01-2017 | Frühling | www. ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

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

Java ist die beste Wahl

Einfacher programmieren
Erste Schritte mit Kotlin

Security
Automatisierte Überprüfung von Sicherheitslücken

Leichter testen
Last- und Performance-Test verteilter Systeme

4 191978 304903 01

Schnell Ticket & Hotelzimmer sichern!

Early-Bird-Preise bis 30.01.2017

28.-30. März 2017 in Brühl

Programm online!

JavaLand 2017

www.javaland.eu

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

Wolfgang Taschner
Chefredakteur Java aktuell

Der iJUG auf dem Weg in den Java Community Process

Der Java Community Process (JCP) ist seit dem Jahr 1998 dafür zuständig, in welcher Form sich die Programmiersprache Java und die gesamte Java-Plattform weiterentwickeln. Jede Erweiterung des Java-Standards findet in Form eines Java Specification Request (JSR) statt, der einfach durchnummeriert ist. Alle aktuellen JSRs sind auf der Webseite des JCP (siehe „https://www.jcp.org") gelistet und einsehbar. Das Executive Committee (EC) im JCP stimmt dann darüber ab, ob vorgeschlagene JSRs gestartet beziehungsweise deren Ergebnisse in den Standard aufgenommen werden.

Das Executive Committee hat damit die Rolle einer Art Vorstandsgremium, dessen wichtigste Aufgabe darin besteht, JSRs auf ihrem Weg durch den Java Community Process zu begleiten. Zudem legt das Executive Committee die Abläufe in der Organisation fest. Bei den vom 1. bis 14. November 2016 anstehenden Neuwahlen werden sechzehn der 25 Sitze im Executive Committee neu vergeben. Für einen dieser Sitze hat sich der iJUG beworben, repräsentiert durch Andreas Badelt, den stellvertretenden Leiter der DOAG Java Community.

Am „Community Day" der diesjährigen JavaOne fand eine öffentliche Sitzung des Executive Committee statt, bei der die Kandidaten für die zu vergebenden Sitze die Gelegenheit hatten, sich der Community zu präsentieren. Andreas Badelt war für den iJUG vor Ort und stellte dessen Hauptanliegen vor: den mehr als 20.000 Java-Entwicklern, DevOps etc., die der iJUG vertritt, eine Stimme im wichtigsten Standardisierungsgremium zu geben; damit verbunden mehr Verantwortung der Community zu überlassen (exemplarisch die momentanen Probleme um Java EE 8) sowie eine generell demokratischere Ausrichtung des JCP zu verfolgen.

Trotz aller Kritik an der Vormachtstellung von Oracle und an der Schwerfälligkeit ist der JCP doch eine funktionierende Institution zur Weiterentwicklung von Java. Dies setzt allerdings auch eine aktive Java-Community voraus. Wie diese sich am JCP beteiligen kann, sagte Patrick Curran, Vorsitzender des JCP, bereits im Interview in der Java aktuell, Ausgabe 04/2011: „Die Leute sollten unbedingt die Specs lesen und Feedback geben. Ich erinnere mich an einen sehr wichtigen JSR vor einigen Jahren. Da haben wir nur fünf Kommentare auf den Early Draft bekommen. Es hilft nichts, wenn sich die Leute über die finale Version beschweren. Sie müssen sich vorher zu Wort melden. Es gibt keine Garantie dafür, dass alle Vorschläge implementiert werden, aber wir beschäftigen uns in jedem Fall damit und geben entsprechend Feedback."

In diesem Sinne wünsche ich mir eine rege Beteiligung am JCP. Für Meinungen dazu bin ich wie immer unter „redaktion@ijug.eu" erreichbar.

Ganz besonders drücke ich natürlich dem iJUG die Daumen, dass er in das Executive Committee gewählt wird.

Ihr

W. Taschner

PS: Kurz vor Druckbeginn gab der JCP das Wahlergebnis bekannt.
Leider hat es für den iJUG nicht gereicht *(siehe Seite 61)*.

8

Der Unterschied von Java EE zu anderen Enterprise Frameworks

21

Erste Schritte mit Kotlin

Kotlin ist eine ausdrucksstarke Programmiersprache, um die Lesbarkeit in den Vordergrund zu stellen und möglichst wenig Tipparbeit erledigen zu müssen

3 Editorial

5 Das Java-Tagebuch
Andreas Badelt

8 Java EE – das leichtgewichtigste Enterprise Framework?
Sebastian Daschner

11 Jumpstart IoT in Java mit OSGi enRoute
Peter Kirschner

16 Graph-Visualisierung mit d3js im IoT-Umfeld
Dr.-Ing. Steffen Tomschke

21 Erste Schritte mit Kotlin
Dirk Dittert

26 Neun Gründe, warum sich der Einsatz von Kotlin lohnen kann
Alexander Hanschke

30 Automatisierte Überprüfung von Sicherheitslücken in Abhängigkeiten von Java-Projekten
Johannes Schnatterer

34 Unleashing Java Security
Philipp Buchholz

41 Last- und Performance-Test verteilter Systeme mit Docker & Co.
Dr. Dehla Sokenou

46 Automatisiertes Testen in Zeiten von Microservices
Christoph Deppisch und Tobias Schneck

50 Continuous Delivery of Continuous Delivery
Gerd Aschemann

56 Technische Schulden erkennen, beherrschen und reduzieren
Dr. Carola Lilienthal

62 „Eine Plattform für den Austausch …"
Interview mit Stefan Hildebrandt

63 JUG Saxony Day 2016 mit 400 Teilnehmern

64 Die Java-Community zu den aktuellen Entwicklungen auf der JavaOne 2016

66 Impressum / Inserentenverzeichnis

34

Innerhalb von Enterprise-Anwendungen spielen Sicherheits-Aspekte eine wichtige Rolle

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java – in komprimierter Form und chronologisch geordnet. Der vorliegende Teil widmet sich den Ereignissen im dritten Quartal 2016.

9. August 2016

Software Foundation für MicroProfile?

Die Community hat gelernt: In der Google-Gruppe von MicroProfile startet Martijn Verburg von der London Java Community eine Diskussion, MicroProfile – gegebenenfalls sehr bald – von einer unabhängigen Software Foundation verwalten zu lassen. Die ersten Antworten der beteiligten Hersteller signalisieren volle Unterstützung für dieses Vorhaben. Die Diskussion geht aber eher in die Richtung, MicroProfile einer existierenden Foundation wie der ASF zu übergeben, statt den mühsamen Weg einer Neugründung zu beschreiten.

https://adtmag.com/Blogs/WatersWorks/2016/08/verburg-on-microprofile.aspx

10. August 2016

„Java EE-VP" stattet JCP-Meeting Besuch ab

Anil Gaur, als Oracle Group Vice President verantwortlich für Java EE und den WebLogic Server, hat als Gast an der Sitzung des JCP Executive Committee teilgenommen und Fragen zur Oracle-Strategie beantwortet. Auch das bringt keine bahnbrechenden Neuigkeiten – ist aber vielleicht ein Zeichen dafür, dass Oracle den JCP weiterhin ernst nimmt. Oracle möchte „HTTP/2, Config, State Management, Eventual Consistency, Multi-Tenancy, O-Auth und OpenID Connect" in der Plattform sehen. Außerdem rede man „mit großen Java-EE-Herstellern" und bald auch „mit Community-Mitgliedern wie Java Champions und Java User Groups". Auf MicroProfile angesprochen, berichtet Anil, dass es tatsächlich Gespräche mit Red Hat gäbe und Oracle die Anstrengungen gerne zusammenführen würde (ohne darauf einzugehen, wie das erfolgen könnte). Mark Little von Red Hat bestätigt die Gespräche und den Wunsch zur Zusammenarbeit – „vielleicht, indem unter Microprofile.io Prototyping stattfindet." Ansonsten gilt weiterhin: Warten auf die JavaOne, dann gibt es hoffentlich Details zu hören.

https://jcp.org/aboutJava/communityprocess/ec-public/materials/2016-08-09/August-2016-Public-Minutes.html

17. August 2016

iJUG-Presseerklärung zu Java EE

Der iJUG hat eine Presserklärung zur Situation um Java EE abgegeben. Dort heißt es unter anderem: „Wenn Oracle zur JavaOne nicht umgehend ausreichend Ressourcen für den JCP und die JSRs vorsieht, dann wird die Community kein weiteres Vertrauen in Oracle setzen und nach anderen Alternativen suchen." Dem ist nichts mehr hinzuzufügen …

http://www.ijug.eu/home-ijug/aktuelle-news/article/ijug-fordert-von-oracle-das-projekt-java-ee-8-zuegig- voranzubringen.html

18. August 2016

„Plug-in-free Web": Applet-API wird „@deprecated"

Es gibt neue Informationen zur Abschaffung von Browser-Plug-ins. Das Oracle-Entwicklungsteam hat das JDK Enhancement Proposal (JEP 289) erstellt mit dem Titel „Deprecate the Applet-API". Der Titel sagt es bereits: Mit JDK 9 wird das Applet-API nicht entfernt, sondern Klassen werden mit „@deprecated" annotiert, um sie dann in einem späteren Release zu entfernen. Early-Access-Releases von JDK 9 enthalten die Änderungen bereits. Die treibenden Kräfte dahinter sind Browser-Entwickler wie Apple und Mozilla, die in Zukunft auf standardisierte „Cross Browser"-Unterstützung von Plug-ins verzichten wollen (auch Flash, Silverlight etc. sind davon betroffen). Natürlich will kein Plug-in-Hersteller zig Browser-spezifische Plug-ins entwickeln. Sobald der Plug-in-Support in allen gängigen Browsern verschwunden ist, gibt es dann auch keinen Grund mehr, Applets zu verwenden. Die Suche nach Alternativen war auch in der vorletzten Tagebuch-Ausgabe schon ein Thema – Neues von Oracle gibt es dazu nicht („auf Java WebStart migrieren"). Inzwischen scheinen aber einige Firmen mit kommerziellen Lösungen in die Nische vorzudringen, etwa „Centralized Java" oder „jpro", die JavaFX in den Browser einbinden. Den umgekehrten Weg – einen auf JavaFX basierenden Browser zu entwickeln, der nativ JavaFX und über das eingebettete WebKit auch HTML5 rendern kann, – beschreitet Bruno Borges mit seinem Open-Source-Projekt WebFX. Es ist keine generelle Lösung für das Problem, der Ansatz könnte aber beispielsweise zum internen Einsatz in Unternehmen eine Alternative sein. Bislang ist allerdings zumindest WebFX nicht über ein persönliches Projekt hinausgekommen.

http://openjdk.java.net/jeps/289

14. September 2016

Java 9 wieder verschoben

Mark Reinhold hat über die OpenJDK-Mailingliste einen aktualisierten Plan für das JDK-9-Release vorgeschlagen. Es seien gute Fortschritte insbesondere beim Projekt Jigsaw erzielt worden, aber für die tiefgreifenden Änderungen sei noch etwas mehr Zeit nötig. Inzwischen käme viel Feedback von anderen Projekten, die das JDK 9 testen, und die Anzahl der offenen Bugs sei deutlich höher als bei JDK 8 zum vergleichbaren Zeitpunkt. Der neue Release-Termin soll Juli 2017 sein. Der Plan wird automatisch angenommen, wenn niemand binnen einer Woche Widerspruch erhebt.

http://mail.openjdk.java.net/pipermail/jdk9-dev/2016-September/004887.html

18. September 2016

Oracle gibt NetBeans ab

Lange Zeit hat sich Oracle den Luxus gegönnt, mit dem JDeveloper (insbesondere für die eigenen Produkte), NetBeans sowie den Plug-ins für Eclipse (wie das „Oracle Enterprise Pack for Eclipse") gleich drei Java-IDEs selbst zu entwickeln beziehungsweise massiv zu unterstützen. Gerüchte, ob nicht eine davon bald über Bord geworfen wird, gab es spätestens seit der Sun-Übernahme. Das geschieht jetzt – aber wohl auf die bestmögliche Weise: NetBeans soll zur Apache Software Foundation wechseln. Laut dem Proposal will Oracle ein Team von Entwicklern zur Verfügung stellen, um die Weiterentwicklung zu gewährleisten – gemeinsam mit anderen existierenden und vielleicht auch neuen Committern, die hier eine neue Chance sehen könnten (etwa um kommerziellen Support zur Verfügung zu stellen). Die Gefahr ist natürlich, dass Oracle die Unterstützung irgendwann zurückfährt und dies nicht durch andere Partner aufgefangen wird. Sollte Oracle die Unterstützung aber langfristig aufrechterhalten, dürfte NetBeans eine rosige Zukunft bevorstehen.

https://wiki.apache.org/incubator/NetBeansProposal

18. September 2016

Java-Keynote auf der JavaOne

Oracle hatte die Community bis zur JavaOne in San Francisco vertröstet, um Details dazu zu präsentieren, wie das steckengebliebene EE 8 wieder auf die Reise geschickt werden kann beziehungsweise soll. Die Erwartungen sind also groß. Ein großer Teil der Java-Keynote am Sonntag vor der eigentlichen Konferenz besteht allerdings aus Präsentationen, die einfach nur zeigen sollen: „Schaut, hier steckt überall Java drin – und es ist alles heute so performant und zuverlässig etc." Bei dem (an sich sehr interessanten) einstündigen Vortrag über den Mars Lander wird dann noch nicht einmal erwähnt, dass Java „drinsteckt". Der sehnlichst erwartete Teil mit den Neuerungen kommt dann deutlich zu kurz – am Ende wird ein Live Coding zu Reactive Programming sogar aus Zeitmangel abgewürgt. Vielleicht hätte sich die Oracle-Marketing-Abteilung etwas mehr zurückhalten sollen. Immerhin gibt es ein paar Ankündigungen: Java wird ein „first-class citizen" für Docker. Und Java EE 8 soll 2017 kommen, mit Fokus auf die Cloud und Microservices. Dafür gibt es zwei neue JSRs: Configuration (im zweiten Anlauf) und Health Check. Der neue MVC-JSR fällt dagegen wohl heraus; er ist auf den Folien nicht vertreten. Java EE 9 soll dann innerhalb eines Jahres folgen – ein sehr ehrgeiziger Plan! Mark Reinhold präsentiert dann noch das kommende SE 9: „coming soon – but not as soon as you'd probably hoped". Der Fokus liegt auf den Modularisierungs-Features aus „Project Jigsaw", mit einer Demo von „jlink", mit dem sich maßgeschneiderte JVMs bauen lassen. Aber er zeigt auch live, was die neue „jshell" alles kann (die interaktive Java-Shell, auch als „REPL" bekannt, kommt jetzt endlich in die Java-Welt). Brian Goetz, Java Language Architect bei Oracle, zeigt neue Features, die die Lesbarkeit von Code erhöhen sollen (beispielsweise Compiler-generierter „Low Value"-Code wie „hashCode()", „toString()" etc. und Verbesserungen bei der dynamischen Type Inference).

https://www.youtube.com/watch?v=Hdn0lqJYFeQ&list=PLPIzp-E1msrZZwFtBIAbC6lB_-eC3WSgo

18. September 2016

Öffentliches Meeting des JCP Executive Committee – iJUG bewirbt sich um einen Sitz

Ein wichtiges Thema gab es noch am Vortag der JavaOne: das öffentliche Meeting des JCP Executive Committee (EC) mit dem einzigen Zweck, alle Kandidaten für die Neuwahlen zum EC im November vorzustellen. Der iJUG hat sich wenige Tage vor der JavaOne zu einer eigenen Kandidatur entschieden. So habe ich die Ehre, vor einem unerwartet gut gefüllten Konferenzsaal zu beschreiben, was der iJUG ist und macht und was wir mit einer Kandidatur bezwecken. In erster Linie natürlich: Die Stimmen von mehr als 20.000 Java-Enthusiasten, die von den aktuell dreißig einzelnen JUGs vertreten werden, in das höchste Gremium des Java-Standards zu bringen. Dazu gehört sicher, insbesondere mit den anderen JUGs im EC die Demokratisierung des JCP voranzutreiben und bessere Lösungen für „steckenbleibende" JSRs in den Prozess zu integrieren, oder dies zumindest auf andere Weise zu vermeiden.

http://www.ijug.eu/home-ijug/aktuelle-news/article/ijug-bewirbt-sich-fuer-sitz-im-executive-committee-des-jcp.html

19. September 2016

JavaOne: Tag 1

Der erste Tag der JavaOne bringt das, was die Java-Keynote ausgelassen hat: Details zu Java EE 8 und der geänderten Roadmap, präsentiert von Linda deMichiel (Specification Lead des EE 8 „Umbrella JSRs"). Sie beginnt mit einem Rückblick auf das EE-8-„Community Survey", das eine wichtige Grundlage für die Themensetzung in EE 8 war; unter anderem war MVC dadurch in den Fokus gerückt. Dann stellt sie den aktuellen Status der einzelnen JSRs aus ihrer Sicht vor – mit ein paar Ausnahmen (wie JMS, Management API) ist da von guten Fortschritten die Rede. Das sehen einige „Guardians" vermutlich anders, aber das Hauptthema dieser Session leitet deMichiel mit den Worten ein: „The world has changed." Cloud, Microservices, „Rapid Evolution of Applications" und DevOps seien jetzt die wichtigen Themen. Daher auch die Änderungen im Fokus von EE 8, mit denen Java EE einen Migrationspfad hin zu Cloud und Microservices anbinden soll. Den JSRs Management API, MVC und JMS wird dabei zu wenig Relevanz zugesprochen. Konkret sollen anstelle von JMS neuere Formen von „Eventing" betrachtet werden; das Management API werde sowieso nicht stark genutzt (ok, aber ist das angesichts des Uralt-API vielleicht ein Henne-Ei-Problem?); über MVC sei viel diskutiert worden, aber Cloud-Apps seien nun mal häufig „headless". Angesichts knapper Ressourcen sollen diese drei JSRs daher nicht weiterverfolgt werden. Wobei: Außer von den aktuellen Spec Leads selbst kann ja die Community nicht daran gehindert werden, diese in Eigenregie weiterzuverfolgen – und warum sollten sie dann am Ende nicht doch in den Standard integriert werden; aber das ist nur mein naiver Gedanke und kein Thema der Session. Als neue JSRs hinzu kommen dann, wie in der Keynote verkündet, Configuration und Health Check.

Für alle anderen JSRs soll sich nichts ändern. Detaillierte Pläne für EE 9 gibt es noch nicht, es soll aber im Wesentlichen die Ansätze von EE 8 vertiefen. Zum Abschluss wirbt deMichiel noch für ein neues Survey auf der GlassFish-Seite, das bis Mitte Oktober läuft. Damit soll Feedback zur Neuplanung von EE 8 eingeholt werden. Ein bisschen komisch ist es am Beispiel „MVC" natürlich schon, ein Survey durchzuführen, damit ein Release zu planen, diese Planung

dann umzuschmeißen und dafür ein neues Survey zu starten. Andererseits steht die Zeit natürlich nicht still und vielleicht zeigen sich die Verschiebungen ja auch im Survey. In der Fragerunde kommt direkt das Thema „MicroProfile" zur Sprache und wie sich Oracle dabei positionieren will. Die diplomatische Antwort von deMichiel: „Wir schauen uns das an und reden mit den Leuten hinter MicroProfile – aber es ist noch nichts entschieden." Historisch haben viele dieser Leute an Java EE gearbeitet und die Ziele seien dieselben. Sei für EE 8 oder EE 9 so etwas wie MicroProfile geplant? Antwort: „Das soll zunächst intern diskutiert werden, aber für EE 9 ist es denkbar, eine stärkere Modularisierung, basierend auf dem dann mit SE 9 verfügbaren „Jigsaw", anzugehen." Weitere Vorträge am ersten Tag vertiefen die Themen noch weiter, etwa „Enterprise Java for the Cloud", wo die Optionen unter anderem für ein Eventing API, „Resiliency" oder reaktives Programmieren betrachtet werden. Nach einem langen Tag darf natürlich die Party nicht fehlen. In diesem Fall die JCP-Party im obersten Stock des Hilton. Hier gibt es nicht nur eine fantastische Aussicht und leuchtende Cocktails, sondern auch die JCP Annual Awards. Diesjährige Preisträger: Member/Participant of the Year: Werner Keil; Outstanding Spec Lead: Dmitry Kornilov; Most Significant JSR: JSR 364, Broadening JCP Membership; Outstanding Adopt-a-JSR Participant: Josh Juneau & Bob Paulin von der Chicago JUG. Herzlichen Glückwunsch!
https://www.youtube.com/results?search_query=javaone+2016

20. September 2016

JavaOne: Tag 2

Der zweite Tag der JavaOne bringt viele interessante Vorträge, etwa „Cloud-Native Java EE" von Payara, der Firma, die nicht nur GlassFish „geforkt" hat und für kommerziellen Support sorgt, sondern auch stark beim MicroProfile engagiert ist.
https://www.youtube.com/watch?v=OtCNZGOMIU0

21. September 2016

JavaOne: Tag 3

Der dritte Tag beginnt mit Stephen Chin, der einen unterhaltsamen Vortrag über „Raspberry Pi with Java 9" hält und dazu seinen Line-Follower-Roboter durch den Vortragssaal fahren lässt (nachdem morgens schon drei hochmotivierte Freiwillige weißes und schwarzes Klebeband fast durch den gesamten Saal verlegt haben). Zu „Eventual Consistency" – auch ein großes Thema in Zeiten von Microservices – gibt es einen ebenfalls interessanten Vortrag. Für die über Jahre partyverwöhnten JavaOne- und OpenWorld-Besucher muss es abends dann natürlich der AT&T-Park (Baseball-Stadion der SF Giants) sein, mit Auftritten von Gwen Stefani und Sting. Die Preislisten an den Bier- und Snack-Ständen schockieren ein wenig, aber diesmal ist ja alles kostenlos (oder: in das JavaOne-Ticket eingepreist).
https://www.youtube.com/watch?v=v95y5CzvcZ4&feature=youtu.be

22. September 2016

JavaOne: Tag 4

Der vierte Tag beginnt mit der unterhaltsamsten Keynote der gesamten Woche, der Community-Keynote. John Duimovich von IBM startet mit den Themen „JVM Performance" und „Microservices" (die Lehr-Plattform „Game On!" ist zum Anschauen empfohlen für alle, die noch keine Microservice-Experten sind). IBM sei ein starker Unterstützer von Java EE 8 und WebSphere Liberty mit seiner Modularisierung sei „the right fit" für Java EE und MicroProfile. Applaus erntet Duimovich für die Ankündigung, dass IBM sein SDK für Java zu Open Source machen werde. Zum Abschluss gibt es eine Demo mit einem Nao-Roboter, der mithilfe eines Service, der unter anderem den Watson-Speech2Text-Service und Alexa integriert, relativ komplexe Fragen beantworten kann. Es folgt die traditionelle Aufführung – diesmal haben Darth Coder und die Duke Troopers Module von Entwicklern gestohlen. Spoiler Alert: James Gosling tritt als Darth Coder auf und es stellt sich am Ende heraus, dass er Dukes Vater ist. Aber die Aufführung hält genügend weitere Lacher bereit. Unbedingt auf YouTube anschauen. Weiteres Highlight des Tages ist das MicroProfile Lunch – eine Podiums-Diskussion mit Vertretern der beteiligten Firmen und JUGs. Bemerkenswert, dass eine ganze Reihe von Oracle-Angestellten im Publikum sitzen und in offizieller Funktion angeregt mitdiskutieren. Auch Reza Rahman von den „Java EE Guardians" ist vertreten, um sein Hauptanliegen „don't diverge – collaborate" vorzubringen. Zumindest dem Verlauf der Diskussion nach könnte es ein gemeinsames Happy End für MicroProfile und Java EE geben. David Blevins als Vertreter von Tomitribe bringt es auf den Punkt, indem er sagt, dass MicroProfile nicht der Standard (EE) ist, aber der Weg dahin sein kann: „This could be the first use case for pre-standardization collaboration."
https://www.youtube.com/watch?v=HdnOlqJYFeQ

23. September 2016

Androiden auf Ceylon gesichtet

Ceylon unterstützt in der neuen Version 1.3 die Entwicklung für Android – und auch die Paketierung für Wildfly Swarm. Die als Java-Alternative entworfene Programmiersprache ist sowohl für den Einsatz in der JVM als auch in Java Script Engines gedacht. Version 1.3. bietet auch hier Neues: Unterstützung für den Node Package Manager (npm). Weitere Highlights sind Plug-ins für IntelliJ IDEA und (natürlich) Android Studio, Docker Images und volle Unterstützung von Java 8.
https://ceylon-lang.org/blog/2016/09/19/ceylon-1-3-0/

Kotlin-Plug-in für NetBeans

Wer mit IntelliJ IDEA arbeitet, kennt JetBrains JVM- und JavaScript-Sprache wahrscheinlich. Neben IDEA und Eclipse wird nun endlich Netbeans per Plug-in unterstützt.
https://github.com/JetBrains/kotlin-netbeans

Andreas Badelt
Leiter der DOAG SIG Java

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

Java EE – das leichtgewichtigste Enterprise Framework?

Sebastian Daschner, Sebastian Daschner – IT-Beratung

Es war einmal eine Zeit, da galten J2EE und insbesondere die Application Server als zu aufgebläht und schwergewichtig. Es konnte ziemlich mühsam und entmutigend für Software-Entwickler sein, diese Technologie zu benutzen. Aber spätestens seitdem der Name zu Java EE geändert wurde, ist diese Annahme nicht mehr wahr. Wo genau liegt der Unterschied von Java EE zu anderen Enterprise Frameworks und was macht ein Framework überhaupt leichtgewichtig?

Einer der wichtigsten Aspekte bei der Auswahl einer Technologie ist der Entwicklungsprozess und die daraus resultierende Produktivität. Programmierer sollten möglichst viel Zeit mit dem Lösen von Problemen und Implementieren von Use-Cases verbringen. Denn genau das schafft am Ende des Tages den Mehrwert eines Produktes beziehungsweise den Profit einer Firma.

Das bedeutet wiederum, dass die Technologien und Methoden die Zeit, die Entwickler mit Warten auf Build-Prozesse, Tests, Deployments, aufwändigem Konfigurieren von Applikationen und Implementieren von nicht-Use-Case-relevanten „Klemptnerarbeiten" sowie Konfigurieren der Build-Umgebung und Abhängigkeiten der Applikation verbringen, minimieren sollten.

Standards

Einer der größten Vorteile, den Java EE gegenüber anderen Frameworks bietet, ist die Standardisierung der APIs. Standards klingen schwergängig und innovationsscheu – und genau das sind sie im Grunde auch, denn der Java Community Process (JCP) hält in den Java Specification Requests (JSR) fest, was sich über längere Zeit davor in der Industrie etabliert hat. Doch genau diese Standards bieten eine Reihe von Vorteilen.

Zusammenspiel der einzelnen Technologien

Die verschiedenen APIs innerhalb des Java-EE-Umbrellas wie zum Beispiel CDI, JPA, JAX-RS, JSONP oder Bean Validation spielen sehr gut zusammen und können out of the box miteinander kombiniert werden. Allen voran sorgt dafür CDI, das als „Kleber" zwischen Komponenten fungiert. Die JSRs beinhalten Voraussetzungen wie: Sobald eine bestimmte Technologie auf dem Container

zur Verfügung steht, muss sie auch mit der anderen Technologie nahtlos zusammenspielen.

Zwei Beispiele: JAX-RS unterstützt JSONP-Typen wie „JsonObject" als Request und Response Entities beziehungsweise integriert nahtlos Bean Validation – passende HTTP-Status-Codes im Falle einer fehlgeschlagenen Validierung inklusive. Validatoren wiederum können per „@Inject" CDI-managed-Beans injizieren und so weiter – Integration der Spezifikationen ist ein großes Augenmerk des Java-EE-Umbrellas.

Diese Ansätze schaffen ein Entwicklerfreundliches und produktives Arbeiten, da sich die Programmierer darauf verlassen können, dass der Application Server die Integrations- und Konfigurationsarbeit leistet. Der Fokus kann somit auf der eigentlichen Business-Logik liegen.

Deklarative, von Convention-over-Configuration getriebene Entwicklung

Der Convention-over-Configuration-Ansatz von Java EE sorgt dafür, dass das Gros der Anwendungsfälle ohne jegliche Konfiguration auskommt. Die Tage von Unmengen an „xml"-Deskriptoren sind vorbei. Für die einfachste Java-EE-Anwendung ist keine einzige „.xml"-Datei mehr notwendig.

Dank der deklarativen Annotations wird ein einfaches, annotiertes POJO zum HTTP Endpoint („@Path") beziehungsweise zur Enterprise Bean („@Stateless") inklusive Transaktionen, Monitoring und Interzeptoren. Diese Ansätze haben sich in der Vergangenheit in vielen Frameworks etabliert und wurden in Java EE zum Standard erhoben.

Externe Abhängigkeiten

Die wenigsten Enterprise-Projekte in der realen Welt kommen ganz ohne im Deployment-Artefakt mitgelieferte externe Bibliotheken aus. Dazu gehören jedoch erfahrungsgemäß eher technisch- als Use-Case-begründete Abhängigkeiten, allen voran Logging- oder Entity-Mapping-Frameworks oder Common Purpose Libraries wie Apache Commons oder Google Guava.

Java EE 7 – insbesondere in Verwendung mit Java 8 – liefert jedoch ausreichend APIs und Funktionalitäten mit, um die gängigen Use-Cases im Enterprise-Umfeld abzudecken. Was nicht out of the box vorhanden ist, kann meist mit minimalem Code gelöst werden, zum Beispiel benötigte, injizierte Configuration per CDI-Producer, Circuit Breaker per Interzeptoren oder aufwändige Collection-Operationen mit Java-8-Lambdas und -Streams.

Natürlich könnte man an dieser Stelle argumentieren, das Rad nicht neu erfinden zu müssen. Er ergibt jedoch wenig Sinn, für eine Handvoll eingesparter Lines of Code Megabyte an Code-Bibliotheken in das Projekt und Deployment-Artefakt zu ziehen. Dabei liegt erfahrungsgemäß das größte Problem noch nicht einmal in der Größe der damit eingeführten direkten Abhängigkeiten, sondern in den transitiven Abhängigkeiten. Letztere kollidieren in vielen Fällen mit anderen im Deployment-Artefakt oder im Application Server vorhandenen Versionen und sorgen für aufwändig zu lösende Konflikte.

Im Endeffekt verbringen die Entwickler mehr Zeit mit dem Konfigurieren und Managen von Abhängigkeiten und eventuellem Ausschließen etwa per Maven „<exclude>"-Direktive als mit der Programmierung und dem Test des Features. Das gilt selbstverständlich in erster Linie für die einfacheren (und doch meisten) Fälle und erfahrungsgemäß dann, wenn Abhängigkeiten technisch und nicht Use-Case-begründet sind.

Ein Beispiel: In den wenigsten Fällen sieht die Business-Anforderung einer Applikation ein bestimmtes Logging-Framework vor, jedoch durchaus eine Integrations-Bibliothek eines externen Systems. Der Unterschied und die Legitimation, ob eine externe Abhängigkeit benutzt werden soll, liegt – neben der Komplexität – also an der fachlichen Notwendigkeit der Lösung. Es empfiehlt sich generell, wenn möglich auf Dependencies zu verzichten, vor der Einführung die Vor- und Nachteile gut abzuwägen und in allen Fällen den Dependency-Tree des Projektes im Auge zu behalten.

Programmieren gegen APIs

Genauso wie die Empfehlung, auf Java-Code-Ebene wenn möglich gegen vorhandene Interfaces statt Implementierungen zu entwickeln, hängen Java-EE-Applikationen im Idealfall nur vom API und nicht von der Container-Implementierung ab. Das minimiert Fehler und Risiken und gewährleistet eine Portabilität zwischen Implementierungen. Auch wenn in Projekten die Portabilität meist nicht in Anspruch genommen wird oder, falls doch, nicht unbedingt reibungslos vonstattengeht, ist sie eher als Rettungsboot denn als Garantie zu verstehen. Gerade für langfristige Projekte im Enterprise-Umfeld ist es unabdingbar, sich auf eine abwärtskompatible, möglichst portable Technologie verlassen zu können. Damit erreicht man langfristig die geringeren Wartungs- beziehungsweise Portierungsaufwände.

Da das Java-EE-API den Application Servern bekannt ist, muss es außerdem nicht beim Deployment mitgeliefert werden. Im Artefakt ausgeliefert wird nur die tatsächliche Businesslogik – mit nur geringem Anteil an „Glue Code" und Cross-Cutting-Concerns.

