }

  public void destory() {
     employee.companyProperty().removeListener(this::companyChanged);
  }

}
```

Listing 12

```
ObservableSet<String> observableSet = FXCollections.observableSet();
observableSet.addListener((SetChangeListener.Change<? extends String> change) ->
  System.out.println(„Content of the collection changed.“)
);
```

Listing 13

```
public class CollectionHolder {
  private final ObservableSet<String> set = FXCollections.observableSet();

  public ObservableSet<String> getSet() {
     return FXCollections.observableSet(set);
  }
  public void add(String element) {
     set.add(element);
  }
}

public class UnmodifiableCollectionsExample {
  public void addListener(CollectionHolder collectionHolder) {
     collectionHolder.getSet().addListener((SetChangeListener.Change<? ex-
tends String> change) ->
        System.out.println(„Content of the collection changed.“)
     );
     collectionHolder.add(„foo“);
  }
}
```

Listing 14

Collections" angelehnt. Unter anderem gibt es Methoden, die eine „read only"-Ansicht einer Collection zurückliefern:

- ObservableList<E> unmodifiableObservableList(ObservableList<E> list)
- ObservableMap<K,V> unmodifiableObservableMap(ObservableMap<K,V> map)
- ObservableSet<E> unmodifiableObservableSet(ObservableSet<E> set)

Hier gilt es aber seit Java 8 ganz genau aufzupassen, denn die zurückgelieferte „observable" Collection ist mit der ursprünglichen Collection nur durch einen „weak"-Listener verbunden. So kann es passieren, das Listener nicht aufgerufen werden, wenn die „read-only"-Collection aus dem Scope fällt *(siehe Listing 14)*.

Da der „CollectionHolder" immer eine neue Instanz der „read only"-Collection zurückliefert und diese im „UnmodifiableCollectionsExample" nicht gespeichert wird, kann die „read only"-Collection des Garbage Collector samt allen Listenern gelöscht werden. Damit wird der Listener nach dem nächsten GC-Durchlauf beim „collectionHolder.add(„foo")" nicht mehr aufgerufen. Dieses Verhalten ist neu seit Java 8, die Dokumentation wurde jedoch leider nicht entsprechend angepasst.

Fazit

Das Konzept von Property und Binding ist sehr mächtig, erfordert allerdings anfangs ein gewisses Umdenken. Die hier beschriebenen Pitfalls sind sicher kein Grund, JavaFX nicht zu verwenden, ganz im Gegenteil – JavaFX ist ein sehr gelungener Nachfolger von Swing.

Jan Zarnikov
jan.zarnikov@irian.at

Jan Zarnikov studierte Informatik an der Technischen Universität Wien. Seit dem Jahr 2010 arbeitet er als Software-Entwickler für Irian Solutions, wo er hauptsächlich mit Java beschäftigt ist. Derzeit ist er als Consultant bei einem JavaFX-Projekt für einen großen Kunden aus Deutschland tätig.

http://ja.ijug.eu/15/3/8

Asynchrone JavaFX-8-Applikationen mit JacpFX

Andy Moncsek, Trivadis AG

JacpFX ist eine JavaFX-basierte Rich Client Platform (RCP) mit Fokus auf einfacher Strukturierung von Applikationen, Kommunikation zwischen Komponenten und Unterstützung von asynchronen Prozessen im User Interface (UI).

UI-Toolkits gehören noch heute zu den wenigen Beispielen in der Software-Entwicklung, die überwiegend „Single-thread" sind, und das aus gutem Grund. „Multi-threading" steigert einerseits die Gefahr von Deadlocks und Race Conditions, andererseits steigt die Komplexität im Toolkit selbst. Das Abstract Window Toolkit (AWT) wurde beispielsweise ursprünglich als „multi-threaded" Java Library entworfen. Angesichts der Komplexität und der ersten Erfahrungen mit den Prototypen ist dieser Ansatz verworfen worden *[1]*. Stattdessen einigte man sich auf das bekannte Event-getriebene Modell im Event Dispatch Thread (EDT). Dieser „single-threaded"-Ansatz führt allerdings traditionell zu den üblichen Problemen der blockierenden UIs bei lang laufenden Prozessen wie zum Beispiel Datenbank-Anfragen.

JavaFX ist in dieser Hinsicht seinem Vorgänger Swing sehr ähnlich. Das Konzept des JavaFX Application Thread entspricht in etwa dem bekannten Event Dispatch Thread (EDT) aus Swing/AWT. Alles läuft auf einem Thread, und wenn dieser beschäftigt ist, blockiert die UI. Natürlich gab und gibt es Lösungen, um lang laufende Prozesse auszulagern. In JavaFX dürfen beispielsweise neue Knoten auch außerhalb des Application Thread erstellt oder manipuliert werden, solange sie nicht im JavaFX SceneGraph eingebunden sind *[2]*.

Mit „SwingWorker" etablierte sich in Swing ein Hilfsmittel, um Prozesse auszulagern und das Ergebnis dem EDT zu übergeben. JavaFX hat diesen Ansatz weiterverfolgt und bietet die Möglichkeit, Prozesse in „Tasks" und „Services" auszulagern. Ein „Task" *(siehe Listing 1)* ermöglicht die einma-

lige Ausführung eines lang laufenden Prozesses im Hintergrund und erlaubt es, dabei Status-Informationen an die GUI weiterzugeben. Ein „Service" hingegen verwendet einen „Task" und kann für wiederkehrende Prozesse eingesetzt werden.

Lang laufende Prozesse sind jedoch nur ein Aspekt, bei dem der Application Thread beachtet werden muss. Callback Events aus anderen Frameworks (etwa bei Verwendung von WebSocket, MQTT oder JMS) müssen Änderungen an der UI ebenfalls über diesen durchführen. Dieser Übergang zum Application Thread wurde in Swing mithilfe der Klasse „SwingUtilities" und der Methoden „invokeAndWait" und „invokeLater" umgesetzt. In JavaFX wird diese Rolle durch die Klasse „javafx.application.Platform" übernommen, allerdings bietet diese nur eine „runLater"-Methode an *(siehe Listing 2)*.

```
Task task = new Task<Void>() {
@Override public Void call() {
static final int max = 1000000;
for (int i=1; i<=max; i++) {
if (isCancelled()) {
break;
}
updateProgress(i, max);
}
return null;
}
};
ProgressBar bar = new ProgressBar();
bar.progressProperty().bind(task.progressProperty());
new Thread(task).start();
```

Listing 1

```
public class MqttGUI extends VBox implements MqttCallback {
private TextField textField;
@Override
public void messageArrived(String s, MqttMessage m) throws Exception {
// schlägt fehl, da nicht in Application Thread !
textField.setText(m.getPayload());
// richtig:
Platform.runLater(()->textField.setText(m.getPayload()));
}
}
```

Listing 2

JacpFX

In einem umfangreichen JavaFX-Projekt wird man sich früher oder später mit beiden Aspekten des JavaFX Application Thread auseinandersetzen müssen. Diese Problematik für Entwickler zu vereinfachen, ist eines der Ziele von JacpFX.

Generell folgt JacpFX einem Komponenten-orientierten Ansatz und unterscheidet dabei zwischen Workbench, Perspective, UI- und Non-UI-Komponenten *(siehe Abbildung 1)*.

Referenzen zwischen der Workbench, ihren Perspektiven und deren Komponenten werden per ID hergestellt. Dabei haben JacpFX-Komponenten nie direkte Abhängigkeiten zu anderen Komponenten und sind somit lose gekoppelt *(siehe Abbildung 2)*.

Das aus dieser Hierarchie resultierende ID-Schema („perspectiveId.componentId") dient gleichzeitig als Adressschema für den Message Bus in JacpFX. Dieser erlaubt es, Komponenten zu benachrichtigen, ohne eine direkte Abhängigkeit zwischen ihnen zu schaffen. Eine eintreffende Nachricht löst dabei folgenden Zwei-Phasen-Lifecycle in der Ziel-Komponente aus *(siehe Abbildung 3)*:

- In der Phase eins („Task") erfolgt die Ausführung der „handle(message)"-Methode der Ziel-Komponente im Worker Thread

```
mvn archetype:generate  -DarchetypeGroupId=org.jacpfx
-DarchetypeArtifactId=JacpFX-simple-quickstart -DarchetypeVersion=2.0.2
```

Listing 3

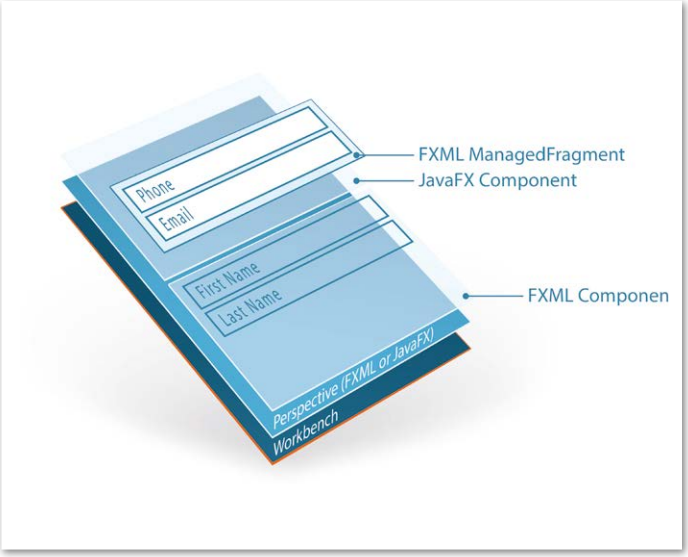


Abbildung 1: JacpFX-Komponenten-Struktur

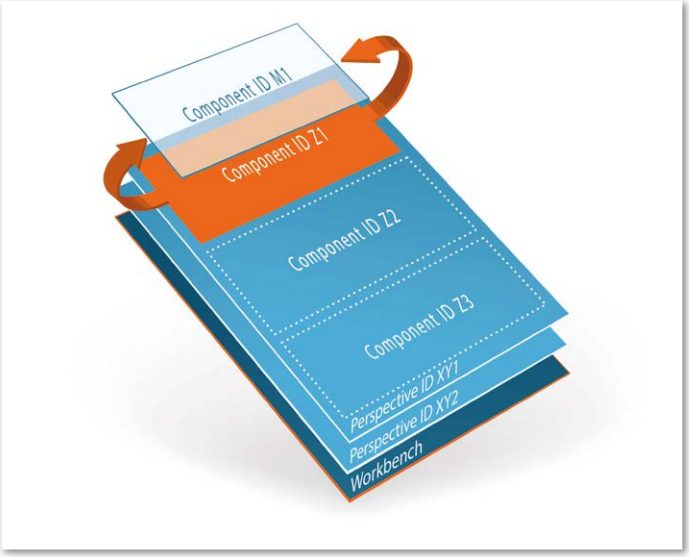


Abbildung 2: Lose Kopplung von JacpFX-Komponenten

- Der Rückgabewert wird anschließend in Phase 2 („State“) der „postHandle(value, message)“-Methode im Application Thread übergeben

Eine Komponenten-Instanz ist in JacpFX immer eindeutig pro Perspektive. Komponenten können jedoch durch mehrere Perspektiven referenziert oder zwischen ihnen verschoben werden. Mit diesem Ansatz soll die Wiederverwendung von Komponenten erleichtert werden, wobei das Adress-Schema eindeutig bleibt.

Die Verarbeitung der eintreffenden Nachrichten erfolgt innerhalb eines Worker Thread und ist immer sequenziell. Da Komponenten auch keine direkten Referenzen untereinander haben, ist eine Synchronisierung von Ressourcen nicht notwendig. Entwickler können den Komponenten-Lifecycle für nebenläufige Prozesse nutzen und müssen sich nicht mit Details der JavaFX-Implementierung und den Fallstricken von Multi Threading auseinandersetzen.

Es folgt ein Beispiel dafür, wie man mit JacpFX eine einfache Chat-Applikation erstellt und darin mit WebSocket-Callback-Events umgeht. Dieser Ansatz kann natürlich auch für klassische Datenbank-Abfragen oder andere lang laufende Prozesse, in denen die GUI nicht blockiert werden soll, übernommen werden *(siehe Abbildung 4)*.

```
public class ApplicationLauncher extends AFXSpringJavaConfigLauncher {
@Override
protected Class<? extends FXWorkbench> getWorkbenchClass() {
return JacpFXWorkbench.class;
}

@Override
protected String[] getBasePackages() {
return new String[]{"package.name"};
}

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

@Override
public void postInit(Stage stage) {
}

@Override
protected Class<?>[] getConfigClasses() {
return new Class<?>[]{BasicConfig.class};
}
}
```

Listing 4

```
@Workbench(id = "id1", name = "workbench",
perspectives = {
BasicConfig.PERSPECTIVE_ONE
})
public class JacpFXWorkbench implements FXWorkbench { … }
```

Listing 5

Erstellen einer JacpFX-Applikation

Zum Erstellen einer JacpFX-Applikation verwenden wir den „JacpFX-simple-quickstart“-Maven-Archetype *(siehe Listing 3)*. Dieser legt eine kleine Beispiel-Applikation an, die im Folgenden zu einer Chat-Applikation angepasst wird. Die wesentlichen Bestandteile der Beispiel-Applikation sind der „ApplicationLauncher“, die „JacpFX-Workbench“, eine FXML-Perspektive, zwei UI-Komponenten und eine Non-UI-Komponente *(siehe Abbildung 5)*.

Der „ApplicationLauncher“ enthält die „main“-Methode und die Angabe, welche Workbench-Klasse verwendet werden soll. Der eingesetzte „ApplicationLauncher“ ist vom Typ „AFXSpringJavaConfigLauncher“ und nutzt Spring 4.x zum Starten aller Komponenten *(siehe Listing 4)*.

Die „JacpFXWorkbench“ dient der Basis-Konfiguration der Applikation. Darin lassen

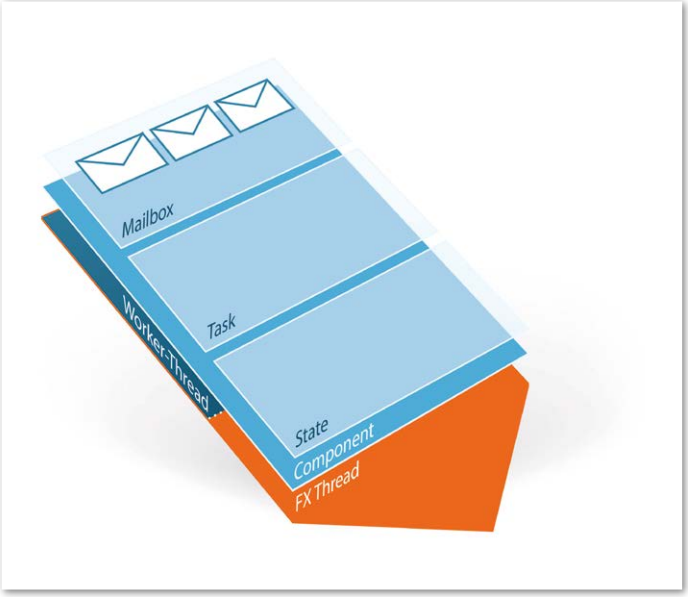


Abbildung 3: JacpFX-Komponenten-Lifecycle

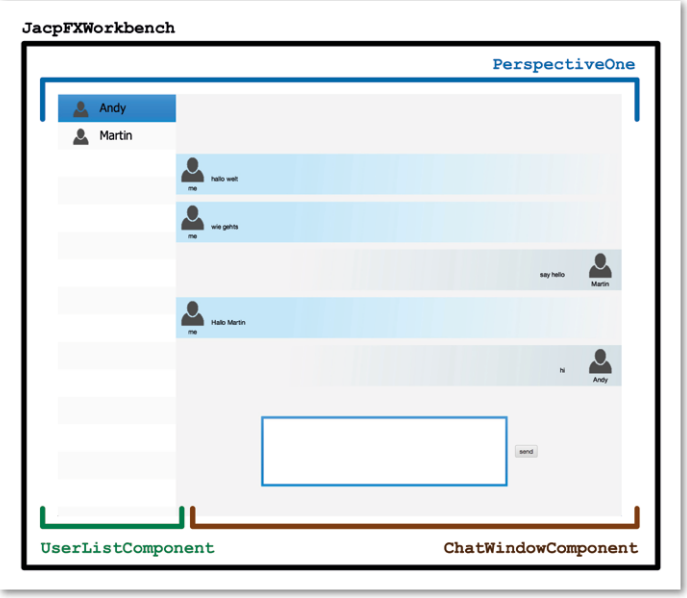


Abbildung 4: JacpFX-Chat-Applikation

sich die Fenstergröße bestimmen, Menüs und Toolbars aktivieren sowie alle Referenzen zu den Perspektiven in unserer Applikation definieren. Für das Beispiel benötigt man nur eine Perspektive und passt die Referenzen in der Workbench entsprechend an *(siehe Listing 5)*.

Die Klasse „PerspectiveOne" wird nun genauer betrachtet *(siehe Listing 6)*. Eine Perspektive ist in JacpFX eine Controller-Klasse mit einer View. Diese definiert das Layout für die aktuelle Ansicht in der Workbench. „PerspectiveOne" verwendet dabei eine FXML-View, alternativ kann die View auch programmatisch mit JavaFX erstellt sein.

In der „@Perspective"-Annotation sind unter anderem Referenzen zu den Komponenten und der verwendeten FXML-View der Perspektive definiert. Eine entscheidende Aufgabe der Perspektive ist die Definition von Platzhaltern, in denen die UI-Komponenten dargestellt werden. Innerhalb der „@PostConstruct"-Methode registriert man dafür im „PerspectiveLayout" zwei JavaFX-„GridPanes", die als UI-Container für die Komponenten zum Einsatz kommen *(siehe Listing 7)*.

Perspektiven dienen ausschließlich der Definition des Layouts und laufen immer im Application Thread, daher sollten hier keine blockierenden Prozesse ausgeführt werden. Nun geht es um die referenzierten JacpFX-Komponenten der Perspektive: Jede Komponente ist einem eigenen Worker Thread zugeordnet, der die eingehenden Nachrichten verarbeitet, die „handle(message)"-Methode ausführt und den Übergang zum Application Thread regelt. Dabei besteht eine UI-Komponente, ähnlich einer Perspektive, aus einer View und einer Controller-Klasse. Die View kann ebenso wie bei Perspektiven als FXML oder in JavaFX vorliegen.

Die Beispiel-Applikation soll auf der linken Seite eine Übersicht aller angemeldeten User und auf der rechten Seite das Chat-Fenster mit den Nachrichten anzeigen *(siehe Abbildung 4)*. Für die Chat-Applikation geht man von zwei WebSocket-Endpunkten aus: „ws://host/users" und „ws://host/chat".

Die „UserListComponent"-Komponente listet alle angemeldeten User im Chat auf. Der Controller dient dabei gleichzeitig als WebSocket Client, der bei eintreffender Nachricht eine Änderung in der GUI vornimmt. Die „ChatWindowComponent"-Komponente hingegen lagert die WebSocket-Implementierung exemplarisch in eine JacpFX-Non-UI-Komponente aus und kommuniziert mit ihr über den JacpFX Message Bus.

Listing 8 stellt einen Ausschnitt der „UserListComponent"-Komponente dar. Die „handle(message)"-Methode wird hier nicht verwendet. Es können aber nach Belieben lang laufende Prozesse ausgeführt werden. Währenddessen wird die Komponenten-View nicht blockiert und die UI ist für den Anwender weiterhin nutzbar. In der „postHandle(value,message)"-Methode fügen wir neue Chat-User ein, die von der WebSocket-@OnMessage-Methode empfangen und an den JacpFX Message Bus weitergeleitet werden *(siehe Listing 9)*.

Für die Chat-Ansicht verwendet man zusätzlich eine Non-UI-Komponente und trennt damit die Service-Implementierung von der UI-Logik. Durch die lose Kopplung aller Komponenten in JacpFX lässt sich so jederzeit die Implementierung schnell auswechseln und beispielsweise durch eine SocketJS-Implementierung austauschen. Aus Sicht der UI-Komponente werden somit Nachrichten mit einer weiteren JacpFX-Komponente aus-

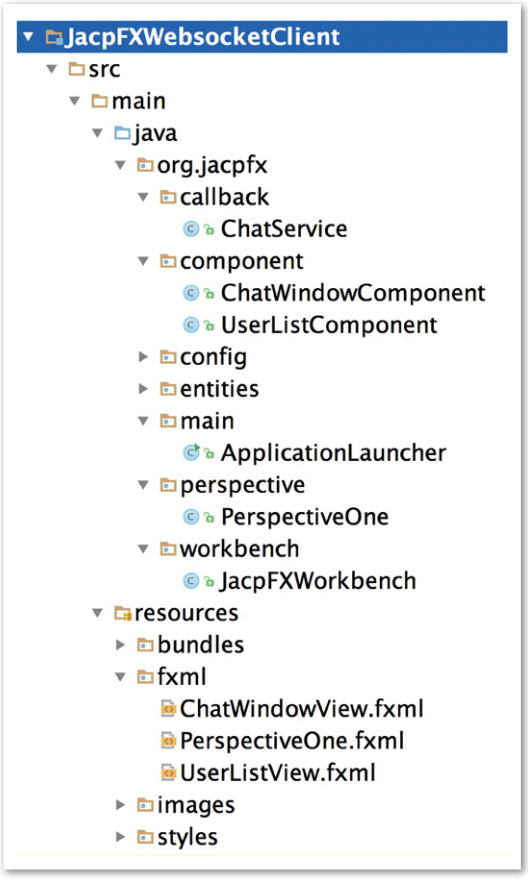


Abbildung 5: JacpFX-WebsocketClient-Struktur

```
@Perspective(id = BasicConfig.PERSPECTIVE_ONE,
name = "contactPerspective",
components = {
BasicConfig.COMPONENT_LEFT,
BasicConfig.COMPONENT_RIGHT,
BasicConfig.STATEFUL_CALLBACK},
viewLocation = "/fxml/PerspectiveOne.fxml",
resourceBundleLocation = "bundles.languageBundle")
public class PerspectiveOne implements FXPerspective {
@FXML
private GridPane left;
@FXML
private GridPane main;
@Resource
public Context context;

@Override
public void handlePerspective(Message<Event, Object> m, PerspectiveLayout l) {
        …
}

@PostConstruct
public void onStartPerspective(final PerspectiveLayout pl, FXComponentLayout l,
ResourceBundle b) {
pl.registerTargetLayoutComponent(BasicConfig.LEFT, left);
pl.registerTargetLayoutComponent(BasicConfig.RIGHT, main);
}

@PreDestroy
public void onTearDownPerspective(final FXComponentLayout layout) {…}
}
```

Listing 6