Schlanke Deployment-Artefakte

Da in einem modernen Java EE-Projekt das API nur als „provided Dependency" verwendet wird, landen im Deployment-Artefakt auch nur die projekteigenen Klassen. Das sorgt für sehr schlanke Artefakte im Kilobyte-Bereich, die wiederum schnelle Build-Zeiten ermöglichen, da der Build-Prozess ohne ständiges Kopieren der Abhängigkeiten auskommt. Der Unterschied kann in der Praxis viele Megabyte und im Build einige Sekunden ausmachen. Summiert man die Zeit, die alle Entwickler beziehungsweise die Continuous Integration Server im Lauf der Projektzeit dadurch mehr aufwenden, dann werden die Unterschiede deutlich. Je häufiger man das Projekt zusammenbaut – gerade im Hinblick auf Continuous Delivery – desto größer diese Ersparnis.

Abgesehen von Build-Zeiten sorgen schlanke Artefakte auch für schnellere Deployment- beziehungsweise Publish-Zeiten. In allen Fällen sind die Moving Parts gering, da die Implementierung des Frameworks eben schon auf dem Application Server vorhanden ist und nicht jedes Mal mit ausgeliefert wird.

Das ideale Framework für Docker

Für moderne, containerbasierte Umgebungen bietet Java EE aus diesem Grund die ideale Struktur. Das Base-Image enthält schon das Betriebssystem, die Java-Runtime und

den Application Server; das Zusammenbauen des Container-Images fügt nur noch die letzte, wenig Kilobyte schlanke Image-Schicht des Deployment-Artefakts hinzu. Diese Zeit- und Speicherplatz-Ersparnis im Vergleich zu Fat-war- oder Standalone-jar-Ansätzen betrifft sowohl das Zusammenbauen als auch das Ausliefern und Versionieren der Images.

Moderne Applikationsserver

J2EE Application Server waren der Inbegriff schwergewichtiger Software in Hinblick auf Start- und Deployment-Zeiten, Installationsgröße und Ressourcen-Footprint. Doch seit der neuen Welt von Java EE entspricht diese Annahme nicht mehr der Wahrheit.

Alle modernen Java EE 7 Application Server wie WildFly, Payara, WebSphere Liberty Profile oder TomEE starten und deployen in wenigen Sekunden. Dafür sorgen fortgeschrittene Modul-Systeme, die nur die benötigten Komponenten initialisieren, und die Tatsache, dass die Deployment-Artefakte sehr schlank gehalten sind.

Die Installationsgröße und der Footprint halten sich sehr in Grenzen. Ein Application Server verbraucht nicht viel mehr Speicher als ein Servlet Container und liefert dann aber schon alle benötigten Enterprise-Funktionalitäten mit. So ist es mit den heutigen Servern möglich und sinnvoll, statt mehrere Deployments auf einem Server einfach mehrere Server mit jeweils nur einer Applikation laufen zu lassen – entweder in einem Container oder als herkömmliche Installation.

Packaging

Was das Packaging angeht, gibt es keinen wirklichen Grund, der noch für „ear"-Archive spricht. Diese Archive beinhalten die Web- und Enterprise-Komponenten noch einmal verpackt als „.war"- beziehungsweise „.ejb-jar"-Archiv. Da der Ansatz, die komplette Applikation auf einem einzigen, dedizierten Applikationsserver laufen zu lassen, voraussetzt, dass ohnehin alle Komponenten vorhanden sein müssen, lässt sich durch den Verzicht auf „.ear"-Archive und die zusätzliche Verpackung Build- und Deployment-Zeit einsparen. Davon abgesehen vermeidet man damit alle Arten von Classloader-Hierarchie-Fehlern, da alle Klassen in einem Kontext vorhanden sind.

In den neuen Microservices- und Cloud-Ansätzen werden Anwendungen vermehrt als Standalone-jar, das sowohl die Applikation als auch die Laufzeit beinhaltet, ausgeliefert. Das ist in der Java-EE-Welt dank Technologien wie WildFly Swarm oder TomEE Embedded möglich.

Aus den genannten Gründen ist es jedoch sinnvoll, wenn möglich die Geschäftslogik, also die Anwendung, von der Implementierung zu trennen. Standalone-jars sind nach Meinung des Autors genau dann ein hilfreicher Workaround, wenn kein Einfluss auf die vorhandene Infrastruktur genommen werden kann, etwa darauf, welche Versionen von Applikationsservern zu Verfügung stehen, oder wenn die Installations- und Operations-Prozesse zu schwergängig sind. In diesen Fällen setzen die Standalone-jars nur eine Java-Runtime voraus und liefern den Rest selbst mit.

Dennoch ist die zu empfehlende Art des Packagings das „.war"-Archiv, das idealerweise nur die Applikation ohne externe Bibliotheken enthält. Ein sehr gutes Beispiel liefert der Maven Java EE 7 Essentials Archetype von Adam Bien.

Fazit

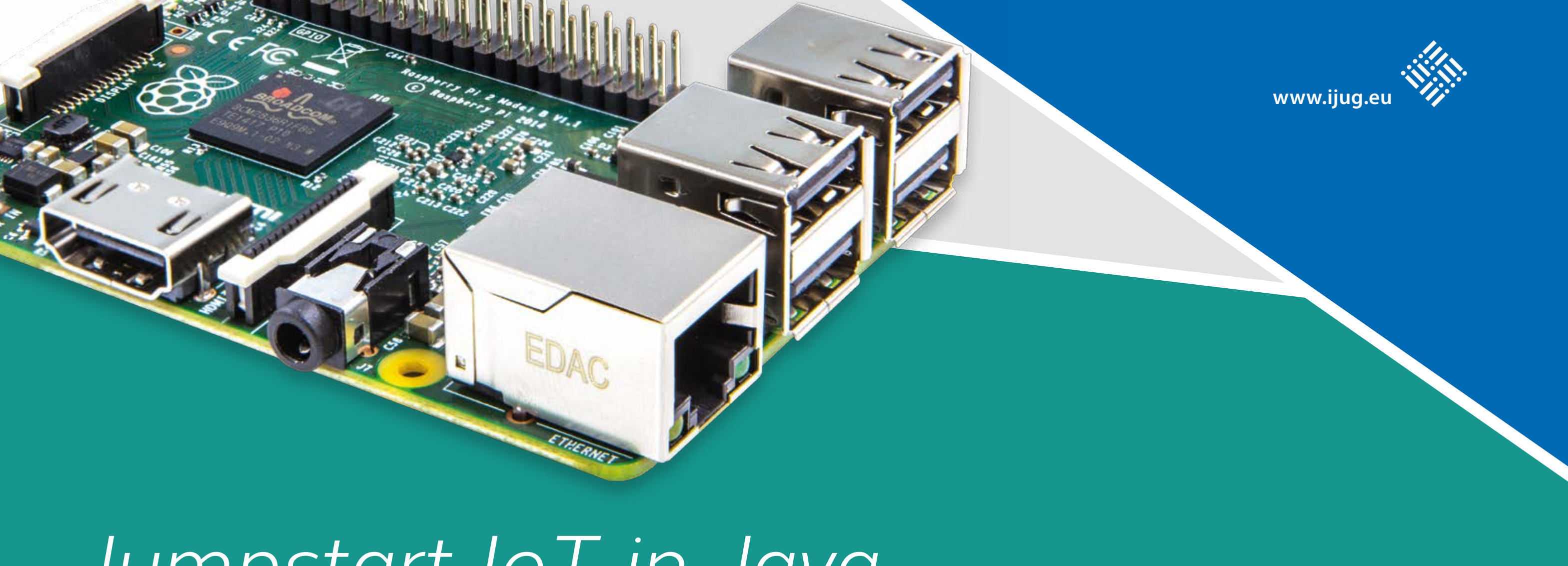
Die Tage von schwergewichtigem J2EE sind schon länger vorbei. Das API des Java-EE-Umbrella bietet ein sehr gutes Entwicklungs-Erlebnis und nahtlose Integration der vorhandenen Standards. Gerade der Ansatz, die Applikationen gegen die APIs zu programmieren und die Implementierung separat zu halten, sorgt in der Praxis für gut performende Entwicklungs- und Auslieferungsprozesse. Mit dem heutigen, modernen Java EE 7 zusammen mit Java 8 haben Entwickler daher eine produktive und ausgereifte Technologie zum Realisieren von Enterprise-Software an der Hand.

Sebastian Daschner
mail@sebastian-daschner.com

Sebastian Daschner arbeitet als freiberuflicher Java-Consultant, Software-Entwickler und -Architekt und programmiert begeistert mit Java (EE). Er nimmt am Java Community Process teil, ist in der JSR 370 Expert Group vertreten und entwickelt an diversen Open-Source-Projekten. Er ist Java Champion und arbeitet seit mehr als sieben Jahren mit Java. Sebastian Daschner evangelisiert Java- und Programmier-Themen unter „https://blog.sebastian-daschner.com" sowie auf Twitter unter „@DaschnerS". Wenn er nicht gerade mit Java arbeitet, bereist er auch gerne die Welt – entweder per Flugzeug oder Motorrad.

Interessenverbund der Java User Groups e.V. (iJUG) vereint bereits 31 Java User Groups

Nach der Übernahme von Sun durch Oracle im Jahr 2009 haben sieben Anwendergruppen den iJUG gegründet. Der Interessenverbund ist auf mittlerweile 31 Mitglieder gewachsen und vertritt damit mehr als 20.000 Entwickler aus Deutschland, Österreich und der Schweiz (siehe Seite 66). Ziel ist die umfassende Vertretung der gemeinsamen Interessen der Java User Groups sowie der Java-Anwender im deutschsprachigen Raum. Ein großer Erfolg des iJUG ist die Durchführung der JavaLand-Konferenz, die bereits im dritten Jahr im Phantasialand in Brühl stattfindet und mittlerweile zu den größten Java-Konferenzen in Europa zählt. Hinzu kommt die Herausgabe der Zeitschrift Java aktuell, die bereits nach einem Jahr einen führenden Platz unter den deutschsprachigen Java-Magazinen erreicht hat. Durch regelmäßige Öffentlichkeitsarbeit und in zahlreichen Gesprächen mit Vertretern von Oracle konnte der iJUG etliche Verbesserungen für die Java-Community erzielen. Themen waren unter anderem Sicherheitslücken in Java, Probleme beim Java-Update sowie Aufforderungen an Oracle hinsichtlich der Weiterentwicklung von Java FX und Java EE.



Jumpstart IoT in Java mit OSGi enRoute

Peter Kirschner, Kirschners GmbH

Das Internet of Things (IoT) ist einer der aktuellen Trends. Dieser Artikel zeigt, wie man für wenig Geld mithilfe eines Raspberry Pi und der OSGi-Distribution enRoute eigene IoT-Anwendungen in Java entwickeln kann. Damit ist man bereits in der Lage, ein vollständiges IoT-System zu entwickeln und zu Hause zu betreiben.

Ausgehend von den Zutaten für unser IoT-Gericht werden wir auf die Entwicklungsumgebung und dann die eigentliche Applikationsentwicklung eingehen. Nachdem die Applikation dann den ersten Sprint hinter sich hat, sehen wir, wie man das Ganze als lauffähiges Archiv exportiert und dann dediziert auf der Hardware laufen lässt. Aus Gründen der Einfachheit wird die folgende Anleitung für Windows als Betriebssystem beschrieben, obwohl osX und Linux mindestens genauso gut funktionieren.

Warum OSGi?

Die OSGi-Alliance [1] wurde im Jahr 1999 gegründet und hat ein dynamisches Komponenten-Modell innerhalb des Java Community Process (JSR-291) entworfen. Das war die Version OSGi R4.1. Danach ging die Entwicklung kontinuierlich weiter und aktuell ist die Version 6 freigegeben. Dabei wird ein besonderes Augenmerk auf Kompatibilität gelegt. „Evolution statt Revolution" wurde mithilfe des konsequenten Einsatzes von Semantic Versioning [2] gelebte Praxis. Dies erlaubt OSGi-Nutzern eine kontinuierliche Entwicklung und Wartung der entwickelten Systeme über Jahre hinweg, ohne dass die Investition in die Basis-Technologie immer wieder überarbeitet werden muss.

Die OSGi-Plattform wurde und wird immer wieder als zu komplex bezeichnet. Diese Komplexität entsteht vor allem durch die Dynamik, die mit verwaltet wird. Mit IoT haben wir aber genau dies als standardmäßigen Anwendungsfall. Sensoren, Gateways und Netzwerke können jederzeit ausfallen. Hier kommt eine der Stärken von OSGi zum Tragen. Es kann zur Laufzeit auf solche Vorfälle reagieren, sich umkonfigurieren und sinnvoll weiterarbeiten.

Es soll hier nicht im Detail auf die vielen Vorteile der OSGi-Architektur eingegangen werden, da dies den Jumpstart ausbremsen würde. Erwähnt werden aber die Aspekte „Modularität", „Architektur" und die „Nano-Services". Details dazu auf der Webseite der OSGi-Alliance [1].

Die Zutaten

Abbildung 1 zeigt im blauen Balken die Hardware, die unser System umfassen kann. Von links nach rechts erfolgt die Beschreibung der Bestandteile:

- *Raspberry Pi* [3]
  Unsere IoT-Basis
- *PC*
  Mit Windows, Linux oder osX als Entwicklungsrechner
- *Optional Sunfounder SensorKit*
  Sensoren für Temperatur, Helligkeit etc.
- *Optional 7-Zoll-Raspberri-Pi-TouchScreen* [3]
  Visualisierung und Interaktion mit der Anwendung

Der grüne Balken enthält den Software-Stack:

- *Raspbian OS*
  Ein auf Debian Linux basierendes Betriebssystem für den Raspberry
- *Java 8*
  Endlich Lamda-Ausdrücke, Stream-API und Java Mission Control
- *Eclipse*
  Sehr flexible, robuste und erweiterbare IDE (auf OSGi basierend)
- *Bndtools*
  Eclipse-Plug-in zur Entwicklung von OSGi-Bundles
- *OSGi enRoute*
  Beispiel für ein serviceorientiertes System, das als Community-Projekt entwickelt wird. Es soll sowohl das OSGi-Programmier-

Modell als auch die vollständige Werkzeugkette entlang des gesamten Software-Lebenszyklus erläutern

Der gelbe Balken zeigt einige der ergänzenden Werkzeuge:

- *GitHub*
  Source-Code-Verwaltung
- *Travis CI*
  Continuous Integration
- *jpm4j*
  Java Package Manager

Abbildung 1: Zutaten in unserem IoT-Stack

```
pi@kipi3_01:~ $ java -version
java version "1.8.0_65"
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) Client VM (build 25.65-b01, mixed mode)
pi@kipi3_01:~ $ curl https://bndtools.ci.cloudbees.com\
> /job/bnd.master/719/artifact/dist/bundles/\
> biz.aQute.jpm.run/biz.aQute.jpm.run-3.0.0.jar >jpm.jar
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1680k  100 1680k    0     0   285k      0  0:00:05  0:00:05 --:--:--  349k
pi@kipi3_01:~ $ sudo java -jar jpm.jar init
Home dir      /var/jpm
Bin  dir      /usr/local/bin
pi@kipi3_01:~ $ jpm version
3.0.0.201509101330
pi@kipi3_01:~ $ sudo jpm install -f biz.aQute.remote.main
pi@kipi3_01:~ $
```

Abbildung 2: Raspberry Pi – one time setup

```
curl https://bndtools.ci.cloudbees.com \
/job/bnd.master/719/artifact/dist/bundles/ \
biz.aQute.jpm.run/biz.aQute.jpm.run-3.0.0.jar >jpm.jar
```

Listing 1

Setup des Raspberry Pi

Das Betriebssystem kann man direkt von der Raspberry-Pi-Webseite als Iso-Image herunterladen und dann etwa mit dem Win-32-Disk-Imager auf eine SD-Karte schreiben. Die aktuellen Versionen von Raspbian enthalten bereits ein vorinstalliertes Java in der Version 8. Nachdem die SD-Karte mit dem Image beschrieben wurde, muss man diese in den Raspberry Pi stecken, ein Netzwerkkabel anschließen (USB-Tastatur und HDMI-Fernseh-/Monitor-Anschluss sind nur zur initialen Konfiguration erforderlich) und dann das Gerät über den microUSB-Anschluss mit Strom versorgen.

Nachdem der Raspberry Pi das erste Mal gebootet hat, ist noch einiges zu konfigurieren. Wir vergeben ein Passwort, den Hostnamen, notieren uns die IP-Adresse oder vergeben eine feste IP und stellen ein, dass er im Shell Mode booten soll. Danach können wir ihn ohne angeschlossene Tastatur und Monitor neu starten.

Alternativ zu dieser initialen Konfiguration kann man bereits die SD-Karte mithilfe des Projekts PiBakery *[2]* mit all diese Einstellungen versehen und direkt ein konfiguriertes ISO-Image auf die Karte schreiben. Auf diesem Wege wäre es auch möglich, unser exportiertes Applikations-Archiv anschließend automatisch aus dem Continuous-Integration-System herunterzuladen und starten zu lassen.

Nach dieser initialen Konfiguration verbinden wir uns vom PC aus mit dem Raspberry Pi via „ssh". Dazu kann man die Git-Bash verwenden oder ein dediziertes Programm wie MobaXterm *[4]*. Nachdem wir verbunden sind, müssen wir noch ein Setup einmalig durchführen *(siehe Abbildung 2)*.

Zuerst prüfen wir mit „java –version", welche Version von Java installiert ist. Dann laden wir den Java-Package-Manager „jpm" herunter *(siehe Listing 1)*. Er muss mit „sudo java -jar jpm.jar init" einmalig initialisiert werden. Anschließend prüfen wir mit „jpm –version" noch die Version von jpm. Zu guter Letzt können wir mit „sudo jpm install -f biz.aQute.remote.main" den Java-Remote-Debug-Agent von Bnd installieren. Jetzt ist der Raspberry Pi für unsere Entwicklungsarbeit vorbereitet und wir widmen uns dem PC.

Der Entwicklungs-PC

Unser PC benötigt ein Java SDK in der Version 8 sowie ein Eclipse SDK, in das zusätzlich das Bndtools-Plug-in *[5]* installiert sein muss. Danach könnten wir das GitHub-Repository auschecken oder das Original-enRoute-IoT-Tutorial *[6]* durcharbeiten.

Als Alternative hat der Autor einen Eclipse-Installer genutzt, um eine vorkonfigurierte IDE namens „IDEfix" mit Eclipse, Bndtools und dem Projekt-Setup für unseren IoT-Jumpstart zur Verfügung zu stellen. Der Eclipse-Installer übernimmt dann die Installation von der IDEfix-Umgebung mit Eclipse und Bndtools sowie das Auschecken und Importieren des IoT-Beispielprojekts namens „osgi.enroute.iot.domotica.application" in die Entwicklungsumgebung. Eine detaillierte Anleitung zur Installation von IDEfix findet sich hier *[7]*. Nachdem

```
pi@kipi3_01:~ $ sudo bndremote -a
Listening for transport dt_socket at address: 1044
```

Abbildung 3: Raspberry Pi – „start bndremote"

wir die Entwicklungsumgebung auf unserem PC aufgesetzt haben, können wir mit der Entwicklung des Systems beginnen.

Die Entwicklung

Das System wird hier auf das „Hello World"-Beispiel beschränkt, da der Fokus darauf liegt, den gesamten Projekt-Lebenszyklus darzustellen, und nicht auf der konkreten Sensor-Anbindung bei IoT-Systemen. Dies könnte man durch das enRoute-IoT-Tutorial *[6]* vertiefen.

Jetzt fokussieren wir uns auf unsere Anwendung, die auf dem RasPi laufen soll. Dazu müssen wir zwei Dinge durchführen. Als Erstes starten wir den „bndremote"-Agent auf dem Raspberry Pi. Dazu nutzen wir erneut die „ssh"-Shell und führen das Kommando „sudo bndremote –a" aus. Der Raspberry PI sollte mit folgender Meldung antworten *(siehe Abbildung 3)*.

Jetzt haben wir erfolgreich den Agent auf Port 1044 gestartet. Mit diesem kann sich unsere Entwicklungsumgebung jetzt verbinden. Das Kommando wird mit Root-Rechten („sudo") ausgeführt, da wir direkt auf die Hardware des Raspberry Pi zugreifen möchten.

Im zweiten Schritt folgt das Anpassen und Starten der Launch-Konfiguration in IDEfix. Um jetzt die Eclipse-IDE direkt mit dem Agent kommunizieren zu lassen, müssen wir die Datei „osgi.enroute.iot.domotica.bndrun" anpassen. Wir öffnen sie und selektieren auf dem „Run"-Tab einmal den „Resolve"-Button. Dieser sorgt dafür, dass ausgehend von unseren „Run Requirements" die Bundles, die zu unserer Applikation gehören, aller transitiven Abhängigkeiten aufgelöst werden. Es erscheint ein „Resolution Results"-Dialog, den wir mit „Finish" akzeptieren.

Jetzt wechseln wir auf den „Source"-Tab und passen die Einstellungen für den Raspberry Pi in der Instruktion „-runremote" an. Eine vollständige Liste der Bndtools-Instruktionen findet sich unter folgender Webseite *[9]* in den Kapiteln 27 und 28.

Wir müssen jetzt jedoch nur den Schlüssel „host" anpassen und den Wert auf die IP oder den Hostnamen des Raspberry-Pi-Systems setzen.

Nachdem wir diese Konfiguration vorgenommen haben, steht dem Start unserer Anwendung auf dem Device nichts mehr im Wege. Hierzu öffnen wir aus dem Menü „Run" und „Run As" den „Bnd Native Launcher". Jetzt werden unsere Bundles auf den Raspberry Pi transportiert und eine OSGi-Laufzeitumgebung gestartet. In diese Laufzeit können unsere Bundles jetzt installiert und gestartet werden. Das wird einen kleinen Augenblick dauern.

Innerhalb der IDE wird jetzt die „Console View" geöffnet und wir bekommen eine OSGi-Shell präsentiert. Es ist die OSGi-Gogo-Shell, mit der wir mit dem System inter-

Apache Felix Web Console
Bundles

APACHE felix

Main OSGi Web Console | Log out

Bundle information: 38 bundles in total - all 38 bundles active

Apply Filter | Filter All | Reload | Install/Update... | Refresh Packages

Id	Name	Version	Category	Status	Actions
0	OSGi System Bundle *(org.eclipse.osgi)*	3.10.100.v20150529-1857		Active	
1	JSON Library *(com.springsource.org.json)*	1.0.0		Active	
2	Apache Commons FileUpload *(org.apache.commons.fileupload)*	1.3.1		Active	
3	Commons IO *(org.apache.commons.io)*	2.4.0		Active	
4	Apache Felix Configuration Admin Service *(org.apache.felix.configadmin)*	1.8.6	osgi	Active	
5	Apache Felix Gogo Command *(org.apache.felix.gogo.command)*	0.14.0		Active	
6	Apache Felix Gogo Runtime *(org.apache.felix.gogo.runtime)*	0.16.2		Active	
7	Apache Felix Http API *(org.apache.felix.http.api)*	3.0.0		Active	
8	Apache Felix Http Jetty *(org.apache.felix.http.jetty)*	3.2.0		Active	
9	Apache Felix Servlet API *(org.apache.felix.http.servlet-api)*	1.1.2		Active	
10	Apache Felix Log Service *(org.apache.felix.log)*	1.0.1		Active	
11	Apache Felix Declarative Services *(org.apache.felix.scr)*	2.0.0	osgi	Active	
12	Apache Felix Web Management Console *(org.apache.felix.webconsole)*	4.2.8		Active	
13	Coordinator *(org.eclipse.equinox.coordinator)*	1.3.100.v20150410-1453		Active	
14	Event Admin *(org.eclipse.equinox.event)*	1.3.100.v20140115-1647		Active	
15	Meta Type *(org.eclipse.equinox.metatype)*	1.4.100.v20150408-1437		Active	
16	org.osgi:org.osgi.service.event *(org.osgi.service.event)*	1.3.1.201505202024		Active	

Abbildung 4: Apache Felix Web Console

```
 1#
 2# OSGI ENROUTE IOT DOMOTICA APPLICATION RUN SPECIFICATION
 3#
 4
 5
 6Bundle-Version:            1.0.0.${tstamp}
 7Bundle-SymbolicName:       osgi.enroute.iot.domotica.application.launch
 8JPM-Service:               domotica
 9
10#-runremote: \
11#   raspberry;\
12#       jdb=1044; \
13#       host=192.168.178.50; \
14#       shell=-1
15
16-runfw:                    org.eclipse.osgi
```

Abbildung 5: „osgi.enroute.iot.domotica.bndrun" für den Archiv-Export

agieren können. Eine detaillierte Beschreibung findet sich hier *[10]*. In der „ssh"- Shell sehen wir in der Ausgabe zweimal „Hello World!". Eines davon kommt aus der DomoticaApplication und eines aus dem DomoticaCommand.

In der IDE öffnen wir jetzt die Source-Datei „DomoticaCommand.java" und ändern den Wert von „Hello World" in „Hello OSGi World". Nach dem Speichern dieser Änderung findet ein sofortiges Update statt. Hierzu werden der Source-Code kompiliert, das neue Bundle gebaut, auf den Raspberry Pi transportiert, das alte Bundle gestoppt und de-installiert, das neue installiert und gestartet. Dies geschieht direkt nach dem Save aus dem Editor. Somit findet ein „Hot Code Replacement" direkt in der Remote-OSGi-Laufzeit statt. Wir sehen dies auch in der „ssh"-Shell. Sie zeigt uns ein „Goodbye World" aus dem DomoticaCommand und direkt danach wieder eine „Hello World" und „Hello OSGi World".

Dieses Vorgehen ermöglicht uns einen sehr schnellen Entwicklungszyklus direkt auf dem Zielsystem mit den exakt gleichen Bundles, die auch spätere Continuous-Integration-Prozesse erzeugen. Dies kann man sich im Ordner „generated" anschauen, weil dort nach dem Speichern immer das aktuelle Bundle erzeugt wird. Dies ist das Bundle, das auch auf das Target kopiert und dort installiert wird.

Fehleranalyse und Debugging

Das Tooling bietet uns auch verschiedenste Möglichkeiten, um Probleme zu analysieren. Um diese zu demonstrieren, stoppen wir den laufenden Prozess über den „Terminate"-Button in der „Console View". In der „ssh"-Shell sehen wir, wie in der Console ein „Goodbye world!" ausgegeben wird. Der Bndremote-Agent selbst läuft aber weiterhin und steht für neue Starts bereit.

In der IDE öffnen wir jetzt die Source-Datei „DomoticaCommand.java" und setzen einen Break-Point in der Methode „activate()". Um diesen aus der IDE heraus anzuspringen, verwenden wir aber diesmal eine andere Launch-Konfiguration namens „debug.bndrun". Dazu öffnen wir diese „bndrun"-Datei, drücken erneut den „Resolve Button" und akzeptieren das „Resolution Results"-Ergebnis. Diesmal starten wir die Launch-Konfiguration über das Menü „Run", „Debug As" und „Bnd Native Launcher".

Nun startet das System erneut und unser Break-Point wird angesprungen. Jetzt können wir den Systemzustand in der IDE wie von Java gewohnt analysieren. Wir lassen das System in unserem Fall jetzt aber einfach weiterlaufen. Die Debug-Launch-Konfiguration nutzt die vorherige und fügt zusätzliche Bundles hinzu, die eine Browser-basierte Fehleranalyse ermöglichen. Hierzu wird die Apache Felix Web Console *[11]* verwendet. Sie erlaubt, das System zur Laufzeit zu analysieren, Fehler zu diagnostizieren und auch Konfigurationen vorzunehmen.

Web Console ist ein sehr mächtiges Werkzeug, um die Komplexität von OSGi in den Griff zu bekommen, indem wichtige Informationen wie die Bundles, Services, Komponenten oder auch aktuelle Konfigurationen dargestellt werden. Um sie aufzurufen, öffnen wir einen Web-Browser und rufen die URL *„http://<RasPi_IP/Hostname>:8080/system/console"* auf. Als Zugangsdaten sind standardmäßig User „admin" und Passwort „admin" hinterlegt. *Abbildung 4* zeigt das erfolgreiche Login.

Hier können wir jetzt über die Menüpunkte verschiedenste Status-Informationen unserer OSGi-Laufzeit abrufen. Dieses Instrument sollte man bei Fehlersuchen immer in Betracht ziehen. Damit schließen wir die Entwicklung unseres Systems ab und gehen zum Deployment und Betrieb über.

Export und Betrieb

Um unsere IoT-Anwendung jetzt auf dem Zielsystem permanent laufen zu lassen, können wir natürlich nicht den Bndremote-Agent nutzen. Daher bietet Bndtools auch dafür eine Option an. Wir sind in der Lage, eine lauffähiges „.jar"-Archiv zu exportieren, das alle unsere Bundles sowie die transitiven Abhängigkeiten enthält. Dieses starten wir dann mit „java -jar <iot-app.jar>.

Hierzu können wir die existierende Bndrun-Datei „osgi.enroute.iot.domotica.bndrun" „-runremote" mit „#" vor den jeweiligen Zeilen auskommentieren. Hierbei ist darauf zu achten, dass der Backslash „\" am Zeilenende eine Fortsetzung der Zeile in der nächsten bedeutet. Somit sollte das Ergebnis wie in *Abbildung 5* aussehen.

Nachdem wir diese Zeile kommentiert haben, können wir auf dem „Run"-Tab oben rechts den „Export"-Button drücken. Im folgenden „Export Wizard Selection"-Dialog wählen wir „Executable Jar" aus und „Next". Jetzt müssen wir noch einen Pfad und Dateinamen angeben (etwa „c:\tmp\iot-app.jar") und dann mit „Finish" abschließen. Wir erhalten in dem angegebenen Pfad nun unsere ausführbare Datei, die wir auf den Raspberry Pi kopieren und dann mit „java -jar iot-app.jar" laufen lassen können.

Fazit, Danksagung und Ausblick

Damit ist unser erster Sprint erfolgreich beendet und das IoT-System am Start. Der Autor hat hoffentlich Appetit auf mehr OSGi und IoT gemacht zu haben. Die Community um OSGi und enRoute ist äußerst aktiv und hilfsbereit. Daher sollte man sich nicht scheuen, Fragen auf den äußerst aktiven Mailinglisten *[12,13]* zu stellen oder den Autor persönlich zu kontaktieren.

Bedanken möchte er sich insbesondere bei zwei Charakteren aus der Community: Neil Bartlett und Peter Kriens. Sie haben ihm mit ihrer Schulung das Tor zu OSGi geöffnet. Davor hatte er bereits sechs Jahre Erfahrung mit Eclipse und sich als Experte bezeichnet. OSGi ist aber wesentlich mehr und diese

beiden haben in Zusammenarbeit mit vielen, nicht explizit genannten, die OSGi-Welt zu einer robusten, extrem vielseitigen und zuverlässigen Umgebung gemacht.

Die Anzahl der auf OSGi basierenden Systeme wächst kontinuierlich. OSGi wird sowohl in IDEs, Webservern und Applikationsservern als auch in Embedded-, HomeAutomation- und IoT-Systemen eingesetzt. Es ist eine etablierte und über fast zwei Jahrzehnte gereifte Technologie, die es immer wieder schafft, neue Konzepte zu integrieren.

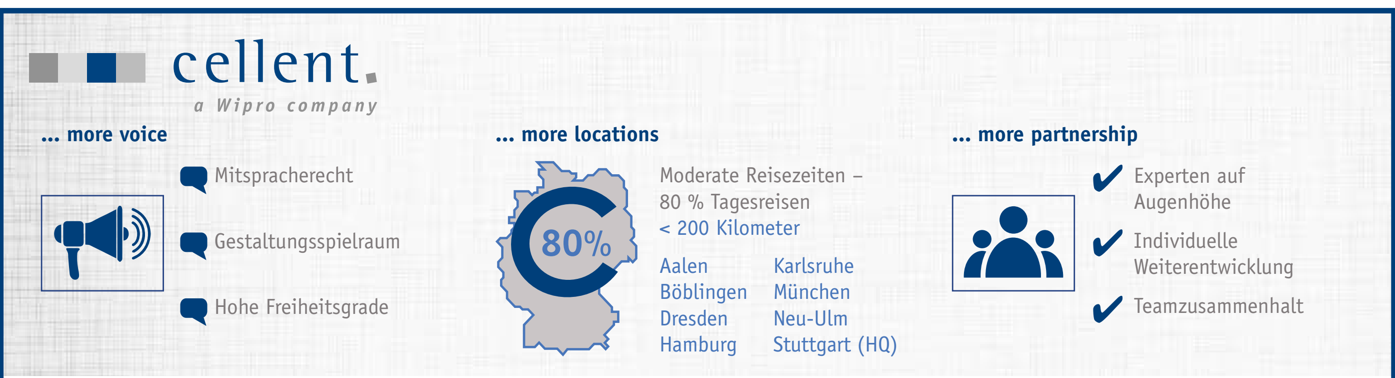
Der Autor hofft, mit diesem Artikel einen kleinen Einstieg zu ermöglichen. Falls ihm das gelungen ist, kann man auf der Webseite *„http://enroute.osgi.org"* weiter schmökern, lernen, Spaß haben sowie robuste Systeme bauen.

Links und Literatur

[1] OSGi Alliance Webseite: *https://www.osgi.org/developer*
[2] OSGi Semantic Versioning: *http://www.osgi.org/wp-content/uploads/SemanticVersioning1.pdf*
[3] PiBakery: *http://www.pibakery.org*
[4] Raspberry Pi Shop: *https://www.raspberrypi.org/products*
[5] MobaXterm: *http://mobaxterm.mobatek.net*
[6] Bndtools: *http://bndtools.org/installation.html*
[7] enRoute IoT tutorial: *http://enroute.osgi.org/tutorial_iot/140-circuits.html*
[8] IDEfix: *http://peterkir.github.io/idefix/bootstrap/conference/jfs2016*
[9] Bnd-Documentation: *http://bnd.bndtools.org*
[10] Gogo shell in OSGi enRoute: *http://enroute.osgi.org/appnotes/gogo.html*
[11] Apache Felix Web Console: *http://felix.apache.org/documentation/subprojects/apache-felix-webconsole.html*
[12] OSGi und enRoute Developer Mailing List: *https://mail.osgi.org/mailman/listinfo/osgi-dev*
[13] Bndtools user forum: *https://groups.google.com/forum/#!forum/bndtools-users*

Peter Kirschner
peter@kirschners.de
Twitter: @peterkir

Peter Kirschner hat seine Mission mit dem Commodore 64 begonnen. Er arbeitet seit vielen Jahren mit Java und OSGi als Entwickler, Architekt, Build- und Release-Engineer in der Industrie. Sein besonderes Interesse gilt nachhaltigen, Plattform-unabhängigen Software-Lösungen basierend auf OSS. Automatisierung und durchdachte Schnittstellen innerhalb des gesamten Software-Lebenszyklus sind aus seiner Sicht der Schlüssel zu einer flexiblen und robusten Software-Entwicklung.



Unser Slogan ist unser Programm. Als innovative IT-Unternehmensberatung bieten wir unseren renommierten Kunden seit vielen Jahren ganzheitliche Beratung aus einer Hand. Nachhaltigkeit, Dienstleistungsorientierung und menschliche Nähe bilden hierbei die Grundwerte unseres Unternehmens.

Zur Verstärkung unseres Teams Software Development suchen wir Sie als

Senior Java Consultant / Softwareentwickler (m/w)

an einem unserer Standorte

Ihre Aufgaben:

Sie beraten und unterstützen unsere Kunden beim Aufbau moderner Systemarchitekturen und bei der Konzeption sowie beim Design verteilter und moderner Anwendungsarchitekturen.
Die Umsetzung der ausgearbeiteten Konzepte unter Nutzung aktueller Technologien zählt ebenfalls zu Ihrem vielseitigen Aufgabengebiet.

Sie bringen mit:

- Weitreichende Erfahrung als Consultant (m/w) im Java-Umfeld
- Sehr gute Kenntnisse in Java/J2EE
- Kenntnisse in SQL, Entwurfsmustern/Design Pattern, HTML/XML/ XSL sowie SOAP oder REST
- Teamfähigkeit, strukturierte Arbeitsweise und Kommunikationsstärke
- Reisebereitschaft

Sie wollen mehr als einen Job in der IT? Dann sind Sie bei uns richtig!
Bewerben Sie sich über unsere Website: **www.cellent.de/karriere**

Graph-Visualisierung mit d3js im IoT-Umfeld

Dr.-Ing. Steffen Tomschke, B-S-S Business Software Solutions GmbH

Der Artikel stellt die visuelle Gestaltung von Graphen im Bereich „IoT" auf Basis des Frameworks d3js vor. Hierzu wird der Einsatz verschiedener Graphen gezeigt, darunter Graphen, basierend auf dem Force-Directed-Algorithmus. Darüber hinaus gibt es Beispiele, wie Anpassungen des Graphen mit d3js möglich sind.

Das Thema „Internet of Things" verbindet viele Facetten – so etwa Big Data, komplexe Netzwerke, M2M-Kommunikation, mobile Endgeräte und Exploration sowie visuelle Darstellung. Die weitreichende Vernetzung, auch über Unternehmensgrenzen hinaus, ist dabei ein zentraler Bestandteil. Graphen sind ein ideales Mittel, um Netzwerke abzubilden und die darin enthaltenen Daten darzustellen. Sie sind neben KPI-Dashboards eine sehr gute Hilfe, um Daten als Informationen visuell aufzubereiten und die Interaktion mit diesen zu ermöglichen. Die Komplexität erfordert dabei größere Darstellungsflächen oder Reduktionsmethoden und damit Abstraktionen in der Menge von Informationen. In beiden Fällen ist der Nutzer damit gezwungen, eine Exploration zu starten und die Interaktion mit dem Graphen zu beginnen.

„Information entsteht als Resultat einer Aktion und dem (erfolgreichen) Transfer von Wissen in einer konkreten Situation und verringert in seiner Wirkung beim Betrachter die Ungewissheit über eine bestimmte Struktur eines nicht erschlossenen Bereichs oder Beziehung" *[1]*.

Graph-Visualisierung, IoT, Big Data und Exploration

Der Mensch ist getrieben, zu sammeln und jegliche Dinge anzuhäufen. Begonnen hat dies mit der Bibliothek zu Alexandrien. Ein erster zentraler Ort, der das Wissen der Welt vereinen sollte. Mit dem Beginn des Internets und der Verbreitung von Smartphones in den 1990er- und 2000er-Jahren stieg die Verbreitung und Ansammlung von Wissen und Informationen.

Heutzutage werden täglich so große Datenmengen erzeugt, dass ein einzelner Mensch nicht in der Lage ist, diese im Laufe seines Lebens zu erfassen. Diese Datenflut ist in der Literatur als Big Data *[2]* beschrieben und durch die zunehmende Vernetzung von Personen und Maschinen im Zeitalter des „Internet of Things" *[3]* verstärkt. Dies bringt uns zu dem Punkt, dass die Daten gefiltert und personalisiert dargestellt werden müssen sowie eine Interaktion mit diesen ermöglicht werden soll. Daraus entstehen Ansprüche und Interessen eines jeden Einzelnen in der Wahrnehmung und im Umgang mit den Daten.

Im Internet erzeugen soziale Netzwerke und M2M-Systeme komplexe Verbindungen. Diese Relationen erzeugen weitere Daten. Damit lässt sich die Entwicklung zum einen in das Wachstum der Menge und die Komplexität von Daten und zum anderen in den Bedarf, Ordnung und Wissen aus den Daten zu bekommen, unterteilen. Ersteres wird durch Big Data und IoT erzeugt, Letzteres durch personalisierte Suche, Aggregation von Informationen und deren Aufbereitung sowie den Umgang in visuellen Interfaces.

Es bedarf der Unterstützung von intelligenten Systemen zur Aufbereitung und Konsolidierung der Informationen – die richtigen Daten zum richtigen Zeitpunkt entsprechend dem Kontext. Im Weiteren ist es nötig, Interaktion mit der komplexen Menge von Daten in Form von interaktiven Graphen zu bieten. Eine freie Exploration der gesamten Datenmenge soll stets dem Nutzer zur Verfügung stehen und durch visuelle Analyse explorierbar gemacht werden *[4]*.

Praxisbeispiel: IoT und Pharma

Im Bereich der Pharma-Industrie sind diese Punkte sehr wichtig. Viele Maschinen er-

zeugen große Datenmengen und müssen präzise aufeinander abgestimmt sein.

Die Kombination von Big Data, IoT und Graph-Visualisierung stellt sich damit folgendermaßen dar: Das IoT stellt und aggregiert die Geräte, die die Daten erzeugen, und bringt diese in einen Zusammenhang. Damit entsteht die Möglichkeit, eine komplexe und große Datenmenge zu erzeugen und zugreifbar zu machen. Die Menge der erzeugten Daten wird als „Big Data“ bezeichnet. Darauf aufbauend steht die Graph-Visualisierung zur Erfassung und Verarbeitung – Big Data wird somit begreifbar und steht zur visuell explorierbaren Analyse bereit.

Das Unternehmen des Autors bietet genau dieses Zusammenspiel von Big Data, IoT und Enterprise Search. Innovative Themen und Technologien ermöglichen tiefe Einblicke, effiziente Analysen und ermöglichen weitreichende Kooperation auf Basis von Informationen.

Diese Visualisierungen und die Analyse stehen im Pharmabereich den Mitarbeitern auf verschiedenen Ebenen mit verschiedenen Sichten zur Verfügung: KPI-basierte Graphen im Management, detaillierte Zusammenhänge zwischen einzelnen Maschinen und Prozessen für Arbeiter an den chemischen Maschinen sowie die komplexen Zusammenhänge von Molekülen und Proteinen für die Forschungsabteilungen. All dies in Form von Graphen, meist in Kombination mit zusätzlichen Informationen oder weiteren Graphen *(siehe Abbildung 1)*.

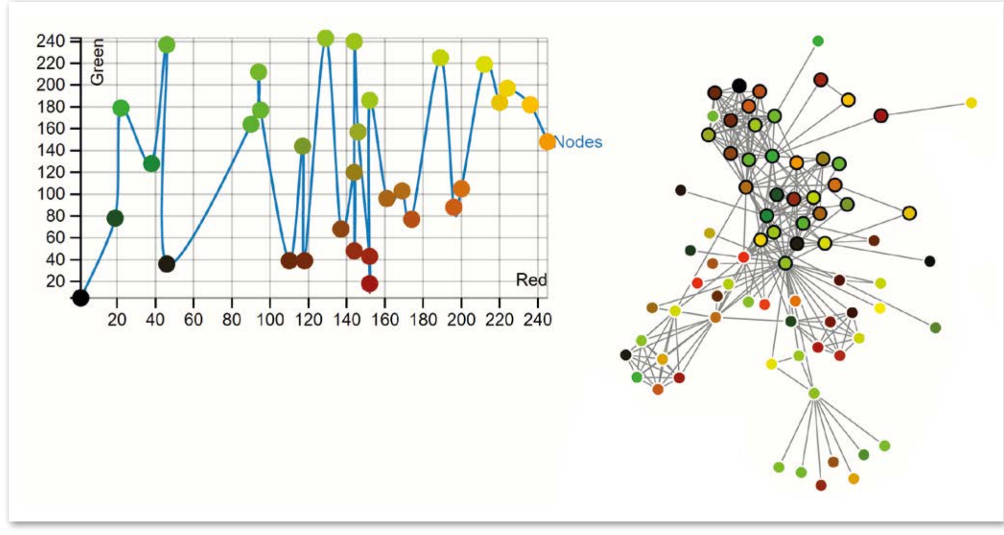


Abbildung 1: Beispiel-Graph aus dem Pharmabereich mit zusätzlicher Analyse (anonymisiert)

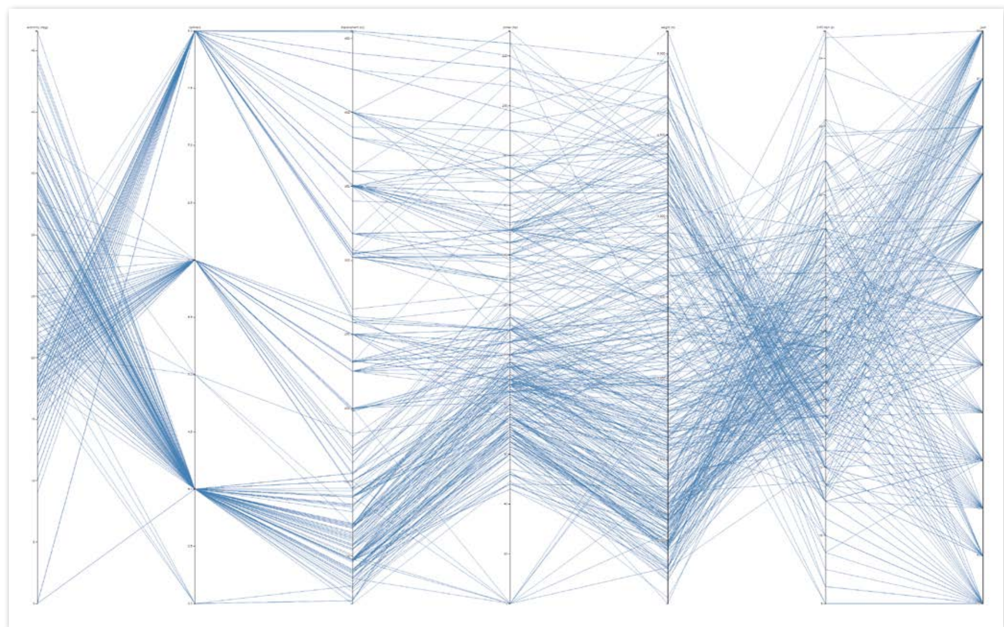


Abbildung 2: Beispiel Parallel-Koordinaten-Graph [13]

d3js als Framework

d3js ermöglicht es, große Datenmengen schnell und mit vielen Möglichkeiten interaktiv gestalten. Diese Dynamik erlaubt es dem Nutzer, mit den Daten zu arbeiten und diese nicht nur als Visualisierung zu erfassen. Die Kombination verschiedener Arten der Visualisierung in d3js bietet damit auch verschiedene, gleichzeitige Sichten auf die Daten. Ebenso wichtig wie auch notwendig beim IoT sind Live-Daten. Die hier vorgestellte Library ermöglicht die Verarbeitung von Live-Daten.

Die JavaScript Library d3.js visualisiert Datensätze und erleichtert die Erstellung von SVG-Grafiken sowie die Manipulation der DOM-Struktur. Sie basiert auf den aktuellen Web-Standards HTML5 *[5]*, CSS3 und SVG *[6]*. Sie ermöglicht verschiedene Typen von Visualisierungen. Dies reicht von Balkendiagrammen über FlowCharts bis hin zu komplexen Graphen, die auf dem Force-Directed-Algorithmus *[7]* basieren *(siehe Abbildungen 2 und 3)*. Als Ausblick für Animationen in der Visualisierung ist dieses Beispiel *[8]* sehr sehenswert; es illustriert die Mächtigkeit von d3js.

Der Zugriff auf die Elemente einer Website (innerhalb der HTML-Struktur) erfolgt mit dem bereitgestellten „selectAll()“-Befehl. Auf diese Weise können alle „<p>“-Elemente einer Website mit dem Befehl „d3.selectAll(„p“).style(„color“, „white“);“ eingefärbt werden. Insbesondere die Kombination mit Funktionen wird in dieser Library erleichtert. *Listing 1* illustriert das Einfärben jedes zweiten „<p>“-Elements.

Für „function(d, i)“ werden die beiden Parameter wie folgt verwendet: „d“ übergibt

```
d3.selectAll("p").style("color", function(d, i) {
  return i % 2 ? "#fff" : "#eee";
});
```

Listing 1

alle eingebundenen Daten als Array an die Funktion, der erste Datensatz steht also für den ersten Node zur Verfügung, der zweite für den zweiten etc. Das Beispiel in *Listing 2* erzeugt in der Website verschiedene (vorgegebene) Schriftgrößen.

Wichtige Funktionen in der Library sind die Methoden „Enter" und „Exit". Sie ermöglichen die Manipulation des Document Object Model (DOM) – es können Elemente hinzugefügt und entfernt werden. *Listing 3* fügt ein neues DOM-Element hinzu.

Wichtig ist, dass das Element „d3.selection" vorher bestimmt wird, um die korrekte Position im DOM zu lokalisieren. Selbiges gilt auch für die „Exit"-Methode: „selection.exit().remove();". Damit wird das aktuell in „d3" selektierte Element aus dem DOM entfernt. Das Beispiel in *Listing 4* zeigt die Erstellung eines einfachen Balken-Diagramms. Als Erstes wird dazu ein neues SVG-Element erzeugt.

Als Datenbasis werden statische Daten genutzt; diese können mit „var dataset = [9, 1, 5, 2];" durch dynamische Daten ersetzt werden. Wie gesagt, iteriert d3js durch das Datenset und bindet dieses an die Nodes. Damit lassen sich die Balken erzeugen *(siehe Listung 5)*.

Dies wird für die Berechnung der Position auf der x-Achse genutzt, sodass jeder Balken sichtbar ist. Anschließend erhält der Balken die Werte als Text zugewiesen *(siehe Listing 6)*. Damit sind neue Elemente entstanden, die nun an ihre Position gebracht werden müssen. Dies muss der Entwickler selbst umsetzen. Eine explizite Bindung und damit eine Vererbung der Position ist nicht vorhanden.

Auch Transitionen von Farben und Formen lassen sich sehr schnell realisieren und in die Website einbringen. *Listing 7* zeigt den Farbwechsel des Hintergrunds.

Im gewählten Szenario „IoT im Pharmabereich" ist der Zusammenhang zwischen Daten wesentlich. Diese großen und komplexen Datenmengen müssen explorierbar gestaltet werden. Wie die Exploration in komplexen Netzwerken erfolgt und welche Auswirkungen verschiedene grafische und interaktive Elemente haben, kann in *[2]* nachgelesen werden. Die dort vorgestellten Methoden sollen auch einen Ausblick geben, wie sich mit d3js die Visualisierung und Interaktion mit komplexen Graphen im IoT-Pharma-Bereich gestalten kann.

Um dem Explorationsverhalten des Nutzers gerecht zu werden und einen adäqua-

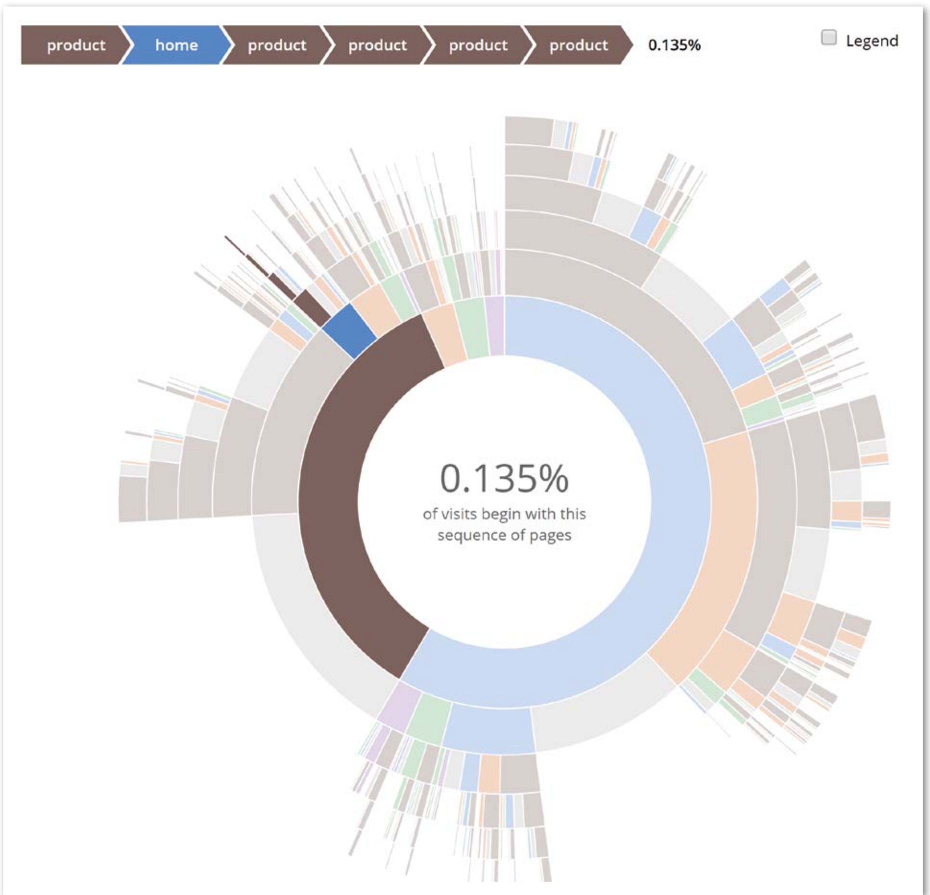


Abbildung 3: Beispiel Sunburst-Graph [14]

```
d3.selectAll("p")
  .data([4, 8, 15, 16, 23, 42])
  .style("font-size", function(d) { return d + "px"; });
```

Listing 2

```
d3.selection.enter().append("div").attr("class", "bar")
  .style("height", function(d){
     …
  })
  .style("margin-top", function(d){
     return (100—d) + "px";
  });
```

Listing 3

```
var svg = d3.select("body")
.append("svg").attr("width", w).attr("height", h);
```

Listing 4

```
svg.selectAll("rect")
.data(dataset).enter().append("rect")
.attr("y", function(d) {
  return h - (d * 2);
})
```

Listing 5

ten Start in die komplexe Darstellung zu geben, wird der Force-Directed-Algorithmus verwendet *[9]*. Dieser baut um ein Ziel-Element alle anderen Elemente des Netzwerks gleichwertig auf, auch wenn die Möglichkeit besteht, dies zu verändern. Der Kern des Force-Directed-Algorithmus *[7]* ist in d3js über das Beispiel in *Listing 8* umgesetzt. Dabei ist „force" der Aufruf für den entsprechenden Algorithmus. Er bindet die entsprechenden Attribute und lädt die Daten mit in die Visualisierung.

Form und Farbe in einem Graphen können geändert werden. Dies ist für alle Graphen in d3js gleich, die aus Knoten und Kanten bestehen. So lässt sich die Farbe der Knoten, die als Kreise gezeichnet sind, mit den Statements „enter()" und „append(„fill")" realisieren. Dies wird direkt an dem Knoten eingebunden *(siehe Listing 9)*.

```
svg.selectAll("text").data(dataset).enter()
.append("text").text(function(d) {
  return d;
})
```

Listing 6

```
d3.select("body").transition()
  .style("background-color", "black");
```

Listing 7

```
var simulation = d3.forceSimulation()
  .force("link", d3.forceLink().id(function(d) { return d.id; }))
  .force("charge", d3.forceManyBody())
  .force("center", d3.forceCenter(width / 2, height / 2));
d3.json("miserables.json", function(error, graph) {
…
function ticked() {
  link
      .attr("x1", function(d) { return d.source.x; })
      .attr("y1", function(d) { return d.source.y; })
      .attr("x2", function(d) { return d.target.x; })
      .attr("y2", function(d) { return d.target.y; });
  node
      .attr("cx", function(d) { return d.x; })
      .attr("cy", function(d) { return d.y; });
  }
}
```

Listing 8

```
var node = svg.append("g")
  .attr("class", "nodes").selectAll("circle")
  .data(graph.nodes).enter().append("circle")
      .attr("r", 5)
      .attr("fill", function(d) { return color(d.group); })
      .call(d3.drag()
          .on("start", dragstarted)
          .on("drag", dragged)
          on("end", dragended));
```

Listing 9

```
setInterval(function(){
nodes.push({
type: d3.svg.symbolTypes[~~(Math.random() * d3.svg.symbolTypes.length)],
      size: Math.random() * 300 + 100
  });
```

Listing 10

Für die Formen bringt d3js ein kleines, überschaubares Portfolio mit. Da es sich hier jedoch um SVG handelt, können diese auch schnell substituiert werden. Der Zugriff auf die Standard-Formen ist über „symbolTypes" und „shape" realisiert *(siehe Listing 10)*.

Für unseren Use Case „IoT-Pharma" ist diese Differenzierung und Kombination der Darstellungsmöglichkeiten wesentlich. Damit lassen sich die verschiedenen Facetten der komplexen Graphen leichter explorierbar und visualisierbar machen. Speziell im Bereich der Interaktion ermöglicht d3js schnelle visuelle Anpassungen. Dies ermöglicht – metaphorisch gesehen – eine direkte Interaktion mit den Daten. Hierzu verwenden wir Zoom und Pan sowie zur Unterstützung in der Exploration die Catch-Funktion.

Die Zoom-Funktion kann in d3js über „d3.zoom()" aufgerufen werden und bietet verschiedene Möglichkeiten, diese zu manipulieren und entsprechend in die Visualisierung mit einzubringen. In Kombination mit Transform und Transition ergibt der Aufruf „selection.transition().duration(750).call(zoom.transform, d3.zoomIdentity);". Eng verbunden mit dieser Funktion ist das „Panning", also das Verschieben eines Graphen auf der visuellen Oberfläche *[10]*.

Aufwändiger und nicht im Standard-Set von d3js enthalten ist die Catch-Funktion. Sie organisiert alle Objekte außerhalb des sichtbaren Bereichs an dessen Rand. Dabei werden die x-y-Attribute entsprechend auf die Koordinaten am Rand projiziert *(siehe Listing 11)*. Dies zeigt implizit, wie auch komplexere Methoden kombiniert werden können und damit die Visualisierung und Interaktion steigt.

Abschließend wird die Kombination verschiedener Graphen gezeigt. Dabei selektiert man aus einem Force-Directed-Graphen eine Teilmenge an Knoten (siehe im Beispiel „d3.select()") und nutzt sie als Input für eine neue Darstellungsform (Diagramm oder weitere Graphen). An dieser Stelle bietet d3js ausschließlich die Schnittstellen für den Datenaustausch *(siehe Abbildung 4)*.

Alternativen

Neben dem d3js-Framework gibt es auch weitere, nennenswerte Visualisierungs-Frameworks und Libraries. Dazu zählen vor allem openUI5 und vis.js. Beide stellen eine ähnliche Auswahl an Funktionen bereit. openUI ist von SAP, wird eher im Business Context für KPIs und Dashboards verwendet und liefert gleich das Responsive-Design mit.

```
for (i = 0; i < force.nodes().length; i++) {
  d = force.nodes()[i];
  if (d.fixed == 1 && !d.selected){
     var restx = d.x % 50;
     if(restx < 25){
        d.x -= restx;
     }else{
        d.x += (50 - restx);
     }
     var resty = d.y % 50;
     if(resty < 25){
        d.y -= resty;
     }else{
        d.y += (50 - resty);
     }
  }
}
d.guix = globVars.xscale(d.x);
d.guiy = globVars.yscale(d.y);
if (globVars.catchmode) {
  d.bordered=false;
  if ((d.guix < 0) || d.guix > globVars.width){
d.guix = Math.max(d.r, Math.min(globVars.width - d.r-
3, d.guix));
     d.bordered=true;
  }
  if ((d.guiy < 0) || d.guiy > globVars.height){
d.guiy = Math.max(d.r, Math.min(globVars.height -
d.r- 3, d.guiy));
     d.bordered=true;
  }
}
```

Listing 11

Personalisierte Dashboards erfüllen diesen Bedarf, indem sie mit allen den Nutzer betreffenden Systemen verbunden sind und die Daten intelligent sowie dem Profil und den Bedürfnissen der Nutzer angepasst darstellen. Diese Darstellung erfolgt in Informations-Streams und KPIs (Dashboards). Der Nutzer ist in der Lage, selbst zu bestimmen, welche Informationen er priorisiert, und bewertet diese. Vis.js hingegen ist einfach gehalten und benötigt einen höheren Aufwand für die gleiche Zielsetzung.

Der Liste können noch weitere hinzugefügt werden wie zum Beispiel morris.js, das aber von Umfang und Relevanz her für den Business Context nicht hinreichend ist.

Fazit

d3js stellt für die Exploration und Interaktion mit komplexen Daten im IoT-Pharma-Kontext eine sehr gute und schnell umsetzbare Visualisierungs-Library bereit. Sie lässt sich gut erweitern und bringt viele Funktionen und Darstellungsformen mit. Mit den aktuellen Webstandards als Basis lässt sie sich auch auf verschiedenen Endgeräten benutzen. Mehr zu diesem Thema ist auf *[11]* zu lesen sowie in dem dazugehörigen Vortrag auf der JUG Saxony *[12]*.

Referenzen

[1] Mark Weiser, The Computer for the 21st Century: *http://wiki.daimi.au.dk/pca/_files/weiser-orig.pdf*
[2] Tomschke, S. (2014), Visualisierungs-und Interaktionskonzept zur graphenbasierten Exploration: *http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-192636*
[3] Open Web Platform Milestone Achieved with HTML5 Recommendation. W3C: *http://www.w3.org/2014/10/html5-rec.html.en*
[4] President's Council of Advisors for Science and Technology, Big Data: Seizing Opportunities, Preserving Values, Executive Office of the President, Mai 2014
[5] HTML 4.0 Specification, W3C Recommendation. Appendix A: Changes between HTML 3.2 and HTML 4.0, A.1.2 Deprecated elements. W3C: *http://www.w3.org/TR/REC-html40-971218/appendix/changes.html#h-A.1.2*
[6] SVG2 Planning Page: *http://www.w3.org/Graphics/SVG/WG/wiki/index.php?title=SVG2_Planning_Page&oldid=6876*
[7] Fruchterman, T. M., & Reingold, E. M. (1991), Graph drawing by force-directed placement. Practice and experience
[8] Dynamische Visualisierungen: *http://bl.ocks.org/mbostock/1256572*
[9] *https://bl.ocks.org/mbostock/4062045*
[10] *https://github.com/d3/d3/blob/master/API.md#zooming-d3-zoom*
[11] Tomschke, S., Graph-Gestaltung mit d3js im IoT-Umfeld, *http://www.bigdata-unleashed.com/20160921/graph-gestaltung-mit-d3js-im-iot-umfeld*
[12] JUG Saxony, Tomschke, S. (2016), Graph-Gestaltung mit d3js im IoT-Umfeld: *https://jugsaxony.org/veranstaltungen/80/*
[13] d3js: *http://bl.ocks.org/syntagmatic/2409451*

Dr.-Ing. Steffen Tomschke
steffen.tomschke@b-s-s.de

Mit mehr als zehn Jahren Erfahrung als Projektmanager und UX-Consultant sowie mehr als fünf Jahren in der Forschung zur Exploration und Visualisierung von Graphen bei der SAP SE berät Dr. Steffen Tomschke verschiedene Großkonzerne im Pharma- und Energie-Sektor. Als Teamlead und UX-Consultant bei der B-S-S ist er in den Bereichen „Information Architecture", „Big Data & IoT" sowie „Enterprise Search" tätig.

DevCamp 2017
8. Februar in Hannover

devcamp.doag.org

Erste Schritte mit Kotlin

Dirk Dittert, 24objects GmbH

Entwickler wünschen sich eine ausdrucksstarke Programmiersprache, um die Lesbarkeit in den Vordergrund zu stellen und möglichst wenig Tipparbeit erledigen zu müssen. Kotlin bietet all das für die JVM. Dieser Artikel erklärt den aktuellen Status der Programmiersprache und gibt einen Einblick in praktische Features.

Kotlin ist eine statisch typisierte Sprache für die Entwicklung von Programmen, die sowohl auf der Java Virtual Machine als auch auf JavaScript Engines ausgeführt werden können. Durch ihr modernes und objektorientiertes Design kann sie von Java-Entwicklern schnell erlernt werden. Sie berücksichtigt auch funktionale Aspekte, beispielsweise dadurch, dass Funktionen vollwertige Sprachelemente sind.

Der Schwerpunkt der Sprache liegt auf der Produktivität des Entwicklers, mit dem Ziel, möglichst viel des ausschweifenden Boilerplate-Codes einzusparen, für den Java berühmt-berüchtigt ist. Üblicherweise ist Kotlin-Code bei gleicher Funktionalität um 20 bis 30 Prozent kürzer, ohne dabei seine gute Lesbarkeit zu verlieren.

Um den reibungslosen Einsatz in großen Software-Projekten zu ermöglichen, ist Kotlin-Code problemlos mit bereits bestehendem Java-Code kombinierbar. Somit können aus Java und Kotlin erzeugte Klassen in beiden Richtungen nahtlos zusammenarbeiten, ohne dass entsprechende Wrapper-Klassen gebaut werden müssen. Kotlin greift ebenfalls auf die Collection-Klassen des JDK zurück, APIs sind also auf beiden Seiten kompatibel.

Die Sprache und auch die zugehörigen Werkzeuge (Compiler-, Maven- und Gradle-Plug-ins, DIE-Plug-ins) werden von JetBrains entwickelt. Kotlin ist Open Source und steht unter der Apache-2-Lizenz. Die Spezifikationen für die Weiterentwicklung werden öffentlich unter Mitarbeit der Community über den Kotlin Evolution and Enhancement Process (KEEP) in einem GitHub Repository erarbeitet *(siehe Kasten auf Seite 25)*.

Erste Schritte

Wer in Entwicklung mit Kotlin hineinschnuppern möchte, kann dies ohne Installationsaufwand online über den Browser tun: Auf der Webseite *„http://try.kotlinlang.org"* kann man die Sprach-Features in einer Testumgebung ausprobieren *(siehe Abbildung 1)*. Dort werden viele Möglichkeiten der Sprache in ausführbaren Syntax-Beispielen erklärt. Über die Kotlin Koans, eine Reihe von 42 Übungsaufgaben, kann man die eigenen Sprachkenntnisse danach weiter vertiefen. Hartgesottene Entwickler haben hier auch die Möglichkeit, die Aufgaben des Advent of Codes *[1]* zu lösen.