```
pl.registerTargetLayoutComponent(BasicConfig.LEFT, left);
pl.registerTargetLayoutComponent(BasicConfig.RIGHT, main);
```

Listing 7

getauscht, ohne weitere Details der Service-Implementierung zu kennen.

Listing 10 zeigt den JacpFX-Chat-Service „ChatService". Diese Non-UI-Komponente läuft immer innerhalb eines Worker Thread und kann somit die UI nicht blockieren. Die „handle(message)"-Methode verarbeitet Chat-Nachrichten, die von der UI-Komponente gesendet werden. Diese Nachrichten werden mithilfe der WebSocket-Session an den Chat-Endpunkt geleitet, der sie an den eigentlichen Adressaten schickt.

Der Rückgabewert der „handle(message)"-Methode wird normalerweise an den Sender zurückgeschickt. In diesem Fall verzichtet man auf eine Antwort und gibt stattdessen „Null" zurück, wodurch eine Rückantwort unterdrückt wird. Die @OnMessage-Methode empfängt eingehende Nachrichten vom WebSocket-Endpunkt und leitet sie an die „ChatWindowComponent"-Komponente weiter.

Listing 11 zeigt einen Ausschnitt der „ChatWindowComponent"-Komponente. In der „postHandle(value,message)"-Methode wird die eingehende Nachricht zum Chat-Fenster hinzugefügt. Während der gesamten Kommunikation braucht man den Application Thread nicht zu beachten. Durch das Routing über den JacpFX Message Bus entfällt ein manuelles Eingreifen zum Synchronisieren der Threads.

Fazit

Mit dieser Beispiel-Applikation ist eine asynchrone, Nachrichten-basierte JavaFX-Applikation entstanden, ohne sich mit den Details des JavaFX Application Threads auseinandersetzen zu müssen. Dieser Ansatz kann ebenso verwendet werden, um klassische Datenbank-Anfragen oder andere lang laufende Prozesse asynchron zu verarbeiten, ohne die UI zu blockieren. Der Nutzer kann beispielsweise eine Suchanfrage starten und währenddessen in einer anderen Perspektive weiterarbeiten.

Dem Entwickler hilft JacpFX, die Applikation zu strukturieren, die Kommunikation zwischen Komponenten zu erleichtern und die Handhabung von asynchronen Prozessen innerhalb der Anwendung zu vereinfachen. Aktuell liegt JacpFX in Version 2.0.1 vor, ein Release 2.1 ist für Ende Sommer 2015 geplant.

Weitere Informationen

[1] https://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html
[2] http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-architecture.htm
[3] http://jacpfx.org
[4] http://jacpfx.org/documentation_main.html
[5] https://github.com/JacpFX/JacpFX-demos/tree/master/JacpFXWebsocketClient

Alle weiteren Listings finden Sie online unter:
www.ijug.eu/go/java_aktuell/201503/listings

```
@DeclarativeView(id = BasicConfig.COMPONENT_LEFT,
name = "SimpleView",
viewLocation = "/fxml/UserListView.fxml",
active = true,
resourceBundleLocation = "bundles.languageBundle",
initialTargetLayoutId = BasicConfig.LEFT)
@ClientEndpoint(decoders = UserDecoder.class)
public class UserListComponent implements FXComponent {
@Resource
private Context context;
@FXML
private ListView userList;

private ObservableList<User> users;
private static final String WEBSOCKET_URL="ws://…";

@Override
public Node handle(final Message<Event, Object> message) {
// runs in worker thread
return null;
}

@Override
public Node postHandle(Node resultNode, Message<Event, Object> m) {
// runs in FX application thread
if(m.isMessageBodyTypeOf(User.class)) {
users.add(m.getTypedMessageBody(User.class));
}
return null;
}

@PostConstruct
public void onPostConstructComponent(FXComponentLayout layout, final Re-
sourceBundle b) {
            …
connectToEndpoint();
}

private void connectToEndpoint() {
…
WebSocketContainer container = ContainerProvider.getWebSocketContainer();
container.connectToServer(this, URI.create(ComponentLeft.WEBSOCKET_URL));

…
     }

@OnMessage
public void onNewUser(User user) {
this.context.send(user);
}
}
```

Listing 8

Andy Moncsek
andy.moncsek@trivaids.com
@AndyAHCP

Andy Moncsek arbeitet als Senior Consultant im Bereich „Application Development" bei der Trivadis AG in Zürich. Neben seinem Fokus auf Java-Enterprise-Applikationen und -Architekturen ist er als Disziplinen-Manager „Java Application Development" tätig. In seiner Freizeit leitet er das Open-Source-Projekt „JacpFX" und ist in vielen Bereichen des Java-Ökosystems unterwegs.

http://ja.ijug.eu/15/3/9

Magnolia mit Thymeleaf – ein agiler Prozess-Beschleuniger

Thomas Kratz, mecom GmbH

Der typische Ablauf bei der Entwicklung einer Webseite: Ein UX-Designer arbeitet ein Layout im PSD-Format aus, die Web-Entwickler setzen das Layout in statischen HTML-Templates um und die CMS-Experten zerlegen diese in Komponenten und machen sie für den Endanwender pflegbar. Gibt es einen Änderungswunsch, wird der Prozess wiederholt. Dieser Artikel zeigt am Beispiel von Magnolia einen Ansatz, bei dem Web-Entwickler und CMS-Experten mit ein und demselben Quellcode arbeiten und so zahllose Roundtrips im Änderungsprozess vermieden oder zumindest stark beschleunigt werden können. Der Ansatz lässt sich leicht auf andere Content-Management-Systeme übertragen.

Betrachten wir das Vorgehen einmal an einem einfachen Beispiel genauer. Ein Web-Entwickler entwickelt eine Teaser-Komponente als statisches Mark-up. Er liefert den CMS-Entwicklern ein HTML-Template sowie das zugehörige CSS beziehungsweise alle JavaScript-Artefakte, die dafür benötigt werden. Für diesen Artikel gibt es ein einfaches Beispiel auf Basis von Twitter Bootstrap *(siehe Listing 1)*.

In der Regel ist das natürlich etwas komplexer als in diesem Beispiel. Für den CMS-Entwickler ist als Teaser-Komponente eigentlich nur der „ ... " von Interesse; er wird dafür ein Komponenten-Template und Pflege-Dialoge erstellen, bei Magnolia typischerweise mit Freemarker oder „.jsp" *(siehe Listing 2)*. Dazu gehört natürlich ein Pflegedialog, der bei Magnolia mit Blossom zum Beispiel wie in *Listing 3* aussehen könnte. Gleichzeitig entsteht dabei noch eine zweite Komponente für den Footer, der ebenfalls pflegbar gemacht werden soll *(siehe Listing 4)*. Auch zum Footer gehört ein Pflege-Dialog, der hier der Kürze wegen nicht zu sehen ist.

Die Problematik

Schon an dem einfachen Beispiel wird deutlich, was hier geschieht. Aus einem einfachen Mark-up entstehen bei der Umsetzung im Magnolia-CMS zahlreiche Artefakte, da die Seite in pflegbare Teilmodule zerlegt werden muss. Tatsächlich sind es noch viel mehr, als hier aufgezeigt, denn es gibt auch noch ein Rahmen-Template für die Seite, in die die Komponenten eingebettet werden; für alle Stellen auf dem Seiten-Template, an denen Komponenten eingefügt werden sollen, entsteht auch noch ein sogenanntes „Area-Template".

Nun wünscht sich der Kunde noch einen Link-Button für den Teaser. Dieser wird nach den Vorgaben der Designer (vermutlich eine neue Version der PSD-Vorlage im statischen Template) gestylt und das ursprüngliche HTML ändert sich *(siehe Listing 5)*.

Der CMS-Entwickler erhält nun den Auftrag, diese Änderung ins Magnolia-System zu übertragen. Dabei behilft er sich womöglich mit einem „Diff"-Werkzeug, um die geänderten Stellen zu identifizieren. Im Alltagsgeschäft ist das freilich nicht nur eine einzelne, sondern womöglich eine Vielzahl von Änderungen, die auf der Seite vorgenommen wurden.

Doch halt: Woher weiß unser Entwickler, welche seiner zahlreichen CMS-Komponenten davon betroffen sind? Die Zuordnung ist aus dem statischen Mark-up nicht erkennbar. In der Praxis wird man es oft mit Hunderten von Komponenten und Dutzenden von Seiten-Templates zu tun haben. Weiß man es nicht auswendig, beginnt eine Suche nach „css"-Klassen und Mark-up-Schnipseln in den „.jsp"-Templates, um die entsprechende Stelle im CMS zu identifizieren und dort die Änderung einzupflegen. Besonders fehleranfällig ist dies, wenn sich gleiche Mark-up-Schnipsel in verschiedenen Templates wiederholen und einfach übersehen werden.

Der Ansatz mit Thymeleaf

Die Thymeleaf-Rendering-Engine verfolgt den Ansatz, dass sich die Templates auch ohne jede Rendering-Engine oder jedes Content-Management-System als korrektes HTML-Markup darstellen und im Browser öffnen lassen. Setzt man die beiden Komponenten nicht als „.jsp"-, sondern als Thymeleaf-Template um, sähe das zum Beispiel wie in *Listing 6* aus.

Der erste auffällige Unterschied ist, dass hier beide Komponenten in einem einzigen Template abgebildet sind. Thymeleaf ermöglicht dies über das Konzept eines „Fragments". Das bedeutet, die beiden Teilbäume mit dem Attribut „th:fragment" können

```
<html lang="en">
<head>
  <meta charset="utf-8"/>
  <title>Bootstrap, from Twitter</title>
  <link href="../../../webapp/docroot/bootstrap/css/bootstrap.css" rel="stylesheet"/>
</head>
<body>
<div class="container">
  <div class="row">
     <div class="span4">
        <h2>Heading</h2>
        <p>Donec id elit non mi porta gravida at eget metus. </p>
     </div>
  </div>
</div>
<hr />
<footer>
  <p>&copy; Company 2015</p>
</footer>
</body>
</html>
```

Listing 1: Statisches Mark-up für einen Teaser

```
<%@ page language="java"%>
<div class="span4">
  <h2>${content.heading}</h2>
  <p>${content.text}</p>
</div>
```

Listing 2: „teaser.jsp"-Template für den Teaser

```
/** Sample Teaser Controller. */
@Template(id = "thymeleaf_proto:components/teaserComponent", title = "Teaser Component")
@Controller
public final class TeaserComponent {
  /**
  * get the template fragment.
  */
  @RequestMapping("/teaserComponent“)
  public String handleRequest() {
     return "templates/jsp/teaser.jsp“;
  }
  /**
     * create the tab.
     */
  @TabFactory("Properties")
  public void createTab(final UiConfig cfg, final TabBuilder tab) {
        tab.fields(
           cfg.fields.text("heading").label("Heading"),
           cfg.fields.text("text").label("Text")
        );
     }
}
```

Listing 3: „TeaserComponent.java“-Pflege-Dialog für den Teaser

```
<html lang=“en“>
<head>
  <meta charset=“utf-8“/>
  <title>Bootstrap, from Twitter</title>
  <link href=“../../../webapp/docroot/bootstrap/css/bootstrap.css“
rel=“stylesheet“/>
</head>
<body>
<div class=“container“>
  <div class=“row“>
     <div class=“span4“>
        <h2>Heading</h2>
        <p>Donec id elit non mi porta gravida at eget metus.</p>
        <p><a class=“btn“ href=“#“>View details &raquo;</a></p>
     </div>
  </div>
</div>
<hr />
<footer>
  <p>&copy; Company 2015</p>
</footer>
</body>
</html>
```

Listing 5: Das angepasste Mark-up

```
<html xmlns:cms=“http://www.eiswind.de/mgnl“ lang=“en“ xmlns:th=“http://www.
thymeleaf.org“>
<head cms:init=““>
  <meta charset=“utf-8“/>
  <title>Bootstrap, from Twitter</title>
  <link th:href=“‘docroot/bootstrap/css/bootstrap.css‘“
     href=“../../../webapp/docroot/bootstrap/css/bootstrap.css“ rel=“stylesheet“/>
</head>
<body>
<div class=“container“>
  <div class=“row“>
     <div th:fragment=“teasercomponent“ class=“span4“>
        <h2 th:text=“${content[heading]}“>Heading</h2>
        <p th:text=“${content[text]}“>Donec id elit non mi porta gravida at eget. </p>
     </div>
  </div>
</div>
<footer th:fragment=“footercomponent“>
  <p th:text=“${content[footertext]}“>&copy; Company 2015</p>
</footer>
</body>
</html>
```

Listing 6: Komponten als Thymeleaf-Template

```
<%@ page language="java"%>
<footer>
  <p>${content.footerText}</p>
</footer>
```

Listing 4: „footer.jsp“-Template für den Footer

einzeln und losgelöst von der Gesamtseite gerendert werden. Abgesehen von den zusätzlichen Attributen aus dem „cms“- beziehungsweise „th“-Namespace ist das Template identisch zur statischen Vorlage.

Hinzu kommt, dass der Browser die Attribute aus dem „th“-Namespace einfach ignoriert, wenn diese Seite direkt geöffnet wird. Das heißt, der Web-Entwickler kann mit diesem Template arbeiten, als wäre es ein statisches Mark-up, und benötigt kein Content-Management-System, um damit zu arbeiten. Er kann dadurch den neuen Link-Button direkt in das Thymeleaf-Template einarbeiten und stylen, die CMS-Entwickler müssen nur noch die Pflege-Dialogfelder und Attribute hinzufügen.

Das Zuordnen von Mark-up zu JSP und das Wiederauffinden der richtigen Stelle entfallen vollständig. Mit etwas Geschick (ein Beispiel dazu finden Sie im Quellcode zum Artikel) lässt sich dasselbe Template sogar als Seiten-Template für Magnolia verwenden, in das die Komponenten später eingepflegt werden können.

Die Realität im Arbeitsalltag

Die Einführung einer neuen Template-Sprache führt in der Regel in einem unter Druck arbeitenden Entwicklerteam nicht nur zu Begeisterung, sondern eher zu zahlreichen Diskussionen und Widerständen. Das Management muss zudem vorübergehend auf eine durch die Lernkurve bedingte verminderte Produktivität eingestimmt werden. Eine noch überschaubare Anzahl von Einführungen des gezeigten Ansatzes in Großbritannien und Deutschland hat dies bestätigt. Ist diese Lernkurve überstanden, zeigt sich jedoch eine deutliche Vereinfachung der Zusammenarbeit zwischen Frontend- und Backend-Entwicklern, das gegenseitige Interesse für das jeweilige Fachgebiet steigt und schließlich beschleunigt sich der Gesamtprozesses, da sich Änderungen schneller umsetzen lassen.

Die technische Umsetzung

Magnolia ist wie die meisten Java-Content-Management-Systeme grundsätzlich darauf ausgelegt, alternative Template-Engines einzubinden. Out of the box kommt

Magnolia schon mit einer Unterstützung für Freemarker und JSP daher. Bei der Einbindung von Thymeleaf gab es lediglich eine technische Bruchstelle, da sich beim Rendering-Prozess von Magnolia alle Komponenten des Templates einen Output-Stream teilen. Thymeleaf dagegen arbeitet intern zuerst alles in einem XML-Baum ab und erst nach Abschluss der Verarbeitung wird dieser XML-Baum in einen Stream geschrieben.

Die vorgestellte Lösung bricht daher die Thymeleaf-Verarbeitung in einzelne Teilschritte auf, die nacheinander in den Magnolia-Stream schreiben. Um diesen Bruch aufzulösen, müsste man tiefer in die Magnolia-Rendering-Engine eingreifen, was hier aus Gründen der Einfachheit und Kompatibilität nicht geschehen ist.

Das Beispiel setzt zudem den Einsatz des Magnolia-Blossom-Moduls (und damit Spring-MVC) voraus. Eine Umsetzung ohne Blossom sollte aber nicht weiter schwierig sein, der Autor steht hier gern mit Rat und Tat zur Seite. Die Implementierung der Integration besteht aus lediglich sechs Klassen für die Komponenten und den Renderer. Den Quellcode sowie ein Beispielprojekt stehen bei github unter *„https://github.com/eiswind/magnolia-thymeleaf-renderer"*.

Einsatz-Szenarien

Der hier beschriebene Ansatz eignet sich vor allem dann, wenn Frontend- (CSS/JavaScript) und Backend- (Java, Magnolia) Kompetenzen auf verschiedene Teams oder Personen verteilt sind. In diesem Fall kann Thymeleaf die beschriebenen Vorteile für die Zusammenarbeit auch in anderen Projekten ausspielen, der Einsatz ist in keinerlei Weise auf das Fallbeispiel mit dem Magnolia-Content-Management-System beschränkt. Man findet sogar für Play/Scala eine Integration der Thymeleaf-Engine im Netz.

Berichte aus dem Anwenderkreis des gezeigten Fallbeispiels haben bestätigt, dass Überzeugungsarbeit im Vorfeld erforderlich ist, wenn im Team noch keine Erfahrung mit Thymeleaf vorhanden ist.

Thomas Kratz
thomas.kratz@eiswind.de

Thomas Kratz arbeitet als Software-Entwickler bei der Hamburger mecom GmbH. Er ist seit mehr als zwanzig Jahren in unterschiedlichen Rollen leidenschaftlich in der Software-Entwicklung tätig. Technologie, Kommunikation und Prozesse sind für ihn unabdingbar miteinander verknüpft.

http://ja.ijug.eu/15/3/10

Clojure – ein Reiseführer

Roger Gilliar, MCS

Java wäre ohne die Leistungsfähigkeit der Java Virtual Machine (JVM) nicht so erfolgreich geworden. Ohne die Popularität von Java hätte sich die JVM aber auch nicht auf das heutige hohe Niveau weiterentwickelt und wäre damit nicht zu der beliebten Zielplattform für alternative Programmiersprachen geworden, die sie heute ist. Clojure ist ein Lisp-Dialekt und eine der bekanntesten Alternativen zu Java auf der JVM.

Acht Seiten – mehr Platz wurde im „LISP 1.5 Programmer's Manual" im Jahr 1962 nicht benötigt, um die Sprache Lisp vollständig zu beschreiben. Damit sollte es doch ein Leichtes sein, auch Clojure auf wenigen Seiten vollständig darzustellen. Würde man sich nur auf die Syntax beschränken, wäre das kein Problem. Dafür reicht ein Absatz. Allerdings machen die Konzepte einer Programmiersprache nur einen verschwindend geringen Teil des Lernaufwands aus.

Aufwändiger ist es, die dazugehörigen Tools, Bibliotheken und APIs kennenzulernen und sachgerecht einzusetzen. Dafür wäre dann schon ein Buch im Umfang von Tolstois „Krieg und Frieden" (ca. 1.500 Seiten) nötig. So viel Platz steht hier leider nicht zur Verfügung, weshalb sich der Artikel auf die Darstellung ausgewählter Konzepte, Werkzeuge und Bibliotheken beschränkt. Die einzelnen Abschnitte sind dabei bewusst knapp gehalten. Sie sollen mehr als Wegweiser durch die Clojure-Welt dienen. Aufgrund der Vielzahl der hier aufgeführten Links sind diese unter *„https://support.mcs.de/development/clojure/resources"* online zu finden.

Was ist Clojure?

Clojure ist eine funktionale, dynamische Programmiersprache. Clojure-Code wird in Byte-Code kompiliert, der auf der JVM ausführbar ist. Außerdem ist Clojure ein Lisp-Dialekt. Soweit die Zusammenfassung der wichtigsten Eigenschaften. Weitere Details stehen unter *„http://clojure.org"*.

Was bedeutet das in der Praxis? Zuerst profitiert Clojure natürlich von den unzähligen Bibliotheken, die für Java verfügbar sind. Durch die Java-Interoperabilität kann man entweder die Bibliotheken direkt verwenden oder Bibliotheken nutzen, die einen Clojure-freundlichen Wrapper zur Verfügung stellen. So lässt sich zum Beispiel das JDBC-API direkt einsetzen oder man verwendet eine Clojure-Bibliothek wie „Yesql" *(siehe „https://github.com/krisajenkins/yesql")*.

Listing 1 zeigt ein erstes Beispiel für ein Clojure-Programm. Das sieht auf den ersten Blick kryptisch aus; die Syntax ist jedoch schnell erklärt: Clojure-Programme bestehen aus Listen, die in geeigneter Weise ausgewertet werden. Geeignet heißt, dass der erste Ausdruck in einer Liste als Funktionsaufruf interpretiert wird. Der Ausdruck „(timbre/info „shuting down...")" bedeutet also: „Rufe die Funktion „info" mit dem Argument „shutting down..." auf."

Bei dem Präfix „timbre/" handelt es sich übrigens um einen Alias. Was in Java „Packages" sind, sind in Clojure „Namensräume", für die man beim Import einen Alias vergeben kann. Das erspart einiges an Tipp-Arbeit, da für „timbre", eine Logging- und Profiling-Bibliothek, der komplette Namensraum „taoensso.timbre" lauten würde *(siehe „https://github.com/ptaoussanis/timbre")*.

Programme als Listen zu repräsentieren, folgt dem Lisp-Credo „code-is-data and data-is-code". Dadurch ergeben sich Möglichkeiten, die weit über das hinausgehen, was in anderen Programmiersprachen mit „Reflection" oder Meta-Programmierung möglich ist. Als erster Einstieg in dieses Thema empfiehlt sich der Screencast von Timothy Baldrige – „Clojure: Deep Walking Macro" *(siehe „https://www.youtube.com/watch?v=HXfDK1OYpco")*.

Noch ein Hinweis zu *Listing 1*: Der Ausdruck „defn" bindet einen Funktions-Ausdruck an ein Symbol. Nach dem Symbol (hier „destroy") folgt ein Vektor mit den erwarteten Funktions-Argumenten. Der Vektor in *Listing 1* ist leer. Die Funktion „destroy" besitzt also keine Funktions-Argumente. *Listing 2* zeigt, wie das Ganze mit einem Funktions-Argument aussehen könnte.

Immutability Rules

Listing 3 zeigt einige einfache Beispiele für die Verwendung einer HashMap und eines Vektors, den am häufigsten in Clojure ver-

```
(defn destroy []
  (timbre/info "server is shutting down...")
  (cronj/shutdown! session/cleanup-job)
  (timbre/info "shutdown complete!"))
```

Listing 1

```
(defn destroy [message]
  (timbre/info message)
  (cronj/shutdown! session/cleanup-job)
  (timbre/info "shutdown complete!"))
```

Listing 2

wendeten Daten-Strukturen. Wie man sieht, können in Clojure auch verschiedene Daten-Typen innerhalb einer Daten-Struktur zum Einsatz kommen. Bezeichner, die mit einem Doppelpunkt beginnen, werden übrigens „keyword" genannt und in Clojure vorzugsweise als Schlüssel für HashMaps verwendet, da für „keyword" eine sehr effiziente „equals"-Implementierung vorhanden ist.

Es gibt allerdings einen entscheidenden Unterschied zu Java. Alle Clojure-eigenen Daten-Strukturen sind unveränderbar. Aus diesem Grund liefert die „assoc"-Funktion, die einer HashMap ein neues Schlüssel/Wert-Paar zuordnet, auch eine neue HashMap zurück. Die ursprüngliche HashMap bleibt unverändert. Ebenso liefert die „conj"-Funktion, die einen neuen Wert an einen Vektor anhängt, einen neuen Vektor zurück. Auch hier wird der ursprüngliche Vektor nicht verändert.

Damit das alles nicht zu sehr auf die Performance geht, findet hier kein naives Kopieren statt. Es werden hingegen „immutable-persistent-data-structures" verwendet, die so weit wie möglich versuchen, Daten-Kopien zu vermeiden. Eine interessante Einführung in dieses Thema steht unter *„https://www.youtube.com/watch?v=I7IdS-PbEgI"* (Immutable Data and React).

Auch wenn das Konzept der unveränderbaren Daten-Strukturen auf den ersten Blick wie eine unnötige Einschränkung wirkt, ergeben sich dadurch nicht zu unterschätzende Vorteile. Der bekannteste dürfte sein, dass Daten, die nicht geändert werden können, automatisch „Thread Safe" sind. Der nicht weniger wichtige Aspekt ist, dass die Verwendung von unveränderbaren Daten-Strukturen ein wichtiger Schritt zu leichter verständlichen und einfacher wartbaren Programmen ist.

Man vergleiche hierzu *Listing 4* und *Listing 5*. In *Listing 4* wird ein Vektor in Clojure an eine Funktion übergeben, in *Listing 5* sieht man ein entsprechendes Beispiel in Java.

Wie sicher kann man sein, dass die Funktion den Vektor oder die Liste nicht ändert? In Clojure kann man sich sicher sein, in Java nicht. „Arrays.asList" liefert zwar eine Liste mit fester Länge zurück, das verbietet aber nicht das Überschreiben bestehender Werte. Folgender Methodenaufruf ist möglich: „einVektor.set(0,4);".