Da hinter der Sprache JetBrains steht, ist die Unterstützung in deren IDEs natürlich am weitesten fortgeschritten. Das Kotlin-Plug-in ist sowohl in der kostenlosen IntelliJ Community Edition als auch in Android Studio enthalten, sodass man mit beiden IDEs sofort mit eigenen Projekten loslegen kann. Dort ist auch ein gut funktionierender Konverter enthalten, um Java-Code nach Kotlin zu konvertieren.

Obwohl die Sprache noch verhältnismäßig jung ist, wird sie bereits von den anderen großen Entwicklungsumgebungen unterstützt: Es gibt ein Plug-in für Eclipse *[2]* und eines für NetBeans *[3]*. Ihr Funktionsumfang ist momentan allerdings noch etwas eingeschränkt.

Einsatzmöglichkeiten

Die Sprache hat sich in letzter Zeit vor allem im Android-Umfeld einen guten Ruf erarbeitet, weil die Entwickler dort auf vielen Geräten auf die Sprachmittel von Java 6 beschränkt sind. Kotlin ist unter dieser Java-Version voll lauffähig und bietet mit Anko eine praktische DSL für die Entwicklung von Android-Applikationen. Mit nur knapp 900 KByte Größe und weniger als 5.000 Methoden stellt die Standardbibliothek eine vergleichsweise kleine Belastung für die Android-Runtime dar.

Kotlin wird jedoch auch in anderen Bereichen aktiv eingesetzt, so lassen sich beispielsweise Spring-Boot-Anwendungen neben Java auch direkt in Kotlin erstellen. Da sowohl Maven als auch Gradle als Buildsysteme unterstützt sind, kann die Sprache auch in bestehenden Projekten problemlos eingesetzt werden. Vor Kurzem kündigte Gradle an, Kotlin als vollwertigen Ersatz für die Skriptsprache Groovy zur Erstellung von Buildskripten einsetzbar zu machen.

Auch für Anhänger des Test Driven Development gibt es bereits mehrere aktiv entwickelte Frameworks. Mit Spek und KotlinTest stehen angepasste Frameworks bereit, während HamKrest eine Portierung der für JUnit beliebten Matcher darstellt.

Highlights für die JVM

Der Schwerpunkt des Artikels liegt auf der Nutzung der Java Virtual Machine. Auch wenn der Sprachkern für beide Ziel-Plattformen identisch ist, gibt es bei der momentan noch experimentellen Kompilierung nach JavaScript einige Besonderheiten zu beachten. Durch konzeptionelle Unterschiede (wie Threads, RMI, Serialisierung auf der JVM und verschiedene Modulsysteme wie CommonJS auf den JavaScript Engines) wird hier aus Platzgründen nur auf die JVM-Seite eingegangen.

Noch ein Wort zur Kompatibilität: Einige Design-Entscheidungen von Java haben sich in den letzten Jahren als wenig vorteilhaft herausgestellt. Kotlin versucht hier eine Balance zwischen Kompatibilität und Weiterentwicklung herzustellen: Kotlin kennt keine Checked Exceptions und es wird auch von der Verwendung von Arrays abgeraten, da sich diese anders als Collections verhalten. Auch die für Neueinsteiger verwirrende Unterscheidung zwischen „int" und „Integer" entfällt in Kotlin. Es gibt allerdings immer Möglichkeiten, zu bestehendem Java-Code kompatibel zu bleiben.

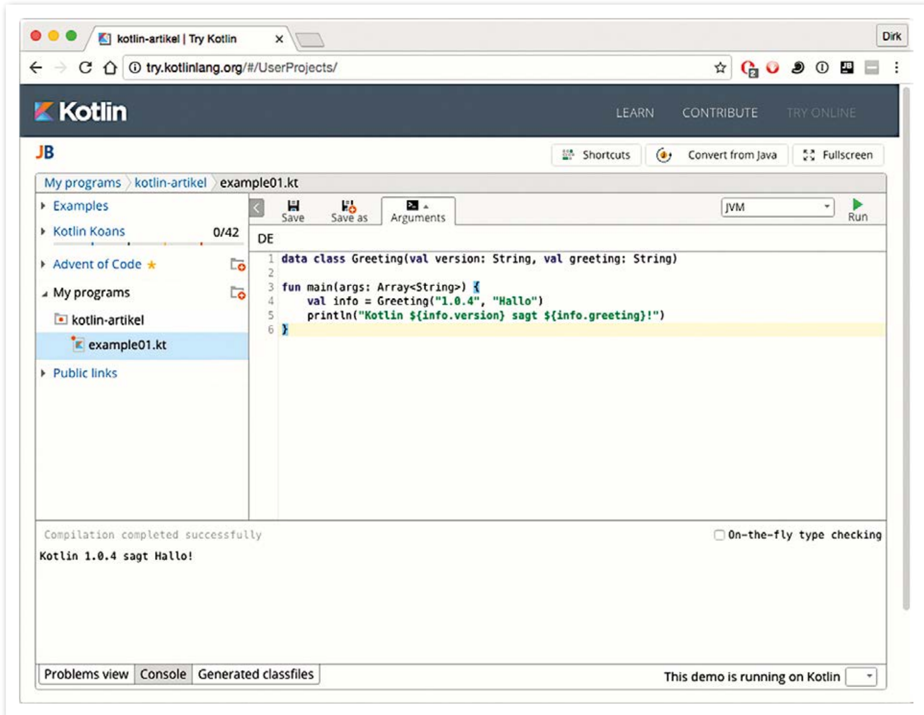


Abbildung 1: Die Seite zum Ausprobieren

```
data class Greeting(val version: String = "1.0.4", val greeting: String)

fun main(args: Array<String>) {
  val info = Greeting(greeting = "Hallo")
  println("Kotlin ${info.version} sagt ${info.greeting}!")
}
```

Listing 1

```
var nullableString: String? = null
 var nonNullableString: String = "Hello world!"
```

Listing 2

```
@NotNull
public String repeat(@NotNull String str) {
  Objects.requireNonNull(str);
  return str + str;
}
```

Listing 3

Produktivität

Das „HelloWorld"-Beispiel vermittelt einen ersten Eindruck von der Sprache *(siehe Listing 1)*. Dem Java-Entwickler fällt sofort auf, dass Kotlin am Ende von Statements keine Strichpunkte benötigt. Das Schlüsselwort „new" entfällt ebenfalls. Auch ist bei der Deklaration von Variablen der Typ auf der linken Seite nicht zu wiederholen. Das verbessert vor allem die Lesbarkeit von generischem Code deutlich. Die Typen stehen in Kotlin immer nach der Variable oder am Ende der Funktions-Deklaration. Generell verwendet Kotlin eigene Typen für grundlegende Datentypen (wie „Int") oder auch für Collections (wie „MutableList"). Diese werden jedoch vom Compiler auf plattformspezifische Typen abgebildet, sodass die Kompatibilität mit bestehendem Java-Code gewährleistet bleibt.

Unveränderliche Variable sind mit dem Schlüsselwort „val" deklariert, was gleichbedeutend mit der Deklaration als „final" in Java ist. Für veränderbare Variablen wird hingegen das Schlüsselwort „var" verwendet.

Während Java das Überladen von Methoden verwendet, um verschiedene Aufruf-Alternativen zu ermöglichen, greift Kotlin dafür auf benannte Parameter in Verbindung mit Vorgabewerten zurück. Parameter können beim Aufruf von Funktionen, Methoden oder Konstruktoren direkt über ihren Namen statt wie in Java nur über ihre Reihenfolge angesprochen werden (im Beispiel „greeting"). Sie können außerdem bei der Deklaration bereits mit Vorgabewerten belegt werden, sodass sie dann beim Aufruf nicht

```
val firstExecution: LocalTime? = ...
// Compilerfehler, da firstExecution möglicherweise null sein kann
println(firstExecution.hour)
if (firstExecution != null) {
  // ok (da firstExecution zuvor geprüft und als val deklariert wurde)
  println(firstExecution.hour)
}
```

Listing 4

```
class Person(val lastname: String, val firstname: String, var age: Int?) {
  val name by lazy { "$lastname, $firstname" }
}
```

Listing 5

```
val p1 = Person("Mustermann", "Max", 42)
val first = p1.component2()
val p2 = p1.copy(firstname = "Thomas")
assert(p1 != p2)
assert(p2.firstname == "Thomas")
```

Listing 6

```
fun countItems(list: List<Item>): Int {
  // ...
}
```

Listing 7

angegeben werden müssen (im Beispiel „version"). Die aus Java bekannten leeren „Weiterleitungsmethoden" können dadurch in Kotlin vollständig entfallen.

Wie im Beispiel am Einstiegspunkt „main" des Programms zu erkennen ist, sind Funktionen (durch das Schlüsselwort „fun" gekennzeichnet) vollwertige Sprachelemente, die auch ohne umgebende Klasse existieren können. Natürlich ist dies nur innerhalb von Kotlin-Code möglich, da die JVM-Methoden nur innerhalb von Klassen erlaubt sind. Der Compiler generiert für freie Funktionen auf der JVM automatisch Klassen, die diese aufnehmen. So können sie dann auch aus Java-Code heraus als statische Methoden aufgerufen werden. Innerhalb von Klassen sind Funktionen wie erwartet auf Methoden abgebildet.

Funktionen, die nur aus einfachen Ausdrücken bestehen, können in Kotlin übrigens sehr kurz formuliert werden: „fun isEven(i: Int) = if (i.mod(2) == 0) „gerade" else „ungerade"". Im Beispiel können die aus Java bekannten „return"-Anweisungen entfallen. Der Compiler leitet sogar den Typ des Rückgabewerts automatisch als „string" ab. Dies soll Entwickler dazu anhalten, Logik im Code in kurze Funktionen mit möglichst sprechenden Namen auszulagern.

Wie man im Beispiel sieht, beherrscht Kotlin die direkte Einbettung von Variablen oder Ausdrücken in Zeichenketten, wie man es etwa aus Groovy kennt. Die Möglichkeit, dass Zeichenketten sich über mehrere Zeilen erstrecken können, ist ebenfalls sehr praktisch.

null

Während „null" in Java eine mögliche Belegung einer Objekt-Referenz ist, ist es in Kotlin fester Bestandteil des Typ-Systems. Kann einer Variable der Wert „null" zugewiesen werden, wird dies in ihrem Typ durch ein „?" markiert, andernfalls ist dies nicht möglich (siehe Listing 2).

Der Compiler fügt automatisch an allen Stellen, an denen möglicherweise „null" an Funktionen, Methoden oder Konstruktoren übergeben werden kann, entsprechende Prüfungen und Annotationen ein. „fun repeat(str: String) = str + str" entspricht also in Java dem Beispiel in Listing 3.

Die Annotationen ermöglichen durch statische Code-Analyse, dass auch bei Aufrufen aus Java-Code heraus nur gültige Werte übergeben werden können. Wenn eine bestehende Code-Basis nicht mit den entsprechenden Annotationen versehen ist, wählt Kotlin für Aufrufe einen pragmatischen Ansatz: Der Code wird genauso behandelt, wie dies auch der Java-Compiler tun würde. Mögliche Verletzungen durch Java-Code werden also weniger streng geahndet und können wie bisher zu Laufzeit-Fehlern führen. Die Praxis hat gezeigt, dass bestehender Code nur so schrittweise migriert werden kann.

An dieser Stelle ist auch eine Reihe von Abkürzungen hilfreich: „str!!.toUpperCase().length" ergibt entweder die Länge der in Großbuchstaben umgewandelten Zeichenkette oder scheitert mit einer Exception. Durch „!!" wird für alle nachfolgenden Aufrufe ein Wert ungleich „null" erzwungen.

Ebenso kann der Aufruf auf den Fall begrenzt werden, dass tatsächlich ein Wert ungleich „null" vorhanden ist: „str?.toUpperCase()?.length" steht entweder für die Länge oder für den Wert „0" (also für den Initialwert des Typs „Int"). Wer schon mit Groovy gearbeitet hat, dem dürften diese Schreibweisen nicht ganz unbekannt sein.

Im Gegensatz zu Java kann der Kotlin-Compiler den Kontrollfluss auswerten, um den Typ einer Variablen möglichst genau einzugrenzen. Dadurch können viele der in Java üblichen Casts entfallen (siehe Listing 4). Diese automatischen Konvertierungen funktionieren übrigens auch bei regulären Typ-Prüfungen durch „is", dem Äquivalent zu „instanceof" in Java.

Properties und Data Classes

Properties funktionieren analog zu anderen Sprachen und vereinfachen die Entwicklung von Java-Beans enorm. Als Entwickler definiert man nur noch die eigentlichen Eigenschaften einer Klasse und der Compiler generiert automatisch die in Java üblichen Getter und Setter (siehe Listing 5).

Im Beispiel erzeugt der Compiler Getter und Setter für die Felder „name", „lastname", „firstname" und „age". Bei Bedarf können diese durch eigenen Code überschrieben werden. Die eigentliche Implementierung der Getter und Setter für „name" wird hier als Beispiel an eine Funktion der Standard-Bibliothek delegiert, die den Wert erst beim ersten Zugriff berechnet. Dieses praktische Feature ermöglicht die Wiederverwendung komplexer Logik aus Gettern und Settern.

Data Classes repräsentieren Wert-Objekte und reduzieren den nach Schema F

produzierten Code noch weiter. Ergänzt man im obigen Beispiel am Anfang das Schlüsselwort „data", dann generiert der Compiler zusätzlich alle üblichen Objektmethoden (also einen Konstruktor, „equals()", „hashCode()" und „toString()") sowie einige Kotlin-spezifische Methoden zum Zugriff auf die einzelnen Bestandteile der Data Class („component1()", „component2()" etc.). Hinzu kommt noch eine Methode „copy" zum Kopieren des Wert-Objekts. Damit entsprechen Data Classes den Case Classes in Scala beziehungsweise der „@Data"-Annotation des Lombok-Projekts *(siehe Listing 6)*.

Data Classes verhalten sich in vielen Bereichen wie normale Klassen, sodass sie durch beliebige eigene Methoden erweitert oder generierte Methoden durch eigene Implementierungen überschrieben werden können.

Funktionen und Erweiterungsfunktionen

Funktionen sind eigenständige Sprachelemente und können in Kotlin auch ohne umgebende Klasse existieren. Sie lassen sich sogar innerhalb von anderen Funktionen deklarieren, was besonders für rekursiven Code praktisch sein kann *(siehe Listing 7)*.

Erweiterungsfunktionen sind ein Kotlin-Sprachmittel zur Erweiterung beliebiger Typen um eigene Methoden. Dies funktioniert mit allen Typen, auch solchen, die als „final" gekennzeichnet sind. Da dies natürlich innerhalb der JVM technisch nicht möglich ist, wird dieser Funktionstyp intern auf statische Methoden abgebildet. Diese Vorgehensweise ermöglicht es außerdem, dass in Kotlin definierte Erweiterungsfunktionen leicht aus Java heraus aufgerufen werden können. *Listing 8* zeigt ein Beispiel für die Deklaration einer Erweiterungsfunktion zum Vertauschen von zwei Elementen einer Liste. Der Aufruf der Erweiterungsfunktion kann dann wie der Aufruf regulärer Objekt-Methoden erfolgen *(siehe Listing 9)*.

Im Gegensatz zu den impliziten Konvertierungen in Scala werden Erweiterungsfunktionen in Kotlin nicht anhand von vordefinierten Suchregeln herangezogen, sondern müssen vor der Verwendung explizit importiert werden. Technisch bedingt werden sie immer statisch aufgelöst, bei Mehrdeutigkeiten erfolgt also die Auswahl durch den deklarierten Typ der Objekt-Instanz. Erweiterungsfunktionen sind also statischen Methoden in Java sehr ähnlich.

Praktische Idiome

Bei der Definition der Erweiterungsfunktion sieht man, dass die jeweilige Instanz der Liste mit „this" angesprochen werden kann. Dieses Konzept trifft man auch bei den Closures wieder an. Auch hier kann man als Entwickler festlegen, in welchem Kon-

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
  val tmp = this[index1] // 'this' entspricht der Instanz der Liste
  this[index1] = this[index2]
  this[index2] = tmp
}
```

Listing 8

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2)
```

Listing 9

```
val s = with(StringBuilder()) {
  append("Starting ")
  append("Countdown ")
  (10 downTo 1).joinTo(this, ", ")
  toString()
}
```

Listing 10

```
val b = JButton().apply {
  isVisible = true
  text = "Hello!"
  minimumSize = Dimension(100, 20)
  addActionListener {
      println("Button was clicked")
  }
}
```

Listing 11

```
findButton()?.let { btn ->
  btn.isVisible = true
  btn.text = "Hello!"
  btn.minimumSize = Dimension(100, 20)
  btn.addActionListener {
      println("Button was clicked")
  }
}
```

Listing 12

```
async {
  val original = asyncLoadImage("image.png") // erzeugt ein Future
  val overlay = asyncLoadImage("overlay.png") // erzeugt ein Future

  // Ausführung wird während dem Ladevorgang der Bilder unterbrochen:
  // Die beiden await Aufrufe blockieren, bis Ladevorgänge abgeschlossen
wurden.
  return applyOverlay(await(original), await(overlay))
}
```

Listing 13

Kotlin im Internet

Die Entwicklung von Kotlin wird von JetBrains offen im Internet vorangetrieben. Die aktuellen Sourcen finden sich auf GitHub *[4]*. Erweiterungsvorschläge für die Sprachen werden ebenfalls auf GitHub im Rahmen von KEEP *[5]* erarbeitet. Für Fragen und Diskussionen stehen sowohl ein lebhafter Slack Channel *[6]* als auch ein Forum *[7]* zur Verfügung. Fehler und Probleme mit der Sprache oder dem Compiler können über einen öffentlichen Issue Tracker *[8]* gemeldet werden.

text der Code ausgeführt werden soll. Dies ermöglicht eine Reihe von praktischen Idiomen („with", „apply" sowie „let") und ist auch bei der Entwicklung von Domain Specific Languages extrem hilfreich. Soll eine Instanz neu erzeugt und mit mehreren Aufrufen passend konfiguriert werden, bietet sich die Verwendung von „with" an *(siehe Listing 10)*.

Im Beispiel wird zuerst ein „StringBuilder" erzeugt und als Parameter übergeben. Alle Aufrufe innerhalb des Lambda-Ausdrucks beziehen sich nun auf diesen Builder, sodass damit mehrere Zeichenketten zusammengefügt werden können. Am Ende wird der Inhalt des Builders über einen Aufruf von „toString()" in ein „String"-Objekt konvertiert und der Variablen „s" zugewiesen. Ähnlich funktioniert „apply", das aber Code auf bereits bestehende Objekte anwendet *(siehe Listing 11)*. Ebenso praktisch ist „let", das einem ermittelten Wert einen Gültigkeitsbereich zur Verfügung stellt, ohne dass man dafür eine eigene Variable deklarieren muss *(siehe Listing 12)*. Im Beispiel wird durch die Methode „findButton()" eine Schaltfläche anhand von bestimmten Kriterien gesucht und zurückgegeben. Diese Schaltfläche wird innerhalb der Closure unter dem Namen „btn" angesprochen. Durch die Verwendung von „?." wird der Code nur dann ausgeführt, wenn von der Methode auch ein Wert ungleich „null" zurückgegeben wird.

Ausblick auf die Version 1.1

Die Entwicklung von Kotlin gliedert sich im Moment in zwei wesentliche Stränge. Mit mehr als 1.000 Bugfixes wurden viele Verbesserungen und Rückmeldungen der Community im stabilen 1.0.x-Entwicklungsstrang des Compilers und der Entwicklungswerkzeuge eingearbeitet. Hier merkt man ganz deutlich, dass diese Sprache nicht nur ein theoretisches Konstrukt ist, sondern auch aktiv innerhalb von JetBrains eingesetzt und durch ein Team von 35 Entwicklern weiterentwickelt wird. Parallel dazu wird momentan an der Version 1.1 gearbeitet, die neue Sprach-Features enthalten wird.

Eine wesentliche neue Funktionalität wird die Integration von Ko-Routinen sein. Darunter versteht man Code, dessen Ablauf bei der Ausführung unterbrochen werden kann, ohne dass der Zustand verloren geht. Dies ist beispielsweise bei asynchronem IO hilfreich. In *Listing 13* werden zwei Bilder (das eigentliche Bild und beispielsweise ein Logo zur Überlagerung) asynchron geladen.

Durch die Integration der Ko-Routinen in die Sprache kann sich der Entwickler auf die eigentliche Geschäftslogik und die zugehörige Fehlerbehandlung konzentrieren, während die korrekte Implementierung der asynchronen Ausführung durch den Compiler und die Laufzeit-Bibliothek sichergestellt ist. Das Besondere an der Implementierung in Kotlin wird sein, dass der Compiler nur den Rahmen für die Integration asynchroner APIs zur Verfügung stellt, die eigentliche Implementierung aber durch verschiedene Frameworks möglich sein wird. Dadurch kann die asynchrone Ausführung beispielsweise durch verschiedene Future-Implementierungen oder asynchrone APIs (wie NIO) übernommen werden.

Neben den Ko-Routinen wird eine Reihe von kleineren Verbesserungen und Erweiterungen mit Version 1.1 Einzug in Kotlin halten. So sollen delegierte Properties nicht nur für Klassen-Variablen, sondern auch innerhalb beliebiger Code-Blöcke möglich sein.

Die Deklaration von Data Classes ist in Version 1.0 stark reglementiert, sodass sie beispielsweise bisher nicht von anderen Klassen abgeleitet werden können. Diese Einschränkungen werden mit Version 1.1 aufgehoben. Darüber hinaus sind dann auch Java 7 und Java 8 offiziell voll unterstützte Plattformen. Dadurch können Default-Methoden in Kotlin-Interfaces enthalten sein. Die Kompatibilitäts-Bibliothek, die in Version 1.0 zur Nutzung der Java-7- und Java-8-APIs notwendig war, entfällt dann.

Mit Version 1.1 soll auch das JavaScript-Backend voll unterstützt werden und besser strukturierten Code generieren. Nachdem bereits mit CommonJS und AMD Unterstützung für JavaScript-Modulsysteme integriert wurde, soll mit dieser Version das Zusammenspiel mit bestehendem JavaScript-Code noch weiter verbessert werden.

Quellen

[1] Advent of Code ist eine Serie von 25 weihnachtlichen Übungsaufgaben, mit denen man seine Fingerfertigkeiten in der Programmierung und dem Verständnis verschiedener Algorithmen unter Beweis stellen kann. Vorsicht: Suchtgefahr! *http://adventofcode.com*
[2] Eclipse Plug-in für Kotlin: *https://github.com/JetBrains/kotlin-eclipse*
[3] NetBeans Plug-in für Kotlin: *https://github.com/JetBrains/kotlin-netbeans*
[4] *https://github.com/JetBrains/kotlin*
[5] *https://github.com/Kotlin/KEEP*
[6] *https://kotlinlang.slack.com*
[7] *https://discuss.kotlinlang.org*
[8] *https://youtrack.jetbrains.com/issues/KT*

Dirk Dittert
dirk.dittert@24objects.de

Dirk Dittert ist Senior-Consultant bei der 24objects GmbH, einem Beratungsunternehmen mit Sitz in Nürnberg. Er ist Java-Entwickler der ersten Stunde und gibt Kunden die richtigen Werkzeuge zur erfolgreichen Umsetzung ihrer Projekte an die Hand. Immer auf der Suche nach Verbesserungsmöglichkeiten ist er bei Kotlin von Anfang an mit Interesse dabei. Er hält Schulungen zu Test Driven Development und Clean Code. Außerdem organisiert er die JUG Nürnberg.

Neun Gründe, warum sich der Einsatz von Kotlin lohnen kann

Alexander Hanschke, techdev Solutions GmbH

Die Programmiersprache Kotlin erfreut sich zunehmenden Interesses, nachdem im Februar dieses Jahres die Version 1.0 erschienen ist. Dies ist kaum verwunderlich, da Kotlin überall dort eingesetzt werden kann, wo eine JVM verfügbar ist, und zusätzlich auch zu JavaScript kompilieren kann.

Kotlin hat den Anspruch, eine pragmatische Alternative zu den etablierten Sprachen zu sein, und hat sich durch geschicktes Cherry-Picking der Features von den Platzhirschen bedient. Wer aus der Java-Welt kommt, erkennt schnell die Vorzüge, die Kotlin bietet; das Wegfallen von Semikolons oder die Type Inference sind dabei noch die kleineren Vorteile. Nachfolgend sind neun Features vorgestellt, die den Einsatz von Kotlin lohnend machen.

Strings

Kotlin verwendet konsequent Objekte und unterstützt die gängigen Typen wie Strings. In Java gestaltet sich das Arbeiten mit Strings oft etwas umständlich, besonders wenn diese sehr lang werden oder selbst Variable verwenden. In der Regel sind Strings mittels „+" konkateniert. Darüber hinaus ist auch der Gebrauch eines „StringBuilder" beziehungsweise „StringBuffer" üblich. Kotlin bietet für die genannten Szenarien eine prägnante Alternative *(siehe Listing 1)*.

Strings können wie üblich durch entsprechende Literale erzeugt werden. Zusätzlich lassen sich innerhalb von Strings weitere Variable mittels „$" referenzieren. Auch komplexe Ausdrücke sind so direkt auswertbar. Allerdings sind diese dann in geschweifte Klammern „${..}" zu setzen.

Die Syntax erinnert an Template Engines wie Thymeleaf oder Velocity. Anders als diese sind die Ausdrücke in Kotlin aber typsicher. Das heißt, der Compiler verifiziert noch vor Ausführung des Programms, dass die verwendeten Ausdrücke valide sind.

Erzeugt man ein String Literal mit jeweils drei Anführungszeichen, sind Backslashes nicht mehr nötig. Zusätzlich kann der String sich über mehrere Zeilen erstrecken, ohne dass es zusätzlicher Konkatenation bedarf *(siehe Listing 2)*.

Klassen und Properties

Klassen sind in Kotlin wie gewohnt mit dem Schlüsselwort „class" deklariert. Anders als in Java sind Klassen in Kotlin per Default „public" und „final". Beispiel: „class Person". Die Deklaration der Klasse „Person" ist gültig, bis hierher jedoch wenig nützlich. Kotlin erlaubt das Initialisieren des primären Konstruktors direkt in der Klassen-Deklaration: „class Person(val name: String, var age: Int)".

Sollen die Sichtbarkeit des Konstruktors verändert oder Annotationen verwendet werden (wie „@Inject"), so muss zusätzlich das Schlüsselwort „constructor" angegeben sein. Die Klasse „Person" ist nun um die Properties „name" und „age" erweitert. Beide sind im Konstruktor deklariert und mit „val" (von „value") beziehungsweise „var" (von „variable") gekennzeichnet. Mit „val" gekennzeichnete Referenzen entsprechen dem aus Java bekannten „final". Eine Neuzuweisung ist nach der Initialisierung nicht mehr möglich.

Sollen Referenzen auch im Nachhinein noch veränderlich sein, kann stattdessen das Schlüsselwort „var" zum Einsatz kommen. Um die Klasse zu verwenden, reicht ein Aufruf *(siehe Listing 3)*. In Kotlin ist das Schlüsselwort „new" zum Erzeugen neuer Objekte nicht nötig. Für das Lesen und Zuweisen von Properties wird die Punktnotation verwendet.

Funktionen

Funktionen werden in Kotlin durch das Schlüsselwort „fun" eingeleitet, gefolgt vom Namen der Funktion, den Parametern sowie dem Rückgabetyp. Analog zu Java ist der Einstiegspunkt in ein Kotlin-Programm die Funktion „main" *(siehe Listing 4)*.

„Unit" ist vergleichbar mit dem aus Java bekannten „void" und kann als Rückgabetyp auch weggelassen werden, da es der Default ist. Funktionsparameter lassen sich mit Werten vorbelegen, die zur Anwendung kommen, sofern sie beim Aufruf nicht überschrieben werden: „fun url(protocol: String = „http", domain: String, port: Int = 80) = „$protocol://$domain:$port"". Die Funktion „url" erzeugt aus den Parametern „protocol", „domain" und „port" eine vollständige URL, wobei nur die Domäne explizit anzugeben ist, da für Protokoll und Port bereits Werte vorgegeben sind.

Anders als die Funktion „main" verwendet „url" eine kompaktere Syntax. Statt Rückgabetyp und geschweiften Klammern wird ein Ausdruck angegeben, der ebenfalls als Rückgabewert fungiert. Die Funktion lässt sich mit „url(„techdev.de")" aufrufen, wobei sich der Compiler direkt beschwert. Dies liegt daran, dass der übergebene String als Wert für „protocol" angenommen wird und der erforderliche Wert für „domain" somit fehlt. Dies lässt sich beheben, indem man den Aufruf zu „url(domain

```
val name = "Kotlin"
println("Hello $name")
```

Listing 1

```
val text = """some text
spanning
multiple
lines"""
```

Listing 2

```
val john = Person("John", 30)
john.age = 31
println("${john.name}, age ${john.
age}")
```

Listing 3

= „techdev.de")" anpasst, womit der Wert explizit für die Domäne verwendet wird.

Data-Klassen

Wer regelmäßig mit Java arbeitet, kennt wahrscheinlich das Projekt „Lombok" *(siehe „https://projectlombok.org")*. Mithilfe von Annotationen übernimmt Lombok das Generieren häufig benötigter Methoden, deren Implementierung in der Regel selten modifiziert wird. Dazu zählen Getter und Setter sowie die Methoden „equals", „hashCode" und „toString". Dieses Konzept hat in Kotlin in Form eines eigenen Schlüsselworts ebenfalls Einzug gehalten. Durch Verwendung von „data" an einer Klassen-Definition werden die zuvor genannten Funktionen generiert. Dies geschieht basierend auf den im Primärkonstruktor definierten Properties: „data class Person(val name: String, var age: Int)". Zusätzlich bekommt die Klasse „Person" nun eine weitere Funktion „copy", mit deren Hilfe Kopien einer Instanz erzeugt werden können *(siehe Listing 5)*.

Die „copy"-Funktion belegt die Werte der Parameter mit den aktuellen Werten der Instanz. Ein Aufruf der Funktion ohne Parameter erzeugt also eine identische Kopie.

Null-Sicherheit

In Java zählen die „NullPointerException" zu den am häufigsten auftretenden Ausnahmen in produktiven Anwendungen *(siehe „http://blog.takipi.com/we-crunched-1-billion-java-logged-errors-heres-what-causes-97-of-them")*. Sie treten auf, wenn zur Laufzeit auf eine Referenz zugegriffen wird, die zu diesem Zeitpunkt „null" ist.

Kotlin versucht dies zu vermeiden, indem schon zur Compile-Zeit verifiziert wird, dass keine Zugriffe auf „null"-Referenzen stattfinden. Damit der Compiler dies überprüfen kann, benötigt er zusätzliche Informationen, was zu geringfügig mehr Aufwand beim Programmieren führt.

Prinzipiell gilt, dass alle Referenzen nicht „null" sind, es sei denn, dies ist explizit gewünscht. Dazu wird in der Typen-Hierarchie zwischen „nullable"- und „nicht-nullable"-Typen unterschieden. Für alle „nicht-nullable"-Typen existiert die Superklasse „Any", die sich analog zu „Object" in Java verhält. Noch darüber findet sich die Klasse „Any?" als Superklasse aller Typen, die potenziell „null" sein können.

Um in unserem Beispiel zu bleiben, so kompiliert der Code „var john: Person = null" nicht. Stattdessen muss die Referenz explizit als „nullable" gekennzeichnet sein: „var john: Person? = null". Beide Typen („Person" und „Person?") unterscheiden sich hinsichtlich der verfügbaren Funktionen. Obwohl „Person?" potenziell „null" sein kann, gibt es eine minimale Menge an Funktionen, die aufgerufen werden kann.

Der Typ „Person" hingegen liefert alle erwarteten Funktionen, die wir implementiert oder per „data" erhalten haben. Um mit „nullable"-Typen zu arbeiten, gibt es die nachfolgenden Konstrukte.

```
fun main(args: Array<String>): Unit {
}
```

Listing 4

```
fun copy(name: String = this.name, age: Int = this.age): Person {
  return Person(name, age)
}
```

Listing 5

```
var person: Person? = null

if (person != null) {
  println(person.name)
}
```

Listing 6

```
var person: Person? = null
println(person?.name)
```

Listing 7

```
var person: Person? = null

person?.let {
  println(it.name)
}
```

Listing 8

```
var person: Person? = null
println(person?.name ?: "elvis")
```

Listing 9

Null-Checks

Analog zu Java können Variablen explizit auf „null" geprüft werden. Dies allein würde keine nennenswerte Verbesserung gegenüber Java darstellen *(siehe Listing 6)*. Der Unterschied ist, dass der Kotlin-Compiler nach dem Check einen Smart Cast durchführt. Der Typ der Referenz „person" wird innerhalb des „if"-Blocks automatisch von „Person?" nach „Person" gecastet und kann entsprechend verwendet werden. Der Check findet nicht atomar statt. Wird die Referenz zwischen der Überprüfung und dem Zugriff wieder auf „null" gesetzt, führt dies unweigerlich zu einer Ausnahme.