```
; ein Semikolon leitet einen Kommentar ein
(def hmap {:key1 1   :key2 2})
                     :key2 2})

(:key1 hmap); liefert den Wert  1
(:key2 hmap); liefert den Wert  2

(assoc hmap :key1 11); liefert hash-map {:key1 11
                                         :key2 2)
(def ein-vektor [1 2 3 4 5 6])
(first ein-vektor); liefert 1
(nth ein-vektor 3); liefert 3
(conj ein-vektor {:k 1}); liefert [1 2 3 4 5 6 {:k 1}]
```

Listing 3

```
(def ein-vektor [1 2 3 4 5])
(do-something ein-vektor)
```

Listing 4

```
List<Integer> einVektor = Arrays.asList(1,2,3,4,5);
doSomething(einVektor);
```

Listing 5

```
;definiere eine Identität vom Typ atom
(def ein-vektor (atom [1 2]))
;ersetze den Inhalt der Identität als atomare Operation
(swap! ein-vektor conj 3)
; ein-vektor enthält nun den Vektor [1 2 3]
```

Listing 6

Allerdings müssen auch Clojure-Programme einen Zustand verwalten, und der sollte veränderbar sein. Aus diesem Grund wird bei Clojure strikt zwischen Wert, Identität und Zustand unterschieden. Ein Wert ist immer unveränderbar, eine Identität kann zu verschiedenen Zeitpunkten jeweils unterschiedliche Werte beinhalten und ein Zustand ist der Wert einer Identität zu einem bestimmten Zeitpunkt.

Klingt noch zu abstrakt? Man stelle sich das am einfachsten so vor: Werte sind unveränderbar. Um Änderungen durchführen zu können, werden Werte in einen Container verpackt. Zustands-Änderung bedeutet dann, dass sich der Inhalt dieses Containers ändern kann. Der Wert und der Container (die Identität) verändern sich hingegen nie. Das Besondere an Clojure ist, dass für den Austausch des Inhalts verschiedene Semantiken verfügbar sind. Die einfachste Variante, der Austausch, ist eine atomare Operation *(siehe Listing 6)*.

Der Vorteil an diesem Konzept ist, dass Zustands-Änderungen durch entsprechende Funktionsaufrufe wie „swap!" deutlich im Source-Code zu erkennen sind. Außerdem sind Änderungen auf Identitäten „Thread Safe". Weitere Beispiele zu diesem Thema finden sich in einem Beitrag von der BOB-Konferenz, die im Januar 2015 in Berlin stattfand *(siehe „https://www.youtube.com/watch?v=IkvA-sz4b_M")*.

Die Qual der Wahl

Wollte man mit einer LISP-ähnlichen Programmiersprache entwickeln, führte lange Zeit kein Weg an Emacs vorbei. Mit „cider" *(siehe „https://github.com/clojure-emacs/cider")* existiert auch für Clojure ein Emacs-Package, das den gewohnten Komfort wie Auto-Completion oder Ausführen von Tests für den Clojure-Programmierer zur Verfügung stellt. Für Emacs-Neulinge lohnt sich ein Blick auf *„https://github.com/bbatsov/prelude"*. In dieser Distribution sind die wichtigsten Packages und Einstellungen für Emacs vorkonfiguriert und erleichtern somit den Einstieg in die Benutzung des Emacs-Editors.

Freunde vom „vi" finden unter *„https://github.com/tpope/vim-fireplace"* die passende Clojure-Unterstützung. Aber auch der komfortverwöhnte Java-Programmierer muss seine gewohnte Umgebung nicht verlassen. Eclipse-Anhänger finden mit „Counterclockwise" ein passendes Eclipse-Plug-in und unter *„http://doc.ccw-ide.org"* alles Wissenswerte zur Installation und Verwendung. Für IntelliJ-Benutzer empfiehlt sich unbedingt

die Installation von Cursive *(siehe „https://cursiveclojure.com/userguide/index.html“)*.

Wer es lieber eine Nummer kleiner möchte, findet mit LightTable *(siehe „http://lighttable.com“)* den passenden Kandidaten. Allerdings gibt es mit der aktuellen Version 0.7.2 zumindest unter OS-X zum Teil erhebliche Performance-Probleme, die mit dem Release 0.8 behoben sein sollten. Entsprechende Hinweise stehen unter *„https://groups.google.com/forum/#!topic/light-table-discussion/PfQ6kCKrj84“*.

Auch für denjenigen, der sich bereits für einen der anderen Editoren entschieden hat, lohnt sich die Beschäftigung mit „LightTable“. Es bietet nämlich die Möglichkeit, Code direkt im Editor auszuführen und sich die Ergebnisse „inline“ anzeigen zu lassen. Wer sich von dieser Funktion erst mal einen Eindruck verschaffen möchte, ohne die Mühen der Installation auf sich zu nehmen, findet auf YouTube entsprechende Tutorial-Videos. Empfehlenswert sind hier insbesondere die Videos von „Misophistful“. So bietet das Video unter *„https://www.youtube.com/watch?v=1f13TTu_X9k“* eine schöne Übersicht über die grundlegenden Konzepte von LightTable sowie eine Demo des Ausführens von Code innerhalb des Editors. Gerade aufgrund dieser Funktion eignet sich LightTable hervorragend dazu, Dinge einfach mal auszuprobieren. Für größere Projekte sind die anderen Editoren und IDEs besser geeignet.

Leiningen – das Clojure Maven

„Maven“ ist in der Java-Welt Standard, um Projekten eine einheitliche Struktur zu geben und alle Abhängigkeiten eines Projekts an zentraler Stelle festzulegen. In Clojure gibt es dafür Leiningen. Im Gegensatz zu der von Maven bekannten XML-Struktur verwendet Leiningen eine ganz normale Clojure-Datenstruktur (HashMap); statt „pom.xml“ heißt die Konfigurationsdatei „project.clj“. *Listing 7* zeigt dazu ein Beispiel. Die Installation von Leiningen ist denkbar einfach und sollte der erste Schritt sein, bevor man Ausflüge in die Clojure-Welt unternimmt. Download und Hinweise zur Installation stehen unter *„http://leiningen.org“*.

Boot – das Clojure Gradle

Für alle, die eher den Ansatz von Gradle und nicht so sehr den deklarativen Ansatz von Maven bevorzugen, gibt es mit Boot eine passende Alternative. Boot ist wie Leiningen ein Build-Tool für Clojure, wertet aber nicht eine Konfigurationsdatei aus, sondern führt Clojure-Programme als Build-Steps aus, die miteinander kombinierbar sind. Um einen Web-Server zu starten, ein Verzeichnis auf Änderungen zu überwachen, dann bei jeder Änderung den Browser zu aktualisieren oder ClojureScript nach JavaScript zu übersetzen und bei Fehlern ein akustisches Signal auszugeben, würde in Boot folgender Aufruf genügen: „boot serve -d target/ watch speak reload cljs-repl cljs -sO none“.

Boot ist zwar schon bei Version 2.x, aber doch insgesamt noch recht jung. So unterstützt Cursive die aktuelle Boot-Version noch nicht. Daher empfiehlt sich immer noch Leiningen für die meisten Aufgaben der Clojure-Programmierung. Bei komplexen Web-Projekten, die neben Clojure auch ClojureScript verwenden, kann der Einsatz von Boot jedoch Vorteile bieten; es empfiehlt sich hier, zweigleisig zu fahren. Für die Definition der allgemeinen Projektstruktur wird Leiningen verwendet. Damit ist man kompatibel zu allen verfügbaren Editoren. Für den Build-Prozess kommt Boot zum Einsatz. In der Boot-Konfigurationsdatei „build.boot“ wird dann allerdings auf Daten aus der „project.clj“ zugegriffen *(siehe Listing 8)*. Weitere Details zu Installation und Verwendung stehen unter *„https://github.com/boot-clj/boot“*.

Clojure und Java

Clojure wird auf der JVM ausgeführt und kann daher auch auf die Java-Standard-Bibliotheken zugreifen. So wird mit „(.toUpper „hello world“)“ die Methode „toUpper“ auf dem String-Objekt ausgeführt. Wer eine Java-ähnliche Syntax bevorzugt, kann das Threading-Makro verwenden: „(println (-> „tesla „ .toUpperCase .trim))“.

Clojure kann natürlich auch beliebige Java-Objekte erzeugen. Statt „new“ schreibt man dafür einen Punkt direkt hinter den Klassennamen: „(println (-> (BigDecimal. „1234“) .intValue))“. Auch die Implementierung eines Interface ist mit dem „proxy“-Makro kein Problem: „(.run (proxy [Runnable] [] (run [] (println "Hi!"))))“.

Diese Beispiele stellen nur einen kleinen Auszug aus den Möglichkeiten dar, die Clojure bietet, um auf Objekte, Klassen und Methoden von Java zurückgreifen zu können. Weitere Details zur Java-Interoperabilität stehen unter *„http://clojure.org/java_interop“* oder als Screencast unter *„https://www.youtube.com/watch?v=TYmC1wzBNfw“*.

Wo bleibt die Struktur?

Gerade am Anfang kann es für den objektorientiert-verwöhnten Programmierer sehr schwierig sein, Clojure-Programme mit ei-

```
(defproject hw-om "0.1.0-SNAPSHOT"
  :description "FIXME: write this!"
  :url "http://example.com/FIXME"

  :dependencies [[org.clojure/clojure "1.6.0"]
     [org.clojure/clojurescript "0.0-2814"]
     [org.clojure/core.async "0.1.346.0-17112a-alpha"]
     [org.omcljs/om "0.8.8"]])
```

Listing 7: „project.clj“

```
(set-env! :dependencies ,[[leiningen-core "2.5.0"]])
(use ,leiningen.core.project)

(eval (read-string (slurp "project.clj")))

(set-env!
 :source-paths   (into #{} (:source-paths project))
 :resource-paths #{"html"}
 :dependencies   (into [] (:dependencies project)))
```

Listing 8: „build.boot“

```
(defn my-system [config]
  (component/system-map
  :datasource (map->DataSourceComp config)
  :pool (component/using
     (map->PooledConnectionComp {})
     [:datasource])))
```

Listing 9

```
(def as-and-bs
  (insta/parser
     "S = AB*
     AB = A B
     A = 'a'+
     B = 'b'+"))
```

Listing 10: EBNF-Grammatik

ner wartbaren Struktur zu versehen. Hier hilft die „component"-Bibliothek von Stuart Sierra *(siehe „https://github.com/stuartsierra/component")*. *Listing 9* zeigt ein Beispiel dafür, wie diese Bibliothek verwendet werden kann. Hier wird deklariert, dass die Komponente „DataSourceComp" initialisiert sein muss, bevor die Komponente „PooledConnectionComp" verwendet werden kann.

Für die ersten kleineren Demo-Programme kann man getrost auf die „component"-Bibliothek verzichten. Für richtige Programme dürfte es hingegen schwierig werden, gute Gründe zu finden, sie nicht zu verwenden. Als Einführung in diese Bibliothek und die verwendeten Konzepte empfiehlt sich der Vortrag von Stuart Sierra von der Clojure/West-Konferenz *(siehe „https://www.youtube.com/watch?v=13cmHf_kt-Q")*.

Grammatiken mit Spaß bauen

Grammatiken bauen und Spaß dabei haben? Mit Instaparse kann einem genau das passieren. Es benutzt die erweiterte Backus-Naur-Form (EBNF) zur Beschreibung der Grammatiken. Dabei ist kein Extra-Schritt zur Generierung des Source-Codes nötig; alles kann direkt in Clojure erledigt werden. *Listing 10* zeigt das erste Beispiel von der Instaparse-Homepage.

Das genügt schon, um einen vollständigen Parser zu definieren, der so benutzt werden kann: „(as-and-bs „aaaaabbbaaaabb")". Als Ergebnis liefert der Parser dann folgende Struktur *(siehe Listing 11)*.

Zur Erinnerung: Eckige Klammern kennzeichnen Vektoren, Schlüsselwörter („keywords") beginnen mit einem Doppelpunkt. Weitere Beispiele und die Dokumentation stehen unter *„https://github.com/Engelberg/instaparse"* oder im Screencast unter *„https://www.youtube.com/watch?v=b2AUW6psVcE"*.

Compojure-API – REST-easy

Compojure ist eine Clojure-Bibliothek, um das Routing innerhalb einer Web-Applikation beschreiben zu können. *Listing 12* zeigt ein Beispiel für die Verwendung, in dem die Route „/hello-world" definiert ist, die beim Aufruf über einen „get"-Request den Inhalt „<h1>Hello World</h1>" zurückliefert.

Compojure ist eine der Standard-Routing-Bibliotheken. Deswegen verwenden viele Bibliotheken Compojure, stellen aber zusätzliche Funktionen zur Verfügung. Eine dieser Bibliotheken ist „Compojure-API". Sie stellt Funktionen zur Daten-Validierung bereit und bietet eine Integration von Swagger und Swagger-UI. Dadurch ist sie hervorragend für die Erstellung von REST-APIs geeignet. Details zu Compojure stehen unter *„https://github.com/weavejester/compojure"*; Beispiele für Compojure-API unter *„https://github.com/metosin/compojure-api"*.

Im Kontext von Clojure ist häufig von Bibliotheken die Rede, und das aus gutem Grund. Statt auf Frameworks wie EE7 oder Spring zu setzen, werden bei der Clojure-Entwicklung sich ergänzende oder aufeinander aufbauende Bibliotheken bevorzugt. Für die Web-Entwicklung kommen daher oft Bibliotheken zum Einsatz, die auf der „Ring"-Bibliothek aufbauen, bei der es sich um eine Bibliothek rund um HTTP handelt. So verwendet auch Compojure die „Ring"-Bibliothek und bietet zusätzlich eine einfache DSL zur Definition von Routen an. Intern verwendet „Ring" übrigens das Java-Servlet-API.

Luminus – Web-Entwicklung leichtgemacht

Eine der Hauptschwierigkeiten beim Erlernen einer neuen Programmiersprache ist zu wissen, welche Aufgaben mit welcher Bibliothek am besten zu lösen sind. Glücklicherweise gibt es für Leiningen viele Templates, die für bestimmte Aufgabenstellungen Projekte mit den passenden Abhängigkeiten für die benötigten Bibliotheken erstellen.

Luminus ist ein solches Template, das die wichtigsten Abhängigkeiten und Plug-ins für ein Webprojekt beinhaltet *(siehe „http://www.luminusweb.net")*. Dank dieses Templates kann mit „lein new luminus hello-world" das Projektgerüst einer Web-Applikation erstellt werden.

Wer sich auch abseits des Computers mit der Web-Entwicklung unter Clojure beschäftigen möchte, findet mit „Web Development with Clojure: Build Bulletproof Web Apps with Less Code" und „Clojure Web Development Essentials" den passenden Lesestoff.

```
[:S
 [:AB [:A "a" "a" "a" "a" "a"] [:B „b" "b" "b"]]
 [:AB [:A "a" "a" "a" "a"] [:B "b" "b"]]]
```

Listing 11

```
(ns hello-world.core
 (:require [compojure.core :refer :all]))

(defroutes app
 (GET "/hello-world" [] "<h1>Hello World</h1>"))
```

Listing 12

```
(ns test-om.core
  (:require [om.core :as om]
            [om.dom :as dom]))

(enable-console-print!)

(def app-state (atom {:text "Hello world!"}))

(om/root
  (fn [app owner]
     (reify om/IRender
        (render [_]
           (dom/h1 nil (:text app)))))
  app-state
  {:target (. js/document (getElementById "app"))})
```

Listing 13: ClojureScript „om"

ClojureScript – Clojure im Browser

ClojureScript ist eine Variante von Clojure, die JavaScript anstelle der JVM als Ziel-Plattform verwendet. Vom Konzept her ähnelt ClojureScript damit Sprachen wie „CoffeeScript" oder „Scala.js". Das Besondere an ClojureScript ist, dass sowohl die Kern-Bibliotheken von Clojure verfügbar sind als auch von vielen Clojure-Bibliotheken angepasste Versionen für ClojureScript existieren. Dadurch können für die Backend- und Frontend-Entwicklung nicht nur die gleiche Programmiersprache, sondern häufig auch die gleichen Bibliotheken verwendet werden.

Damit der aus den ClojureScript-Programmen erzeugte JavaScript-Code nicht zu umfangreich wird, setzt ClojureScript auf die Closure Tool-Suite von Google, die einen Optimierer enthält, der nicht benutzten Source Code entfernt. ClojureScript ist übrigens nicht nur im Browser, sondern auch unter „node.js" einsetzbar. Mit „om" *(siehe „https://github.com/omcljs/om")* und „reagent" *(siehe „https://github.com/reagent-project/reagent")* gibt es Bibliotheken, die einen einfachen Zugriff auf die immer populärer werdende „React.js"-Bibliothek *(siehe „http://facebook.github.io/react")* bieten. *Listing 13* zeigt ein Beispiel, das ClojureScript und „om" verwendet. Details zu ClojureScript stehen unter *„https://github.com/clojure/clojurescript"*.

Wer bereits Leiningen installiert hat, kann ganz einfach mit ClojureScript und „om" loslegen:

- „lein new mies-om hello-world"
- „cd hello-world"
- „lein cljsbuild auto" (übersetzt ClojureScript nach JavaScript)
- Dann die Datei „index.html" aus dem Verzeichnis „hello-world" im Browser öffnen

Wer sich für Single-Page-Applikationen mit ClojureScript interessiert, sollte sich unbedingt „re-frame" genauer anschauen *(siehe „https://github.com/Day8/re-frame")*. Diese Bibliothek basiert auf „reagent", bietet aber Erweiterungen, um auch bei größeren Applikationen eine saubere Software-Architektur beizubehalten.

Fazit

Der Streifzug durch die Welt von Clojure ist am Ende angekommen. Natürlich gibt es noch viel mehr zu entdecken. Für den Anfang sollte man allerdings mit den wichtigsten Stichworten versorgt worden sein:

- Clojure ist ein Lisp-Dialekt
- Clojure läuft auf der JVM
- Clojure bevorzugt unveränderbare Daten-Strukturen („immutable data")
- Statt Maven und Gradle gibt es Leiningen und Boot
- In Clojure werden Bibliotheken statt Frameworks bevorzugt
- Für Eclipse gibt es das „Counterclockwise"-Plug-in
- Für IntelliJ gibt es „Cursive"
- Parsen geht ganz einfach mit „Instaparse"
- Ring ist die Basis-Bibliothek für Web-Anwendungen
- Luminus ist ein Leiningen-Template für Web-Anwendungen
- ClojureScript ist Clojure für den Browser
- Für Single-Page-Applikationen gibt es „re-frame", das auf „reagent" basiert
- ClojureScript verwendet die Closure-Tools von Google

Wozu das alles? Clojure ist anders, aber auf eine sehr angenehme Art und Weise. Selbst wenn man keine Gelegenheit hat, Clojure in aktuellen Projekten einzusetzen, lohnt sich eine eingehende Beschäftigung mit den Konzepten. So profitieren auch die Java-Programme von unveränderbaren Daten-Strukturen und einer geordneten Verwaltung des Zustands.

Roger Gilliar
roger.gilliar@mcs.de

Roger Gilliar ist Ausbilder und Senior-Software-Engineer bei MCS in Hamburg. Sein Schwerpunkt liegt schon seit mehr als zehn Jahren beim Einsatz von Java- und JavaScript-Technologien.

http://ja.ijug.eu/15/3/11

JavaFX-GUI mit Clojure und „core.async“

Falko Riemenschneider

Die Programmierung von JavaFX mithilfe von Clojure erfordert ein Umdenken, will man die Stärken dieser funktionalen Programmiersprache ausnutzen. Der Artikel stellt nach einer kurzen Charakterisierung von Clojure zwei zentrale GUI-Architektur-Ideen vor, um Ziele wie testbare Präsentationslogik und eleganten Umgang mit inhärenter Asynchronität in GUIs zu erreichen.

Clojure steht auf drei fundamentalen Säulen, deren Kenntnis den Zugang zur Sprache deutlich erleichtert.

Clojure ist ein Lisp

Die Syntax eines Lisp ist ungewohnt, aber einfach. Ihre Basis ist die S-Expression „(something-callable arg1 arg2 arg3)“. Zu sehen sind vier Symbole in einem Klammerpaar. Das erste Symbol ist immer etwas Aufrufbares, also entweder eine Funktion oder eine der wenigen „special forms“, die quasi fest in die Sprache eingebaut sind, oder ein Makro. Man stelle sich ein Makro als eine Art eingebetteten Code-Generator vor, der zur Compile-Zeit ausgeführt wird. Die übrigen drei Symbole sind die Argumente. Statt eines Symbols kann jedes der vier Teile wiederum eine S-Expression oder ein Daten-Literal sein. *Listing 1* zeigt, wie die Syntax für Daten-Literale von S-Expressions abweicht. Der Clou ist, dass die Syntax für Daten-Literale auch verwendet wird, um beispielsweise eine neue Funktion zu definieren *(siehe Listing 2)*.

Die gezeigte S-Expression ist eine Liste, deren erstes Symbol „defn“ ein Makro bezeichnet, dann folgen der Name der Funktion, ein Kommentar-Text und zum Schluss die Symbole der Funktionsparameter „a“ und „b“ in eckigen Klammern, also in einem Vektor. Der Ausdruck „(+ a b)“ ruft die Funktion „+“ mit zwei Argumenten auf.

Man verwendet also die gleichen Syntax-Bausteine für Code, die man sonst auch für Daten benutzt. Diese Eigenschaft reduziert die Programmierung von Makros (also Software-Generatoren) auf die gewohnte Arbeit mit Daten-Strukturen. Tatsächlich basiert der Sprachumfang von Clojure mit Ausnahme der vierzehn „special forms“ nur auf Makros und Funktionen, was im Umkehrschluss bedeutet, dass man Clojure ohne fremde Hilfe selbst erweitern kann, sollte man einmal Sprachfeatures vermissen.

Clojure-Code ist letztlich also nur eine Menge von S-Expressions, die Symbole, Daten-Literale und wiederum S-Expressions enthalten. Man braucht auch keine Sorgen wegen der ungewohnten Klammerung haben. In der Lisp-Welt sind Werkzeuge zum strukturellen Editieren schon lange weit verbreitet. Schon nach zwanzig Minuten Übung balanciert man die Klammern mit allen gängigen IDEs oder Editoren ohne Abzählen und nimmt die Klammern kaum noch wahr.

Clojure ist eine funktionale Programmiersprache

Heutzutage warten beinahe alle neueren Sprachen mit funktionalen Features auf, weshalb es hier noch etwas präziser wird. Zunächst ist der Kerngedanke der funktionalen Programmierung (FP), möglichst viel Code eines Systems in Form von seiteneffektfreien, kontextunabhängigen Funktionen zu schreiben.

Solche Funktionen bieten in hohem Maße Komponierbarkeit, Wiederverwendbarkeit und Testbarkeit. Aber natürlich können nutzbringende Softwaresysteme nicht auf Seiteneffekte (wie Ein- und Ausgabe) verzichten. Es gibt also durchaus Stellen innerhalb eines auf FP basierenden Systems, die die Arbeit erledigen müssen. Der Trick ist, diese so zu kapseln, dass man wenig Mühe hat, den restlichen Code komponierbar und testbar zu gestalten.

Ein wichtiges Instrument, um das Schreiben dieser komponierbaren Funktionen zu erleichtern, sind einheitliche Datenstrukturen mit durchgängiger Wert-Semantik: Die Zahl „42“ ist beispielsweise ein Wert. Diesen können wir nicht ändern, genauso wenig wie ein Datum oder einen String. Im Gegensatz dazu stehen Variable, deren Inhalte wir ändern können. Instanzen von Daten-Strukturen in Clojure (also Maps, Vektoren, Sets) sind ebenfalls Werte und damit unveränderlich. Das hat nicht nur Vorteile bezüglich Multi-Threading. Wenn wir ein Symbol „a“ als Vektor „[1 2 3]“ definieren und im Anschluss eine Funktion „foo“ damit aufrufen,