Safe Calls

Neben den expliziten Checks existieren noch die „Safe Calls" *(siehe Listing 7)*. Im Beispiel prüft der Compiler, ob „person" „null" ist. Ist dies der Fall, wird der gesamte Ausdruck auf „null" gesetzt. Anderenfalls fährt die Auswertung fort und es wird auf den Namen einer Person zugegriffen. Durch diese Syntax lassen sich auch verkettete Aufrufe auswerten, ohne eine „NullPointerException" auszulösen.

Eleganter wird der Zugriff, wenn man zusätzlich die Funktion „let" verwendet, die in der Kotlin-Standardbibliothek enthalten ist *(siehe Listing 8)*. Wie zuvor wird überprüft, ob die Referenz „null" ist. Falls nicht, wird die Funktion „let" aufgerufen und in ihr die Referenz unter dem Namen „it" verfügbar gemacht.

Assertions

Ist man sich sicher, dass eine Referenz nicht „null" ist, kann man den Compiler dazu zwingen, die Referenz entsprechend zu behandeln. Dazu verwendet man zwei Ausrufezeichen „!!". Die auffällige Art dieser Syntax ist keineswegs zufällig und soll nur in Ausnahmefällen genutzt werden, da ein Irrtum doch wieder zu einer Ausnahme zur Laufzeit führt.

Elvis-Operator

Der Elvis-Operator prüft einen Ausdruck auf „null" und gibt ihn zurück, falls er nicht „null" ist. Anderenfalls wird ein vordefinierter Wert zurückgegeben *(siehe Listing 9)*.

Mittels Safe Call wird überprüft, ob die Referenz auf den Namen einer Person „null"

ist. Da dies der Fall ist, wird der String „elvis" zurückgegeben. Daher eignet sich der Elvis-Operator, um Defaultwerte zu vergeben. Doch auch der folgende Aufruf ist gültig und kann beispielsweise zur Validierung verwendet werden *(siehe Listing 10)*.

```
var person: Person? = null
person?.name ?: throw IllegalArgumentException("name is required")
```

Listing 10

Pattern Matching

Kotlin bietet mit der „when"-Expression die Möglichkeit zum Pattern Matching. „when" ist ein Ausdruck, der ein Ergebnis liefert, abhängig davon, welcher Zweig durchlaufen wird *(siehe Listing 11)*. Anders als in Java ist hier kein „break" nötig, da immer nur ein Zweig ausgeführt wird und dieser das Ergebnis des Ausdrucks bestimmt.

Im Beispiel lässt sich das Ergebnis eines Web-Aufrufs analysieren. Abhängig davon, welcher Status-Code zurückgeliefert wurde, übersetzt die Funktion „resolve" den Status in einen lesbaren String.

```
fun resolve(status: Int): String {
  return when (status) {
     in 100..199 -> "info"
     in 200..299 -> "success"
     in 300..399 -> "redirect"
     in 400..499 -> "client error"
     in 500..599 -> "server error"
     else -> "unknown"
  }
}
```

Listing 11

Die Funktion verwendet Ranges, um einen Wertebereich zu beschreiben. Dabei sind die Grenzen jeweils inklusive. Die Range „100..199" enthält somit alle Ganzzahlen von 100 bis einschließlich 199. Im Erfolgsfall sollte der Status-Code den Wert „200" besitzen und der zweite Zweig ausgeführt werden *(siehe Listing 12)*.

```
fun analyse(any: Any?): Any {
  return when(any) {
     is Int    -> any.inc() // Int.inc()
     is String -> any.reversed() // String.reversed()
     else      -> any.toString() // Any.toString()
  }
}
```

Listing 12

Abfragen können wie gezeigt die verwendete Referenz auch auf einen bestimmten Typ prüfen. Wird eine Verzweigung ausgeführt, so führt der Compiler auch hier einen Smart Cast durch und innerhalb des Zweiges kann die Referenz entsprechend verwendet werden, ohne einen expliziten Cast durchführen zu müssen. Innerhalb eines „when"-Blocks müssen die Zweige erschöpfend sein, es muss also für jede Option ein Zweig existieren. In der Regel lässt sich dies durch einen „else"-Zweig sicherstellen.

Erweiterungsfunktionen

Neben den bereits beschriebenen Funktionen gibt es in Kotlin zusätzlich die Erweiterungsfunktionen. Sie erweitern bestehende Typen um zusätzliche Funktionalität, wobei es keine Rolle spielt, ob diese Typen Teil der eigenen Code-Basis oder etwa in einer verwendeten Bibliothek definiert sind. So lässt sich die Klasse „java.util.Date" erweitern, um sie in ein „java.time.LocalTime" zu überführen *(siehe Listing 13)*.

```
import java.time.LocalDate
import java.time.ZoneId
import java.util.Date

fun Date.local(): LocalDate {
  return this.toInstant().atZone(ZoneId.systemDefault()).toLocalDate()
}
```

Listing 13

Erweiterungsfunktionen geben als Teil ihres Namens immer den Typ an, für den sie definiert werden, hier „Date". Abgesehen davon sehen die Deklarationen identisch zu den bereits behandelten Funktionen aus. Innerhalb einer Erweiterungsfunktion verweist „this" jeweils auf die Instanz des erweiterten Typs.

Erweiterungsfunktionen werden statisch aufgelöst, können also nicht überschrieben oder vererbt werden. Existiert eine gleichnamige Member-Funktion, so bekommt diese immer Vorrang.

Operatoren überladen

Kotlin ermöglicht das Überladen von Operatoren. Dazu existieren für jeden Operator bestimmte Funktionen, die es zu implementieren gilt, beispielsweise „plus" für den Operator „+". Um Konflikte mit Funktionen gleichen Namens zu vermeiden, muss zusätzlich das Schlüsselwort „operator" verwendet werden.

Als Beispiel dient eine Klasse, die einen zweidimensionalen Vektor beschreibt: „data class Vector(val x: Double, val y: Double)". Basierend auf dieser Klasse lässt sich die Vektor-Addition definieren *(siehe Listing 14)*.

Tabelle 1 stellt die Operatoren und ihre zu implementierenden Funktionen gegenüber. Dabei gilt wie gewohnt „Punktrechnung vor Strichrechnung" und die ersten drei Operationen haben eine höhere Präzedenz als die zwei folgenden. Der Range-Operator hat dabei die geringste Präzedenz.

Analog zur Addition lässt sich auch die Multiplikation mit einem Skalar implementieren *(siehe Listing 15)*. Dabei ist jedoch zu beachten, dass die Kommutativität verletzt wird; es gilt nur „a * b", jedoch nicht zwingend auch „b * a". Sofern diese Eigenschaft benötigt wird, kann der Operator analog auf der Klasse „Double" mithilfe einer Erweiterungsfunktion definiert werden.

Interoperabilität

Eine der Stärken von Kotlin ist die nahtlose Interoperabilität von und zu Java. Aufrufe von Java aus Kotlin heraus funktionieren ohne Probleme, doch wie verhält es sich beim Auf-

```
operator fun plus(other: Vector): Vector {
  return Vector(this.x + other.x, this.y + other.y)
}
```

Listing 14

```
operator fun times(scalar: Double): Vector {
  return Vector(this.x * scalar, this.y * scalar)
}
```

Listing 15

```
fun boom() {
  throw IOException(„boom“)
}
```

Listing 16

Operator	Funktion
a * b	times
a / b	div
a % b	mod
a + b	plus
a – b	minus
a..b	rangeTo

Tabelle 1

rufen von Kotlin aus Java heraus? Kotlin erzeugt für jede Datei eine Java-Klasse mit gleichem Namen, der zusätzlich das Kürzel „Kt“ angefügt ist. Eine Datei namens „Extensions.kt“ wäre demnach als „ExtensionsKt“ in Java verfügbar. Soll die Datei unter einem anderen Namen verfügbar sein, wird dies dem Compiler mitgeteilt. Dazu ist als erste Anweisung eine entsprechende Annotation anzugeben, etwa „@file:JvmName(„Extensions“)“.

Sind in der Datei globale Funktionen definiert, können diese als statische Methoden auf dem Java-Typ aufgerufen werden: „fun String.hide() = „*“.repeat(this.length)“. Die Erweiterungsfunktion „hide“ wandelt einen String in einen String gleicher Länge um, der nur aus Sternchen „*“ besteht. Der Java-Aufruf lautet „Extensions.hide(„password“);“. Da es nicht möglich ist, bestehende Typen in Java zu erweitern, erwartet die statische Methode „hide“ als einzigen Parameter den String, den es zu verstecken gilt.

In Kotlin definierte Klassen lassen sich anhand ihres Namens verwenden. Für unveränderliche Properties („val“) werden die Getter-Methoden automatisch erzeugt. Im Falle von veränderlichen („var“-) Properties wird zusätzlich ein Setter erzeugt.

Eine weitere Besonderheit von Kotlin ist, dass es keine Unterscheidung zwischen Checked und Unchecked Exceptions macht. Daher müssen Checked Exceptions auch nicht als Teil der Funktionssignatur angegeben werden. Dies führt allerdings zu Problemen, wenn eine Funktion eine Checked Exception auslöst (siehe Listing 16).

Aus Java heraus lässt sich die Funktion ohne Probleme aufrufen. Versucht man allerdings, die „IOException“ zu behandeln, beschwert sich der Compiler, da die Exception nicht Teil der Signatur ist. Um dies zu ändern, dient die Annotation „@Throws(IOException::class)“. Sie bewirkt, dass die Exception der Signatur hinzugefügt wird, sodass diese in Java behandelt werden kann.

Fazit

Kotlin weist in jedem Fall ein paar interessante Features auf und ist in der aktuellen Version (1.0.4) bereits für den produktiven Einsatz geeignet. Die Sprache wird aktiv weiterentwickelt und Vorschläge können über den Kotlin Evolution & Enhancement Process (KEEP) in Form von Pull Requests eingereicht werden (siehe „https://github.com/Kotlin/KEEP“). Es bleibt abzuwarten, wie sich Kotlin im Backend-Bereich behaupten kann.

Alexander Hanschke
alexander.hanschke@techdev.de

Alexander Hanschke arbeitet seit dem Jahr 2013 bei der techdev Solutions GmbH in Karlsruhe, wo er Kunden bei der erfolgreichen Umsetzung von IT-Projekten unterstützt.

APEX Connect 2017
9. bis 11. Mai 2017 in Berlin

http://apex.doag.org

Automatisierte Überprüfung von Sicherheitslücken in Abhängigkeiten von Java-Projekten

Johannes Schnatterer, Triology GmbH

Sicherheit in Software-Projekten ist ein schwer zu beherrschendes Thema, wie regelmäßig in den Medien erscheinende Berichte über Datenlecks beweisen. Für Software-Entwickler ist es schwer zu überblicken, welchen Einfluss ihre tägliche Arbeit an nicht unmittelbar sicherheitsrelevanten Themen auf die Sicherheit ihrer Anwendung hat.

Einen Überblick über die häufigsten Makel in Punkto Sicherheit bieten die Top Ten des Open Web Application Security Project (OWASP, *siehe „https://www.owasp.org/index.php/Top_10_2013-Top_10"*), eine Non-Profit-Organisation, die sich zum Ziel gesetzt hat, die Sicherheit im Web zu verbessern. Hier zeigt sich zudem, dass auch nicht selbst geschriebener Code Sicherheitslücken enthalten kann. Diesen kann man jedoch mit wenigen Schritten überprüfen. Der Artikel zeigt, wie man mit geringem Aufwand zumindest über die bekannten Sicherheitslücken in verwendeten Abhängigkeiten von Dritten auf dem Laufenden bleibt.

Grundlage bildet die National Vulnerability Database (NVD, siehe *„https://nvd.nist.gov")*, eine Art Datenbank für Sicherheitslücken. Sie wird gepflegt vom National Institute of Standards and Technology (NIST), eine mit der deutschen Physikalisch-Technischen Bundesanstalt (PTB) vergleichbare US-amerikanische Bundesbehörde, die auch bekannte Verschlüsselungsalgorithmen wie DES und AES standardisiert.

Die NVD beinhaltet unter anderem die Schwachstellen, die im Industrie-Standard „Common Vulnerabilities and Exposures" (CVE) erfasst werden. Die bereits erwähnte OWASP bietet eine Anwendung an, die automatisiert Jar-Dateien mit der NVD abgleicht. Dieser sogenannte „OWASP Dependency-Check" (DChc, *siehe „https://www.owasp.org/index.php/OWASP_Dependency_Check")* wurde für die Kommandozeile sowie für Ant, Maven, gradle, sbt, Jenkins und SonarQube implementiert.

Nachfolgend ein Lösungskonzept, um die Abhängigkeiten einer Java-Anwendung mithilfe von Maven und Jenkins regelmäßig auf neue Sicherheitslücken zu durchsuchen. Nach einmaliger Einrichtung sind keine weiteren Schritte mehr notwendig. Die Information über neu gefundene Sicherheitslücken erfolgt dann per E-Mail. Dabei kann es sich entweder um neu bekannt gewordene Sicherheitslücken in bestehenden Abhängigkeiten handeln oder um bereits bekannte Sicherheitslücken, die in neu hinzufügten Abhängigkeiten existieren.

Erster Schritt: Maven-Plug-in ohne Konfiguration verwenden

Eine erste Überprüfung der Abhängigkeiten der eigenen Anwendung kann ohne weitere Konfiguration in wenigen Sekunden angestoßen werden, beispielsweise mittels Maven-Plug-in (MVN, *siehe „http://jeremylong.github.io/DependencyCheck/dependency-check-maven/configuration.html")* durch „mvn org.owasp:dependency-check-maven:1.4.0:aggregate". Dabei gilt es zu beachten, dass „dependency-check-maven-plugin" zunächst eine lokale Kopie der CVE-Datenbank in Form einer H2-Datenbank herunterlädt und sie im lokalen Maven-Repository speichert. Aufgrund der Anzahl bekannter Sicherheitslücken resultiert dies in einer mehrere Hundert Megabytes umfassenden Datei. Das initiale Erzeugen kann, abhängig von verfügbarer Bandbreite und Rechenleistung, einige Minuten dauern. Nach erfolgreichem Abschluss steht der Bericht unter „target/dependency-check-report.html". Ein Beispiel für einen solchen Bericht findet man auf der Website des Projekts (DChck-Sample, *siehe „http://jeremylong.github.io/DependencyCheck/general/SampleReport.html")*.

Automatisierung mit Jenkins

Um ohne weiteres Zutun über neu gefundene Sicherheitslücken informiert zu werden, bietet sich eine regelmäßig durchgeführte, automatische Überprüfung an. OWASP bietet für den CI-Server Jenkins ein Plug-in an, das die Überprüfung durchführen und deren Ergebnisse auswerten kann. Unter anderem können die gefundenen Sicherheitslücken in Jenkins visualisiert, das Ergebnis eines Jobs von der Anzahl gefundener Sicherheitslücken abhängig gemacht und die Entwicklung der Anzahl gefundener Sicherheitslücken über die Builds in einem Diagramm dargestellt werden. Diese Auswertung ist in einer Post-Build-Action im Jenkins-Job konfiguriert. Die Überprüfung der Abhängigkeiten selbst kann durch einen durch das Jenkins-Plug-in bereitgestellten Build-Step oder per Maven durchgeführt werden.

Da die Überprüfung einige Zeit in Anspruch nimmt, sollte sie nicht bei jedem Commit/Push stattfinden. Ein Nightly-Build oder ein einmal pro Woche durchgeführter Build bieten sich daher an. Die Überprüfung per Maven Goal statt mit Jenkins-Build Step hat den Vorteil, dass die zentrale Konfiguration in der „pom.xml" erfolgt, wie in *Listing 1* gezeigt.

Mit dieser Konfiguration müssen in Jenkins nur noch die folgenden Maven Goals mit Parameter aufgerufen werden: „mvn clean install

```
<properties>
  <dependency-check-format>HTML</dependency-check-format>
</properties>
<build>
  <plugins>
     <plugin>
        <groupId>org.owasp</groupId>
        <artifactId>dependency-check-maven</artifactId>
        <version>1.4.0</version>
        <configuration>
           <format>${dependency-check-format}</format>
        </configuration>
     </plugin>
  </plugins>
</build>
```

Listing 1: Maven-Konfiguration mit parametrisierbarem Format des Reports

Build Settings

E-mail Notification

Recipients a@b.c

Send e-mail for every unstable build

Send separate e-mails to individuals who broke the build

Send e-mail for each failed module

Post-build Actions

Publish OWASP Dependency-Check analysis results

Dependency-Check results war/target/dependency-check-report.xml

Fileset includes setting that specifies the generated raw Dependency-Check XML report files, such as **/dependency-check-report.xml. Basedir of the fileset is the workspace root. If no value is set, then the default **/dependency-check-report.xml is used. Be sure not to include any non-report files into this pattern.

Run always

By default, this plug-in runs only for stable or unstable builds, but not for failed builds. If this plug-in should run even for failed builds then activate this check box.

Detect modules

Determines if Ant or Maven modules should be detected for all files that contain warnings. Activating this option may increase your build time since the detector scans the whole workspace for 'build.xml' or 'pom.xml' files in order to assign the correct module names.

Health thresholds 100% 0%

Configure the thresholds for the build health. If left empty then no health report is created. If the actual number of warnings is between the provided thresholds then the build health is interpolated.

Health priorities Only priority high Priorities high and normal All priorities

Determines which warning priorities should be considered when evaluating the build health.

Status thresholds (Totals) All priorities Priority high Priority normal Priority low

0

If the number of total warnings is greater than one of these thresholds then a build is considered as unstable or failed, respectively. I.e., a value of 0 means that the build status is changed if there is at least one warning found. Leave this field empty if the state of the build should not depend on the number of warnings.

Compute new warnings (based on the last successful build unless another reference build is chosen below)

Default Encoding

Default encoding when parsing or showing files. Leave this field empty to use the default encoding of the platform.

Trend graph You can define the default values for the trend graph in a separate view.

Delete

Abbildung 1: Das Ergebnis der Post-Build-Action

```
<configuration>
  <!-- Skip artifacts not bundled in distribution (provided scope) -->
  <skipProvidedScope>true</skipProvidedScope>
  <!-- Suppress false positives or dependencies that cannot be changed for specific reasons.-->
  <suppressionFile>suppress-cves.xml</suppressionFile>
</configuration>
```

Listing 2: Maven-Konfiguration mit suppressionFile und provided Scope

org.owasp:dependency-check-maven:check -Ddependency-check-format=XML". Dadurch wird eine Datei „target/dependency-check-report.xml" im Workspace generiert, die man von einer entsprechenden Post-Build-Action verarbeiten lassen kann. Abschließend muss in dieser noch konfiguriert werden, dass der Job fehlschlägt oder instabil wird, sobald eine Sicherheitslücke entdeckt wird *(siehe Abbildung 1)*. In Zusammenarbeit mit der von Jenkins angebotenen E-Mail-Benachrichtigung ist sichergestellt, dass man ohne weiteres Zutun per E-Mail über neu gefundene Sicherheitslücken informiert wird.

Die Praxis

Abschließend noch einige Details, die in der täglichen Arbeit mit dem Dependency-Check relevant sein können. Es empfiehlt sich, den Dependency-Check auf dem Maven-Modul durchzuführen, das das eigentlich veröffentlichte Artefakt (.jar, .war, .ear etc.) herstellt. Dadurch werden auch nur die wirklich relevanten Abhängigkeiten auf Sicherheitslücken durchsucht und nicht die Abhängigkeiten von Modulen, die beispielsweise nur für Tests im Einsatz sind. Abhängigkeiten im Maven-Scope-Test werden in der aktuellen Version des Maven-Plug-ins standardmäßig nicht durchsucht.

Anders sieht es mit Abhängigkeiten in den Scopes „provided" und „runtime" aus. Insbesondere bei „provided" können sie unerwünscht sein. Wenn man beispielsweise durch die Kundenumgebung an ein bestimmtes Servlet-API gebunden ist, ist man gegen die gegebenenfalls darin enthaltenen Sicherheitslücken machtlos. Insofern kann es durchaus sinnvoll sein, diesen Scope auszuschließen. Dies kann in der Konfiguration von Maven angepasst werden *(siehe Listing 2)*.

Alternativ lassen sich bestimmte CVEs auch für einzelne Abhängigkeiten unterdrücken. Diese sind in einer speziellen Datei spezifiziert. Der XML-Code für das Unterdrücken lässt sich direkt aus dem im ersten Schritt generierten HTML-Bericht kopieren. Dies kann insbesondere sinnvoll sein, wenn es sich um False Positives handelt, also um gefundene Sicherheitslücken, die auf die Abhängigkeit gar nicht zutreffen. Ein Beispiel

```
<?xml version="1.0" encoding="UTF-8"?>
<suppressions xmlns="https://jeremylong.github.io/DependencyCheck/dependency-suppression.1.1.xsd">
  <suppress>
     <!-- This Jackson-related issue seems to be related to module jackson-dataformat-xml, which is not used here.
It is considered a false positive. See https://github.com/jeremylong/DependencyCheck/issues/517 In addition it was
fixed in version 1.7.4., see https://github.com/FasterXML/jackson-dataformat-xml/issues/199 -->
     <notes><![CDATA[file name: jackson-core-2.8.1.jar]]></notes>
     <sha1>fd13b1c033741d48291315c6370f7d475a42dccf</sha1>
     <cve>CVE-2016-3720</cve>
  </suppress>
  <suppress>
     <!-- Same as other Jackson-related issue -->
     <notes><![CDATA[file name: jackson-annotations-2.8.0.jar]]></notes>
     <sha1>45b426f7796b741035581a176744d91090e2e6fb</sha1>
     <cpe>cpe:/a:fasterxml:jackson:2.8.0</cpe>
  </suppress>
</suppressions>
```

Listing 3: Beispielhaftes suppressionFile

dafür ist der Generalverdacht (CVE-2016-3270, *siehe „https://github.com/jeremylong/DependencyCheck/issues/517“)* gegen alle Module des Jackson-Frameworks, obwohl nur in einem eine Sicherheitslücke auftritt. Es ist daher generell sinnvoll, jede gefundene Sicherheitslücke auf Korrektheit zu überprüfen. In jedem Fall sollte kommentiert werden, weshalb ein CVE unterdrückt wird, um zu späteren Zeitpunkten die Gründe für den Ausschluss nachvollziehen zu können. In den *Listings 2 und 3* wird diese Möglichkeit für den Ausschluss aufgezeigt.

Fazit

Der Artikel zeigt, wie man mit wenigen Schritten dauerhaft über bekannte Sicherheitslücken von Abhängigkeiten informiert bleibt. Dies sorgt nicht für absolute Sicherheit, entschärft aber mit geringem Aufwand einen der zehn gängigsten Makel. Insofern ist dies uneingeschränkt für alle Java-Projekte zu empfehlen.

Die hier beschriebene, auf Maven und Jenkins basierende Lösung stellt eine Möglichkeit für das Herausfinden von Sicherheitslücken dar. Aufgrund der vielen verfügbaren Implementierungen des OWASP-Dependency-Checks lassen sich vergleichbare Lösungen auch für viele andere Tools realisieren. Ein komplett lauffähiges Beispiel, das die oben beschriebenen Fälle sowie eine Abhängigkeit mit nicht ausgeschlossener Sicherheitslücke enthält, steht bei GitHub *(siehe „https://github.com/triologygmbh/dependency-check“)*.

Johannes Schnatterer
johannes.schnatterer@triology.de

Johannes Schnatterer ist Solution Architect bei der TRIOLOGY GmbH in Braunschweig. Technologisch ist er dort in den Bereichen „Java EE“ und „Web“ tätig und versucht, mit besonderem Fokus auf Qualität, Open-Source-Enthusiasmus, einem Hauch von Pedantismus und der Pfadfinderregel die IT-Welt jeden Tag ein bisschen besser zu machen.

Veranstaltungen der im iJUG organisierten Java User Groups auf Erfolgskurs

Mit einer Rekordbeteiligung von rund 1.700 Teilnehmern fand am 7. Juli 2016 zum 19. Mal das Java Forum in Stuttgart statt. Die Java User Group Stuttgart hatte 49 Vorträge in sieben parallelen Tracks organisiert. Zudem waren 34 Aussteller vor Ort, darunter auch der Interessenverbund der Java User Groups e.V. (iJUG). Abends gab es die Gelegenheit, sich bei verschiedenen BoF-Sessions (Birds of a Feather) mit Gleichgesinnten zu treffen, um über ein bestimmtes Thema zu diskutieren und sich auszutauschen.

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

Das Java Forum Nord öffnete am 20. Oktober 2016 in Hannover seine Pforten. Die eintägige, nicht-kommerzielle Konferenz in Norddeutschland mit Themenschwerpunkt Java ist für Entwickler und Entscheider. Mit mehr als 25 Vorträgen in parallelen Tracks und einer Keynote wurde ein vielfältiges Programm zu einem unschlagbaren Preis geboten, der regionale Bezug bietet zudem interessante Networking-Möglichkeiten. Gestaltung und Organisation wurde von den lokalen Java User Groups (Bremen, Göttingen, Hamburg, Hannover, Kassel, Ostfalen) in Kooperation mit der Java User Group Deutschland e.V./Sun User Group Deutschland e.V. als offiziellen Veranstaltern durchgeführt. Es kamen 400 Besucher.

Auch das Datum für die JavaLand 2017 steht bereits fest. Sie findet vom 28. bis 30. März 2017 an gewohnter Stätte im Phantasialand Brühl statt.

Unleashing Java Security

Philipp Buchholz, esentri AG

Innerhalb von Enterprise-Anwendungen spielen Sicherheits-Aspekte eine wichtige Rolle. Nur autorisierte Benutzer oder allgemein autorisierte Entitäten dürfen auf sensible Daten oder Systemteile zugreifen.

Oftmals machen rechtliche und organisatorische Rahmenbedingungen eine Autorisierung, also eine Zugriffskontrolle, notwendig. Dieser Artikel erläutert zunächst die grundlegende Sicherheits-Architektur von Java mit den beteiligten Klassen und deren Verwendung. Darauf aufbauend wird auf die Authentifizierung und Autorisierung, basierend auf der JAAS-Spezifikation, eingegangen. Abschließend sind die grundlegenden Möglichkeiten der JASPIC-Spezifikation im Kontext von Java-EE-Anwendungen erläutert.

Die grundlegende Sicherheits-Architektur innerhalb von Java basiert auf der Gruppierung von Quellcode nach dessen Herkunft und der Zuordnung von Berechtigungen zu diesen Gruppierungen. Auf der JVM ausgeführter Quellcode wird aus diesem Grund in sogenannte „ProtectionDomains" eingruppiert. Jede Klasse kann immer nur einer ProtectionDomain angehören. Dieser ProtectionDomain, dargestellt durch eine Instanz der Klasse „java.security.ProtectionDomain" *(siehe „https://docs.oracle.com/javase/8/docs/api/java/security/ProtectionDomain.html")*, werden Berechtigungen zugeordnet. Berechtigungen sind durch Instanzen einer Ableitung der Klasse „java.security.Permission" dargestellt.

System-Domain

Der Quellcode des JDK wird immer in einer speziellen ProtectionDomain, der System-Domain, ausgeführt und besitzt alle verfügbaren Berechtigungen *(siehe „http://docs.oracle.com/javase/8/docs/technotes/guides/security/spec/security-spec.doc2.html")*.

Alle verfügbaren Berechtigungen sind über die spezielle Implementierung „java.security.AllPermission" abgebildet. Wird diese Permission gewährt, impliziert das die Zuteilung aller verfügbaren Berechtigungen. Es ist wichtig zu beachten, dass innerhalb der System-Domain alle Klassen enthalten sind, die Zugriffe auf externe Ressourcen durchführen. Das umfasst unter anderem die Klassen der Packages „java.io" und „java.nio" für den Zugriff auf das Dateisystem.

Application-Domain

Applikations-Quellcode wird davon separiert in einer eigenen ProtectionDomain ausgeführt und erhält die für die entsprechende CodeSource konfigurierten Berechtigungen. Diese Berechtigungen werden über eine Implementierung von „java.security.policy" *(siehe „https://docs.oracle.com/javase/8/docs/api/java/security/Policy.html")* ausgelesen. Die im JDK mitgelieferte Default-Policy-Implementierung liest zu diesem Zweck eine Textdatei. Die Konfiguration von Berechtigungen über diese „.policy"-Files ist weiter unten beschrieben.

Wie beschrieben, enthält die System-Domain alle Klassen, die erforderlich sind, wenn Zugriffe auf externe Ressourcen wie das Dateisystem durchgeführt werden sollen. Da die Klassen einer Applikation sich in einer getrennten ProtectionDomain befinden, muss beim Zugriff auf externe Ressourcen von der Application-Domain in die System-Domain gewechselt werden.

Bei solch einem Übergang dürfen der Application-Domain keine nicht erteilten Berechtigungen gewährt werden. Die Applikation darf keinen Dateisystemzugriff durchführen können, wenn ihr nicht explizit die „java.io.FilePermission" zugeordnet wurde, auch wenn die System-Domain implizit alle Berechtigungen besitzt. Aus diesem Grund wird zwischen den Permissions der Application-Domain und der System-Domain ein Intersect *(siehe „https://en.wikipedia.org/wiki/Intersection_(set_theory)")* vorgenommen *(siehe Abbildung 1)*.

Das Venn-Diagramm zeigt, dass nach dem Intersect nur die Menge von Berechtigungen erteilt wird, die allen ProtectionDomains gewährt wurde. Wird der Application-Domain also die „java.io.FilePermission" nicht explizit gewährt, kann kein Zugriff auf das Dateisystem stattfinden und eine „java.lang.SecurityException" resultiert. Damit ist sichergestellt, dass einer Applikation beim Übergang von einer ProtectionDomain mit mehr Rechten in eine ProtectionDomain mit weniger Rechten keine nicht zugeteilten Berechtigungen gewährt werden.

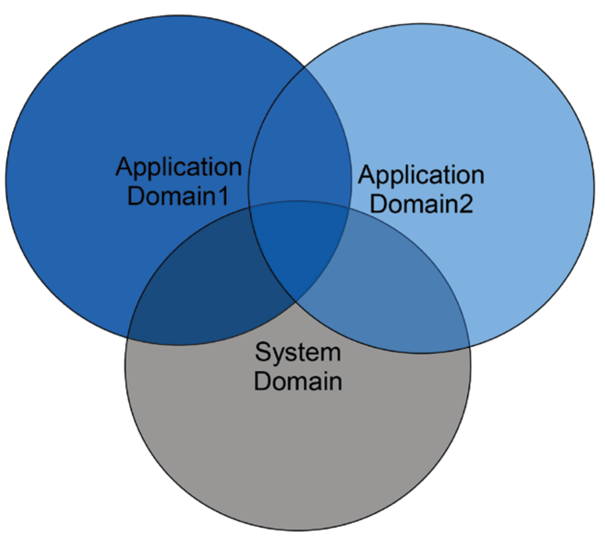


Abbildung 1: Intersect von ProtectionDomains

Zentrale Klassen für Sicherheitsüberprüfungen

Für die Überprüfung von Berechtigungen sind innerhalb des JDK mehrere Klassen zentral verantwortlich. Innerhalb einer Laufzeitumgebung gibt es höchstens einen aktiven SecurityManager. Dieser wird über die Methode „System.setSecurityManager(securityManager : java.lang.SecurityManager) : void" innerhalb der aktuellen JVM installiert und kann über den zugehörigen Getter abgerufen werden.

Der Aufruf von „checkPermission(permission : java.security.Permission) : void" prüft, ob dem Aufrufer die übergebene Berechtigung gewährt wird oder nicht. In diesem Fall wird immer der aktuelle SecurityContext verwendet. Diese Methode wird durch „checkPermission(permission : java.security.Permission, context : Object) : void" überladen. Sie erlaubt das Überprüfen einer Berechtigung unter Verwendung eines definierten SecurityContext.

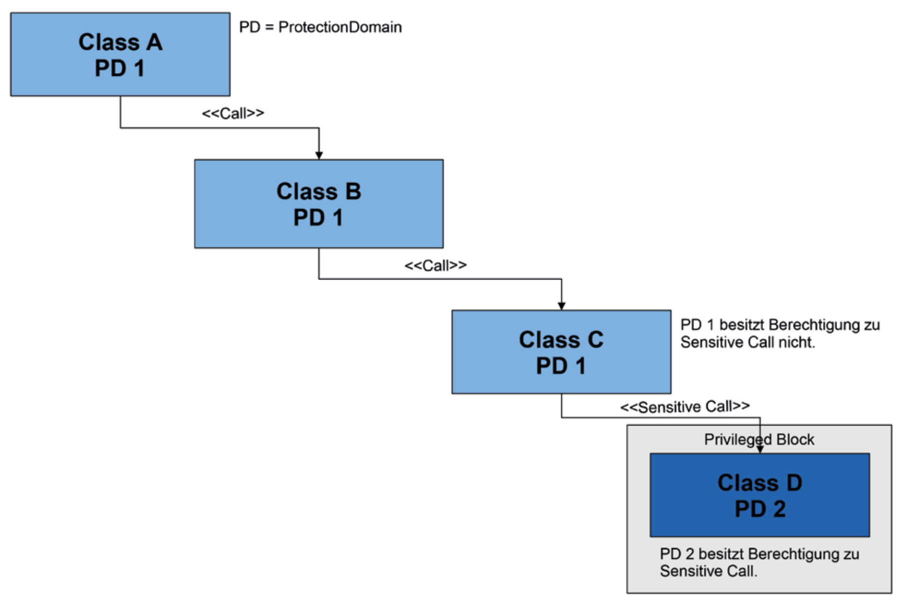


Abbildung 2: Beispiel zur Verwendung des „privileged block"-API

Bei speziellen Anforderungen die nicht über benutzerdefinierte Berechtigungen abgedeckt werden können, kann von „java.lang.SecurityManager" geerbt und eine seiner Methoden überschrieben beziehungsweise weitere Methoden für die Durchführung von Zugriffskontrollen hinzugefügt werden. Diese Vorgehensweise ist allerdings im Normalfall nicht notwendig und nicht empfehlenswert. In fast allen Fällen können über das Permission-System und eine entsprechende Policy-Implementierung alle Anforderungen abgedeckt werden.

Innerhalb von Java-SE-Anwendungen ist ein SecurityManager mit dem Setter des System-Singletons zu installieren oder die JVM mit dem Argument „-Djava.security.manager" zu starten. Ansonsten werden keine Permissions überprüft. Der SecurityContext und die Definition eigener Permissions sind weiter unten erläutert.

Ist kein SecurityManager installiert, kann es sein, dass bestimmte Funktionen nicht zur Verfügung stehen. Beispielsweise kann RMI für entfernte Methoden-Aufrufe nicht verwendet werden. Ohne installierten SecurityManager wird bei Verwendung von RMI eine Exception geworfen. Innerhalb einer Java-EE-Anwendung ist immer ein SecurityManager installiert.

Der AccessController wird von der Klasse „java.lang.SecurityManager" verwendet, um die eigentliche Überprüfung von Berechtigungen durchzuführen. Die „checkPermission"-Aufrufe des SecurityManager werden hierzu an den AccessController delegiert. Der AccessController bezieht den aktuellen SecurityContext in die Überprüfung mit ein.

Der SecurityContext wird über die native Methode „getStackAccessControlContext() : AccessControlContext" ermittelt. Der zurückgegebene AccessControlContext stellt den SecurityContext dar. Der AccessControlContext besteht aus den ProtectionDomains der Klassen, die sich im Stack des aktuellen Threads befinden. Aus diesen ProtectionDomains wird der Intersect der assoziierten Permissions gebildet. Die resultierende Menge von Berechtigungen bildet die Grundlage für Berechtigungsprüfungen. Außerdem übernimmt der AccessController die Markierung von Quellcode als privilegiert *(siehe Abbildung 2)*.

Bei der Verwendung des „privileged block"-API ist von einem Szenario auszugehen, bei dem innerhalb des Call-Stacks ein Übergang zwischen ProtectionDomains stattfindet. Wie beschrieben, findet in diesem Fall ein Intersect der Permissions statt. Dies verhindert, dass eine Anwendung Berechtigungen einer ProtectionDomain erhält, die dieser nicht erteilt wurden. Wurde der Applikation die benötigte Berechtigung nicht erteilt, wird eine SecurityException geworfen.

Aus unterschiedlichen Gründen kann es aber notwendig sein, dass ein sensitiver Aufruf trotzdem durchgeführt werden muss. Beispielsweise könnte es notwendig sein, dass eine Anwendung zur grundsätzlichen Funktionsweise Konfigurationen aus Dateien lesen muss. Es könnte ebenfalls sein, dass eine Bibliothek entwickelt wird, die dann innerhalb einer nicht bekannten Ausführungsumgebung, in der bestimmte Berechtigungen nicht garantiert werden können, zum Einsatz kommt. Für diesen Fall wurde das „privileged block"-API in das JDK integriert. Damit ist es möglich, eine sensitive Operation auszuführen, obwohl der Applikation die Berechtigung selbst nicht erteilt wurde. Hierzu muss der aufgerufene Code und jeder in der Folge aufgerufene Code die benötigte Berechtigung besitzen.

Für die Implementierung solch einer Funktionalität werden privilegierte Codeblöcke eingerichtet. Dazu muss das Interface „java.security.PrivilegedAction" oder „java.securi-

```
...
Configuration configuration = AccessController.doPrivileged(new PrivilegedAction<Configuration>() {
  public Configuration run() {

     /* This will check FilePermission for reading the designated file. */
     try(InputStream configurationInputStream = Files.newInputStream(Paths.get(configurationFileUri),StandardOpenOp
tion.READ)) {
         /* Implement logic to read configuration here. */
         }

     }
  });
...
```

Listing 1

```
public class EditCustomerDataPermission extends BasicPermission {
  public EditCustomerDataPermission(String name) {
    super(name);
  }
}
```

Listing 2

```
SecurityManager securityManager = System.getSecurityManager();
securityManager.checkPermission(new EditCustomerDataPermission(„editAll“));
```

Listing 3

ty.PrivilegedExceptionAction" implementiert sein. Der Unterschied zwischen den beiden besteht darin, dass aus Implementierungen von „java.security.PrivilegedExceptionAction" eine Checked Exception geworfen werden kann, im anderen Fall nicht. Anschließend wird eine der „doPrivileged"-Methoden der Klasse AccessController mit dieser Implementierung aufgerufen. Der innerhalb der Implementierung eines der oben genannten Interfaces befindliche sensitive Aufruf wird damit als privilegiert ausgeführt und markiert.

Beispiel-Implementierung eines Privileged Blocks

Berechtigungen, die dem Quellcode *(siehe Listing 1)* eines privilegierten Blocks erteilt werden, stehen nach der Ausführung dieses Blocks nicht mehr zur Verfügung. Privilegierte Blöcke sollten nur sparsam und nach genauer Überlegung eingesetzt werden. Es ist immer erst zu prüfen, ob die Konfiguration über eine Policy durchgeführt werden kann.

Wie bereits erwähnt, stellen Implementierungen der Klasse „java.security.Permission" Berechtigungen für den Zugriff auf geschützte Ressourcen dar. Geschützte Ressourcen sind beispielsweise Dateien im Dateisystem oder Sockets für den Netzwerk-Zugriff. Je nach Szenario kann eine geschützte Ressource aber auch das Anzeigen bestimmter Daten innerhalb einer UI oder das Ausführen von Administrations-Funktionen darstellen.

Eine Permission besteht immer aus einem Namen und wird abhängig von der Implementierung zusätzlich durch eine ActionList näher beschrieben. Diese wird für die Beschreibung von Aktionen in Bezug auf eine geschützte Ressource verwendet und ist optional. Je nach Dokumentation wird für den Namen einer Berechtigung auch Target synonym verwendet. Beispiele für Aktionen sind „read", „write" oder auch „editCustomerData".

Innerhalb des JDK gibt es bereits diverse Permission-Implementierungen, die den Zugriff auf die wichtigsten geschützten Ressourcen beschreiben. Beispielsweise existiert eine „java.io.FilePermission", um den Zugriff auf das Dateisystem abzudecken. Weitere wichtige Permissions sind „java.net.SocketPermission" für den Zugriff auf einen Socket, „java.net.NetPermission" oder auch „java.sql.SQLPermission". Die genannten Permissions sind nur eine kleine Auswahl bereits vorhandener Berechtigungen. Eine komplette Liste zeigt die Vererbungshierarchie von „java.security.Permission". Die bereits vorhandenen Permissions werden bei Verwendung der geschützten Ressourcen unter Verwendung des installierten SecurityManager geprüft.

Erstellen von benutzerdefinierten Permissions

Bevor auf die Verwendung von Permissions eingegangen wird, soll die Implementierung einer benutzerdefinierten Permission gezeigt werden *(siehe Listing 2)*. In diesem Fall wird die abstrakte Basisklasse „java.security.BasicPermission" erweitert. Sie erbt von „java.security.Permission" und implementiert bereits alle abstrakten Methoden. Ableitungen dieser Klasse stellen sogenannte „Named-Permissions" dar. Das sind Berechtigungen, die bereits durch ihren Namen komplett beschrieben sind; eine zusätzliche ActionList ist nicht notwendig. Daher reicht in der Ableitung ein Konstruktor, der als Parameter den Namen der Berechtigung aufnimmt.

Um benutzerdefinierte Permissions im Quellcode zu überprüfen, wird die aktuell installierte Instanz des SecurityManager abgerufen und die „checkPermission"-Methode mit einer Instanz der benutzerdefinierten Permission aufgerufen *(siehe Listing 3)*. Im Beispiel wird geprüft, ob die oben definierte Berechtigung „EditCusomterDataPermission" erteilt wurde. Wurde die Berechtigung nicht erteilt, wird eine SecurityException geworfen; ist die Berechtigung erteilt, kehrt der Aufruf fehlerfrei zurück.

Zuweisung von Permissions über Policies

Die Zuweisung von Permissions zu CodeSources erfolgt über Implementierungen der abstrakten Basisklasse „java.security.Policy". Die statische Factory-Methode „Policy.getPolicy()" instanziert und liefert die aktuell in der JVM konfigurierte Policy-Implementierung. Die zu verwendende Policy-Implementierung wird über die System-Property „policy.provider" konfiguriert. Dieser ist der vollqualifizierte Klassenname der gewünschten Policy-Implementierung zugewiesen. Ist hier nichts konfiguriert, wird die Default-Policy-Implementierung verwendet. Diese liest „.policy"-Dateien aus. Die Default-Implementierung befindet sich in der Klasse „sun.security.provider.PolicyFile".

Die zugewiesenen Berechtigungen werden über mehrere „.policy"-Dateien konfiguriert. Es gibt eine globale, systemweite „.policy"-Datei und eine benutzerspezifische „.policy"-Datei. Die systemweite steht unter „$JAVA_HOME/lib/security/java.policy", die benutzerspezifische unter „$USER_HOME/.java.policy". Die Platzhalter sind durch die entsprechenden systemspezifischen Pfade zu ersetzen. Anwendungsspezifische „.policy"-Dateien werden über das JVM-Argument „-Djava.security.policy=$POLICY_FILE" definiert.

Innerhalb von „.policy"-Dateien lässt sich über einen „keystore"-Eintrag ein Keystore definieren, der Public- und Private-Keys enthält. Dieser Eintrag ist relevant, wenn signierter Quellcode bestimmte Berechtigungen erhalten soll. Auf diese Aspekte wird hier nicht weiter eingegangen.

Über „grant"-Einträge werden bestimmtem Quellcode Berechtigungen zugewiesen. Die Berechtigungen werden innerhalb eines „grant"-Eintrags als „permission"-Einträge angegeben. *Listing 4* zeigt die komplette Syntax eines „grant"-Eintrags.

Die Angaben „SignedBy" und „CodeBase" sind optional und können auch weggelassen werden. „SignedBy" wird für die Vergabe von Berechtigungen für signierten Quellcode verwendet; „CodeBase" definiert die Herkunft von Quellcode. Hier kann beispielsweise Bezug auf ein „.jar"-File oder auch eine URL genommen werden. Die Principal-Angaben werden im Abschnitt über JAAS erläutert.

Der „grant"-Eintrag in *Listing 5* erteilt der kompletten Anwendung die Berechtigung, die Datei „main.config" zu lesen; außerdem wird die Berechtigung zum Editieren aller

Kundendaten erteilt. Wie solch eine Konfiguration um Principals und damit um angemeldete Benutzer verfeinert wird und wie sie eine vollständige Authentifizierung und Autorisierung von Benutzern ermöglicht, steht im Abschnitt über JAAS.

Authentifizierung und Autorisierung in Java SE unter Verwendung von JAAS

JAAS steht für „Java Authentication and Authorization Service" und ist ein Standard, der in Java-SE-Anwendungen für die Authentifizierung und Autorisierung von Entitäten, beispielsweise Benutzern, verwendet wird. Die Schnittstellen und Klassen des JAAS-Standards sind im Package „javax.security.auth" definiert.

JAAS baut auf der oben beschriebenen Sicherheits-Architektur auf und erweitert diese um die Möglichkeit, Zugriffskontrollen auf Basis der Entität durchzuführen, die den Quellcode aktuell ausführt. Das kann nicht nur ein interaktiver Benutzer, sondern auch ein Servicekonto sein.

Eine Anwendung, die JAAS verwendet, bleibt bei korrekter Verwendung unabhängig von der verwendeten Authentifizierungs-Technologie. Durch eine einfache Konfiguration kann eine neue Authentifizierungs-Technologie ähnlich wie ein Plug-in eingehängt werden. JAAS basiert auf Pluggable Authentication Modules (PAM, *siehe „https://de.wikipedia.org/wiki/Pluggable_Authentication_Modules")*.

Instanzen der Klasse javax.security.auth.Subject *(siehe „https://docs.oracle.com/javase/8/docs/api/javax/security/auth/Subject.html")* stellen Entitäten dar, die den Quellcode aktuell ausführen. Diese sind durch ein Set von Instanzen der Klasse "javax.Security.Principal" beschrieben. Außerdem enthält ein Subject das Set von öffentlichen und privaten Credentials. Öffentliche Credentials könnten Public-Keys und private Credentials Private-Keys sein. Ein Subject wird als Ergebnis einer Authentifizierung erstellt und um die beschreibenden Principals ergänzt.

Wie erläutert, sind Principals mit Subjects assoziiert und identifizieren ein Subject näher. Ein Principal könnte die Domäne eines angemeldeten Benutzers, der Benutzername oder auch eine E-Mail-Adresse sein. Für das Durchführen der Authentifikation muss als Erstes eine Konfiguration erstellt werden, die die zu verwendende Authentifizierungstechnologie definiert. Diese wird durch die abstrakte Basisklasse „javax.security.auth.login.Configuration" repräsentiert. Das JDK liefert bereits eine Default-Implementierung in der Klasse „sun.security.provider.

```
grant signedBy "signer_names", codeBase "URL",
  principal principal_class_name "principal_name",
  principal principal_class_name "principal_name"
  // Weitere principal-Einträge möglich
{

     permission permission_class_name "target_name", "action",
        signedBy "signer_names";
     permission permission_class_name "target_name", "action",
        signedBy "signer_names";
     // Weitere permission-Einträge möglich
};
```

Listing 4

```
grant {
  permission java.io.FilePermission "main.config", "read";
  permission de.bu.EditCustomerDataPermission "editAll";
};
```

Listing 5

```
Applikationsname {
  com.sun.security.auth.module.UnixLoginModule required;
};
```

Listing 6

ConfigFile" mit. Diese liest eine Konfigurationsdatei aus. Eine solche Konfigurationsdatei kann wie in *Listing 6* aussehen.

„Applikationsname" steht für einen frei definierbaren Identifikator, der für das Auslesen der korrekten Konfiguration verwendet wird. Innerhalb dieses Eintrags wird eine beliebige Menge von „javax.security.auth.spi.LoginModule"-Einträgen konfiguriert.

Ein „LoginModule" stellt eine Authentifizierungs-Technologie dar. Im Beispiel liefert das „UnixLoginModule" die Authentifizierungs-Informationen des Unix-Betriebssystems, auf dem die Anwendung ausgeführt wird. Ein Gegenstück für Windows-Betriebssysteme existiert als „com.sun.security.auth.module.NTLoginModule". So können beispielsweise Gruppen, in denen ein Benutzer Mitglied ist, zur Autorisierung innerhalb einer Java-SE-Anwendung verwendet werden.

Das „required"-Flag hinter „LoginModule" bedeutet, dass die Authentifizierung über dieses LoginModule erfolgreich durchgeführt werden muss. Weitere Möglichkeiten sind „requisite", „sufficient" und „optional". Die genaue Bedeutung dieser weiteren Flags steht in der JavaDoc der Configuration-Klasse. Beim Start der Anwendung wird der Dateiname dieser Datei über den Parameter „-Djava.security.auth.login.config" übergeben. Damit ist die Konfiguration aktiv.

Ergänzen der Policy-Implementierung um Principals

Um die ermittelten Identifizierungs-Informationen eines Subject, die Principals, für Autorisierungen verwenden zu können, müssen diese innerhalb eines „.policy"-Files ergänzt werden *(siehe Listing 7)*.

In diesem Fall können Subjects, die der Gruppe mit der GroupID *(siehe „https://en.wikipedia.org/wiki/Group_identifier")* „32456" angehören, die Datei „main.config" lesen und alle Kundendaten editieren. Man verwendet also für die Konfiguration von Zugriffsberechtigungen innerhalb der „policy"-Datei die Principals, die anhand eines LoginModule ermittelt werden. Durch das Hinzufügen weiterer „grant"- oder „permission"-Einträge lassen sich die Zugriffsberechtigungen beliebig fein gestaltet.

Login durchführen

Um die Authentifizierung durchzuführen, muss eine Instanz der Klasse „javax.security.auth.login.LoginContext" instanziiert werden. Auf dieser Instanz wird die „login"-Methode mit dem innerhalb der Konfiguration hinterlegten Applikationsnamen aufgerufen. Dadurch lassen sich die konfigurierten Login-Modules laden. Diese werden der Reihe nach evaluiert. Ist die Authentifizierung nicht erfolgreich, wird eine „javax.security.auth.login.LoginException" geworfen.

War der Vorgang erfolgreich, kehrt der Aufruf zurück und das fertige Subject kann über den Aufruf des Getters „getSubject()" abgerufen werden. Um eine Aktion unter der Identität eines bestimmten Subject auszuführen, ist jetzt eine der „doAs"- oder „doAsPrivileged"-Methoden aufzurufen. Der Unterschied besteht darin, dass die „doAs"-Methoden die Berechtigungen des Subject mit den Berechtigungen des aktuellen „SecurityContext" durch einen Permission-Intersect kombinieren *(siehe Listing 8)*.

Der Code deutet an, wie eine Authentifizierung innerhalb einer Anwendung aussehen kann. Der LoginContext wird hier als private Variable auf Klassenebene hinterlegt und in der „authenticate"-Methode erstellt. Soll nun ein geschützter Aufruf unter Verwendung der Berechtigungen des Subject erfolgen, wird „Subject.doAsPrivileged" mit dem aktuellen Subject aufgerufen. Die „ProtectionDomain" des Codes, der die Authentifizierung mit einem Aufruf von „LoginContext.login()" durchführt, und des Codes, der im Kontext eines bestimmten Subject ausgeführt wird, müssen getrennt sein.

Wenn eine Trennung dieser beiden ProtectionDomains nicht erfolgt, wird zwischen den Berechtigungen der beiden Domains ein Intersect durchgeführt. Das hat zur Folge, dass Berechtigungen, die nur in einer ProtectionDomain vorhanden sind, nicht wirksam werden. Es wäre also nicht möglich, Berechtigungen nur einem bestimmten Benutzer aufgrund seiner Principals zu erteilen. Zur Trennung muss die Methode „Subject.doAsPrivileged" mit „null" für den „AccessControlContext" aufgerufen werden. Der Aufruf erstellt einen neuen „AccessControlContext". Dieser enthält nur die Berechtigungen des Subject. Wird eine der „doAs"-Methoden verwendet, findet angesprochener Intersect statt. Ein Zugriff, der erteilt werden sollte, wird dann nicht erteilt.

Hinweise bei der Konfiguration von Permissions

Bei der Konfiguration von Policies für eine Anwendung ist es ratsam, allgemeine Berechtigungen, die für das Öffnen von Dateien, das Laden von Bibliotheken etc. notwendig sind, innerhalb der systemweiten „.policy"-Datei zu definieren. Innerhalb der applikations- beziehungsweise benutzerspezifischen „.policy"-Datei werden dann nur spezialisierte Berechtigungen hinterlegt. Sollte es Probleme bei der Konfiguration der Permissions geben, kann die JVM mit dem Parameter „-Djava.security.debug=access" gestartet werden. Damit ist eine genaue Analyse der erteilten und verweigerten Berechtigungen möglich. Hilft das nicht, kann auch ein Breakpoint innerhalb der „checkPermission"-Methode des AccessController gesetzt werden. Die Analyse der Variablen ermöglicht einen detaillierten Einblick in die gebildeten „ProtectionDomains" und deren Rechte. Das Konzept des Permission-Intersect sollte verinnerlicht werden.

Verwendung von Callbacks für die Kommunikation mit der Anwendung

Wie in der Einführung erwähnt, ist es wichtig, dass Anwendungen unabhängig von der Authentifizierung bleiben und dass keine enge Kopplung zu einer bestimmten Authentifizierungstechnologie entsteht. Aus diesem Grund werden Callbacks für die Kommunikation zwischen Anwendung und LoginModules verwendet.

Nachdem ein LoginModule instanziiert wurde, ist es durch einen Aufruf der initialize-Methode zu initialisieren. In diesem Aufruf wird eine Instanz von „javax.security.auth.callback.CallbackHandler" übergeben. Dieser wird von der aufrufenden Applikation implementiert. Innerhalb der „handle"-Methode ist hier auf unterschiedliche Callbacks zu reagieren, um den LoginModules Authentifizierungs-Informationen zu übermitteln.

Innerhalb des JDK gibt es bereits einige Implementierungen solcher Callbacks. Beispielsweise fragt die Klasse „javax.security.auth.callback.PasswordCallback" das Passwort eines Benutzers ab.

Authentifizierung und Autorisierung in Java EE unter Verwendung von JASPIC

Innerhalb von Java EE gab es lange keinen einheitlichen Standard für die Authentifizierung und Autorisierung innerhalb eines Containers. Viele Hersteller haben aus diesem Grund die JAAS-Spezifikation in einer für ihre jeweilige Java-EE-Implementierung spezifischen Form ausgelegt. Das führt dazu, dass Java-EE-Anwendungen teilweise nicht zwischen Java-EE-Containern portabel sind. Seit dem Jahr 2002 wurde an der JASPIC-Spezifikation im Rahmen des JSR-196 *(siehe „https://www.jcp.org/en/jsr/detail?id=196")* gearbeitet. JASPIC steht für Java Authentication Service Provider Interface for Containers. Die erste finale Version dieses JSR wurde im Jahr 2013 veröffentlicht.

JASPIC definiert auf Basis eines abstrakten, nachrichtenbasierten Verarbeitungsmodells einen Authentifizierungs- und Autorisierungsstandard zur Verwendung innerhalb eines Java-EE-Containers. Dieser wird über Profile für bestimmte Verarbeitungsmodelle spezialisiert; so ist beispielsweise innerhalb der Spezifikation ein Profil für die Request-Response-Verarbeitung über HTTPServlets unter dem Namen „Servlet Container Profile" definiert.

Zentrale Klassen der JASPIC-Spezifikation

Innerhalb der JASPIC-Spezifikation sind einige zentrale Klassen und deren Verantwortlichkeit innerhalb des Verarbeitungsmodells definiert *(siehe Abbildung 3)*. Die Klassen sind innerhalb

```
// "Grant" entry for specified unix group (GID)
grant com.sun.security.auth.UnixNumericGroupPrincipal "32456" {
  permission java.io.FilePermission "main.config", "read";
  permission de.bu.EditCustomerDataPermission "editAll";
};
```

Listing 7

```
private LoginContext loginContext;
…
private void authenticate() throws LoginException {
  this.loginContext = new LoginContext(APPLICATION_NAME);
  loginContext.login();
}
…
Subject.doAsPrivileged(this.loginContext.getSubject(), new
PrivilegedAction<Void>() {     @Override
  public Void run() {
     /* Open restricted application here. */
     return null;
  }
}, null);
```

Listing 8

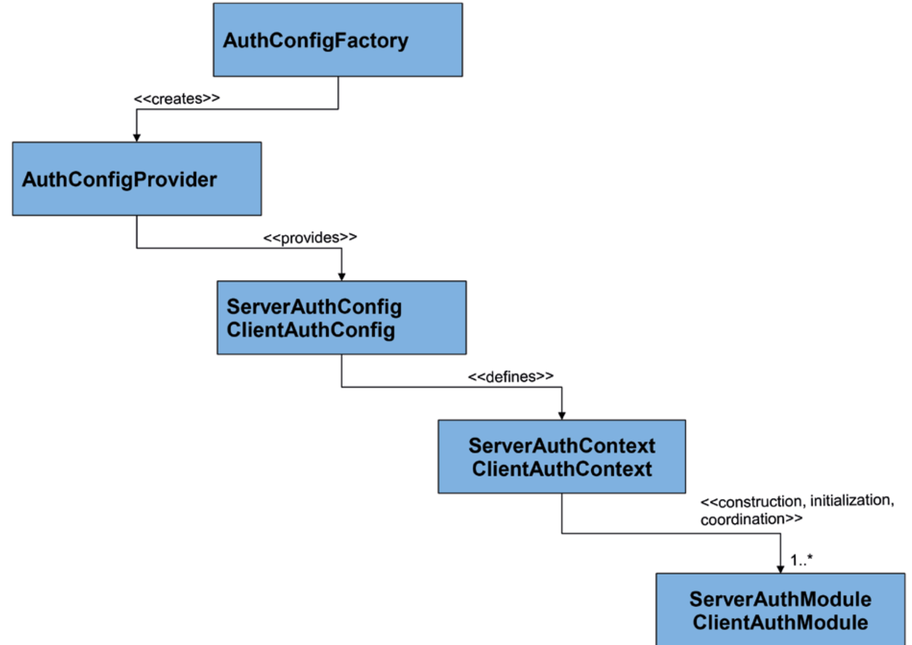


Abbildung 3: Zentrale Klassen der JASPIC-Spezifikation

des Package „javax.security.auth.message" und dessen Sub-Packages definiert.

Die Klassen „AuthConfigFactory", „AuthConfigProvider", „ServerAuthConfig" und „ClientAuthConfig" sind für das Erstellen und Konfigurieren der verwendeten „ServerAuthContext"- und „ClientAuthContext"-Klassen sowie der „ServerAuthModule" und „ClientAuthModule" verantwortlich. Diese sind nur zu implementieren, wenn nicht auf den Möglichkeiten eines bestehenden Containers aufgesetzt wird. Innerhalb eines Containers wie Tomcat ab Version 9 *(siehe „https://tomcat.apache.org/tomcat-9.0-doc/config/jaspic.html")* oder Oracle WebLogic 12c *(siehe „https://docs.oracle.com/cd/E24329_01/web.1211/e24422/jaspic.htm#SECMG643")* muss nur ein „ServerAuthModule" oder „ClientAuthModule" implementiert und konfiguriert sein.

Die Klassen mit dem Präfix „Server", nehmen Authentifizierungs- beziehungsweise Autorisierungsvorgänge gemäß dem in JASPIC definierten, nachrichtenbasierten Verarbeitungsmodell auf Serverseite vor. Klassen mit dem Präfix „Client" übernehmen entsprechende Vorgänge auf der Clientseite.

Mappen der Autorisierungs-Informationen beziehungsweise Principals auf Java-EE-Rollen

Die Autorisierung innerhalb der Java-EE-Spezifikation basiert auf Rollen. Die für eine Komponente, einen Methoden-Aufruf oder auch das Aufrufen einer Servlet-URL benötigten Rollen werden entweder programmatisch oder deklarativ definiert. Bei der deklarativen Vorgehensweise können Annotationen oder Deployment-Deskriptoren eingesetzt werden. Details dazu stehen in den Dokumentation von Oracle.

JASPIC verwendet genauso wie JAAS innerhalb der Authentifizierungs-Module Callbacks, um Informationen an die aufrufende Anwendung weiterzugeben. Der große Unterschied ist, dass die aufrufende Anwendung in diesem Fall der Java-EE-Container ist. Dieser stellt auch die Callbacks zur Verfügung und verarbeitet die gelieferten Informationen in einer für den jeweiligen Container passenden Weise.

Der „javax.security.auth.message.callback.GroupPrincipalCallback" liefert der Laufzeit-Umgebung die Rollen, der „javax.security.auth.message.callback.CallerPrincipalCallback" den Remote-User und damit den Aufrufer der Anwendung. So lassen sich zum Beispiel Rollen, die deklarativ für einen Methodenaufruf mit der Annotation „@RolesAllowed" definiert sind, gegen die Rollen beziehungsweise Gruppen prüfen, die der entsprechende Callback liefert.

Programmatisch kann innerhalb eines Servlet-Containers der Aufruf von „HttpServletRequest.isUserInRole(role : String) : boolean" für das Überprüfen verwendet werden. Innerhalb eines EJB-Containers lässt sich das gemappte Principal über die Methode „getCallerPrincipal() : java.security.Principal" abrufen. Die Überprüfung, ob der Aufrufer einer bestimmten Rollen angehört, kann mit der Methode „isCallerInRole(role : String) : boolean" durchgeführt werden.

Fazit

Die Sicherheits-Architektur von Java SE deckt viele Anwendungsfälle über bestehende oder selbst implementierte Permissions ab. Oftmals muss hier eine Einstiegshürde überwunden werden, bis das Konzept der „Protection-Domains" und des „Permission-Intersect" klar ist. Das liegt oftmals daran, dass das Permission-System durch das Fehlen eines Security-Manager nicht aktiv ist. Durch die Entkopplung der Applikation von der Authentifizierungs-Technologie über Callbacks und das Einhängen von LoginModules kann eine Applikation vollständig unabhängig von einer konkreten Technologie gehalten und später einfach durch einen weiteren Konfigurationseintrag geändert beziehungsweise erweitert werden.

JASPIC adaptiert und standardisiert dieses Modell für Java-EE-Container. Aktuell ist es so, dass viele Hersteller die JASPIC-Spezifikation implementieren, indem sie Wrapper um ihre bisherigen Authentifizierungs- und Autorisierungs-APIs für die Erfüllung von JASPIC bereitstellen. Die Verbreitung und Adaption von JASPIC hat seit dem Jahr 2013 stetig zugenommen. Innerhalb von WebLogic, GlassFish oder auch Tomcat stehen funktionierende Implementierungen bereit. Ein weiterer Schritt hin zu mehr Portabilität von Anwendungen zwischen Java-EE-Containern ist damit getan.

Philipp Buchholz
philipp.buchholz@esentri.com

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

Last- und Performance-Test verteilter Systeme mit Docker & Co.

Dr. Dehla Sokenou, GEBIT Solutions

Moderne Virtualisierungs-Umgebungen haben das Potenzial, die Software-Entwicklung zu revolutionieren. Neben der immer engeren Verzahnung von Entwicklung und Betrieb (Stichwort „DevOps") unterstützen Lösungen wie Docker auch andere Phasen und Tätigkeiten im Software-Entwicklungsprozess, etwa den Test. Hierbei profitiert insbesondere der Last- und Performance-Test.

Auch wenn Testen inzwischen aus der Exotenecke in der Gegenwart angekommen ist, verursachen Tests immer noch einen nicht unerheblichen Anteil an den Kosten der Software-Entwicklung. Test-Automatisierung hilft, diese Kosten in den Griff zu bekommen, und sollte, wo immer möglich, den Vorzug vor manuellen Tests erhalten. Dabei kann Automatisierung sowohl bei der Vorbereitung, beim Aufsetzen der Test-Umgebung, beim Deployment der zu testenden Anwendung wie auch bei der eigentlichen Testausführung und -auswertung zum Einsatz kommen.

Neben den fachlichen Tests auf Unit-, Integrations- und System-Ebene müssen zusätzliche Tests das nichtfunktionale Verhalten eines Systems berücksichtigen. Ein Beispiel dafür – neben anderen – sind Last- und Performance-Tests, die das Verhalten des Systems in seinen Grenzbereichen und außerhalb seiner Grenzen überprüfen. Der erreichbare Automatisierungsgrad ist hier im Gegensatz zu anderen nichtfunktionalen Tests besonders hoch.

Verteilte Systeme stellen beim Test eine besondere Herausforderung dar, weil zusätzlich zum Verhalten eines einzelnen Systems die Kommunikation innerhalb des Gesamtsystems eine große Rolle spielt. Für den Test sollte eine möglichst realitätsnahe Umgebung zur Verfügung zu stehen. Allerdings ist es meist nicht möglich, Szenarien mit 10.000 oder mehr vollwertigen Knoten zu Testzwecken aufzusetzen. Wie testet man also die Performance solcher Systeme möglichst realistisch und im Idealfall automatisiert?