```
"a string"            ;; ein String
[1 :a "a text"]       ;; ein Vektor bestehend aus einer Zahl
                      ;; einem Keyword und wieder einem String
{1 "Foo" 3 "Baz"}     ;; eine assoziative Map mit 2 Zuordnungen
#{4711 3.141 1/5 \a}  ;; eine Menge mit 4 Elementen
,(\a "list of" 3)     ;; eine Liste mit 3 Elementen
```

Listing 1

```
(defn addiere-zahlen
  "Liefert die Summe zweier Zahlen."
  [a b]
  (+ a b))
```

Listing 2

also „(def a [1 2 3])" und „(foo a)", dann ist nach Beendigung von „foo" garantiert, dass „a" und der Vektor „[1 2 3]" unverändert sind. Um dies zu wissen, brauchen wir nicht in „foo" hineinzuschauen, was mentalen Aufwand verringert.

Wer nun befürchtet, dass der Umgang mit unveränderlichen Datenstrukturen sehr teuer ist, kann beruhigt werden. Alle Daten-Strukturen verwenden „structural sharing", womit das Erzeugen von Kopien denkbar günstig wird. Diese sogenannten „persistenten Datenstrukturen" sind seit Ende der 1990er Jahre bekannt *[1]*.

Auch Clojure-Funktionen sind Werte: Wir können sie ad hoc erzeugen oder als Argumente in Funktions-Aufrufe hineinreichen. Es gibt zudem praktische Werkzeuge, um aus bestehenden Funktionen neue abzuleiten, wie „comp" („hintereinanderschalten") oder „partial" („neue Funktion mit fixierten Argumenten ableiten").

Der Hauptteil eines Systems in Clojure sind also Funktionen, die neue Werte aus bestehenden Werten berechnen. Zusätzlich bietet Clojure eine Art von Thread-sicheren „Variablen" („vars", „atoms", „agents" und „refs") an, die an wenigen ausgewählten Stellen genutzt werden, um einen Programmzustand in Form zugeordneter Werte zu halten. Clojure geht also sehr bewusst mit dem Zustand und dessen Änderung um, was Programme nachvollziehbarer und Thread-sicher macht. FP in Clojure basiert also auf mächtigen Datenstrukturen mit Wert-Semantik und Funktionen, die ebenfalls wie Werte behandelt werden.

Einfachheit als zentrales Ziel

Rich Hickey, der Erfinder von Clojure, unterscheidet zwischen „simple" im Sinne von „es macht genau eine Sache" und „easy" im Sinne von „es ist vertraut". Damit ist die „Einfachheit" eines Konzepts, eines Sprach-Features oder einer Bibliothek objektiv bestimmbar, während Vertrautheit subjektiv ist. Viele Entscheidungen im Clojure-Sprach-Design gehen auf das Bestreben zurück, die Dinge einfach zu halten. Die Trennung von Werten und Variablen ist ein Beispiel dafür, ebenso die Tatsache, dass das Dispatching von Funktionsaufrufen nicht an Typen gebunden sein muss, dass Daten und Funktionen unabhängig sind und es keine Objekte gibt. Die Motivation ist, dass einfache Bausteine leichter zu verstehen und miteinander zu kombinieren sind als komplexe Bausteine. Das führt mitunter zu Ansätzen, die Java-Entwicklern noch fremd sind und daher Umdenken und Übung erfordern.

Funktionale Programmierung von GUIs

Eine GUI, die beispielsweise auf JavaFX basiert, widerspricht auf den ersten Blick der Idee der FP, wie Clojure sie vertritt. Letzten Endes ist eine Benutzungsschnittstelle veränderlicher Zustand pur, doch Clojure bevorzugt seiteneffektfreie Funktionen, die Daten transformieren, also neue Werte aus bestehenden erzeugen.

In der Tat: Wenn man das klassische GUI-Architektur-Muster „Model View Controller" (MVC) anwendet, wird man Clojure wie eine imperative anstatt einer funktionalen Sprache nutzen, womit man am Ende weniger gewinnt als erhofft. Die Ursache liegt darin, dass viele MVC-Beschreibungen suggerieren, dass beispielsweise der Controller das Modell ändert oder dass Modell-Änderungen vermittels Observer Änderungen in der View erzeugen.

Abbildung 1 zeigt die Illustration von MVC. Entlang der Pfeile schicken sich Instanzen Nachrichten, um Aktualisierungen durchzuführen. „A ändert B" ist in imperativen Sprachen zu finden, die im Jahr 1980 zum Einsatz kam, um die ersten MVC-Implementierungen zu erstellen *[2]*. Ein Muster, das davon implizit ausgeht, wird sich – kaum überraschend – nicht gut in die Welt der funktionalen Programmierung eingliedern.

Wenn man sich jedoch von MVC löst und nochmal mit freiem Kopf über GUI nachdenkt, kann man im ersten Schritt zu folgender Aussage gelangen: Eine GUI ist eine Repräsentation des System-Zustands. Um eine Repräsentation eines System-Zustands „state" zu erzeugen, verwendet man eine Funktion, die aus den Daten eine „view", also etwas für den Benutzer Sichtbares produziert.

Der zweite Schritt ist die Feststellung, dass der Benutzer mit dieser „view" ja mehr tun kann, als sie nur zu betrachten: Er kann durch sie den Systemzustand beeinflussen, indem er ein „event", also ein Ereignis auslöst, etwa einen Buchstaben in ein Textfeld eingibt oder eine Schaltfläche antippt. Es ist daher eine zweite Funktion notwendig, die „state", also die Daten des aktuellen Systemzustands, und ein „event" akzeptiert und daraus eine nachfolgende Version „state'" des System-Zustands errechnet. Ein GUI-Programm besteht also aus zwei idealisierten Funktionen mit Pseudo-Signaturen *(siehe Listing 3)*.

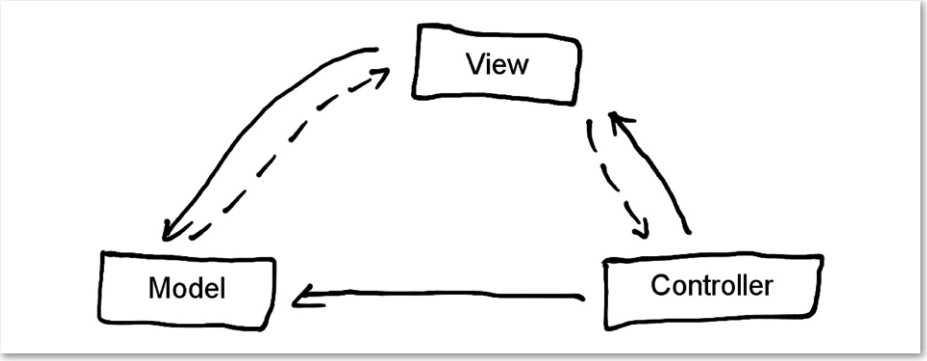


Abbildung 1: Model-View-Controller

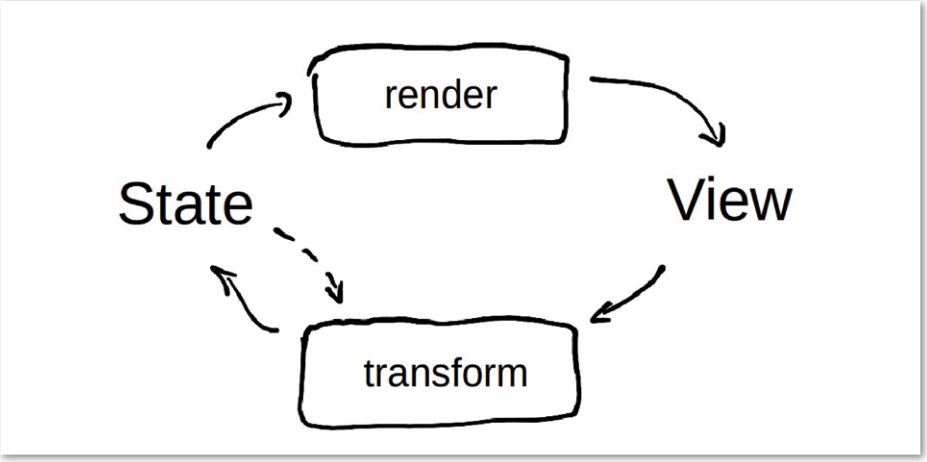


Abbildung 2: Unidirektionaler Datenfluss zwischen Zustand und View

Abbildung 2 veranschaulicht den resultierenden unidirektionalen Datenfluss durch die beiden Funktionen hindurch. Der entscheidende Unterschied zu MVC ist hier, dass das, was der Controller in MVC ist, nun eine Funktion „transform" ist, die Daten anhand von Ereignissen transformiert und damit keinen Zugriff auf die GUI hat. Die Aufgabe, aus dem System-Zustand etwas für den Benutzer Sichtbares zu produzieren, ist vollständig auf die Funktion „render" übertragen worden. Dieses in JavaFX seiteneffektbehaftete Rendering muss also eine Stage, falls sie noch nicht existiert, neu erzeugen und diese mit den Daten des Systemzustands aktualisieren. Ein Rendering, das uns diese Arbeit komplett abnimmt, erlaubt es uns, die eigentliche Ereignisverarbeitung so zu programmieren, dass sie sich ohne GUI-Roboter automatisiert testen lässt.

Bisher wurde der Systemzustand „state", der die Eingabe beider Funktionen ist, noch nicht näher betrachtet. Eine Funktion erzeugt die Daten, die von einer Stage repräsentiert werden *(siehe Listing 4)*.

Diese Funktion produziert eine Map, die eine Beschreibung der JavaFX-Controls, die darzustellenden Daten, ein Mapping und Validierungsregeln enthält. Dies ist der initiale Zustand, der durch die von „render" zu erzeugende Stage später angezeigt und durch „transform" mit jedem Ereignis in einer neuen Version fortgeschrieben wird.

Diese glasklare Trennung des Darstellungsproblems von der Ereignisverarbeitung kann man auch in „HTTP Request Response"-Web-Anwendungen beobachten. Sie ist weit in der Spiele-Programmierung verbreitet und findet auch in der JavaScript-Welt immer mehr Anhänger, nachdem Facebook im Jahr 2013 „React.js" veröffentlicht hat *[3]*.

```
render: state → view
transform: state, event → state'
```

Listing 3

Asynchronität in GUIs

JavaFX wie auch viele andere Rich-Client-Plattformen bis hin zum Web-Browser basieren auf einer zentralen, in nur einem Thread ablaufenden Ereignisverarbeitungsschleife. In JavaFX werden Ereignisse im sogenannten „Application Thread" abgearbeitet, damit Probleme von Nebenläufigkeit bei der Aktualisierung des „Scene Graph" ausgeschlossen sind. Wird diese Abarbeitung durch eine lang laufende Berechnung oder einen Fremdsystem-Aufruf blockiert, hat der Benutzer das Gefühl, die Anwendung sei „eingefroren". Das führt letztlich dazu, dass GUI-Anwendungen fast immer Multi-Threading enthalten und Entwickler entsprechend versiert sein müssen, um keine schwer reproduzierbaren Fehler einzubauen.

JavaFX bietet hier mit den Klassen „Task" und „Service" aus „javafx.concurrent" Unterstützung an. Um diese zu nutzen, wird der lang laufende Code beispielsweise in eine „Task"-Instanz eingepackt. Um aus diesem, in einem anderen Thread ausgeführten Code auf den „Scene Graph" zuzugreifen, muss dieser wiederum in ein „Runnable" verpackt sein und mittels „Platform.runLater()" an den Application Thread übergeben werden. Die lang laufende Ereignisverarbeitung wird also in mehrere Callbacks zerlegt, damit diese nebeneinander, also asynchron in Worker und Application Thread ausgeführt werden können.

Auch Modalität zwischen Stages würde ohne technischen Sonderweg in JavaFX sofort zu Asynchronität führen, denn Modalität eines Dialogs bedeutet, dass etwa ein Befehl wie „Datei öffnen" so lange nicht fortgesetzt werden kann, bis der Benutzer die Datei ausgewählt hat. Die Ereignisverarbeitung muss also warten, bis der modale Dialog geschlossen ist. Aber Warten im Application Thread ist aus den genannten Gründen kein gangbarer Weg. JavaFX löst dieses Problem durch eine zweite, temporäre Ereignisverarbeitungsschleife in „Stage.showAndWait()".

Aus diesen Ausführungen ergibt sich, dass eine besondere Behandlung für die Verarbeitung lang laufenden Codes in GUIs unvermeidbar ist. Die Mittel, die in JavaFX zur Verfügung stehen, zwingen einen, den Code in Callbacks zu zerlegen, bei denen nicht klar ist, ob und wie man noch automatisiertes Testen der Ereignisverarbeitung anwenden kann.

```
(defn item-editor-spec
  [data]
  (-> (v/make-view "item-editor"
     (window "Item Editor"
        :modality :window
        :content
        (panel "Content" :lygeneral "wrap 2, fill"
                  :lycolumns "[|100,grow]"
           :components
           [(label "Text")
              (textfield "text" :lyhint "growx")
              (panel "Actions" :lygeneral "ins 0"
                        :lyhint "span, right"
                 :components [(button "OK")
                        (button "Cancel")])])))
(assoc   :mapping (v/make-mapping :text ["text" :text])
        :validation-rule-set (e/rule-set :text (c/min-length 1))
        :data data)))
```

Listing 4

```
(go-loop [xs []]
  (let [x (<! ch)]
     (println "Got" x ", xs so far:" xs)
     (recur (conj xs x))))
```

Listing 5

CSP und „core.async"

Ein anderer interessanter Weg, um der aus dem Single-Thread-Modell resultierenden Asynchronität zu begegnen, besteht darin, sie als Grundlage der GUI zu akzeptieren. Dies fällt mit entsprechenden Mitteln, die Clojure bietet, leicht. Die Bibliothek „core.async" *[4]* erlaubt die Anwendung eines Programmiermodells namens „Communicating Sequential Processes" (CSP), das im Jahr 1978 erstmals veröffentlicht wurde *[5]*. Die Kernidee besteht darin, ein System als eine Sammlung von leichtgewichtigen, nebenläufigen Prozessen zu verstehen, die über Kanäle miteinander interagieren.

Um in Clojure einen Kommunikationskanal zu definieren, reicht ein Ausdruck wie „(def ch (chan))". Ein Kanal ist standardmäßig immer blockierend, er ist also auch ein Synchronisationspunkt zwischen Prozessen: Lesender wie schreibender Prozess warten hier aufeinander. Um einen Wert innerhalb

eines Prozesses in einen Kanal zu schreiben, reicht ein Ausdruck wie „(>! ch 42)".

Neben Funktionen, mit denen man aus einem Kanal etwas lesen oder schreiben kann, verwendet man das „go"- oder „go-loop"-Makro, um einen nebenläufigen Prozess zu programmieren. Das folgende Beispiel in *Listing 5* zeigt einen denkbar einfachen Prozess, der einen Wert aus „ch" liest, diesen auf der Konsole in einem Text ausgibt und im „recur" ein lokales Symbol „xs" an einen neuen Wert bindet, bevor er wieder auf einen Wert aus „ch" wartet.

Wie wir sehen, braucht dieses Programmiermodell keine sonderliche Zeremonie. Das „go"-Makro zerlegt den enthaltenen Code und erzeugt zur Übersetzungszeit eine Zustandsmaschine, sodass sich alle Prozesse einen gemeinsamen Threadpool teilen. Damit ist es möglich, Zehntausende aktive Prozesse in einer JVM zu benutzen, worauf wir bzgl. GUI glücklicherweise nicht angewiesen sind.

Prozess und Kanal nutzen

Es sei nochmal kurz daran erinnert, dass man GUI-Programme mit zwei idealisierten Funktionen formulieren will:

- *Render*
  state view
- *Transform*
  state, event state'

Die Idee ist, einen einzigen Prozess zu haben, der das Rendering im Application Thread übernimmt, also quasi die Funktion „render" ausführt, und zusätzlich für jede Stage einen weiteren Prozess, der den „state" einer Stage hält und dessen Transformation anhand von Ereignissen übernimmt. *Abbildung 3* vermittelt einen Überblick über die Threads, Prozesse und Kanäle.

Der JavaFX-Rendering-Prozess hat einen Eingabekanal, durch den alle Stage-Prozesse den jeweils letzten „state" an den JavaFX-Rendering-Prozess übergeben. Die Funktion „render" verwendet das JavaFX-API nun, um eine Stage, falls noch keine existiert, anhand des „state" zu konstruieren und die Daten in die Controls zu übertragen. Sie ist damit vollständig generalisiert.

Jeder Stage-Prozess hat einen Eingabekanal, der von allen Event-Handlern, die an Controls der Stage hängen, verwendet wird, um ein Ereignis zur Verarbeitung weiterzuleiten. Die Verarbeitung verwendet den im Prozess gekapselten „state" und ein „event" wie eine geänderte Eingabe in einem Textfeld und liefert eine neue Version des „state". Die individuelle Präsentationslogik einer Stage befindet sich in einer Handler-Funktion *(siehe Listing 6)*.

Man sieht, dass auch die Präsentationslogik in einem „core.async go"-Block liegt. Man kann nun innerhalb eines Handlers einen blockierenden Aufruf eines Service ausführen *(siehe Listing 7)*.

In dem „let"-Block wird ad hoc ein Kanal erzeugt, der in einem „Future", also einem Stück nebenläufigen Code verwendet wird, um das Ergebnis des „expensive-service" aufzunehmen. Der „let"-Block hat als Ergebnis diesen Kanal; mit „<!" wird blockierend aus dem Kanal gelesen. Von diesem Blockieren ist der JavaFX Application Thread nicht betroffen, es ist also nur die Ereignisverarbeitung in dieser Stage blockiert.

Möchte man dagegen, dass die Ereignisverarbeitung dieser Stage weiterläuft, kann man das Ergebnis des „expensive-service" in den „events"-Kanal des Stage-Prozesses leiten, womit es sich wie ein gewöhnliches Ereignis in die Ereignisverarbeitung einreiht *(siehe Listing 8)*.

Selbstredend kann man sehr technisch anmutende Idiome in Makros kapseln, sodass der Handler Code verständlich bleibt. Der wesentliche Aspekt ist, dass man durch die Verwendung von „core.async" die Ereignisverarbeitung grundsätzlich in einem eigenen Thread durchführt, wodurch man im Handler keine Rücksicht mehr auf den JavaFX Application Thread nehmen muss.

Fazit

Es wurden zwei wichtige Ideen vorgestellt, die von der in Java gewohnten Weise abweichen, GUIs zu programmieren. Sie werden bewusst genutzt, um auch in GUI-Code möglichst viele seiteneffektfreie, leicht testbare

```
(defn item-editor-handler
  [state event]
  (go (case ((juxt :source :type) event)
     ["OK" :action]
     (assoc state :terminated true)
     ["Cancel" :action]
     (assoc state
     :terminated true
     :cancelled true)
     state)))
```

Listing 6

```
(<! (let [ch (chan)]
  (future (put! ch (expensive-service)))
  ch))
```

Listing 7

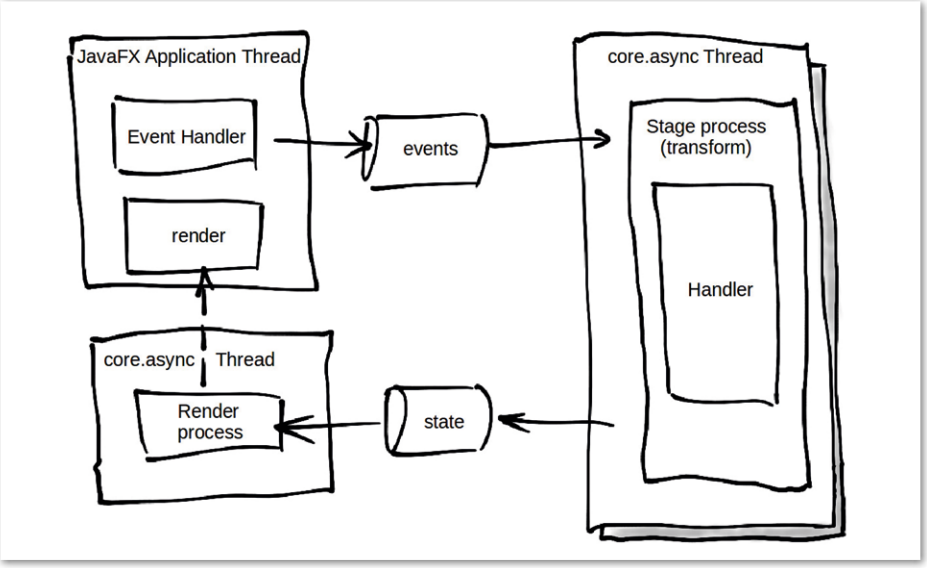


Abbildung 3: Prozesse und Kanäle in einem Rich Client