Moderne Virtualisierungs-Umgebungen wie Docker bieten sich hier als Lösung an. Aber auch die Verwendung von VMs als Test-Umgebung hat weiterhin ihre Daseinsberechtigung. Virtualisiert wird hierbei das zu testende System. Wenngleich Virtualisierungs-Umgebungen auch noch in anderer Hinsicht beim Testen hilfreich sein können, etwa bei der Virtualisierung der Build- und Test-Umgebung, ist dies nicht Gegenstand dieses Artikels. Nachfolgend

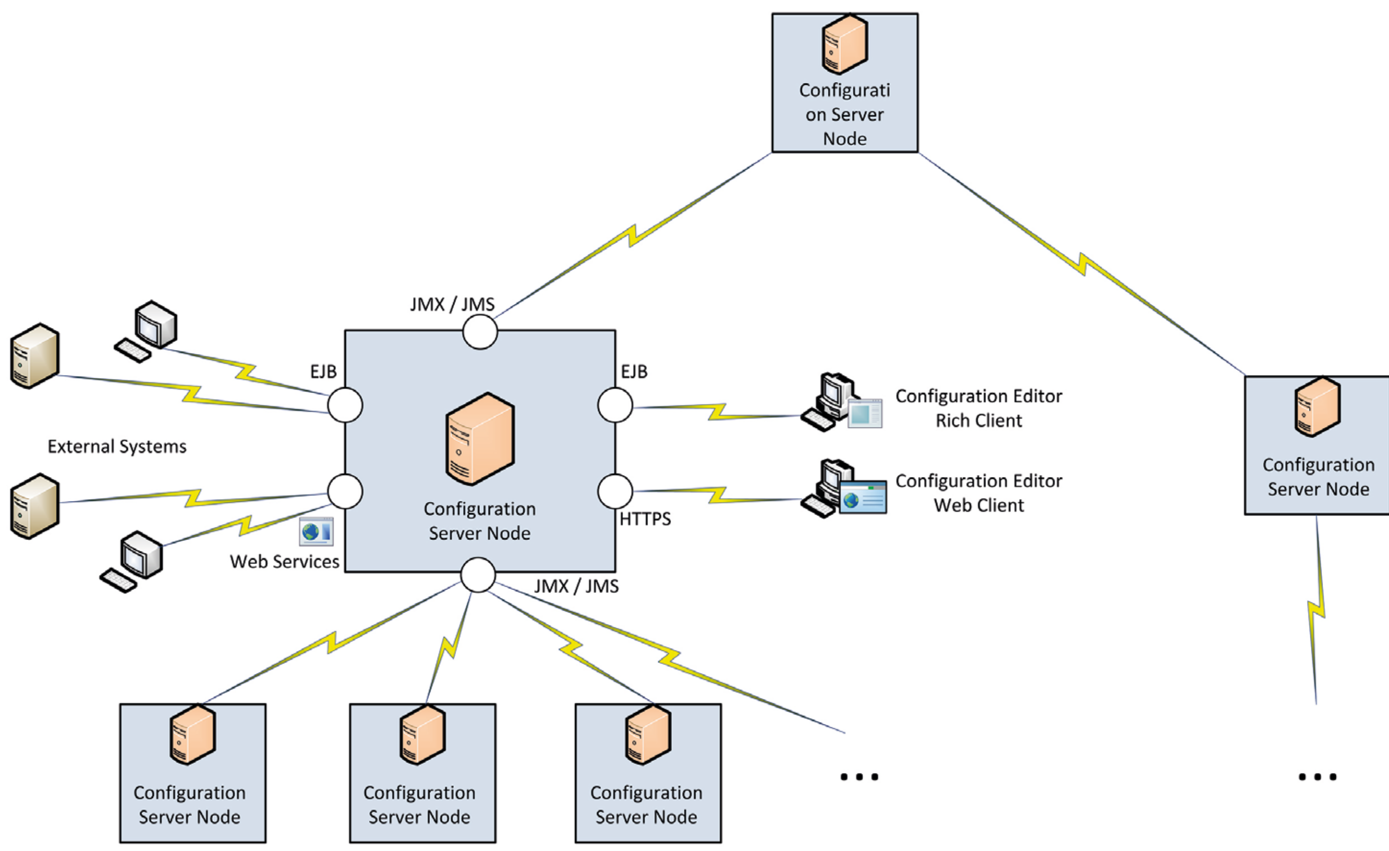


Abbildung 1: Komponenten des Konfigurationssystems

sind verschiedene Szenarien anhand eines Praxisbeispiels aufgezeigt und bewertet.

Ein Praxisbeispiel

Eines der Produkte aus dem Unternehmen des Autors ist ein weltweit verteilt eingesetztes System zur Erstellung und Verteilung von Anwendungskonfigurationen *(siehe Abbildung 1)*. Es besteht im Kern aus einer Server-Komponente mit diversen Schnittstellen, einem Web-Client und einem Rich-Client. Ein Anwendungsfall ist beispielsweise die Übertragung wichtiger Betriebsparameter für Kassen- oder Warenwirtschaftssysteme, die in verschiedenen Ländern und Regionen zu unterschiedlichen Zeiten in unterschiedlichen Ausprägungen definiert sein sollen.

Aus Gründen der Betriebssicherheit und Lastverteilung ist üblicherweise die Server-Komponente an verschiedenen Standorten verteilt im Einsatz. Die einzelnen Server-Instanzen kommunizieren miteinander, einerseits um Betriebsdaten auszutauschen, andererseits um Konfigurationsdaten weltweit zu replizieren und somit lokal zur Verfügung zu stellen. Die einzelnen Server-Instanzen bilden dabei eine Baumstruktur mit einem Top-Level- und Kind-Knoten über mehrere Ebenen; dabei kann ein Netzwerk leicht 10.000 und mehr Knoten umfassen. Für die Kommunikation der Knoten untereinander werden aktuell JMS- beziehungsweise JMX-Schnittstellen zur Verfügung gestellt, zukünftig alternativ auch weitere, etwa Webservice-Schnittstellen. Die Kommunikation erfolgt dabei primär zwischen Eltern und den direkten Kindern, für einzelne Funktionen aber auch zwischen Eltern und indirekten Kindern.

Es gibt verschiedene Arten von Clients des Systems. Die im Rahmen des Projekts entwickelten eigenen Clients dienen zur Erstellung und Änderung der Konfigurationen sowie zur Überwachung des Server-Netzwerks und nutzen dazu spezielle eigene Schnittstellen zur Kommunikation. Daneben gibt es auch eine Reihe bekannter sowie eine Reihe unbekannter externer Systeme. Diese nutzen zur Kommunikation entweder die öffentlichen EJB- oder die Web-Service-Schnittstellen. Dabei umfassen die bekannten externen Systeme sowohl projektbezogene Eigenentwicklungen als auch Systeme von Drittherstellern.

Ein Schwerpunkt liegt auf dem Last- und Performance-Test des gezeigten Systems. Es war unter anderem aufgefallen, dass unter bestimmten Bedingungen wie zu geringer Netzwerk-Bandbreite bei gleichzeitig geringem Speicherplatz die verwendete Messaging-Lösung (HornetQ) instabil wurde. Um jedoch das Szenario zu testen und Optimierungen an der Software und der HornetQ-Konfiguration vornehmen zu können sowie Empfehlungen für den Betrieb zu geben, war es notwendig, eine möglichst realistische Test-Umgebung aufzubauen. Im ersten Schritt wurden VMs eingesetzt, im zweiten Docker-Container.

Last- und Performance-Test auf VMs

Um die Umgebung, bei der die Probleme auftraten, möglichst realistisch nachbilden zu können, wurden VMs aufgesetzt, die die Realität soweit wie möglich abbildeten. Die Basis bildete ein spezieller, exklusiv für Last-und Performance-Test genutzter VMware-ESXi-Host. Auf diesem können Projekte, die eine entsprechende Testumgebung brauchen, VM-Templates anlegen, die anschließend für den eigentlichen Test vervielfältigt werden.

Als Betriebssystem für die VMs kam das beim entsprechenden Kunden verwendete Betriebssystem (Windows Server) zum Einsatz. Auf einer Template-VM wurde die gesamte notwendige Software inklusive Application-

Abbildung 2: Simulation von Netzwerk-Problemen

Server und Datenbank installiert. Haupt- und Festplattenspeicher wurden soweit reduziert, dass sie dem realen System entsprachen. Um Netzwerk-Probleme wie geringe Bandbreite, Package-Loss und Verbindungsabbrüche zu simulieren, kamen zwei Werkzeuge zum Einsatz: NetLimiter *[1]* zur Begrenzung der verfügbaren Bandbreite und clumsy *[2]*, um wahrscheinlichkeitsgesteuert Netzwerk-Probleme zu erzeugen *(siehe Abbildung 2)*.

Da das Serversystem bereits die Möglichkeit einer Auto-Initialisierung bot, um das Aufsetzen der Server-Instanzen zu erleichtern, konnte diese für den Last- und Performance-Test in leicht abgewandelter Form wiederverwendet werden. Es reichte also aus, auf dem VM-Template einen Server so einzurichten, dass er mit dem System gestartet wird. Nach der Vervielfältigung ermittelten die gestarteten Server-Instanzen ihre eigene Identität und ihre Position im Netzwerk anschließend automatisch, sodass ein manuelles Aufsetzen des Netzwerks überflüssig wurde.

Auch das Monitoring der einzelnen Instanzen erfolgte teilweise über bereits vorhandene Funktionalität. Jeder Server meldet regelmäßig seinen Status an seine Eltern-Knoten; auch die im Augenblick zu verarbeitende Last ist Teil dieser Statusberechnung. Damit kann über den Top-Level-Knoten der Zustand des gesamten Netzwerks überwacht werden.

Stehen solche Funktionen nicht bereits zur Verfügung, müssen sie bereitgestellt werden. So ist beispielsweise ein manuelles Aufsetzen einer großen Anzahl von Knoten des verteilten Systems in der Regel mit viel Aufwand verbunden. Ein System, das beim Testen nicht überwacht werden kann, ist nicht testbar, da Testbarkeit neben der Steuerbarkeit der Eingaben auch die Überwachung der erwarteten Ergebnisse verlangt.

Der ESXi-Host wurde für den Test vollständig ausgelastet, es konnten knapp über sechzig Server erstellt und betrieben werden. Dies reichte für ein realistisches Szenario aus, sodass in einem ersten Schritt die HornetQ-Konfiguration optimiert werden konnte. So wurde zum Beispiel zur Reduzierung des benötigten Arbeitsspeichers das HornetQ-Paging eingeschaltet. Weitere Optimierungen wurden am System selbst vorgenommen, um die Datenreplikation bei größeren Datenmengen zu entlasten. So hat man auch eine weitere Stage zur Zwischenspeicherung von Nachrichten in einer Datenbank eingeführt, bevor sie an andere Instanzen weitergereicht werden. Als Nebeneffekt hat die Einführung der zusätzlichen Stage einen Austausch des Transport-Layers HornetQ durch eine andere Technik möglich gemacht, sodass nun alternativ unter anderem auch Web-Services genutzt werden könnten.

Eine Umgebung auf Docker

Der Einsatz von VMs für realitätsnahe Tests war aus unserer Sicht unabdingbar, allerdings konnte aufgrund der Limitierungen des ESXi-Hosts keine große Menge von Instanzen

betrieben werden. Eine Unterstützung von deutlich mehr Instanzen war jedoch eine der Anforderungen, die validiert werden mussten. Es war also eine Möglichkeit notwendig, deutlich mehr als die bisher im Test verwendeten sechzig Instanzen zu unterstützen. Es wäre natürlich möglich gewesen, die benötigten Ressourcen für den Test temporär zu mieten. Einer Nutzung von Docker *[3]* wurde allerdings der Vorzug gegeben.

Da das vorgestellte System in Java implementiert ist, auf einem Standard-Application-Server (JBoss, WildFly) läuft sowie unterschiedliche Datenbanken unterstützt, ist ein Betrieb auf einem Linux-Betriebssystem möglich, zumal Linux zu den explizit unterstützten Plattformen gehört.

Man hat daher entschieden, das System als Docker-Instanzen auf Debian-Basis aufzusetzen. Das offizielle Debian-Image für Docker ist sehr minimal gehalten und somit ein erster Schritt, um möglichst viele Instanzen betreiben zu können. Zudem wurden die Teile des Systems, die für den Last- und Performance-Test nicht relevant sind, durch Mock-Implementierungen ersetzt oder gar nicht erst eingerichtet. Die Blattknoten im Server-Netzwerk wurden beispielsweise als reine Datensenken implementiert, da insbesondere die Last auf dem Top-Level-Knoten sowie den Zwischen-Knoten von Interesse war, da es bisher nur dort bisher zu Auffälligkeiten gekommen war. Der Einsatz von Mocks und Minimal-Implementierungen sollte je nach eigenen Anforderungen kritisch hinterfragt werden, da gegebenenfalls die Ergebnisse des Tests nur bedingt oder gar nicht auf das real im Einsatz befindliche System übertragen werden können.

Docker bietet zwei unterschiedliche Methoden, ein System aufzusetzen und zu vervielfältigen. Beide basieren zunächst einmal auf einem Image. Die öffentliche Docker-Registry *[4]* stellt eine Menge von offiziellen und inoffiziellen Images zur Verfügung, sodass dies ein guter Startpunkt für das eigene System ist. Auf Basis eines Images lässt sich nun ein laufender Container starten.

Die erste Möglichkeit nimmt ein vorhandenes Image, also in unserem Fall ein Debian-Image, und startet es als Container. Anschließend kann dort die notwendige Software installiert, daraus wiederum ein Image erzeugt und dieses in einer Registry für die weitere Verwendung abgelegt werden. Aus Erfahrung des Autors ist dieses Vorgehen zwar das einfachere, allerdings kann es dazu kommen, dass gegebenenfalls auch Software installiert wird, die für den eigentlichen Zweck nicht notwendig ist, sondern lediglich zur Unterstützung des Administrators bei der Einrichtung des Containers dient, zum Beispiel die Installation eines Texteditors, um mal schnell den Inhalt einer Datei anzupassen. Dies kann den Container und damit das daraus erzeugte Image unnötig aufblähen. Es ist also die entsprechende Disziplin bei der Einrichtung des Containers geboten.

Die zweite Möglichkeit ist die Verwendung eines Docker-Files. Dieses basiert ebenso auf einem vorhandenen Image und beschreibt alle Befehle, um den Container mit notwendiger Software auszustatten und diese laufen zu lassen. Der Vorteil ist, dass man sich hier in der Regel auf die essenziell notwendigen Befehle beschränkt, um das Docker-File nicht unnötig anwachsen zu lassen. Die Installation eines Texteditors hätte hier zum Beispiel gar keinen Vorteil.

Ein weiterer Vorteil der Verwendung von Docker-Files ist die einfache Anpassung an sich ändernde Versionen von genutzter Software. Ändert sich etwa die Version des Application-Servers oder der Datenbank, so ist einfach das Docker-File entsprechend anzupassen und erneut auszuführen.

Zu beachten ist, dass jeder Befehl im Docker-File ein neues Image erzeugt und im Docker-Cache speichert. Wenn dieses bereits vorhanden ist, werden die entsprechenden Docker-Befehle nicht ausgeführt; es wird stattdessen auf das vorhandene Image im Cache zurückgegriffen. Dies sollte im Auge behalten werden, wenn beispielsweise der Download von Installationsdateien von einem Build-Server in einem Docker-File erfolgen soll. Dazu sollte einer der Docker-File-Befehle „ADD" oder „COPY" verwendet werden, die den Cache invalidieren, wenn sich der Inhalt und damit die Checksumme der hinzugefügten oder kopierten Dateien geändert haben.

Für Remote-Dateien sollte allerdings „wget" oder „curl" verwendet werden, denn „ADD" unterstützt zwar den Download von Remote-Dateien, aber keine Authentifizierung. Zudem können mit „ADD" heruntergeladene Archive nicht im gleichen Befehl wieder gelöscht werden, was wiederum die Imagegröße negativ beeinflusst.

Alternativ lässt sich der Cache deaktivieren, dann werden allerdings die Befehle im Docker-File immer vollständig abgearbeitet, wodurch sich das Aufsetzen wiederum verlangsamt. Um die Anzahl der erzeugten Images klein zu halten, sollte man Befehle – wo möglich – zusammenfassen; dazu werden Befehle einfach mit „&&" verknüpft. *Abbildung 3* zeigt einen Ausschnitt aus der Ausführung des Docker-Files, bei dem die PostGreSQL-Datenbank aufgesetzt und der Port 5432 containerübergreifend verfüg-

```
Step 9 : RUN apt-get update && apt-get -y install postgresql-${PGVERSION} postgresql-client-${PGVERSION}
 ---> Using cache
 ---> 7fd0c0260d4d
Step 10 : USER postgres
 ---> Using cache
 ---> 253bfc7b8b29
Step 11 : RUN /etc/init.d/postgresql start &&    psql --command "CREATE USER test WITH SUPERUSER PASSWORD 'test';" &&    createdb -O test test
 ---> Running in f708d2ca31a1
Starting PostgreSQL 9.4 database server: main.
CREATE ROLE
 ---> 221e15ae9f4d
Removing intermediate container f708d2ca31a1
Step 12 : RUN echo "host all  all    0.0.0.0/0  md5" >> /etc/postgresql/${PGVERSION}/main/pg_hba.conf
 ---> Running in 280966343f8f
 ---> 9d99b6240cbf
Removing intermediate container 280966343f8f
Step 13 : RUN echo "listen_addresses='*'" >> /etc/postgresql/${PGVERSION}/main/postgresql.conf
 ---> Running in 1c5367701898
 ---> ce4c41f34a8a
Removing intermediate container 1c5367701898
Step 14 : EXPOSE 5432
 ---> Running in 3b0cf5b2b657
 ---> a233038dad9e
```

Abbildung 3: Ausschnitt aus einem Docker-File-Run

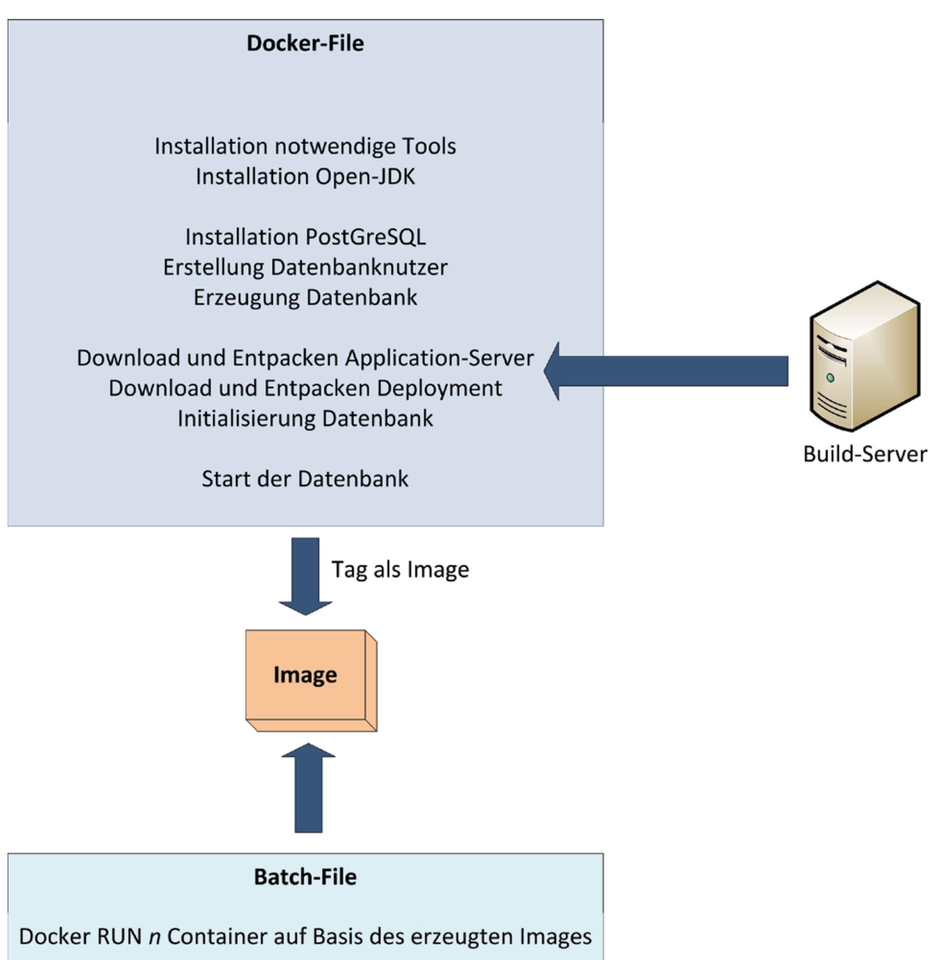


Abbildung 4: Aufsetzen der Umgebung mit Docker

bar gemacht wird („EXPOSE"). Wird auf den Cache zurückgegriffen, so ist dies bei der Ausführung des Docker-Files ersichtlich.

Der gesamte Prozess des Aufsetzens mithilfe eines Docker-Files ist schematisch in *Abbildung 4* dargestellt. Zunächst wird die benötigte Software installiert. Anschließend werden der Application-Server, eine angepasste Version mit allen benötigten Modulen und der passenden Konfiguration sowie das Deployment für den Application-Server vom Build-Server heruntergeladen. Das so erzeugte, entsprechend getaggte Image wird nun mit dem Befehl zum Starten des Application-Servers sowie der Identität des jeweiligen Containers gestartet.

Die Last- und Performance-Tests wurden auf demselben ESXi-Host durchgeführt wie die Tests mithilfe von VMs. Es konnten knapp dreihundert Docker-Container auf dem Host betrieben werden. Das Verhältnis von Instanzen auf VM zu Instanzen auf Docker beträgt also 1:5. Dabei wurde der ESXi-Host vom Docker-Host noch nicht einmal vollständig ausgelastet, sodass hier noch Spielraum nach oben ist.

Fazit

Tests unter Verwendung von VMs haben somit ebenso ihre Berechtigung wie Tests mithilfe von Docker. Beide bieten Vor- und Nachteile und eignen sich – je nach Testziel – beide gut für den Last- und Performance-Test.

Last- und Performance-Tests auf VM-Basis sind immer dann sinnvoll, wenn eine möglichst realistische Abbildung der Wirklichkeit gefordert ist und diese sich mithilfe von Docker nicht abbilden lässt. Gerade Betriebssysteme, die wie ältere Windows-Versionen keine Docker-Unterstützung bieten, lassen sich nur auf einer VM aufsetzen. So ist unter anderem das Verhalten bei vielen offenen Netzwerk-Verbindungen unter Windows grundlegend anders als unter einem Unix-basierten System. In diesem Fall hätte ein Test in Linux-basierten Docker-Containern keinerlei Aussagekraft für das Verhalten des realen Systems unter Windows.

Allerdings haben VMs den Nachteil, dass sie eher schwergewichtig sind. Es ist immer ein ganzes Betriebssystem notwendig, während Docker-Container sich Ressourcen teilen können. Zudem dauert das Aufsetzen der VMs relativ lange, da jeweils eine vollständige VM diverse Male geklont und angepasst werden muss.

Container sind sehr viel leichtgewichtiger, der Start eines Containers aus einem Image geht sehr schnell und auch das Aufsetzen der Images mithilfe von Docker-Files ist durch die Verwendung des Cache ein schneller Prozess. Zudem ermöglichen Docker-Files eine einfache Anpassung der erzeugten Images und damit der erzeugten Container bei sich ändernden Software-Versionen, während auf einer VM die Installation eines Updates notwendig ist.

Wird der Test wiederholt, startet jeder Test in einem Docker-Container „clean", ein Bereinigen oder eine Neuinstallation ist also nicht notwendig. Bei Verwendung von VMs müssen diese entweder neu geklont werden oder alternativ muss zumindest eine Herstellung des Initialzustands des zu testenden Systems erfolgen, beides eher aufwändige Prozesse. So sind in den meisten Fällen Docker-Container dem Einsatz von VMs beim Last- und Performance-Test vorzuziehen.

Referenzen

[1] clumsy: *https://jagt.github.io/clumsy*
[2] NetLimiter: *https://www.netlimiter.com*
[3] Docker: *https://www.docker.com*
[4] Offizielle Docker-Registry: *https://hub.docker.com*

Dr. Dehla Sokenou
dehla.sokenou@gebit.de

Dr. Dehla Sokenou promovierte im Jahr 2005 an der Technischen Universität Berlin über das Thema „UML-basiertes Testen objektorientierter Systeme". Seit Anfang 2006 ist sie als Senior Software Consultant bei GEBIT Solutions am Standort Berlin tätig. Neben Projektleitung, Konzeption und Entwicklung großer objektorientierter Softwaresysteme mit modellbasierten Methoden umfassen ihre Schwerpunkte modellgetriebenes Requirements Engineering und modellbasiertes Testen. Seit dem Jahr 2016 ist sie stellvertretende Sprecherin des Arbeitskreises „Testen objektorientierter Programme / Model-Based Testing" der GI-Fachgruppe „Test, Analyse und Verifikation von Software".

Automatisiertes Testen in Zeiten von Microservices

Christoph Deppisch und Tobias Schneck, ConSol Software GmbH

Die Software-Entwicklung ist im Wandel. Immer schneller, immer häufiger, immer einfacher müssen neue Features in Produktion gebracht werden. Große, schwergewichtige Alleskönner werden durch mehrere kleine, individuelle Services ersetzt. Jeder Microservice bildet einen Aspekt der gesamten Fachlichkeit ab und lässt sich deshalb unabhängig entwickeln und warten. Welche Auswirkungen hat diese veränderte Sicht einer Software-Architektur auf die Qualitätssicherung in der Entwicklung? Ist hier auch alles einfacher, schneller und besser? Die Antwort lautet: „Ja und nein". Während die dezentralisierten Services Vorteile für die Testbarkeit und mehr Flexibilität versprechen, ist die Integration der Services eine Herausforderung im automatisierten Test.

„Eine Software muss ausreichend getestet werden." Dieser Grundsatz ist so alt wie das erste „Hello World" – aber in Zeiten von Microservices und Continuous Delivery treffender denn je. Software muss immer schneller den Weg in den Markt finden. Änderungen, Bugfixes und Features sollen im kontinuierlichen Rhythmus in möglichst kurzen Abständen in Releases gepackt und dem Nutzer zur Verfügung gestellt werden. Dieses Vorgehen hat auch unweigerlich Auswirkungen auf die Qualitätssicherung in der Software-Entwicklung. Man kann es sich nicht mehr leisten, vor einem Release aufwändige Tests zu fahren, um dann kurz vor der Deadline diverse Bugs zurück an die Entwicklung zu melden, in der diese dann umständlich in weiteren Testzyklen verifiziert werden müssen.

Die Qualitätssicherung muss voll automatisiert und kontinuierlich schon während der Entwicklung stattfinden. „Continuous Integration" nennt sich dieser Ansatz, in dem alle Tests vollständig automatisiert ablaufen und fest in den Build-Lifecycle einer Anwendung integriert sind. Mit jeder Code-Änderung werden alle Tests ausgeführt und geben bei eventuell aufgetretenen Fehlern schnelles Feedback an die Entwickler.

Im Bereich der Unit-Tests ist diese Automatisierung bereits als Standard etabliert. Die Entwickler erstellen die Unit-Tests entwicklungsbegleitend und integrieren sie fest in den Build-Lifecycle mit Tools wie Maven [1], Gradle [2] und Jenkins [3].

Testbedarf auf mehreren Ebenen

Unit-Tests sind leider nicht in der Lage, alle Aspekte einer Software ausreichend unter Test zu stellen. Sie durchlaufen, wie der Name (Unit = Einheit) schon sagt, kleine Code-Abschnitte, also eine Klasse oder Methode. Dies findet in kompletter Isolation zu anderen Einheiten statt. Alle Abhängigkeiten zu Klassen, Komponenten oder Ressourcen werden im Test durch Mocks ersetzt. Diese Isolation hat viele Vorteile. Die Unit-Tests sind sehr schnell und ohne großen Aufwand wiederholt ausführbar. Eine direkte Folge dieser Isolation ist aber auch, dass wir die Interaktion und Integration mit anderen Komponenten nicht getestet haben. Wichtige Aspekte wie User Interfaces, Konfiguration, Dependency Injection, Ressourcen-Verwaltung und Schnittstellen nach außen bleiben damit ungetestet.

Vor allem die zuletzt genannten Schnittstellen bekommen im Sinne von Microservices eine immer größere Bedeutung. Die individuellen Services tauschen untereinander Daten über Schnittstellen wie HTTP REST oder JMS aus. Die Schnittstellen folgen zwar gewissen Regeln, befinden sich aber auch stetig im fachlichen Umbau. Die Zusammenarbeit mehrerer Services und die reibungslose Integration der Services untereinander sind daher enorm wichtige Aspekte für die Qualitätssicherung.

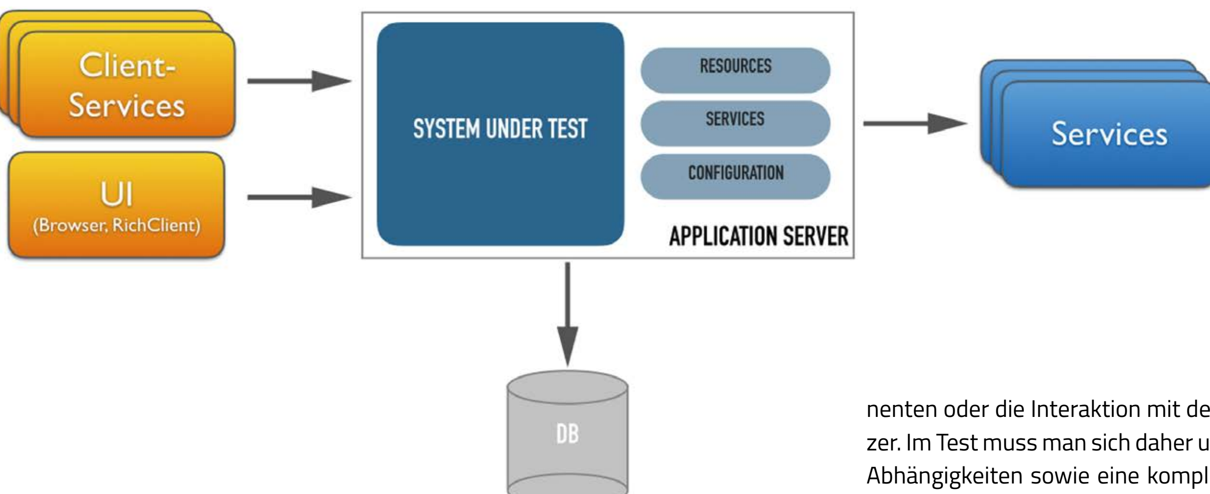


Abbildung 1: Beispiel-Infrastruktur im Integrations- beziehungsweise UI-Test

Wenn alle Units für sich allein erfolgreich getestet wurden, heißt das noch lange nicht, dass auch die Zusammenarbeit aller Units reibungslos funktioniert. Hier helfen nur die Integrations- beziehungsweise Acceptance-Tests, die genau das Zusammenspiel mehrerer Komponenten als primäres Testziel verfolgen. Hinzu kommen die UI-Tests, die sich mit der Benutzeroberfläche einer Software beschäftigen und die richtige Anzeige der Daten testen. Auch hier ist ein funktionsfähiges Backend mit entsprechenden Daten in der Datenhaltung Voraussetzung für den sinnvollen Test. Natürlich müssen diese Tests im Sinne des Continuous Delivery auch vollautomatisiert werden; dies bringt einige Herausforderungen mit sich.

Herausforderungen bei der Automatisierung

Integrations- beziehungsweise UI-Tests haben einen entscheidenden Nachteil. Die in den Unit-Tests praktizierte Isolation ist hier leider nicht mehr gegeben. Ziel der Tests ist ja genau die Integration mehrerer Komponenten oder die Interaktion mit dem Benutzer. Im Test muss man sich daher um diverse Abhängigkeiten sowie eine komplexere Infrastruktur kümmern.

Abbildung 1 zeigt eine solche Infrastruktur als Beispiel. In der Regel wird eine zu testende Software-Komponente (System-Under-Test, kurz „SUT") in einem Application Server oder einer ähnlichen Laufzeitumgebung eingerichtet und konfiguriert. Hinzu kommen diverse Abhängigkeiten zu anderen Services und der Datenhaltung (zum Beispiel zur Datenbank). Mehrere Clients konsumieren im Test nun die vom SUT angebotenen Services. Das SUT greift wiederum auf die Persistenz in der Datenbank oder auf weitere Services zu.