```
(future (put! events
     {:type :call
     :value (expensive-service-call)}))
```

Listing 8

Funktionen schreiben zu können, was sich auch in einem Prototyp besichtigen lässt *[6]*.

Die erste Idee ist die konsequente Verlagerung aller GUI-Zugriffe in eine Rendering-Funktion, die durch pure Daten, die ein vollständiges Modell der Stage und ihrer Daten bilden, gesteuert wird. Die zweite Idee ist die Verwendung des Programmiermodells CSP, das es erlaubt, den inhärent asynchronen GUI-Code natürlicher zu formulieren, als man das mit Callbacks allein je erreichen könnte. Clojure ermöglicht damit einen aus Architektursicht äußerst reizvollen Entwurf, der die häufig überbordende Komplexität in Rich Clients reduzieren hilft.

Referenzen

[1] C. Okasaki, „Purely Functional Data Structures", Cambridge University Press, 1998

[2] G.E. Krasner, S.T. Pope, „A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System", ParcPlace Systems, 1988

[3] React.js: „http://facebook.github.io/react"

[4] core.async: „https://github.com/clojure/core.async"

[5] C.A.R. Hoare, „Communicating Sequential Processes", Prentice-Hall, 1985

[6] async-ui-Prototyp: „https://github.com/friemen/async-ui"

Falko Riemenschneider
falko.riemenschneider@arcor.de

Falko Riemenschneider arbeitet als Software-Architekt als leitender Software-Architekt bei doctronic in Bonn. Er schreibt regelmäßig auf „http://www.falkoriemenschneider.de".

http://ja.ijug.eu/15/3/12

Java-Dienste in der Oracle-Cloud

Dr. Jürgen Menge, ORACLE Deutschland B.V. & Co. KG

Die Bedeutung des Themas „Public Cloud" ist trotz aller Vorbehalte unbestritten: Die Landschaft für Cloud Computing wird im Jahr 2017 zunehmend von Plattform- und Datenbank-Services dominiert werden, die hybride Infrastrukturen unterstützen. Zu diesen Ergebnissen kommt eine Studie, die IDG Connect im Auftrag von Oracle durchgeführt hat – ein Grund mehr, sich anzuschauen, was die Oracle-Cloud im Bereich „Java" anzubieten hat.

In der Cloud unterscheidet man zwischen Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS) und Data as a Service (DaaS). Für Entwickler sind neben dem Database-Service vor allem der Developer-, der Mobile- und der Java-Cloud-Service interessant. Der Java Cloud Service (JCS) hat drei Ausprägungen:

- SaaS Extension
- Virtual Image
- Java Cloud Service

Bei SaaS-Extension handelt es sich um einen reinen PaaS, bei dem die Version 10.3.6 des Oracle WebLogic Server (WLS) mit den ADF-Bibliotheken von 11.1.1.7.1 zur Verfügung steht. Die dazu passende Version des JDeveloper 11.1.1.7.1 ist eine spezielle Cloud-Version der IDE. Die Bezeichnung „SaaS Extension" wurde gewählt, weil auf Basis dieses Service-Erweiterungen der Oracle-Standard-Software (E-Business Suite beziehungsweise Fusion Applications) betrieben werden können. Es lassen sich auch andere Arten von Applikationen einrichten, sofern sie mit der geforderten Version entwickelt wurden. Während dies bei ADF-Applikationen die Version 11.1.1.7 beziehungsweise 11.1.1.7.1 sein muss, fallen bei Web-Services die Abhängigkeiten von der Version des JDeveloper weniger ins Gewicht, solange die jeweilige Version des Web-Service-Standards unterstützt wird. Insgesamt sind aber die Konfigurations- und damit Einflussmöglichkeiten in Bezug auf den Service relativ gering.

Bei den beiden anderen Java Cloud Services handelt es sich dagegen um erweiterte IaaS, bei denen eine WLS-Umgebung zum Deployment von Java-Applikationen zur Verfügung steht.

Zum Test der Cloud Services gibt es einen Trial-Account für eine begrenzte Zeit *(siehe „https://cloud.oracle.com/java")*.

Dr. Jürgen Menge
juergen.menge@oracle.com

Dr. Jürgen Menge hat bei Oracle zunächst sechs Jahre als Trainer für die Entwicklungs-Werkzeuge (Designer und Forms) gearbeitet, um dann als technischer Berater in den Lizenzvertrieb zu wechseln. Sein fachlicher Schwerpunkt sind nach wie vor die Entwicklungs-Werkzeuge – sowohl die klassischen (Forms, Reports, Designer) als auch die neuen Standard-basierten Werkzeuge (JDeveloper/ADF, BI Publisher etc.).

http://ja.ijug.eu/15/3/13

Highly scalable Jenkins

Sebastian Laag, adesso AG

Continuous Integration (CI) Server werden heutzutage von vielen Projekten genutzt, um Qualitätsstandards der eigenen Software dauerhaft sicherzustellen. Jenkins [1] ist eines der populärsten Produkte in diesem Bereich. Es kann um Hunderte Plug-ins erweitert werden, um den meisten Anforderungen eines kontinuierlichen Build-Prozesses gerecht zu werden.

Wenn die Auslastung von Jenkins – bedingt durch eine große Anzahl von Jobs oder die häufige Ausführung von Builds sehr hoch ist, lässt sich die Last mithilfe von Slaves verteilen. In großen Software-Projekten entstehen so schnell Instanzen mit einer zweistelligen Anzahl von Slaves. Die Herausforderung ist hier die einheitliche Einrichtung und deren Verwaltung. Dies ist gerade bei sehr heterogenen Anforderungen an Betriebssystem und zu nutzende Software ein erheblicher Aufwand.

Der Artikel stellt Strategien zum Umgang mit großen Jenkins-Instanzen vor und zeigt dabei verschiedene Umsetzungsmöglichkeiten bei der Einrichtung und Verwaltung von Jenkins-Master und -Slaves. Zudem werden Plug-ins für die erleichterte Administration einer Jenkins-Umgebung demonstriert.

Entlastung für den Master

Wenn das Projekt weiter wächst und immer mehr Anwendungen Continuous Integration verwenden, kommt der Punkt, an dem sich die Build-Warteschlange mehr und mehr füllt. Das führt zu langen Wartezeiten auf Feedback des letzten Check-ins in die Versionskontrolle.

Kurzfristig lässt sich dieser Zustand durch die Freigabe weiterer sogenannter „Executor" in Jenkins lösen. Aber selbst die am besten ausgerüstete Instanz ist irgendwann überlastet. Danach kann eine weitere Entlastung durch die Einführung von Slaves erreicht werden. Ein Slave wird auf einer vom Master separierten Instanz eingerichtet und kann über verschiedene Wege mit dem Master verbunden werden. Dabei stehen SSH, JNLP oder die Ausführung auf Windows als Dienst zur Auswahl.

Zudem kann hier entschieden werden, ob ein Slave für alle Jobs zur Verfügung stehen soll oder nur für solche, die explizit für den Slave konfiguriert sind. Letzteres ist dann sinnvoll, wenn eine heterogene Infrastruktur mit verschiedenen Betriebssystemen im Einsatz ist.

Verschiedene Slaves lassen sich unter einem „Label" gruppieren. In der Job-Konfiguration können anschließend durch die Eingabe des Labels alle Slaves adressiert werden. Der Jenkins-Master entscheidet dann je nach Auslastung, auf welchem Slave der jeweilige Build ausgeführt wird.

Windows-Slaves und plötzlich Probleme

Je nach Projekt werden für verschiedene Builds oder Tests Windows-Slaves eingesetzt. Diese können als Dienst oder per JNLP mit dem Master verbunden werden. Bei der Verwendung von Windows passiert es häufig, dass sich der Jenkins-Prozess auf dem Slave unerwartet beendet. Meist lässt sich das Problem auf das Transportprotokoll JNA zurückzuführen.

Bis vor Kurzem verwendete Jenkins eine veraltete Version von JNA. Eine manuelle Aktualisierung kann hier Abhilfe schaffen. Dafür muss das JAR „jna-3.5.0.jar" im Ordner „war\WEB-INF\lib" durch die neueste Version 4.1.0 ersetzt werden. Diese lässt sich zum Beispiel über verschiedene öffentlich zugängliche Maven-Repositories herunterladen.

Wichtig ist es, danach erst einen Neustart des Masters und danach aller Slaves vorzunehmen. Bei der Aktualisierung von Jenkins kann es unter Umständen vorkommen, dass die veraltete Version wie-

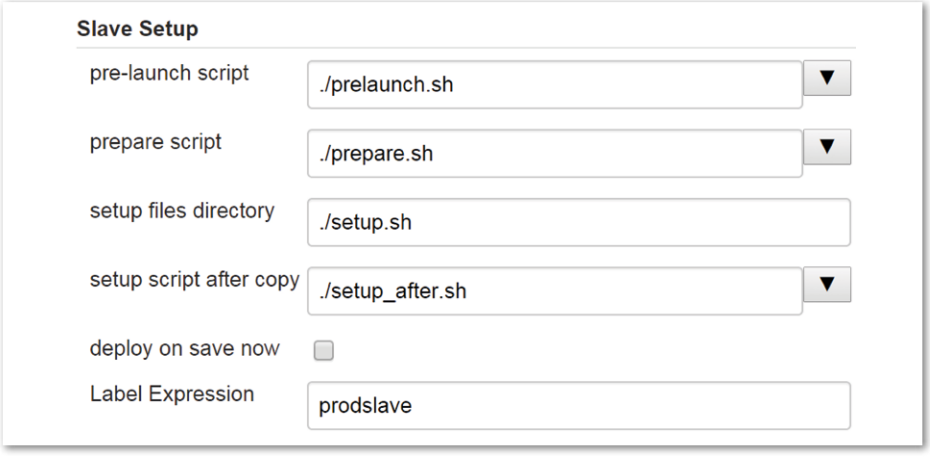


Abbildung 2: Konfiguration des Slave-Setup-Plug-ins

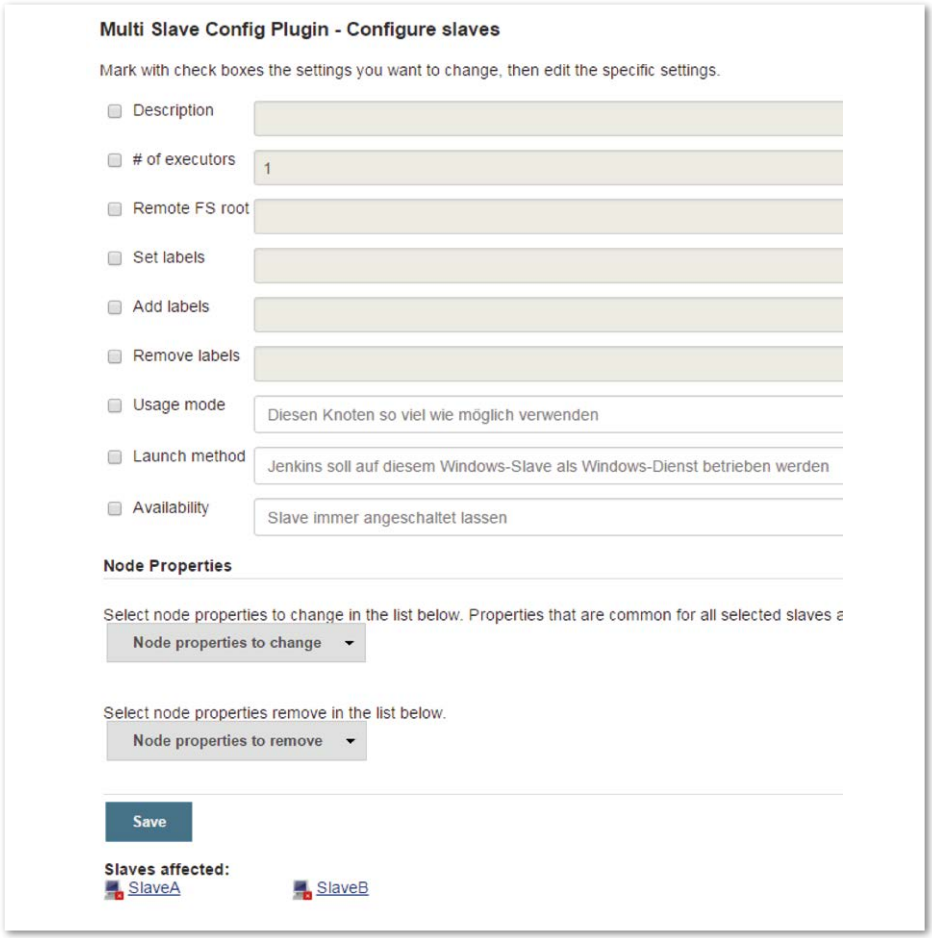


Abbildung 1: Gleichzeitige Konfiguration unterschiedlicher Slaves

der eingespielt wird. Falls dies der Fall ist, muss die Aktualisierung von JNA erneut erfolgen.

Nur ein Master und dann Wartungsarbeiten?

In überschaubaren Umgebungen mit einem Master und wenigen Slaves lassen sich Wartungsarbeiten noch zeitnah umsetzen und führen zu geringen Ausfallzeiten der Infrastruktur. Je nach Anzahl von Jobs und Häufigkeit von Builds ist es durchaus sinnvoll, von der üblichen Jenkins-Strategie mit einem Master und vielen Slaves abzuweichen. Hier ist der Einsatz mehrerer Master-Instanzen empfehlenswert. Dabei kann die Einrichtung der Jobs nach verschiedenen Typen wie Build, Integrationstest oder Deployment unterschieden werden. Alternativ lässt sich bei unterschiedlichen Teams auch eine Installation pro Team einrichten.

Wenn nun ein Jenkins-Master für Wartungsarbeiten abgeschaltet werden muss, können die anderen Instanzen weiterhin genutzt werden. Zudem ist die Verteilung von Builds auf eine große Anzahl von Slaves für den Jenkins-Master sehr Speicher-intensiv und kann schnell zur Belastung für die Infrastruktur werden.

Allerdings ist zu beachten, dass durch die Verwendung mehrerer Master-Instanzen auch der Wartungsaufwand steigt, da jedes Jenkins-System für sich bearbeitet werden muss. In den folgenden Abschnitten werden Plug-ins vorgestellt, mit denen der Wartungsaufwand reduziert und das Anlegen verschiedener gleichartiger Jobs erleichtert werden kann.

Configuration Slicing

Je nach Anzahl der Jobs in einer Jenkins-Installation kann eine Veränderung der Konfiguration mehrerer Jobs mit viel Aufwand verbunden sein. Das Configuration-Slicing-Plug-in *[2]* bietet eine gute Möglichkeit, die Konfiguration mehrerer Jobs gleichzeitig zu verändern. Dabei können Parameter wie verwendetes JDK, genutzter Slave, Log-Rotate-Einstellung, die Maven-Version und einige mehr für die gewünschten Jobs auf einmal angepasst werden.

Bei der Benutzung der Plug-ins (Verwaltung -> Configuration Slicing) sind auf einer Übersichtsseite alle Konfigurations-Parameter aufgelistet. Nach Auswahl eines Parameters werden auf der linken Seite die Einstellung und auf der rechten die entsprechend konfigurierten Jobs eingetragen. Durch Verschieben der Job-Namen von einem zum anderen Parameter und anschließendes Speichern werden alle Jobs gleichzeitig verändert. Viele Plug-ins bringen die passende Erweiterung für das Configuration Slicing direkt mit, sodass per Plug-in hinzugefügte Parameter ebenfalls für mehrere Jobs gleichzeitig angepasst werden können.

Multi Slave Config

Das Multi-Slave-Config-Plug-in *[3]* ermöglicht das Anlegen und Bearbeiten mehrerer sogenannter „dumb slaves" gleichzeitig. So lassen sich Parameter wie die Anzahl der Executor, das „Home-Verzeichnis" oder Umgebungs-Variablen für die Slaves verändern. Doch nicht nur Einstellungen können geändert, sondern auch Slaves offline genommen oder wieder verbunden werden. Zudem kann man die angelegten Slaves auch auf einmal entfernen *(siehe Abbildung 1)*.

Slave-Setup-Plug-in

Bevor ein Build auf einem Slave gestartet werden kann, sind meist Vorbedingungen wie die korrekte Version eines JDK, des Build-Tools oder anderer Skripte notwendig. Das Slave-Setup-Plug-in *[4]* schafft die genannten Bedingungen. Dafür bietet es die Möglichkeit, ein Skript vor der (Wieder)-Verbindung des Jenkins-Masters mit dem Slave oder beim Speichern der Jenkins-Konfigurationsseite auszuführen.

Für jede Konfiguration der Skripte wird in der Jenkins-Verwaltung auch ein Label hinterlegt. Dessen Name muss mit dem des Slaves übereinstimmen, damit die Skripte ausgeführt werden *(siehe Abbildung 2)*.

Docker Slaves

Das Docker-Plug-in *[5]* stellt eine weitere Möglichkeit zur Ausführung von Jobs bereit. Damit kann pro Build ein eigener Docker-Container gestartet, dann der Build ausgeführt und der Container wieder zerstört werden. Der Vorteil hier ist die Kapselung der Builds. Jeder Build wird in seinem eigenen Container ausgeführt und hat somit keine Seiteneffekte auf andere Builds. Dies spart auch Speicherplatz, da der Docker-Container

Project name	Jobs	Builds all	Builds locked	Workspace
ChildChildJobA	130 KB	116 KB	7 KB	-
ChildJobA	63 KB	53 KB	-	-
ParentJob	43 KB	34 KB	-	-
ChildJobB	28 KB	18 KB	-	-
MultiJob	-	-	-	-
Total	**265 KB**	**222 KB**	**7 KB**	**-**

Abbildung 3: Anzeige des Speicherplatzverbrauchs

inklusive Slave und Workspace nach Ausführung eines Builds komplett entfernt wird.

Dafür muss nicht nur das Plug-in installiert, sondern auch eine Docker-Umgebung eingerichtet sein. Dazu gehört die Einrichtung eines Docker-Image mit SSH-Daemon, JDK und einem Benutzer zum Einloggen. Bei der Einrichtung der Umgebung ist die richtige Docker-Version entscheidend. Diese muss mit dem Plug-in kompatibel sein. Erfahrungsgemäß ist die aktuelle Version von Docker nicht zwangsläufig mit dem Jenkins-Plug-in kompatibel.

Job DSL

Das übliche Vorgehen beim Anlegen immer weiterer ähnlicher Jobs ist das Erstellen einer Kopie eines „Vorlage"-Jobs. Sollen sich allerdings bestimmte Parameter ändern, wird es irgendwann schwierig, die Konsistenz zwischen den Jobs sicherzustellen.

Das Job-DSL-Plug-in *[6]* verfügt über ein Kommando-API zur Generierung von Jobs. Hier werden Groovy-Skripte verwendet, um das Anlegen und Umkonfigurieren von Jobs durchzuführen. Zur Verwendung des Plug-ins werden ein Free-Style-Job erstellt und der Build-Schritt „Process Job DSLs" hinzugefügt. In diesen muss ein Groovy-Skript mit Job-DSL-Kommandos *[7]* eingefügt sein. Es dient als Vorlage für weitere Generierungen.

Nach Ausführung des Jobs werden die generierten Jobs angezeigt. Alternativ kann man auch den Pfad zur Datei innerhalb des Workspace angeben. Somit ist es möglich, das verwendete Vorlage-Skript zu versionieren und durch den Job auschecken zu lassen. Auf diese Weise können Jobs auf verschiedenen Jenkins-Installationen immer wieder angelegt oder Konsistenzen zwischen ihnen dauerhaft sichergestellt werden.

Jenkins und die richtige Version

Bei der Einrichtung einer Jenkins-Umgebung kann von Anfang an viel falsch gemacht werden. Dies beginnt bei der Wahl der Jenkins-Version. Jenkins bietet einen Long-Term-Support (LTS) und eine Version ohne langfristigen Support an. Die LTS-Version erfährt seltener Neurungen, ist dafür aber ausführlich getestet und enthält nur selten Fehler. Aus eigener Erfahrung empfiehlt der Autor immer den Einsatz der LTS-Version.

Wichtig ist dabei die Aktualisierungsstrategie. Wie in vielen Systemen üblich, sollten Updates nur dann vorgenommen werden, wenn sie notwendig sind und zu einer spürbaren Verbesserung führen. Auf der offiziellen Webseite von Jenkins existiert ein Change-Log, das über Neuerungen informiert. Dies bietet zudem ein Community-Rating, das einen Hinweis darauf gibt, ob die entsprechende Version verwendet oder ausgelassen werden sollte.

Viele Plug-ins – viele Probleme

Wesentliche Features wie die Verwendung verschiedener Benachrichtigungs-Arten, der Einsatz einer Build-Pipeline, die Überwachung des Speicherverbrauchs oder der Einsatz von SVN oder Git müssen per Plug-in nachgerüstet werden. Die große Auswahl von Plug-ins ist verführerisch und lädt schnell dazu ein, immer mehr davon zu installieren. Doch hier ist Vorsicht geboten. Einige Plug-ins sind nur unzureichend getestet und führen schnell zu Problemen. Dabei kommt es sogar vor, dass sich Plug-ins nicht korrekt deinstallieren lassen und die gesamte Jenkins-Instanz lahmlegen, da die Plug-in-Parameter nicht korrekt aus der Job-Beschreibung entfernt wurden.

Auch bei Updates von Plug-ins sollte man vorsichtig vorgehen. Manch eine Aktualisierung führt zu Inkompatibilitäten zu anderen Plug-ins oder sogar dazu, dass eine ganze Funktionalität nicht mehr verwendet werden kann. Ist dies der Fall, kann über das Plug-in-Center eine Wiederherstellung der alten Plug-in-Version erfolgen. Wer vor einer Aktualisierung Informationen über Änderungen bekommen möchte, findet über das Plug-in-Center die Wiki-Seite des jeweiligen Plug-ins.

Wartungsarbeiten

Nach langer Betriebsdauer einer Jenkins-Installation wird erfahrungsgemäß der Speicherplatz knapp. Das liegt unter anderem daran, dass beim Löschen eines Jobs der Workspace nicht entfernt wird. Gleiches gilt für die Umbenennung von Jobs. So entstehen schnell Workspace-Leichen, die nicht automatisch verschwinden. Jenkins bietet hier keine automatische Möglichkeit, diese aufzuräumen. Das Disk-Usage-Plug-in *[8]* identifiziert die Platzverschwender. Dabei wird der Speicherplatz von Workspace und Builds festgestellt. Die Daten sind ein guter Indikator für Aufräumarbeiten *(siehe Abbildung 3)*.

Ein weiterer, häufig gemachter Fehler ist die fehlende Einrichtung einer automatischen Lösch-Strategie von Builds. Jenkins bietet in der Job-Konfiguration die Einstellung „discard old builds" an. Hier kann eine Limitierung auf eine bestimmte Anzahl von Builds oder eine Speicherung von „x" Tagen eingerichtet werden. Das spart nicht nur jede Menge Speicherplatz, sondern sorgt auch dafür, dass Jenkins optimal arbeiten kann. Jeder Build wird intern von Jenkins verwaltet und sorgt daher für eine Verlangsamung des Systems.

Fazit

Beim Aufbau einer Jenkins-Instanz ist die richtige Strategie entscheidend. Dabei sollten Rahmenbedingungen wie die Anzahl von Teams und die Verschiedenheit ihrer Anforderungen an Jenkins betrachtet werden. Sind die Anforderungen extrem unterschiedlich, ist es sinnvoll, mehrere Master zu verwenden. Sind die Anforderungen sehr ähnlich und Wartungsarbeiten sind nur selten zu erwarten, kann eine große Anzahl von Slaves verwendet werden.

In jedem Fall sollte die LTS-Version von Jenkins eingesetzt werden, um Probleme mit Instabilitäten zu vermeiden. Jenkins bietet eine Vielzahl von Plug-ins, um den Wartungsaufwand zu reduzieren. Dazu gehören die gleichzeitige Konfigurationsänderung mehrerer Jobs oder Slaves sowie eine Skript-Sprache für das Anlegen und Aktualisieren von Jobs. Alternativ können mit dem Configuration-Slicing-

Parameter mehrere Jobs gleichzeitig aktualisiert werden.

Die Verwendung von Docker bietet eine gute Möglichkeit, die Seiteneffekte bei der Ausführung gleichzeitiger Jobs zu minimieren. Das Multi-Slave-Config-Plug-in erleichtert die Konfiguration von Slaves und spart somit Zeit beim Anlegen, Anpassen oder Löschen selbiger. Insgesamt sollte an regelmäßige Wartungsarbeiten der Jenkins-Umgebung gedacht werden. Zudem ist die Installation jedes Plug-ins zu überdenken.

Weitere Informationen

[1] https://jenkins-ci.org
[2] https://wiki.jenkins-ci.org/display/JENKINS/Configuration+Slicing+Plugin
[3] https://wiki.jenkins-ci.org/display/JENKINS/Multi+slave+config+plugin
[4] https://wiki.jenkins-ci.org/display/JENKINS/Slave+Setup+Plugin
[5] https://wiki.jenkins-ci.org/display/JENKINS/Docker+Plugin
[6] https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin
[7] https://github.com/jenkinsci/job-dsl-plugin/wiki/Job-DSL-Commands
[8] https://wiki.jenkins-ci.org/display/JENKINS/Disk+Usage+Plugin

Sebastian Laag
sebastian.laag@adesso.de

Sebastian Laag (Dipl. Inf., Univ.) ist als Senior Software-Engineer bei der adesso AG in Dortmund tätig und arbeitet derzeit als Product Owner in einem Telematik-Projekt. Er hat bereits verschiedene Artikel im Continuous-Integration-Umfeld veröffentlicht und ist darüber hinaus leidenschaftlicher Fan von Borussia Dortmund.

http://ja.ijug.eu/15/3/14

Vaadin – der kompakte Einstieg für Java-Entwickler

Gelesen von Daniel Grycman

Dieses Buch ist das erste Einsteigerwerk in deutscher Sprache, das sich mit dem Versionsstand 7 des Vaadin-Frameworks befasst. Dabei ist kein umfassendes Nachschlagewerk entstanden; die Autoren sind sich dessen aber bewusst und verweisen entsprechend in ihrem Vorwort auf das englischsprachige „Book of Vaadin" als Referenzwerk.

Die Kapitel-Aufteilung folgt keinem konsequenten Aufbau. Aus diesem Grund ist es dem Leser möglich, die einzelnen Kapitel getrennt voneinander zu lesen, ohne einen roten Faden zu vermissen. Am Anfang gehen die Autoren den Fragen nach, was Vaadin genau ist, wie es funktioniert und wie die entsprechende Vaadin-Architektur aussieht. Es folgt ein Kapitel mit jeweils einer kurzen Beschreibung verschiedener UI-Komponenten, wobei folgende Struktur verwendet wird: Bezeichnung, Funktion und Einsatz der Komponente im Quelltext.

Die nachfolgenden Kapitel fünf bis acht gehen auf die Punkte „Data Binding", „Server Push", „Layout", „Styling mit CSS/Sass" und „Navigation" ein. Für alle Beschreibungsteile gilt das Credo des Untertitels. Alle Inhalte werden dem Leser in kompakter Form präsentiert.

In Kapitel neun, das sich mit Architektur- und Entwurfsmuster-Konzepten in Bezug auf Vaadin auseinandersetzt, kommt es zu einem Bruch mit der bisherigen Kapitelstruktur. Es wird eine Beispiel-Anwendung geboten, an der die einzelnen Konzepte dargestellt werden. Die Autoren erläutern dabei die konkrete Implementierung anhand der beiden Architekturmuster Model-View-Presenter (MVP) und Model-View-ViewModel (MVVM), wobei sie bei MVP noch eine Unterteilung nach den Varianten „Passive View" und „Supervising Controller" vornehmen. Zudem wird noch auf weitere Entkopplungs- und Kapselungsmöglichkeiten durch Event-Bus und CDI eingegangen. An diesem Kapitel merkt der Leser, wie sehr es den Autoren auf den Praxis-Einsatz des Vaadin-Frameworks ankommt.

Die letzten Kapitel befassen sich mit den Themen „Add-ons", „Buildmanagement mit Maven" und „automatisiertes Testen". Hier wird, wie schon in den Kapiteln vier bis acht, ein fundiertes Basiswissen vermittelt.

Titel:	Vaadin – Der kompakte Einstieg für Java-Entwickler
Auflage:	1. Auflage 2015
Autoren:	Joachim Baumann, Daniel Arndt, Frank Engelen, Frank Hardy, Carsten Mjartan
Verlag:	dpunkt.verlag. Heidelberg
Umfang:	280 Seiten
Preis:	34,90 Euro eBook (downloadbar) im Preis enthalten
ISBN:	978-3-86490-206-2

Fazit

Die Autoren ermöglichen einen Einstieg, der aufgrund des geringen Umfangs zwar relativ kurz erscheint, aber auf knappem Raum viele Informationen bietet. Verbesserungsvorschläge für eine aktualisierte zweite Auflage wären zum einen die Platzierung des Maven-Kapitels vor dem Architektur-Kapitel, da Vorwissen über die Verwendung von Maven benötigt wird, und zum anderen eine tiefere Darstellung der Datenbank-Anbindung.

Wer zum Erlernen einer neuen Technologie eine durchgestylte Beispiel-Anwendung benötigt, sollte dieses Buch nicht unbedingt lesen. Wer bereit ist, abseits eines vorgegebenen Weges mitzudenken, wird an diesem Buch seine Freude haben.

Daniel Grycman
daniel.grycman@bilsteingroup.com

First one home, play some funky tunes!

Pascal Brokmeier, OPITZ CONSULTING Deutschland GmbH

Das Internet der Dinge (IoT) ist in aller Munde und bringt der Heim-Automatisierung sowohl im Bastlerumfeld als auch auf kommerzieller Seite frischen Wind in die Segel. Dabei bietet sich die Java-Plattform geradezu an, um die in fast jeder Architektur auffindbaren Gateways zu treiben und die Logik näher an die Grenze der realen Welt zu bringen. Ausgehend von diesen Gegebenheiten entstand ein Heim-Automatisierungs-Projekt auf Basis günstiger Hardware und gängiger Java-Enterprise-Technologien, das zeigt, wie Entwickler bestehendes Backend-Wissen für zukünftige IoT-Projekte einsetzen können.

Wer kennt das nicht: Licht angelassen, Soundsystem über Nacht eingeschaltet oder vergessen, die Heizung herunterzudrehen … Motivationen für die Heim-Automatisierung sind lange bekannt und bereits seit einiger Zeit gibt es Lösungsansätze zur automatisierten Steuerung von Gebäuden. Das Problem dabei: Diese Systeme waren bisher meist unverhältnismäßig teuer, basierten auf kabelgebundenen Bus-Systemen oder nutzten proprietäre und schwer erweiterbare Systeme.

Die Entwicklungen der letzten Jahre lassen jedoch einen Wechsel erhoffen. Geräte wie der Raspberry Pi bieten Bastlern eine günstige Basis, um vollwertige Java-Server-Applikationen zu betreiben. Offene APIs wie RESTful-Webservices führen zu einfacherer Interoperabilität. Mithilfe dieser Services ist es möglich, herstellerübergreifende Architekturen zu entwickeln. Eine einfache Suche auf Google nach „DIY home automation" zeigt, dass es unglaublich viele Projekte gibt. Viele davon setzen auf dem Raspberry Pi als Hardware-Plattform auf.

Raspberry PI und Java

Der Prototyp eines Projekts zur Heim-Automatisierung, das der Artikel näher vorstellt, basiert auf Raspberry Pi und Java-Framework. Ziel ist es, abhängig von Benutzer-Präsenzen und Wetterverhältnissen bestimmte Licht-Einstellungen vorzunehmen oder Musik abzuspielen. Technischer ausgedrückt: Es sollen kontextsensitive Regeln erstellt werden, die an Aktionen gekoppelt sind. Dafür ist von Anfang an eine offene Architektur wichtig. Diese soll folgende Aufgaben erfüllen:

- Die Steuerung weiterer Geräte-Typen ermöglichen
- Andere Informationsquellen integrieren
- Neue Regeln zur Kombination von Event-Erzeugern und Akteuren erfassen

Ein weiterer Aspekt sind die Hardware-Kosten: Hier soll so wenig Geld wie möglich investiert werden. Für einfachste Anwendungsfälle lassen sich die in vielen Haushalten bereits vorhandenen, simplen 433-MHz-Funksteckdosen integrieren. Diese reichen aus, um einfache Leuchten und ältere elektronische Geräte anzuschalten.

Die Architektur des Prototyps

Hersteller wie Oracle und Intel setzen bei ihren IoT-Architekturen auf Gateways, um die Brücke zwischen den Dingen und komplexerer IT wie Serveranwendungen, Big Data etc. zu schlagen. Oft wird auf diesen Gateways bereits beträchtliche Intelligenz ausgeführt, um zum Beispiel Events zu filtern oder lokale Steuerungen vorzunehmen. Dieses Konzept wird im Prototyp aufgegriffen; bereits mit kleinen Gateways ist also einiges möglich.

Die zentrale Einheit des Systems bildet eine Spring-Applikation auf einem Raspberry Pi, gehostet durch einen Jetty-Server. Sie stellt ein REST-API zur Steuerung von Geräten und zur Änderung der Benutzer-Zustände zur Verfügung und kann entweder manuell mit einer Mobile App angesprochen oder durch andere Dinge genutzt werden.

Ein zweiter Raspberry Pi ermittelt, ob sich Benutzer im Haushalt befinden. Er hostet eine Oracle-Event-Processing-Applikation (OEP). Taucht ein Benutzer im lokalen Netz auf, wird diese Information an die Spring-Applikation weitergegeben. Diese prüft daraufhin ihre Regeln und schaltet entsprechende Gerätegruppen *(siehe Abbildung 1)*.

Struktur der Spring-Applikation

In der Spring-Applikation sind Camunda BPM, ein Prozessautomatisierungs-Framework, sowie Drools, eine Rule Engine, integriert. In

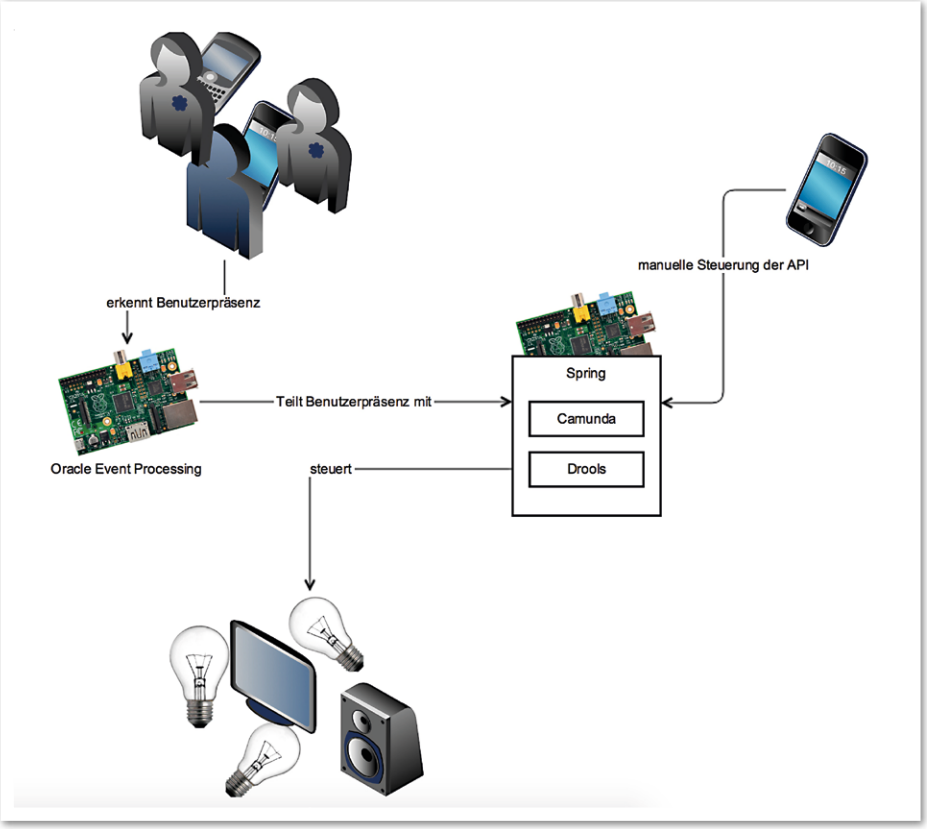


Abbildung 1: System-Übersicht

Drools können diverse Regeln zur Steuerung der Beleuchtung und der Geräte hinterlegt werden, etwa „Alle Bewohner zu Hause + draußen dunkel oder regnerisch = Wohnzimmer-Beleuchtung an". Oder „Nur Person X daheim = Licht im Büro an + leise Radiomusik".

Der Spring-Server ist in Schichten aufgeteilt. Sobald die Logik des Event-Processing eine Änderung bei der Benutzerpräsenz erkennt, wird diese durch einen API-Aufruf an den Spring-Server übermittelt. Die Camunda-Prozesse greifen für ihre Tätigkeiten auf die darunterliegenden Services zu und orchestrieren verschiedene Handlungen *(siehe Abbildung 2)*.

Zur Verdeutlichung: Wird ein Event übermittelt, der besagt, dass ein Benutzer das Haus betreten hat, wird ein entsprechender Prozess gestartet. Hier wird nun geprüft, welche Geräte geschaltet werden sollen. Dazu wird auf die Regeln von Drools zugegriffen. Diese geben vor, welche Geräte für diesen Benutzer geschaltet werden. Während diese Regeln aktuell noch abhängig von der jeweiligen Präsenzsituation schalten, lassen sie sich zukünftig beliebig erweitern. So könnte eine Regel heißen: „Wenn Benutzer X nach Hause kommt, niemand sonst daheim ist, es draußen dunkel ist oder es laut Wetterservice gerade regnet, dann soll das Licht im Wohnzimmer angehen und Musik mit dem Thema funky' abgespielt werden" *(siehe Listing 1)*.

Dieses Regelwerk wird aus der Camunda-Lösung heraus aufgerufen. Nachdem alle Regeln durchlaufen sind, werden die gewünschten Geräte geschaltet und die auszuführenden Befehle ausgeführt. Konkret wird aus der Persistenz-Schicht ein Objekt des Typs „Musicbox" geladen und bei diesem die Methode „searchAndPlay(„funky")" ausgeführt. Der in der Service-Schicht liegende Adapter für die Musicbox schickt diesen Befehl anschließend an den Raspberry Pi, der an das Soundsystem angeschlossen ist. Dieser spielt Musik ab, die via Google Music zur Suche „funky" gefunden wurde *(siehe Abbildung 3)*.

Funksteckdosen aus Java steuern

Um die Funksteckdosen zu steuern, muss man sich als Java-Entwickler ein wenig aus seiner gewohnten Umgebung heraustrauen. Der Raspberry Pi bietet mit seinen GPIO-Pins eine Hardware-Schnittstelle zur Steuerung eines 433-MHz-Funkmoduls. Es existiert eine C++-Bibliothek „wiringpi" *[1]*, um die Steuerung der GPIO-Pins zu vereinfachen. Mit „rcswitch-pi" *[2]* können dann Funksteckdosen in C angesteuert werden. Dazu müssen die 2x5-Bit-Codes für die gängigen ELRO-Chip-Steckdosen übergeben werden. Nachdem „wiringpi" im System installiert wurde und somit unter „/usr/local/lib" liegt, können diese Bibliotheken genutzt werden.

Um diese Funktionalität in Java möglich zu machen, ist ein Java Native Interface (JNI) erforderlich. Dazu erstellt man eine Java-Klasse, etwa „NativeRCSwitchAdapter, und schreibt native Methodenköpfe: „public native void switchOn(String group, String channel);"

Anschließend kompiliert man die Klasse mit „javac" und lässt C-Header-Dateien mittels „javah com.opitz.jni.NativeRCSwitchAdapter" generieren. So entsteht eine C-Header-Datei mit allen Funktions-Deklarationen. Diese Funktionen sind gebunden an die Methoden aus der Java-Klasse. Wird die oben beschriebene Methode aufgerufen, führt die JVM die Funktion im C-Code aus. Tiefer soll jetzt nicht in die Entwicklung in C eingestiegen werden. Die durch „javah" generierten Funktions-Deklarationen sind jedoch erwähnenswert *(siehe Listing 2)*.

Das erste Argument im Skript ist ein Pointer zum JNI-Interface. Das zweite stellt eine Referenz zum Java-Objekt dar, die letzten

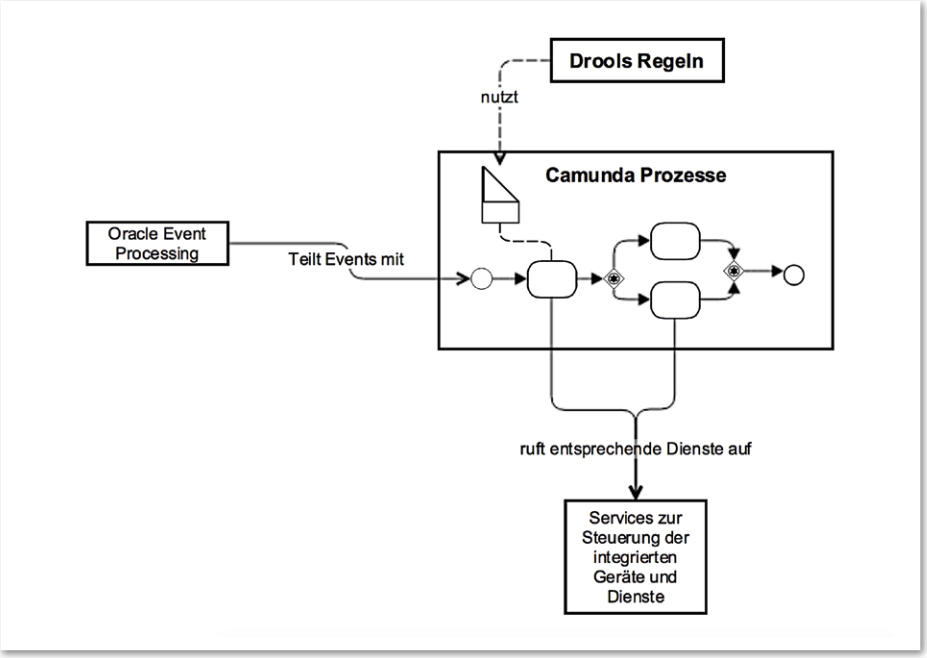


Abbildung 2: Zusammenspiel Camunda BPM, OEP, Services, Drools

```
//Regel schaltet Geräte aus Gruppe "Wohnzimmer-Licht" sowie musicbox an
rule "Apply light theme for Pascal"
when
  u : User( username == "Pascal" )
  u.isAlone()
  WeatherService.isRaining() || WeatherService.isDark()
then
  HashSet<String> devicesToSwitchOn = new HashSet<String>();
  devicesToSwitchOn.add("Wohnzimmer-Licht");
  devicesToSwitchOn.add("Wohnzimmer-musicbox");
  insert(devicesToSwitchOn);
end

//Regel spielt Musik mit Thema funky in Wohnzimmer ab
rule "Apply funky theme for Pascal"
when
  u : User( username == "Pascal" )
  u.isAlone()
  WeatherService.isRaining()
then
  HashSet<Command> commands = new HashSet<Command>();
  ArrayList<String> parameters = new ArrayList();
  parameters.add("funky");
  commands.add(new Command("Musicbox" "Wohnzimmer-musicbox", "searchAnd-
Play", parameters));
  insert(commands);
end
```

Listing 1

beiden sind die eigenen Strings, die der nativen Methode übergeben wurden. Da die String-Objekte nur als Referenz übergeben werden, muss man diese über das „JNIEnv" erfragen. Nach dem Absetzen der Funksignale muss man die String-Objekte noch über „env->ReleaseStringUTFChars(jsGroup, csGroup);" freigeben, damit der Garbage Collector diese aufräumen kann. Andernfalls entstände unter Umständen ein Memory-Leak *[3]*.

Native Bibliotheken in Java laden

Nachdem der C-Code implementiert ist, muss er noch in eine Shared Object Library kompiliert und ebenfalls unter „/usr/local/lib" abgelegt werden. Würde man den Java-Code nun ausführen, bekäme man einen „UnsatisfiedLinkError". Dies liegt daran, dass die JVM noch die entsprechend aufzurufende Bibliothek laden muss. *Listing 3* zeigt, wie das im Projekt gelöst wird.

Dafür bekommt die Datei in der Share-Object-Bibliothek den Namen „libRCSwitchAdapter.so" und wird unter einem der JVM bekannten Bibliotheken-Pfad abgelegt. Nun kann man die native Methode aus Java heraus aufrufen. Diese nutzt anschließend die C++-Funktionen der genannten Bibliotheken, um die Funkwellen der 433-MHz-Signale zu modulieren und damit die Steckdosen zu steuern.

Da die Steckdosen nicht zurückmelden, ob sie wirklich geschaltet haben, empfiehlt es sich, aktiv zu testen, ob die Signale auch überall ankommen. Ein besonders starkes Sendemodul oder eine gute Antenne sowie mehrere Wiederholungen des Sendevorgangs helfen, die erfolgreiche Schaltung sicherzustellen.

Dies zeigt einen wichtigen Punkt auf, den man sich als Entwickler zu Herzen nehmen sollte: Im Kontext des IoT kann TCP/IP-basierte Kommunikation ein Luxus sein. Die Ausprägungen der eingesetzten Geräte sind unterschiedlich – angefangen von einem einfachen Drucksensor bis zu einem hochkomplexen PKW gibt es somit auch viele verschiedene Kommunikationstechnologien. Fehleranfällige, langsame oder sehr einfache Kommunikationswege und deren Einschränkungen sollten also bei bestimmten Anwendungsfällen ausführlich bedacht werden. Im gezeigten Prototyp lassen sich durch das wiederholte Senden der Funkbefehle in der Applikationsebene Schwächen der zugrunde liegenden Kommunikationstechnologie kompensieren.

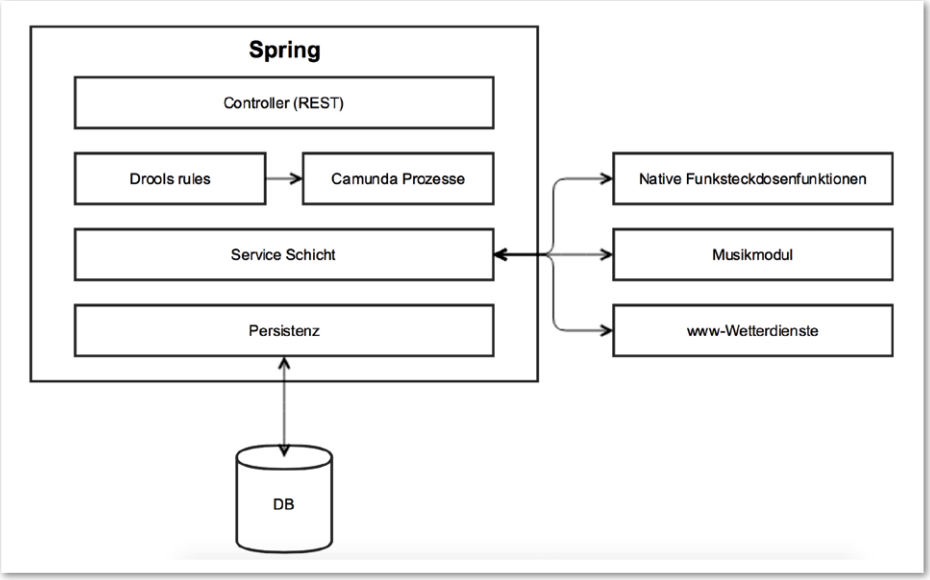


Abbildung 3: Spring-Applikation

Musik-Modul

Um bei der Heimkehr eines Bewohners neben einem warmen Licht auch angenehme Musik abzuspielen, ist ein Musik-Player erforderlich. Hier bietet sich die Pi Musicbox *[4]* an. Dabei handelt es sich um ein Projekt von GitHub, das einen vielseitigen Music-Player auf einen Raspberry Pi bringt und Dienste wie „Spotify", „Google Music" oder „AirPlay" unterstützt. Der Raspberry Pi kann an eine einfache Stereoanlage angeschlossen und mittels Funksteckdosen eingeschaltet werden. Sobald das System hochgefahren ist, startet das Lieblings-Webradio oder es werden über die Schnittstelle des Music Player Daemon (MPD) spezielle Songs oder Playlists ausgewählt *[5]*. Zur Steuerung der MPD-Schnittstelle aus Java heraus gibt es ebenfalls eine „JavaMPD"-Bibliothek. So kann der Spring Server den Musik-Pi wie oben beschrieben auf Basis von Regeln kontextabhängig steuern.

Benutzer-Präsenz und Oracle Event Processing

Die Benutzer-Präsenz-Erkennung ist einfach gehalten. Sie prüft lediglich, ob bekannte, den Benutzern zugeordnete Geräte im Netzwerk gefunden werden. Ein Protokoll namens „ARP" macht dies möglich. Es ist in der Internet Protocol Suite im Link-Layer, also sehr Hardware-nah angesiedelt und überprüft das Netzwerk auf Media-Access-Control-Adressen (MAC-Adressen).

Während sich IP-Adressen oft ändern können und nicht jeder Benutzer eine Client-seitige Applikation auf seinen Geräten installiert haben muss, kann man mithilfe dieses Filters für MAC-Adressen die Präsenz des Client-Geräts unabhängig von der jeweiligen Software feststellen. Hier soll beispielhaft ein Event-Netzwerk vom Oracle Event Processing (OEP) erläutert werden. *Abbildung 4* zeigt das beschriebene Netzwerk.

Der OEP-Pi pingt regelmäßig alle Geräte im Netzwerk an („NetworkNodeAdapter") und vergleicht anschließend die MAC-Adressen der Geräte (ausgelesen aus dem ARP-Cache des Betriebssystems) mit den ihm bekannten Geräten („UserDeviceProcessingBean"). Der Zustand wird für alle relevanten Geräte geprüft („StateCalculatingBean"). Hat sich dieser innerhalb der letzten

```
JNIEXPORT void JNICALL Java_com_opitz_jni_NativeRCSwitchAdapter_switchOn
  (JNIEnv *, jobject, jstring, jstring);
```

Listing 2

```
static{
  System.loadLibrary("RCSwitchAdapter");
}
```

Listing 3

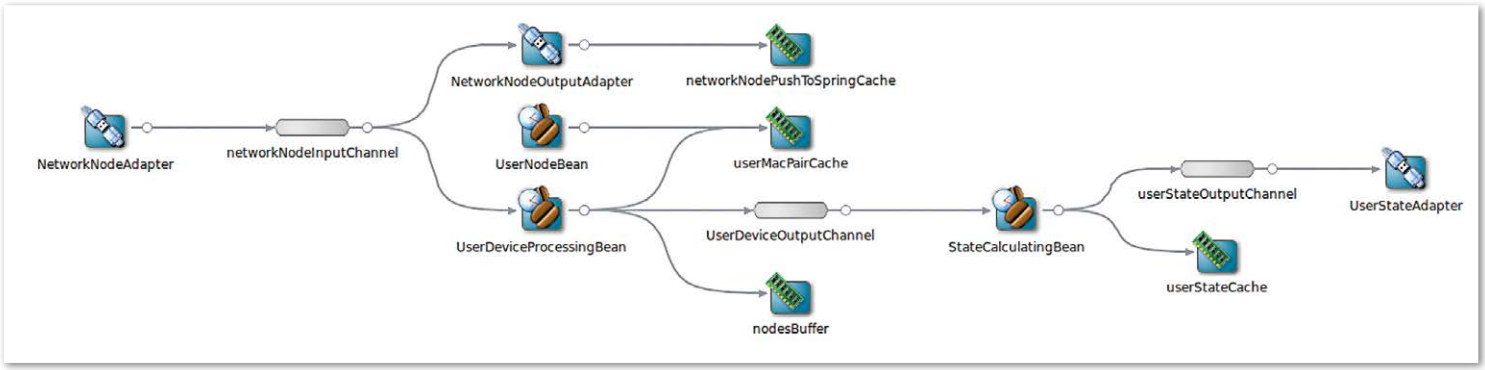


Abbildung 4: Event-Netzwerk

Minuten geändert? Wenn dies der Fall ist, wird ein Event an den Spring Pi geschickt („UserStateAdapter"). Ist der Zustand gleich geblieben, wird kein Event weitergegeben.

Nun ist klar, dass eine Benutzer-Präsenz-Prüfung auf Basis von IP-fähigen Geräten im Netzwerk heute meist noch Zukunftsmusik ist. Während die Präsenz der meisten 14-Jährigen wohl auf diese Weise ermittelt werden könnte, ist dies bei anderen Personengruppen nicht der Fall. Außerdem kann es vorkommen, dass ein Gerät im Flugmodus ist oder keine WLAN-Verbindung besitzt. Das System würde in diesen Fällen also irrtümlich unterstellen, der Benutzer sei nicht mehr anwesend, und dementsprechend Geräte ausschalten. Um diese Schwachstellen zu korrigieren, wird eine Mischung aus Geo-Fencing, Bewegungsmelder oder iBeacons eingesetzt. So kann man verschiedene Event-Typen kombinieren und mit höherer Wahrscheinlichkeit bestimmen, ob sich jemand im Gebäude befindet, um welche Person es sich handelt – und bis zu einem gewissen Grad an Genauigkeit sogar, an welchem Ort sie sich im Gebäude aufhält.

Diese Schwachstelle beschreibt ein noch häufig anzutreffendes Problem im IoT-Kontext: Physische Nähe-Relationen sind schwer in Erfahrung zu bringen. Während die Wissenschaft im Bereich „Mobile Wireless Sensor Network Localization" bereits einige Algorithmen zur Lokalisation von Netzwerk-Knoten ohne Nutzung von Energie-intensivem GPS bietet *[6]*, ist die Praxis noch weit von standardisierten Lokalisierungsmethoden entfernt.

Lessons Learned

Java-Entwickler können mit ihrem bestehenden Skillset auch im Embedded-Bereich bereits einiges leisten. Mit kleinen Abstechern in nativen Code sind auch mit Java direkte Hardware-Interaktionen möglich. Plattformen wie der Raspberry Pi bieten außerdem eine mächtige Grundlage, um komplexere Intelligenz ohne teure Hardware und mit geringem Energie-Aufwand näher an die Grenze zwischen IT und Realität zu bringen. Mithilfe des Event Processing konnte eine architektonische Taktik verfolgt werden, mit der sich aus den massiven Datenflüssen in Richtung „Backend" die wichtigsten Informationen herausfiltern und somit die Last für das Backend reduzieren lassen.

Neben diesen positiven Entwicklungen zeigen sich aber auch noch einige Probleme: Geräte im IoT haben häufig Leistungs-, Kommunikations-, oder Energie-Restriktionen. Diese Faktoren muss der Entwickler mit bedenken und durch entsprechende Maßnahmen kompensieren.

Auch die Kommunikations-Protokolle und -Technologien sind sehr verschieden und sollten für jedes System gründlich evaluiert und verglichen werden. Allein der genannte, noch recht simple Prototyp enthält verschiedene Kommunikationstechnologien: RESTful APIs via HTTP-Requests und WiFi, 433-MHz-Funkkommunikation, MPD, ein spezielles auf TCP aufbauendes Protokoll und ein proprietäres Protokoll, von Philips übertragen über „Zigbee". Viele dieser Technologien bieten aber auch Vorteile gegenüber schwergewichtigen Alternativen. Ein Blick beispielsweise auf CoAP 6LoWPAN und IEEE 802.15.4 zeigt dies.

Fazit

Die Kostenfrage der Heim-Automatisierung ist im Grunde schon beantwortet. Die notwendige Hardware ist in ihrer elementaren Form günstig und leistungsstark. Fertige, bereits am Markt erhältliche Anwendungen sind hingegen meist proprietär und teuer. Es fehlt noch an zufriedenstellenden Software-Systemen, um die vielen verschiedenen Geräte-Typen zu integrieren und für Nicht-Software-Entwickler benutzbar zu machen.

Open-Source-Projekte können dabei helfen, einige gute Projekte herauszustellen, die diverse Plattformen integrieren und so eine einheitliche Schnittstelle bieten. Die dafür hilfreichen Frameworks und Engines sind bereits in Form von Rule Engines und Prozess-Frameworks verfügbar. Die Kabel der verbreiteten BUS-Systeme sind jedoch definitiv Funkverbindungen gewichen. Hier gibt es allerdings wieder viele Standards und Protokolle, die es zu integrieren gilt.

Weitere Informationen

[1] https://projects.drogon.net/raspberry-pi/wiringpi
[2] https://github.com/r10r/rcswitch-pi
[3] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/design.html#wp16696
[4] http://www.woutervanwijk.nl/pimusicbox
[5] http://mpd.wikia.com/wiki/Music_Player_Daemon_Wiki
[6] http://doi.acm.org/10.1145/2677855.2677858 und http://dx.doi.org/10.1007/s00779-013-0692-9

Pascal Brokmeier
pascal.brokmeier@opitz-consulting.com

Pascal Brokmeier arbeitet seit vier Jahren als Software-Entwickler bei der OPITZ CONSULTING Deutschland GmbH, daneben studiert er Wirtschaftsinformatik an der Universität zu Köln. Über mobile Applikationen und Java-EE-Anwendungen hat er sich in den letzten zwei Jahren auf das Internet der Dinge spezialisiert und analysiert hierzu aufkommende Architekturen sowie technische und wirtschaftliche Entwicklungen. Über seine Erkenntnisse hält er regelmäßig Vorträge auf Konferenzen.

http://ja.ijug.eu/15/3/15

Verarbeitung bei Eintreffen: Zeitnahe Verarbeitung von Events

Tobias Unger, Yenlo Deutschland GmbH

„Die maximale Anzahl der Anmeldeversuche wurde erreicht." So einfach und klar die Meldung ist, so kompliziert kann es sein, die Anzahl der Anmeldeversuche zu ermitteln. Anmeldeversuche können von verschiedenen Quellen wie Portalen oder mobilen Anwendungen (Apps) initiiert werden und dennoch soll die Berechnung möglichst zeitnah („Realtime") erfolgen. Idealerweise ist die Berechnung vor dem nächsten Anmeldeversuch abgeschlossen und der Account somit schon gesperrt. Complex Event Processing (CEP) stellt einen Weg dar, diese Events zeitnah zu verarbeiten. Dieser Artikel erläutert CEP auf Basis von Siddhi als CEP-Prozessor und WSO2-CEP als CEP-Server.

Menschen sind von Events umgeben. Das Handy vibriert aufgrund einer eintreffenden Nachricht, ein Warnton signalisiert uns das Schließen der Türen einer U-Bahn oder das Auto warnt uns mit unzähligen Warnleuchten vor Benzinmangel, Minusgraden oder nicht angeschnallten Beifahrern. Eingebettete Systeme wie in Autos arbeiten schon lange Zeit Event-orientiert. Die Quellen für Events sind hier meist Sensoren. Der Abstandssensor übermittelt in regelmäßigen Intervallen den Abstand zum vorausfahrenden Wagen. Durch Aggregation dieser Events erkennt das Auto, ob sich der Abstand vergrößert oder verringert. Verringert sich der Abstand unter einen bestimmten Wert, kann der Bremseinheit signalisiert werden, den Wagen zu bremsen, bis der Abstand einen gewissen Wert wieder überschreitet. Durch Aggregation der Events kann auch bestimmt werden, wie schnell sich der Abstand verringert, und somit die Stärke der Bremsung bis hin zur Notbremsung ermittelt werden.

Genau hier setzt CEP an. Es ermöglicht, einzelne Events oder Ströme von Events kontinuierlich zu aggregieren und daraus höherwertige Events zu erzeugen. In obigem Beispiel werden aus den einfachen Abstands-Events höherwertige Events wie „Wagen bremsen" oder „Wagen beschleunigen" erzeugt. Dazu versucht CEP, Muster, Beziehungen und Daten-Abstraktionen zwischen den Events zu ermitteln und unmittelbar durch das Senden eines Events darauf zu reagieren.

Dennoch sind CEP-Systeme aktuell noch nicht weit verbreitet; besonders im Umfeld von Unternehmensanwendungen wie CRM, ERP, HR und bei der Integration der genannten Systeme kommt CEP noch nicht oft zum Einsatz. Anwendungsfälle, die eine zeitnahe Reaktion erfordern, werden meist als Batches realisiert, die wiederholend in kurzen Abständen ausgeführt werden. Mit Siddhi *[1]* und dem WSO2 Complex Event Processor *[2]* besteht hier die Möglichkeit, CEP als zentrale Komponente der IT-Infrastruktur zu etablieren.

Siddhi und WSO2-CEP stehen unter Apache-2-Lizenz, sind also Open Source. Siddhi ist eine in Java geschriebene CEP-Engine, die in eigene Anwendungen direkt eingebettet werden kann. WSO2-CEP erweitert Siddhi um Management-Funktionen und stellt eine Menge von Adaptoren bereit, um Events über verschiedene Protokolle und in verschiedenen Formaten empfangen zu können.

Event-Driven Architecture und Big Data als Treiber

Durch zunehmende Integration von Systemen wird auch in klassischen Anwendungsfeldern wie im Order-Management auf Event-Orientierung gesetzt. Anstatt beim Eingang einer Bestellung jedes einzelne System, wie das Inventory Management oder das Payment, einzeln aufzurufen, über die Bestellung zu informieren und unter Umständen blockierend auf eine Antwort zu warten, publiziert das Order-Management schlicht einen Event, auf den sich die anderen Systeme registrieren können.

Diese Systeme können selbst wiederum Events publizieren. Das führt zu einer losen Kopplung zwischen den Systemen, da ein System nur noch wissen muss, für welche Events es sich interessiert – unabhängig von der Tatsache, durch welches System diese publiziert wurden. Auch muss das System die Interessenten für eigene Events nicht kennen, es muss die Events einfach publizieren. Der dieser Idee von Systemdesign folgende Architekturstil wird auch Event-driven Architecture (EDA) genannt. Deren besonderes Kennzeichen ist die mit dem Architekturstil inhärent verbundene Asynchronität. Eine genauere Beschreibung dieses Architekturstils ist unter *[3]* zu finden.

Im Rahmen einer EDA kommt CEP die Aufgabe zu, Zusammenhänge zwischen den Events sichtbar zu machen. Erfolgen beispielsweise viele teure Bestellungen mit der gleichen Kreditkarte innerhalb weniger Minuten, kann dies selbst ein Event von Interesse sein, da die Möglichkeit besteht, dass hier mit gestohlenen Kreditkartennummern bestellt wird.

Hier verspricht CEP im Gegensatz zu anderen Lösungen *[4]* die Möglichkeit, Tausende Events in nahezu Echtzeit verarbeiten zu können, weshalb CEP immer wieder in Kombination mit dem Schlagwort „Big Data" genannt wird. Im Gegensatz zu bestehenden Lösungen zum Verarbeiten von Event-Strömen wie Apache Storm bieten CEP-Systeme noch den weiteren Vorteil der deklarativen Anfrage-Sprache.

Der WSO2-CEP-Server

Auch wenn Siddhi in Java implementiert ist, spielt das für den Anwender des WSO2-

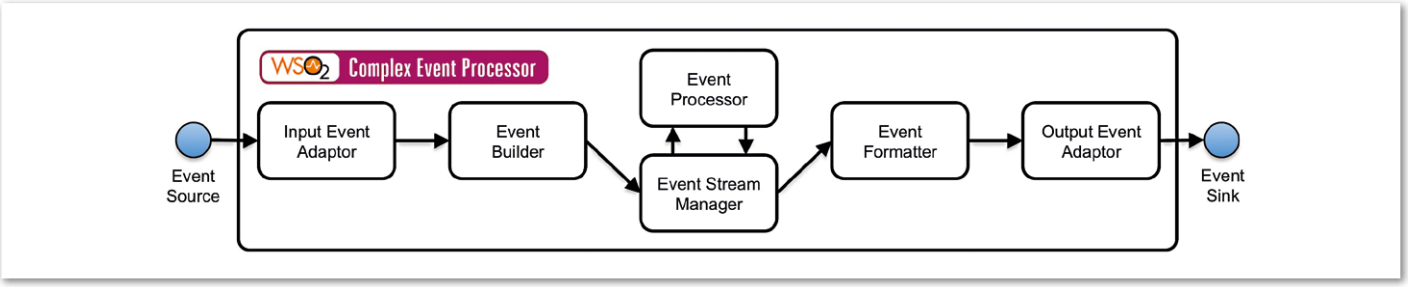


Abbildung 1: Architektur des WSO2-CEP-Servers (Quelle: WSO2)

CEP-Servers eine untergeordnete Rolle, solange die Engine selbst nicht erweitert werden soll. *Abbildung 1* zeigt seine Architektur und den prinzipiellen Weg der Events durch den Server.

Die Events werden vom „Input Event Adaptor" empfangen. Adaptoren stehen für die meisten der gängigsten Protokolle wie HTTP, SOAP, JMS oder E-Mail bereit. Für die Integration mit anderen WSO2-Produkten steht ein spezieller „WSO2 Event Adaptor" zur Verfügung, der Events über Thrift empfängt. Eine komplette Liste der existierenden Adaptoren ist in der Dokumentation zu finden *[5]*. Darüber hinaus können für andere Protokolle spezifische Adaptoren selbst entwickelt werden.

Der „Input Event Adaptor" reicht die Nachrichten an einen „Event Builder" weiter. Dessen Aufgabe besteht darin, die eingehenden Events in das CEP-Server-eigene Event-Format zu übersetzen. Als Format für die Events werden standardmäßig WSO2Event, XML, JSON, MAP und Text unterstützt. Event Builder unterstützen auch einfache Mapping-Operationen. Für XML wird dafür XPATH eingesetzt und für JSON JSONPath.

Nach Verlassen des Event Builder erreicht der Event den „Event Stream Manager". Dieser ist zuständig für die Verwaltung aller Event Streams und stellt den zentralen Hub für alle Events dar. Hier werden auch die Definitionen der Event Streams verwaltet. Diese sind immer im WSO2-eigenen Format spezifiziert *[6]*. Auf Basis dieser Definitionen werden Events an die „Processing Engine" weitergeleitet. Event Streams können auch mit „In-Flows" und „Out-Flows" assoziiert werden, um Events mithilfe verschiedener Protokolle und Formate empfangen und publizieren zu können.

Der „Event Processor" ist zuständig, um die Querys auszuführen, deren Definition später noch zu sehen ist. Im Falle des WSO2-CEP-Servers sind die Querys Teil eines „Execution-Plans". Dabei wird für jede „Query" eine „Siddhi Engine" gestartet. Hier ist gut der Zusammenhang zwischen WSO2-CEP-Server und Siddhi zu erkennen: Siddhi ist die CEP-Engine, die der WSO2-CEP-Server benutzt.

Ausgehende Events reicht der Event Stream Manager an einen „Event Formatter" weiter. Dieser kann das interne WSO2-Event-Format in andere Formate wie XML, JSON oder Text transformieren und auch einfach Mappings analog zum „Event Builder" durchführen.

Final ist es die Aufgabe des „Output Event Adaptor", die ausgehenden Events über verschiedene Protokolle zu versenden. Standardmäßig werden hier dieselben Protokolle wie auch beim „Input Event Adaptor" unterstützt. Auch hier ist es möglich, andere Protokolle durch Implementierung eines spezifischen Protokoll-Adaptors zu verwenden.

Querys schreiben

Nachdem der Weg der Events durch den Server betrachtet wurde, folgt nun der wichtigste Teil, das Schreiben von „Querys". Ein wichtiges Element dabei ist die „Event Definition". Diese spezifiziert, welche Attribute ein Event besitzt. Leider ist der Begriff „Event Definition" innerhalb des WSO2-CEP doppelt belegt. Im vorigen Abschnitt wurde die „Event Stream Definition" von WSO besprochen. Die hier vorgestellte „Event Definition" wird von Siddhi intern verwendet. Benutzer des WSO2-CEP brauchen sich darüber keine Gedanken zu machen, da die Definitionen hier automatisch synchronisiert werden. Dennoch soll sie hier der Vollständigkeit halber vorgestellt werden. *Listing 1* zeigt eine Siddhi Event-Definition. Innerhalb von Siddhi wird jeder Stream über einen Namen identifiziert.

Siddhi-Querys weisen eine Ähnlichkeit zu SQL-Querys auf. Hier werden mittels „select" allerdings keine Tabellen, sondern die Streams selektiert, aus denen gelesen werden soll. Mittels „insert into" werden Events in Streams geschrieben.

Siddhi unterstützt mehrere Operatoren, um Events zu selektieren. Über einen Filter

```
define stream AccessLogEntry(clientip string, request string, ...)
```

Listing 1

```
select from AccessLogEntry[clientip == '127.0.0.1' and verb == 'GET']
insert into LocalAcces clientip, verb, request;
```

Listing 2

```
from AccessLogEntry[response == '401']#window.time( 1 min )
select clientip, request, count(response) as requestNumber
group by clientip, request
insert into Unauthorized for all-events;
```

Listing 3

```
from TickEvent[symbol=='IBM']#window.length(2000) as t unidirectional
join NewsEvent#window.time(500) as n
on t.symbol == n.company
insert into JoinedStream
```

Listing 4

kann eine Bedingung für einen Attributwert definiert werden. Dabei stehen „=", „<" und „>" als Vergleichsoperatoren zur Verfügung. Durch Verknüpfung einzelner Bedingungen mittels der Operatoren „and", „or" und „not" können komplexe Bedingungen erzeugt werden. *Listing 2* zeigt die Selektion nach einer IP-Adresse und des HTTP-Verbs.

Ströme von Events können theoretisch unendlich viele Events enthalten. Um eine Teilmenge dieser Events zu selektieren, bietet Siddhi die Möglichkeit, sogenannte „Windows" zu bilden. Siddhi unterstützt dabei Windows, die auf Basis der Anzahl der Events (etwa die letzten 100 Events), der Zeit (etwa alle Events innerhalb der letzten Minute) oder der Attribut-Werte (etwa alle Events mit gleicher IP-Adresse) definiert werden.

Eine genaue Beschreibung der Windows findet sich in der Dokumentation. Anzumerken ist hier, dass temporale Windows als Basis zur Berechnung des Windows den Zeitstempel verwenden, an dem der Event von Siddhi empfangen wurde. Wird eine Berechnung des Windows auf Basis des Zeitstempels bei Erstellung des Events benötigt, kann in neueren Siddhi-Versionen auf das „External Time Window" zurückgegriffen werden.

Das Ergebnis eines Windows ist eine Menge von Events, auf die weitere Operatoren angewendet werden können. Die Ausgabe der Events kann entweder erfolgen, sobald alle Events zusammengefasst sind („Batch Window") oder immer wenn ein neuer Event eintrifft („Sliding Window"). *Listing 3* zeigt ein Beispiel, bei dem ein Sliding Window erzeugt wird, das alle Events sammelt, die innerhalb einer Minute eintreffen.

Darüber hinaus besteht die Möglichkeit, mehrere Streams durch einen „Join"-Operator zu kombinieren. Standardmäßig wird dieser ausgeführt, sobald auf einem der Ströme ein Event ankommt. Ist dies nicht gewünscht, kann mithilfe des Schlüsselworts „unidirectional" ein Stream definiert werden, der den „Join" bei Eintreffen eines Events auslöst. Ein „Join" wird dabei als „INNER JOIN" ausgeführt. Zu beachten ist, dass mindestens ein Strom ein Window definiert haben muss. *Listing 4* zeigt einen Join, der ausgeführt wird, sobald ein „TickEvent" eintrifft.

```
from every a1 = PizzaOder
  -> a2 = PizzaOder[custid=a1.custid]
insert into ReturningCustomers
  a1.custid as custid a2.ts as ts
```

Listing 5

```
define partition oderParition by PizzaOrder.id, PizzaDone.oid, PizzaDeliv-
ered.oid
select from PizzaOder as o ->PizzaDone as p -> PizzaDelivered as d
insert into OrderTimes (p.ts-o.ts) as time2Preprae, (d.ts-p.ts) as time2De-
livery
  partition by oderParition
```

Listing 6

```
// Create Siddhi Manager
SiddhiManager siddhiManager = new SiddhiManager();
siddhiManager.defineStream("define stream cseEventStream ( symbol string,
price float, volume int )");
siddhiManager.addQuery("from  cseEventStream [ price >= 50 ] " +
  "select symbol, price " +
  "insert into StockQuote ;");
siddhiManager.addCallback("StockQuote", new StreamCallback() {
        @Override
        public void receive(Event[] events) {
           EventPrinter.print(events);
        }
     });
InputHandler inputHandler = siddhiManager.getInputHandler("cseEventStream");
inputHandler.send(new Object[]{"IBM", 75.6f, 100});
Thread.sleep(500);
siddhiManager.shutdown();
```

Listing 7

Siddhi verfügt auch über die Möglichkeit, Muster von nacheinander auftretenden Events zu prüfen. Die dazu verwendeten Operatoren heißen „Pattern" und „Sequence". Sie unterscheiden sich darin, dass „Pattern" das Unterbrechen des Musters durch andere Events erlaubt, während das Muster bei „Sequence" exakt auftreten muss. *Listing 5* zeigt ein „Pattern", das es ermöglicht, wiederkehrende Kunden zu erkennen.

Um Querys zu beschleunigen oder zu verteilen, bietet Siddhi die Möglichkeit, „Partitionen" zu bilden. Dabei werden die Events, basierend auf einer Bedingung, in Partitionen eingeteilt. *Listing 6* zeigt ein Beispiel, bei dem die Bestellungen nach ihrer ID partitioniert werden, um anschließend zu berechnen, wie viel Zeit zwischen Bestellung und Auslieferung vergeht.

Das Bilden von Partition ergibt Sinn, falls sich ein Query innerhalb einer Partition schneller berechnen lässt, als es auf den ganzen Strom von Events anzuwenden. Dies ist bei dem Query in *Listing 6* der Fall. Hier erzielt die Partitionierung noch den Effekt, dass das Query einfacher zu designen ist, da es keine Events aus anderen Bestellungen berücksichtigen muss. Partitionen sind des Weiteren ein Mittel zur Skalierung, da Partitionen sich auf verschiedene Knoten verteilen lassen.

Zu guter Letzt bieten „Event Tables" die Möglichkeit, Events in einer Tabelle zwischenzuspeichern. Events können in eine Event Table eingefügt, daraus gelöscht und durch eine Join-Operation mit eintreffenden Events kombiniert werden. Tabellen sind immer dann sinnvoll, falls Events über längere Zeit zur Verarbeitung vorgehalten werden sollen (etwa Tage oder Wochen) und dadurch die im Speicher gehaltenen „Windows" an ihre Grenzen stoßen. Architektonisch sollte der Einsatz von „Event Tables" gut durchdacht sein, da hier je nach Szenario eine Batch-Verarbeitung besser geeignet ist.

Siddhi ohne WSO2-CEP

Siddhi lässt sich auch außerhalb des WSO2-CEP-Servers betreiben. *Listing 7* zeigt, wie Siddhi innerhalb eines Java-Programms instanziiert werden kann. Die zentrale Komponente stellt der „SiddhiManager" dar. Dieser ermöglicht es, Streams und Querys zu definieren sowie Events zu senden und zu empfangen.

Ein Beispiel

Wer Lust hat, Siddhi und WSO2-CEP auszuprobieren, dem steht auf GitHub *[7]* das

in der Einleitung genannte Access-Log-Beispiel in drei Varianten zur Verfügung. Die ersten zwei Varianten benutzen WSO2-CEP, die dritte verwendet Siddhi direkt innerhalb eines Java-Programms. Die beiden WSO2-CEP-Beispiele unterscheiden sich darin, dass im ersten Fall die grafische Konfigurations-Oberfläche genutzt wird und im zweiten Fall XML-Konfigurationsdateien erzeugt werden, die direkt auf dem Server eingesetzt werden können.

Fazit

Siddhi ermöglicht den Einstieg in CEP und somit in die zeitnahe Verarbeitung von Events. Siddhi kann dabei als Teil des WSO2 Complex Event Processor genutzt werden oder direkt in eigene Applikationen eingebunden sein. Welche der zwei Varianten letztlich benutzt wird, hängt von den jeweiligen Anforderungen des Projektes ab. Als CEP-Schnellstart eignet sich der WSO2-CEP besser, da er dank grafischer Administrationsoberflächen eine deutlich geringere Einstiegshürde bietet.

Referenzen und Quellen

[1] https://github.com/wso2/siddhi
[2] http://wso2.com/products/complex-event-processor/
[3] http://www.eaipatterns.com/docs/EDA.pdf
[4] http://srinathsview.blogspot.ch/2014/12/real-time-analytics-with-big-data-what.html
[5] https://docs.wso2.com/display/CEP310/WSO2+Complex+Event+Processor+Documentation
[6] http://wso2.com/products/complex-event-processor
[7] https://github.com/ungerts/siddhi-cep-example

Tobias Unger
tobias.unger@yenlo.com

Tobias Unger (Dipl.-Inf.) verfügt über mehr als 10 Jahre Erfahrung in den Bereichen Enterprise Architecture, BPM und Java Enterprise. Sein aktueller Fokus liegt auf Design und Implementierung von Integrationsplattformen auf Basis von WSO2-Middleware. Er ist Autor zahlreicher Publikationen und als Sprecher auf Konferenzen aktiv.

http://ja.ijug.eu/15/3/16

Unbekannte Kostbarkeiten des

Heute: Dateisystem-Überwachung

Bernd Müller, Ostfalia

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

Einige Systeme und Werkzeuge beobachten das Dateisystem, um bei Änderungen bestimmte Aktionen auszuführen. Seit Java 7 ist eine derartige Möglichkeit zur Überwachung im SDK eingebaut und es ist sehr einfach, sie zu verwenden.

Motivation

Der WildFly-Application-Server beobachtet ein bestimmtes Verzeichnis, um beim Erzeugen, Verändern oder Löschen einer Datei mit einer der Endungen .jar/.war/.ear ein Deployment, Redeployment oder Undeployment durchzuführen. Andere Systeme oder Werkzeuge besitzen ähnliche Funktionalitäten, um beispielsweise den Überlauf eines Dateisystems zu verhindern beziehungsweise frühzeitig Alarm zu schlagen. Seit Java 7 enthält das SDK

Interfaces und entsprechende Implementierungen, die es Entwicklern erlauben, derartige Funktionalitäten sehr einfach zu nutzen. Grundlage dafür sind die Interfaces „WatchService" und „WatchKey" im Package „java.nio.file".

„WatchService" und „WatchKey"

Ein „WatchService" beobachtet Änderungen an Dateisystem-Objekten, die zuvor registriert wurden. Änderungs-Events werden indirekt durch einen „WatchKey" repräsentiert. Indirekt deshalb, weil ein derartiger „WatchKey" einen Zustand besitzt, der abgefragt werden kann; es gibt jedoch keine Events und Event-Listener, wie es sonst in Java üblich ist. Ein Grund hierfür ist, dass Dateisystem-Events eventuell in einer höheren Frequenz entstehen, als sie abgearbeitet werden können. Man hat sich daher für einen Abfragemodus entschieden, der es zum einen erlaubt, Events aufzureihen, aber auch, Dateisystem-Events einfach „zu vergessen".

Die Verwendung

Nachdem eine „WatchService"-Instanz erzeugt wurde, wird sie für ein bestimmtes Verzeichnis registriert. Bei der Registrierung sind die interessierenden Events anzugeben: „ENTRY_CREATE", „ENTRY_DELETE" und „ENTRY_MODIFY", Konstanten der Klasse „WatchEvent.Kind<Path>", um auf Dateisystem-Events zu registrieren, oder „OVERFLOW", eine Konstante der Klasse „WatchEvent.Kind<Object>", um auf verlorene beziehungsweise verworfene Events zu registrieren.

Mit den Methoden „poll()" oder „take()" lassen sich „WatchKeys" erfragen. Die zweite Methode ist blockierend, wartet also auf das nächste Event. Nachdem ein „WatchKey" erfragt wurde, folgt die eigentliche Verarbeitung des Events. Im einführenden Beispiel geht es um das Deployment einer EE-Anwendung oder die Reaktion auf zur Neige gehenden Speicherplatz. Nach der Verarbeitung wird der „WatchKey" zurückgesetzt. *Listing 1* zeigt das beschriebene Verfahren exemplarisch.

Die Verarbeitung des Events besteht im Beispiel aus der Ausgabe der Event-Art (Methode „kind()"), also einer der oben genannten Konstanten, und der Ausgabe des Namens der Datei (Methode „context()"), die sich entsprechend verändert hat. Man kann sich sicher leicht vorstellen, wie man hier seine eigenen Anforderungen realisiert. Ein umfangreicheres Beispiel, bei dem rekursiv ein ganzer Verzeichnisbaum beobachtet wird, steht im Java-Tutorial *[1]*.

Plattform-Abhängigkeiten

Nach dem Javadoc des Interface „WatchService" soll eine Implementierung die nativen Möglichkeiten des zugrunde liegenden Betriebssystems für die Überwachung des Dateisystems verwenden und nur, falls das Betriebssystem eine derartige Funktionalität nicht anbietet, Polling oder andere Alternativen als Fall-Back-Lösung verwenden. Die im OpenJDK und Oracle-SDK verwendete Implementierung für Linux, die Klasse „sun.nio.fs.LinuxWatchService", realisiert ganz offensichtlich die direkte Betriebssystem-Verwendung, da auf dem Fedora-Rechner des Autors keinerlei Rechenaufwände des Prozesses feststellbar sind, selbst wenn zum Beispiel das Log-Verzeichnis des WildFly-Application-Servers überwacht wird.

Fazit

Seit Java 7 gibt es im SDK die Möglichkeit, Änderungen im Dateisystem, also das Neuanlegen, Löschen oder Ändern von Dateien, zu überwachen und beim Eintreten eines solchen Ereignisses mit einem beliebigen Methodenaufruf darauf zu reagieren. Das API ist äußerst schlank und sehr einfach zu verwenden.

Literatur

[1] Watching a Directory for Changes: http://docs.oracle.com/javase/tutorial/essential/io/notification.html

Bernd Müller
bernd.mueller@ostfalia.de

Bernd Müller ist Professor für Software-Technik an der Ostfalia (Hochschule Braunschweig/Wolfenbüttel). Er ist Autor mehrere Java-EE-Bücher, Sprecher auf nationalen und internationalen Konferenzen und engagiert sich in der JUG Ostfalen sowie im iJUG.

http://ja.ijug.eu/15/3/17

```
WatchService watcher = FileSystems.getDefault().newWatchService();
Path path = Paths.get("."); // zu beobachtendes Verzeichnis
path.register(watcher, ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);
for (;;) {
   WatchKey key = watcher.take();
   for (WatchEvent<?> event : key.pollEvents()) {
       System.out.println("Name: " + event.kind());
       System.out.println("Path: " + event.context());
   }
   key.reset();
}
```

Listing 1

Alle unbekannten Kostbarkeiten

Bernd Müller veröffentlicht seit Jahren die "Unbekannten Kostbarkeiten des SDK" in der Java aktuell. Die früheren Beiträge sind auf seiner Website zu finden *(siehe „http://www.pdbm.de/person/publikationen.html#Artikel+in+Zeitschriften+und+Tagungsb%C3%A4nden")*.

„Ich finde es großartig, wie sich die Community organisiert …“

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

Wie ist die Java User Group Dortmund organisiert?
Hendrik: Aktuell gibt es noch keinen Verein oder ähnliches. Die letzten zweieinhalb Jahre haben Ansgar und ich die Termine organisiert und in Dortmunder Unternehmen freiwillige Sponsoren gefunden. Seit etwa einem Jahr sind wir Mitglied bei der iJUG. Da wir aber vor allem in den letzten eineinhalb Jahren stark gewachsen sind, das Programm stark erweitert haben und auch in Zukunft noch weitere zusätzliche Events neben unseren Vortragsabenden organisieren wollen, planen wir aktuell die Gründung eines Vereins.
Ansgar: Wir sind im Moment beide in einer Phase, in der wir beruflich sehr eingespannt sind, und merken, dass wir mit der Organisation der JUG an unsere Grenzen stoßen. Daher steht oben auf unserer Tagesordnung neben der formellen Gründung eines Vereins auch die Erweiterung des Organisations-Teams. Dazu haben wir aber erste Schritte in die Wege geleitet und denken, dass wir Anfang 2015 eine gute Lösung für die JUG finden werden.

Was sind die Ziele der Java User Group Dortmund?
Hendrik: Initial war sicherlich das einzige Ziel, andere Java-Entwickler in Dortmund kennenzulernen und eine lokale Community zu gründen. Nachdem wir aber im Jahr 2014 neben den monatlichen Vorträgen auch weitere Events wie einen Workshop und eine Mini-Konferenz mit sechs Vorträgen anbieten konnten, sind die Ziele auch gewachsen. Neben der Community-Arbeit ist es uns wichtig, Wissen zu vermitteln und Java-Entwickler durch die kostenlosen Vorträge und Workshops zur Weiterbildung zu animieren. Ab dem Jahr 2015 wird dann noch ein neues Ziel die Mitgestaltung von Java- und Open-Source-Software sein. Wie das Ganze genau aussehen soll, ist aktuell noch in der Diskussion, aber es gibt durch JSR-Mitarbeit oder Hackergarten-Termine ja genug Möglichkeiten für User Groups, sich aktiv einzubringen.
Ansgar: Für mich steht nach wie vor der lokale beziehungsweise regionale Gedanke im Vordergrund. Es geht darum, Java-Interessierte im Raum Dortmund zu vernetzen und eine Möglichkeit zu bieten, sich regelmäßig zum Thema auszutauschen und durch die Vorträge neue Impulse aufzunehmen.

Das Jahr 2015 wird sicherlich super interessant und spannend. Hendrik hat ja schon ein paar von unseren Ideen angesprochen. In unseren Köpfen geistern aber auch noch andere Ideen rum. Wir haben zum Beispiel noch keine vernünftige Plattform gefunden, mit der wir die JUG gut und einfach organisieren können. Hier könnten wir uns durchaus ein längeres Projekt vorstellen.

Wie viele Veranstaltungen gibt es pro Jahr?
Hendrik: Unser Ziel ist es, pro Monat einen Termin anzubieten. Anfangs gab es immer einen Vortrag und einen Stammtisch-Termin im Wechsel. Seit dem Jahr 2013 haben wir es dann aber bis auf einige Ausnahmen geschafft, jeden Monat einen Vortrag anzubieten. Darüber hinaus hatten wir in den Jahren 2013 und 2014 jeweils ein Special Event. So haben wir letztes Jahr im Sommer eine Mini-Konferenz mit mehreren Vorträgen und anschließender Grillparty organisiert. Dank unserer Hauptsponsoren adesso AG und TynTec sogar komplett kostenlos für die Teilnehmer.
Ansgar: Die Stammtische waren in der Gründungsphase wichtige Treffen, auf denen wir vieles planen und besprechen konnten. Als wir es dann geschafft haben, regelmäßiger Vorträge zu organisieren, merkten wir, dass die Stammtische eine eher kleine Gruppe angesprochen haben. Wir organisierten daraufhin zugunsten der Vorträge keine Stammtische mehr.

Was bedeutet Java für euch?
Hendrik: Ich habe Java schon in der Schule gelernt und bin nach dem Studium dann auch direkt als Java-Entwickler in das Berufsleben gestartet. Durch das Schreiben von Artikeln, Büchern und der Organisation der JUG nimmt Java sogar einen Teil meiner Freizeit ein. Ich denke, das wäre nicht möglich, wenn ich nicht voll hinter der Programmiersprache stehen würde. Ich persönlich könnte mir momentan auf jeden Fall keinen anderen Beruf und kein anderes Hobby vorstellen.
Ansgar: In der Schule wurde bei uns noch Turbo Pascal gelehrt; später an der Uni war Java dann die Lehrsprache. Obwohl Java sowohl im Beruf als auch in der Freizeit die Sprache meiner Wahl ist, versuche ich auch immer wieder über den Tellerrand zu gucken. In den letzten zwei Jahren kamen zum Beispiel immer wieder Frontends mit Javascript und viel AngularJS hinzu. Java ist aber in den meisten meiner Projekte die Basis und stellt für mich trotz der Vielfalt an neuen JVM-Sprachen immer noch eine sehr gute Lösung dar.

Welchen Stellenwert besitzt die Java-Community für euch?
Hendrik: Für mich hat die Community einen sehr hohen Stellenwert. Das Ganze hört

bei den User Groups aber nicht auf. Durch den JCP kann die Community sich aktiv an der Weiterentwicklung von Java beteiligen. Hierdurch entstehen viele Innovationen. Als Beispiel möchte ich die diesjährigen „Duke Choice Award"-Gewinner JavaFX-Ports und DukeScript anführen. JavaFX-Ports ist der Port von JavaFX auf Android und DukeScript ist eine auf JavaScript basierte JVM die das Ausführen von Java Anwendungen im Browser ermöglicht. Beides sind Community-getriebene Open-Source-Projekte mit sehr hohem Innovationsgrad und Erfolg.

Ansgar: Ich finde es großartig, wie sich die Community organisiert und wie mit dem JavaLand ein Event geschaffen wurde, der es den JUGs und der Community ermöglicht, sich einmal im Jahr auch persönlich zu treffen. Ich finde es wichtig, dass die Kontakte zu anderen JUGs aufgebaut und erhalten werden können. In NRW sind wir da sicherlich ganz gut dran mit fünf großen JUGs und einem relativ dichten Autobahnnetz.

Wie sollte sich Java weiterentwickeln?

Hendrik: Grundsätzlich finde ich, dass Oracle hier mittlerweile einen sehr guten Job macht. Auf Konferenzen wie der JavaOne kann man sehen, dass die Community Oracle mittlerweile vertraut und auch dessen Marschrichtung von Java unterstützt. Ich kenne einige der Java-Entwickler von Oracle persönlich und bin der Meinung, dass hier sehr fähige Leute am Werk sind, die Notwendigkeiten für neue Sprach-Features und APIs erkennen und diese (vielleicht nicht immer für das gewünschte Release) professionell umsetzen. Durch JSRs und den JCP wird die Weiterentwicklung von Java aber auch nicht rein durch Oracle gelenkt, sondern ist zum Teil auch ein Ergebnis der Community. Sollte man trotzdem mit einer geplanten Entwicklung mal nicht zufrieden sein, gibt es immer noch die Möglichkeit, seine Meinung und Ideen in Mailing-Lists oder durch Patches (OpenJDK) kundzutun.

Ansgar: Ich würde mich freuen, wenn die Entwicklungen an der Plattform zügiger erfolgen. Oft werden Features erst in anderen Projekten entwickelt und dann in die Plattform übernommen. Dadurch hinkt die Sprache oft hinterher und Frameworks wie Spring sind weiterhin notwendig, um aktuelle Entwicklungen nutzen zu können. Vieles von dem, was später Standard wird, musste vorher woanders ausprobiert werden.

Wie sollte Oracle eurer Meinung nach mit Java umgehen?

Hendrik: Grundsätzlich machen sie schon vieles richtig. Ich persönlich würde noch etwas an der Zugänglichkeit des OpenJDK arbeiten. Es wär ein Traum, wenn die offenen Java-Sourcen bei GitHub liegen und die Jira-Versionen mal auf den neuesten Stand gebracht würden.

Ansgar: Da kann ich Hendrik nur zustimmen. Eine weitere Öffnung wäre sicherlich von Vorteil. Ich denke auch, die Prozesse der Mitgestaltung könnten schlanker gestaltet werden. Es ist im Moment doch noch recht kompliziert, in den JCP-Prozess einzusteigen, auch wenn vieles davon sicherlich den rechtlichen Problemen geschuldet ist.

Wie sollte sich die Community gegenüber Oracle verhalten?

Hendrik: Offenheit und Unterstützung sind hier sicherlich die beiden wichtigsten Punkte. Nur durch aktive Mitarbeit kann die Community erreichen, dass Java und die JVM auch in Zukunft das moderne und innovative Produkt bleiben.

Ansgar: Ich glaube bei der breiten Nutzung von Java ist es wichtig, dass Oracle stark mit der Community verbunden ist, wenn diese zum Erfolg von Java beitragen soll. Es ist wichtig, dies auch immer wieder deutlich zu machen und auch von Oracle einzufordern.

http://ja.ijug.eu/15/3/18

Ansgar Brauner
a@jugdo.de
http://www.jugdo.de

Zur Person: Ansgar Brauner

Ansgar arbeitet bei REWE Digital in Köln und entwickelt dort E-Commerce-Lösungen für die REWE Group. Er fing auf einem alten Commodore Plus/4 an zu programmieren und ließ im Jahr 2012 mit Hendrik Ebbers die JUG Dortmund wieder aufleben. Neben Java und AngularJS freut er sich über die Möglichkeiten, die der DevOps-Gedanke Entwicklern eröffnet. Abseits vom Computer geht er gerade mit dem BVB durch die schwere Zeit im unteren Tabellendrittel und hofft auf eine starke Rückrunde.

Hendrik Ebbers
h@jugdo.de
http://www.jugdo.de

Zur Person: Hendrik Ebbers

Hendrik (@hendrikEbbers) ist Senior Java Architekt bei der Canoo Engineering AG und hat mehrere Jahre Erfahrung in der Entwicklung von Java-Anwendungen. Sein Haupt-Interesse liegt hierbei in den Bereichen Frontend, Middleware und DevOps. Hendrik leitet als aktives Mitglied der Java-Community die Java User Group Dortmund und spricht auf internationalen Konferenzen und User Groups. Zusätzlich veröffentlicht er Fachartikel in Print- und Online-Medien. Auf www.guigarage.com bloggt Hendrik regelmäßig über Architekturansätze im Bereich „JavaFX". Sein Buch „Mastering JavaFX 8 Controls" ist diesen Sommer bei Oracle Press erscheinen. Hendrik war einer der Featured Speaker auf der JavaOne 2014.

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://jugda.wordpress.com

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

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.

iJUG Verbund

www.ijug.eu

Impressum

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

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

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

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

Titel, Gestaltung und Satz:
Alexander Kermas,
DOAG Dienstleistungen GmbH

Anzeigen:
Simone Fischer
anzeigen@doag.org

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

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

Titelfoto: © destinacigdem / 123rf.com
Foto S. 16: © Daniel Villeneuve / 123rf.com
Foto S. 21: © Do Ra / 123rf.com
Foto S. 36: © mur34 / 123rf.com
Foto S. 54: © rahultiwari3190 / 123rf.com
Foto S. 62: © alexmit / 123rf.com

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.

APEX connect by DOAG

Zwei Tage, nur ein Thema.

Konferenz für APEX-Begeisterte

9. & 10. Juni 2015 in Düsseldorf

DOAG
Deutsche ORACLE-Anwendergruppe e.V.