Diese Infrastruktur ist in einem automatisierten Test vollständig aufzubauen. Clients und Backend-Services müssen im Test entsprechend simuliert werden. UI-Tests betätigen die Benutzeroberflächen und stellen sicher, dass die gelieferten Daten entsprechend angezeigt sind. Die Test-Durchführung ist hier weitaus komplexer als in den Unit-Tests. Trotzdem müssen alle Tests automatisiert und vor allem effektiv wiederholbar sein, um die geforderten Release-Zyklen im Continuous Delivery einhalten zu können.

```
public class Greeter {
  public void sayHello(PrintStream to, String user) {
     to.println(sayHello(user));
  }

  public String sayHello(String user) {
     return "Hello, " + user + "!";
  }
}

@RunWith(Arquillian.class)
public class SayHelloTest {

  @Deployment
  public static JavaArchive createDeployment() {
     return ShrinkWrap.create(JavaArchive.class)
        .addClass(Greeter.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
  }

  @Inject
  Greeter greeter;

  @Test
  public void should_say_hello() {
     Assert.assertEquals("Hello, Yoda!",
        greeter.sayHello("Yoda"));
     greeter.sayHello(System.out, "Yoda");
  }
}
```

Listing 1

Tools für den Integrationstest

Zum Glück gibt es eine Reihe von Tools und Frameworks, die bei der Umsetzung dieser Aufgaben unterstützen. Arquillian *[4]* ist ein Open-Source-Tool, um eine Java-Anwendung in einem Container automatisiert zu testen. Das Framework bietet dafür ein hervorragendes Container-Lifecycle-Management sowohl für gängige Java Application Server (WildFly, GlassFish, Tomcat, JBoss AS etc.) als auch für Docker-Container-Umgebungen. Arquillian kümmert sich im Vorfeld eines Tests um den Start des Containers und das automatische Deployment der zu testenden Java-Archive (JAR, WAR oder EAR). Nach dem Deployment stehen der Java-Anwendung während des Tests diverse Container-Res-

sourcen wie JNDI, JMS, CDI Injection und EJB-Ressourcen zur Verfügung *(siehe Listing 1)*.

Ähnlich nützlich für den automatisierten Infrastruktur-Aufbau ist das Docker-Maven-Plug-in von fabric8 *[5]*. Damit lassen sich Docker-Images im Maven-Build-Lifecycle bauen und im Docker-Container starten. Somit können beispielsweise Datenbanken und Komponenten wie ein JMS Message Broker vollautomatisch vor dem Test gestartet werden.

Ebenfalls möglich ist das automatische Deployment der SUT-Komponente in einem Application Server. Damit wird die komplette Infrastruktur für den Test automatisch im Maven Build initialisiert. Nach dem Testdurchlauf können die Komponenten entsprechend automatisch wieder gestoppt werden, um eine saubere Test-Umgebung zu hinterlassen.

Das Docker „fabric8 maven"-Plug-in in *Listing 2* zeigt die Konfiguration für eine MySQL-Datenbank sowie für einen AqtiveMQ Message Broker, die jeweils in einem Docker-Container ausgeführt werden. Beide Container können über das Plug-in im Maven-Lifecycle automatisch gestartet und gestoppt werden. Dabei greift das Beispiel auf fertige Docker-Images der einzelnen Komponenten zurück. Damit lassen sich die für den Test benötigten Komponenten einfach und bequem starten. Der Maven-Build-Prozess wartet sogar, bis die Container erfolgreich hochgefahren wurden, bevor die Testfälle gestartet werden.

Nun fehlen noch diverse Clients und Backend-Services, die während des Tests Daten abfragen beziehungsweise Daten liefern. Dafür bietet sich ein weiteres Java-Open-Source-Framework mit dem Namen „Citrus" *[6]* an. Damit ist man in der Lage, nachrichtenbasierte Schnittstellen client- und serverseitig im Test zu simulieren. Citrus stellt dafür als Framework fertige Komponenten für das Senden und Empfangen von Nachrichten zur Verfügung. Diese sogenannten „Endpoints" gibt es für HTTP REST, JMS, SOAP, FTP, RMI, XML, JSON und viele weitere Transportwege und Datenformate. Citrus bietet eine Java-DSL, um die Testlogik für den Austausch von Nachrichten über diverse Schnittstellen zu definieren. Die Anweisungen sind einfach in einen Arquillian- oder JUnit-Test integrierbar *(siehe Listing 3)*.

Das Citrus-Beispiel zeigt zunächst die Konfiguration der Citrus-Endpoints für den Test. Im Test werden diese entsprechend referenziert. Der Test umfasst eine einfache HTTP-REST-Request-Response-Kommunikation, wobei Citrus als Client das SUT über die Schnittstelle aufruft und eine entsprechende HTTP-200-Ok-Response empfangen möchte. Als zweiten Schritt erwartet Citrus eine eingehende Mail-Nachricht, die auf dem SUT ausgelöst wurde. Im Test werden die realen Schnittstellen des SUT bedient und somit auf die korrekte Funktionsweise überprüft. Citrus überprüft alle eingehenden Nachrichten gegen ein erwartetes Template, das sowohl Body- als auch Header-Informationen beinhalten kann.

Tools für den UI-Test

Im Bereich der UI-Tests ist das wohl bekannteste Web-Testing-Framework Selenium *[7]*. Es ist in der Lage, Interaktionen eines Benutzers im Browser zu simulieren. Dabei werden Document-Object-Model-Elemente (DOM) wie Buttons, Textfelder oder Links anhand von Namen, IDs oder CSS-Informationen identifiziert und angesteuert. Selenium-Tests lassen sich dabei über JUnit sehr gut in den etablierten Build-Lifecycle integrieren und automatisch ausführen.

Sollen neben rein Web-basierten Inhalten auch Rich-Clients wie PDF Reader oder native Anwendungen wie ein Mail-Client Gegenstand eines automatisierten Oberflächentests sein, bietet sich der Einsatz eines Frameworks wie Sakuli *[8]* an. Es kann neben den DOM-basierten Elementen auch Inhalte außerhalb

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <configuration>
     <logDate>default</logDate>
     <verbose>true</verbose>
     <images>
        <image>
           <alias>mysqldb</alias>
           <name>mysql:5.5</name>
           <run>
              <namingStrategy>alias</namingStrategy>
              <ports>
                 <port>3306:3306</port>
              </ports>
              <env>
                 <MYSQL_ROOT_PASSWORD>admin</MYSQL_ROOT_PASSWORD>
                 <MYSQL_DATABASE>todolist</MYSQL_DATABASE>
                 <MYSQL_USER>todo</MYSQL_USER>
                 <MYSQL_PASSWORD>secret</MYSQL_PASSWORD>
              </env>
              <wait>
                 <log>MySQL init process done</log>
                 <time>60000</time>
              </wait>
              <log>
                 <enabled>true</enabled>
                 <color>green</color>
              </log>
           </run>
        </image>
        <image>
           <alias>activemq-broker</alias>
           <name>consol/activemq-5.12:latest</name>
           <run>
              <namingStrategy>alias</namingStrategy>
              <ports>
                 <port>61616:61616</port>
                 <port>8161:8161</port>
              </ports>
              <wait>
                 <http>
                    <url>http://localhost:8161</url>
                    <method>GET</method>
                    <status>200</status>
                 </http>
                 <time>60000</time>
              </wait>
              <log>
                 <enabled>true</enabled>
                 <color>green</color>
              </log>
           </run>
        </image>
     </images>
  </configuration>
</plugin>
```

Listing 2

```
<citrus-http:client id="reportingClient"
  request-method="GET"
  request-url="http://localhost:8080/report/services"/>

<citrus-mail:server id="mailServer"
  port="${mail.server.port}"
  auto-accept="true"
  auto-start="true"/>

@Test
public class ReportingIT extends TestNGCitrusTestDesigner {

  @Autowired
  private HttpClient reportingClient;

  @Autowired
  private MailServer mailServer;

  @CitrusTest
  public void shouldSendMailForLargeOrder() {
      echo("Add 1000+ order and receive mail");

      variable("orderType", "chocolate");
      variable("amount", "1001");

      http().client(reportingClient)
          .send()
          .put("/reporting")
          .queryParam("id", "citrus:randomNumber(10)")
          .queryParam("name", "${orderType}")
          .queryParam("amount", "${amount}");

      http().client(reportingClient)
          .receive()
          .response(HttpStatus.OK);

      echo("Receive report mail for 1000+ order");

      receive(mailServer)
          .payload(new ClassPathResource("templates/mail.xml"))
          .header(CitrusMailMessageHeaders.MAIL_SUBJECT, "Congratulations!");

      send(mailServer)
          .payload(new ClassPathResource("templates/mail_response.xml"));
  }
}
```

Listing 3

des Browser-Fensters ansteuern und im Test verifizieren. Je nach Bedarf macht sich Sakuli hierbei die Bildmuster-Erkennung zunutze und ist damit in der Lage, alle nativen Rich-Client-Anwendungen über User-Aktionen per Maus oder Tastatur auf der nativen UI des Betriebssystems zu bedienen *(siehe Listing 4)*.

Für die Ausführung der Tests ohne Anzeige („headless“) bietet Sakuli fertige Docker-Images mit vorinstalliertem Chrome und Firefox, die dann sogar die Ausführung der Tests in parallel laufenden Containern ermöglichen. So können die oftmals lang laufenden UI-Tests im Parallelbetrieb deutlich schneller ausgeführt werden. Ein komplettes Beispiel mit lauffähigem Continuous Integration Build ist auf GitHub *[9]* verfügbar.

Wartung und Pflege

Die Test-Automatisierung im Unternehmen nimmt ihren Lauf und immer mehr automatisierte Tests werden erstellt und kontinuierlich mit jeder Änderung ausgeführt. Daraus ergibt sich ein gewisser Pflegeaufwand für die Tests, der nicht zu unterschätzen ist.

Der entwickelte Code wird ständigen Änderungen und Anpassungen unterzogen. Features und fachliche Abläufe, die gestern noch aktuell waren, werden heute schon anders interpretiert und implementiert. Die Pflege der Tests muss daher entwicklungsbegleitend stattfinden. Wenn ein Entwickler eine Code-Änderung durchführt, muss auch die Testlogik bei Bedarf entsprechend nachgezogen werden. Im Idealfall kann der Entwickler alle Tests problemlos lokal bei sich auf dem Rechner ausführen, um eventuelle Testfehler zu analysieren und gegebenenfalls zu beheben, bevor der Check-in Fehler in der Continuous-Integration-Build-Pipeline auslöst.

Es ist jedoch kaum zu vermeiden, dass Fehler im Continuous Build auftreten – ausgelöst von fehlgeschlagenen Tests. In diesem Fall empfiehlt es sich, zeitnah Feedback an die Entwickler zu geben, damit diese die Test-Ergebnisse leichter den letzten Code-Änderungen zuordnen können. Dadurch lassen sich die Fehler viel besser analysieren und nachvollziehen.

Ein roter Continuous Build, der über Tage hinweg keine Beachtung findet, führt zu einer generellen Müdigkeit und Gleichgültigkeit gegenüber dem Build-Status. Wenn sich verschiedene Probleme und Fehler im Build dann auch noch überlagern, ist eine Fehler-Analyse und -Behebung deutlich erschwert. Bei hoher Test-Automatisierung ist eine kontinuierliche Anpassung und Fehlerbehebung als zwingend notwendig anzusehen. Dieser Grundsatz muss im Team etabliert werden und in den täglichen Arbeitsrhythmus fest integriert sein.

Fazit

Die vollständige Automatisierung aller Tests und die kontinuierliche Ausführung im Continuous Integration Build sind essenzielle Voraussetzungen für schnelle und häufige Lieferungen einer Software. Nur so kann die Entwicklung jederzeit die Release-Fähigkeit einer Anwendung beurteilen. Mit den vorgestellten Tools ist eine Automatisierung auch in

```
//open print preview
env.type("p", Key.CTRL);
screen.find("save_button").highlight().click().type(Key.ENTER);

//open pdf in new tab and validate
env.type("tl", Key.CTRL).paste(getPDFpath()).type(Key.ENTER);
screen.waitForImage("pdf_place_order.png", 5).highlight();
[
  "pdf_blueberry.png",
  "pdf_caramel.png",
  "pdf_chocolate.png"
].forEach(function (imgPattern) {
  screen.find(imgPattern).highlight();
});
```

Listing 4

den Bereichen der Integrations-, Acceptance- und UI-Tests möglich. Damit hat man nicht nur eine gute Absicherung gegen Bugs, sondern auch Sicherheit bei zukünftigen Anpassungen.

Mit zunehmender Automatisierung wächst jedoch auch die Verantwortung für alle Beteiligten im Hinblick auf die Pflege der Tests. Fehlgeschlagene Tests dürfen nicht zu lange aufgeschoben werden, damit ein roter Build nicht zur akzeptierten Gewohnheit wird.

Die Art und Weise, wie wir Software entwickeln, befindet sich im Wandel – stets mit dem Ziel, Flexibilität und Effizienz zu optimieren. Da ist es nur logisch, dass die Qualitätssicherung denselben Mechanismen unterzogen wird und die gleiche Beachtung im Unternehmen finden muss.

Weiterführende Weblinks

[1] https://maven.apache.org
[2] https://gradle.org
[3] https://jenkins.io
[4] http://arquillian.org
[5] https://github.com/fabric8io/fabric8-maven-plugin
[6] http://www.citrusframework.org
[7] http://www.seleniumhq.org
[8] https://github.com/ConSol/sakuli
[9] https://github.com/toschneck/sakuli-example-bakery-testing

Christoph Deppisch
christoph.deppisch@consol.de

Christoph Deppisch arbeitet als Consultant und Software-Architekt bei der ConSol Software GmbH und verfügt über mehr als zehn Jahre Erfahrung bei der Umsetzung großer Enterprise-Projekte vor allem im Middleware-Integration-Umfeld. Als aktiver Open-Source-Entwickler ist er für das Test-Framework „Citrus" verantwortlich. In letzter Zeit beschäftigt er sich vor allem damit, welche Einflüsse Microservices und Container-Technologien auf die Continuous-Delivery-Pipeline haben.

Tobias Schneck
tobias.schneck@consol.de

Tobias Schneck ist seit zehn Jahren in der IT-Branche tätig. Er sammelte weitreichende Erfahrungen in den Bereichen der IT-Administration bis hin zur Entwicklung von IT-Speziallösungen. In seiner derzeitigen Position als Software-Consultant bei der ConSol Software GmbH ist er Mitbegründer des Open-Source-Testing-Frameworks „Sakuli" und spezialisierte sich als Java-Entwickler auf den Bereich „Test-Automatisierung". Er ist Konferenz-Speaker sowie Organisator der MeetUp-Gruppe „Agile Testing @Munich" und interessiert sich für neue, innovative Technologien.

Continuous Delivery of Continuous Delivery

Gerd Aschemann, Freiberufler

Zahlreiche Organisationen nutzen für ihre Software-Entwicklung bereits Continuous Integration oder sogar Continuous Delivery zum Fertigen, Prüfen und Ausliefern ihrer Software-Artefakte. Dazu betreiben sie eine Plattform, die selbst aus Software-Komponenten und zahlreichen Anpassungen für den spezifischen Lieferprozess der Organisation besteht. Konsequentes Handeln bedeutet hier, auch die Plattform selbst kontinuierlich liefern zu können, also Continuous Delivery of Continuous Delivery zu betreiben. Der Artikel zeigt, wie dieser Prozess via „Platform as Code" von einer DevOps-Organisation umgesetzt werden kann.

Wer kennt das nicht? Bob konnte den Build-Prozess nicht beenden. Kurz vor Abschluss des Sprints ist plötzlich alles „rot", Builds sind nicht durchgelaufen, Tests gescheitert, Artefakte nicht im Repository gelandet, Instanzen wurden nicht neu gestartet. Im morgendlichen Daily schauen alle betreten zu Boden. Fingerpointing ist verpönt und erstmal hat auch keiner eine Idee, wen er überhaupt anklagen könnte.

Bob selbst sagt nichts. Bob ist das Maskottchen des Teams. Er wird von allen liebe-

voll gehegt und gepflegt. Wenn es ihm gut geht, liegt er schnurrend in der Ecke und macht brav seine Arbeit. Wenn er aber nicht richtig gefüttert wird, fährt er sehr schnell Zähne und Klauen aus und zeigt dem Team, wer den längeren Atem hat. Bob ist kein Kätzchen, aber genauso eine Diva. Bob heißt mit vollem Namen „Bob the Builder" und ist ein ausgewachsener Jenkins. Er könnte auch zur Familie TeamCity, Bamboo oder einigen anderen gehören. Seine Spielkameraden heißen Nexus, SCM-Manager, GitHub, SonarQube, sie tollen gerne im Red-Hat-Wald oder auf der Debian-Wiese herum, oder auf Cloud 7. Manchmal ist auch Streit mit den Kameraden der Grund für Bobs schlechte Laune oder ein Gewitter über den Wolken.

Das Team hat es an solchen Tagen schwer. Hat die Missstimmung damit zu tun, dass der neue Praktikant letzte Woche bei allen Jobs seine Mail-Adresse einpflegen und dabei prüfen sollte, ob alle Maven-Builds mit der neuen Version laufen? Da hat man schnell mal in der Konfigurationsmaske ein Häkchen umgeschossen. Oder funktionieren die neuen Plug-ins, die seit ein paar Tagen genutzt werden, nur, solange man nicht gerade einen Release-Build macht? Sind vielleicht beim Test des geänderten Release-Prozesses vor ein paar Tagen Tags im Git nicht wieder weggeräumt worden? Leider ist das ganze System so komplex, dass man neue Plug-ins oder Änderungen am Prozess nur am lebenden Herzen, also der einzigen gepflegten Jenkins-Instanz, dem Produktivsystem, testen kann.

So geht das jedenfalls nicht mehr weiter! Warum sollen nur die Laufzeit-Umgebungen der erstellten Software in Entwicklung, Test und Produktion als „Cattle" betrieben werden statt als „Pet" *[1]*, aber nicht die Continuous Delivery Platform (CDP) selbst auch? Warum nicht die CDP als „Platform as Code" (PaC) bereitstellen, vollautomatisch testen und bei Bedarf neu oder partiell ausrollen?

Configuration Management

„Configuration Management" wird in der IT mehrdeutig benutzt. Entwickler assoziieren damit oft „Source Code Management" (SCM) mithilfe von Systemen wie CVS, SVN oder Git. Der IT-Betrieb denkt eher an das Management der Infrastruktur.

Vergleicht man die Tätigkeiten von Entwicklern (Dev) bei der Software-Entwicklung und Administration/Betrieb (Ops) im Konfigurationsmanagement, sieht man leicht eine einheitliche Struktur auf abstrakter Ebene:

- In *Abbildung 1* durchläuft der Entwickler einen Zyklus, in dem er Code bearbeitet, testet und bei Zufriedenheit freigibt (liefert); bei Fehlern startet er den Zyklus erneut.
- In *Abbildung 2* ändert der Configuration Manager eine Konfiguration, prüft sie und gibt sie bei Erfolg zur Nutzung frei (Lieferung). Bei Nicht-Erfolg fängt er von vorne an.

Ein wesentlicher Unterschied ist, dass ein Dev typischerweise erst mal in ein SCM einliefert und seine Arbeit durch eine Vielzahl weiterer Schritte wie automatische Tests (Integrationstest, Performance-Test, Security-/Exception-Test), manuelle Qualitätssicherung (Code Review, explorative Tests) und automatisierte Messungen (Qualitätsmetriken für die Code-Qualität und Architektur etc.) geprüft wird.

Die Arbeit von Ops unterliegt hingegen häufig keiner so umfangreichen und vielschichtigen Qualitätssicherung. Dabei gibt es zahlreiche Systeme für den automatisierten Aufbau von Infrastruktur wie Chef *[2]*, Puppet *[3]* oder Ansible *[4]*, man spricht hier auch von „Infrastructure as Code" (IaC, *[5]*). Von diesen CM-Systemen wird der Aufbau beziehungsweise die Konfiguration der Infrastruktur (Server, Storage etc.) gut abgedeckt.

Für die Software-Entwicklung selbst haben sich im Java-Umfeld Systeme wie Maven oder Gradle etabliert, die es ermöglichen, den konkreten Build-Prozess eines Artefakts und Teile seiner Qualitätssicherung (Modul-Tests, Erfassen von Qualitätsmetriken, Architektur-Prüfungen etc.) in Code zu gießen. Ihre Konfigurationen liegen als XML- oder Groovy-Code vor und lassen sich ebenfalls automatisiert qualitätssichern (etwa Konsistenzprüfung von Maven POMs, Abstraktion/Vereinheitlichung des Build-Prozesses). Versucht man, CM in verschiedene Ebenen zu zerlegen, ergibt sich etwa folgender Aufbau:

- *Infrastruktur*
  - OS/Linux (Windows, OSX, Android, Embedded etc.)
  - Services (DB, DNS, Web etc.)
- *Plattform*
  - Build Server (Jenkins, Bamboo, Travis etc.)
  - Source Code Repository (SVN, SCM-Manager, GitLab, GitHub etc.)
  - Artifact Repository (Nexus, Artifactory etc.)
  - QS-Management (FindBugs/Checkstyle/PMD, Surefire/Failsafe, SonarQube etc.)
- *Application*
  - Maven, Gradle etc.

Wie gesagt, gibt es auf Infrastruktur- und Application-Ebene gute Mechanismen zur Automatisierung. Was häufig fehlt, ist der kontinuierliche, ganzheitliche Aufbau einer Plattform und eine hinreichende, auf Automatisierung beruhende Qualitätssicherung.

Continuous Delivery

Continuous Delivery setzt frei nach Martin Fowler *[8]* vier Anforderungen um:

- Software ist jederzeit deploybar
- Deploybarkeit hat Vorrang vor Erweiterungen
- Aussagefähigkeit über Production Readiness nach jedem Change

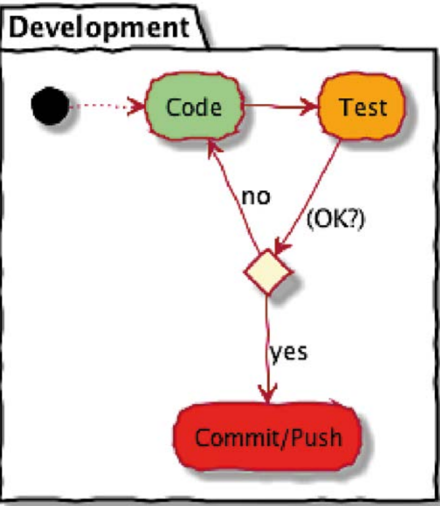


Abbildung 1: Tätigkeit des Entwicklers

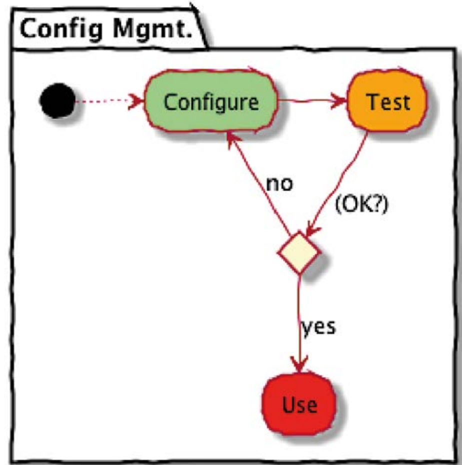


Abbildung 2: Tätigkeit des Configuration Managers

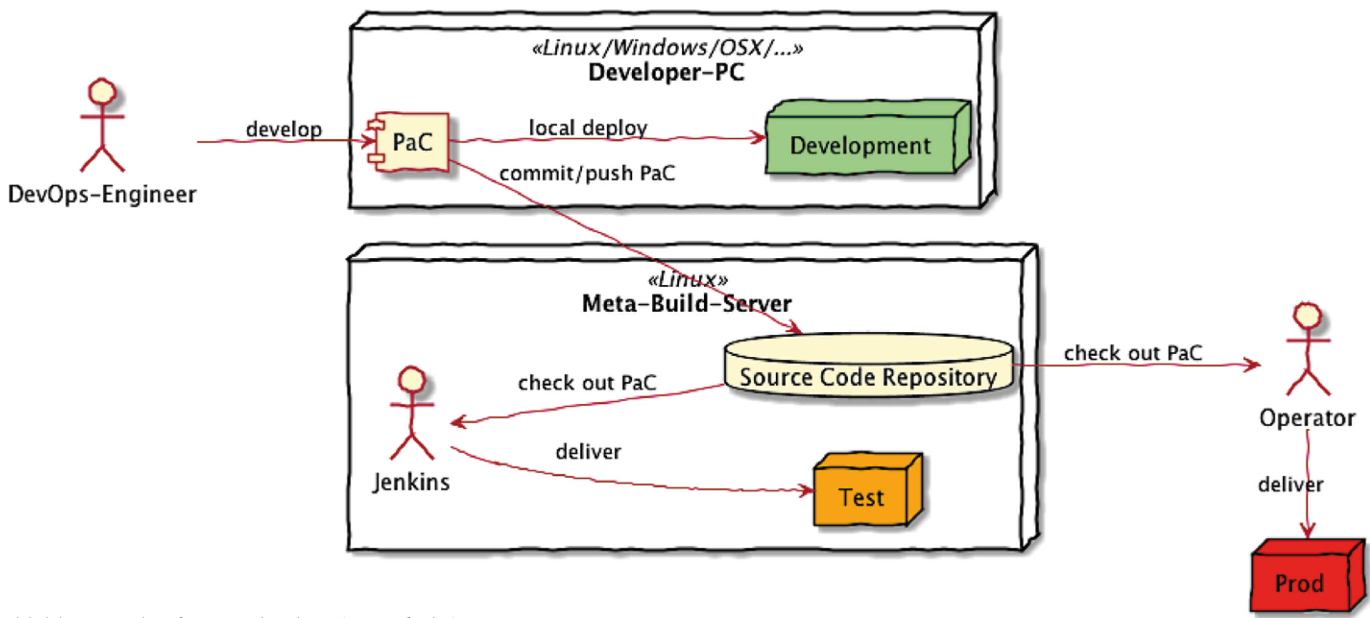


Abbildung 3: Plattform-Architektur (vereinfacht)

- Push-Button-Deployment beliebiger Versionen in beliebige Umgebungen

Diese lassen sich weitgehend auch auf die CDP übertragen. Die letzten beiden Punkte sind konkreter beziehungsweise ergänzt:

- Änderungen an Infrastruktur- und Plattform-Komponenten ergeben sich häufig auch hinter den Kulissen. Betrachtet man beispielsweise allein Jenkins und seine Plug-ins in der jeweils neuesten Version, so gibt es fast täglich Änderungen, die sich auf Abhängigkeiten und Konfiguration auswirken. Komplexe Build-Prozesse erfordern oft eine Vielzahl von Plug-ins, die voneinander abhängen. Hier muss die Toolchain kontinuierlich auf einen aktuellen Stand gehoben werden. Fasst man sie erst nach Monaten oder Jahren wieder an, kann man kaum noch eine verlässliche Aussage über ihr Zusammenspiel machen und hat sehr hohe Regressionsaufwände oder ein entsprechendes Risiko.
- Es muss sicher kein Deployment auf Knopfdruck beliebiger Versionen in beliebige Umgebungen möglich sein, etwa ein Downgrade um mehrere Versionen. Vernünftig erscheint eher ein regelmäßiges Upgrade in kleinen (kontinuierlichen) Schritten.

Außerdem sei darauf hingewiesen, dass Continuous Delivery nicht mit Continuous Deployment gleichzusetzen ist. Ersteres meint die Fähigkeit, jederzeit in Produktion gehen zu können, Letzteres bedeutet, es auch fortlaufend zu tun.

Plattform-Architektur am Beispiel

Die folgenden Beispiele sind mit konkreten Komponenten wie Jenkins [6] als CI-Server oder Nexus als Artefakt-Repository und Puppet als CM-Systemen umgesetzt, hierzu gibt es auch eine Referenz-Implementierung auf GitHub [7]. Fast alles lässt sich in ähnlicher Form auch mit anderen, analogen Komponenten und CM-Systemen umsetzen, etwa mit Ansible statt Puppet. Die hier getroffene Auswahl basiert auf Erfahrungen und Vorarbeiten des Autors, ohne eine Wertung zu beinhalten. Jede Komponente hat erfahrungsgemäß ihre jeweiligen Stärken und Schwächen. Will man die Implementierung einer CDP starten, muss man zudem die vorhandenen Skills und bereits eingesetzte Tools in der jeweiligen Organisation beachten. Abbildung 3 zeigt eine vereinfachte Architektursicht (Außensicht zur Kontext-Abgrenzung):

- Der Plattform-Entwickler (DevOps-Engineer) entwickelt auf seinem lokalen Rechner die CDP als Code (PaC). Die CDP selbst läuft in einer VM (lokal oder in einer Cloud).
- Wenn seine lokalen Tests erfolgreich sind, checkt er den PaC in ein SCM ein beziehungsweise pusht seine Änderungen an das zentrale Source-Code-Repository seines Teams.
- Der (Jenkins) Build Server checkt regelmäßig den neuesten Stand aus, baut eine Test-Instanz auf und führt auf dieser alle nötigen Qualitätsprüfungen (Tests etc.) durch.
- Erfolgreiche Konfigurationen können vom Betrieb (Ops) ausgecheckt und ins produktive System überführt werden.

Für den DevOps-Engineer sieht die Innensicht der Plattform vereinfacht aus wie in Abbildung 4. Der Jenkins soll Source-Code aus einem SCM-Repository (hier: SCM-Manager [9]) auschecken, Artefakte bauen und in ein Artefakt-Repository deployen (hier: Nexus [10]).

Die Plattform kann natürlich aus weiteren Komponenten bestehen, etwa einem SonarQube-Server und einer assoziierten Datenbank. Die Beispiele beschränken sich jedoch auf Jenkins und Nexus.

Referenz-Plattform als VM mit Vagrant

Der Aufbau der Plattform erfolgt für den Entwickler in einer virtuellen Maschine (VM), ebenso wie der Aufbau der Test-Instanz(en). Die Produktiv-Umgebung kann ebenfalls als VM aufgebaut sein oder auch auf einer physischen Maschine (oder mehreren). Die Konfiguration einer VM erfolgt mit Vagrant [11]. Vagrant abstrahiert von verschiedenen VM-Technologien, den sogenannten „Providern". Es gibt unter anderem Provider für folgende Bereiche:

Abbildung 4: Komponenten der CDB (Beispiel)

- VirtualBox, ein kostenloses Produkt von Oracle, das für Windows, Linux und OSX verfügbar ist
- VMware oder Parallels (kommerzielle Produkte)
- Qemu/Libvirt (Open-Source-Virtualisierungen unter Linux)
- Cloud-Instanzen (Amazon und weitere)

Im Beispiel wird VirtualBox genutzt, teilweise sind Fragmente für andere Provider im Code enthalten. Vagrant startet eine VirtualBox mit einem vorgefertigten OS-Image (im Beispiel Ubuntu 16.04 LTS). Nach erfolgreichem Start und einigen Initialisierungen (Port-Weiterleitungen, Verzeichnis-Freigabe beziehungsweise -Montage etc.) startet Vagrant innerhalb der VM einen sogenannten „Provisioner". Das ist im einfachsten Fall ein Shell-Skript; Vagrant kann auch eines der genannten CM-Tools nutzen. Es können auch mehrere Provisioner nacheinander zum Einsatz kommen. Vagrant wird durch ein Vagrantfile gesteuert, das Ruby-Syntax hat *(siehe Listing 1)*. Die wichtigsten Parameter sind hier:

- Der Name der VM, der auch gleichzeitig der Hostname ist (vm.hostname bzw. vbox.name)
- Das ausgewählte Basis-Image (vm.box)
- Der zugewiesene Hauptspeicher (2048 MB)
- Port-Weiterleitungen (hier: Der Port des zu installierenden Jenkins wird auf Port 50080 des lokalen Hosts weitergeleitet, man kann später also über *„http://localhost:50080"* auf den in der VM laufenden Jenkins zugreifen)

Starten lässt sich das Ganze mit dem Befehl „vagrant up" in der Kommandozeile, sofern VirtualBox *[12]* und Vagrant auf dem lokalen Rechner installiert sind.

Provisionierung: Module und Aggregate

Nachdem die VM gestartet ist, kann Vagrant über SSH auf die VM zugreifen und startet darüber auch die weitere Provisionierung. Ab diesem Zeitpunkt kann man sich auch über den Befehl „vagrant ssh" auf der VM einloggen, Vagrant legt entsprechende User und SSH-Keys an. Bei Nutzung von VirtualBox und einigen anderen Providern wird zudem das lokale Verzeichnis (in dem das Vagrantfile liegt) als „/vagrant" in die VM montiert – alle hier liegenden Dateien sind in der VM zugreifbar. Das machen sich die Provisioner zunutze.

In der Referenz-Implementierung wird ein Shell-Skript als Provisioner genutzt, das seinerseits wiederum weitere Skripte und Puppet aufruft. Die genannten CM-Tools sind zwar bedeutend leistungsfähiger als einfache Shell-Skripte, sie sind aber nicht für einen lokalen Einsatz auf einer einzelnen (virtuellen) Maschine ausgerichtet. Puppet arbeitet zum Beispiel normalerweise in einem Master-Agent-Verbund: Konfigurationen werden auf einem zentralen Master gepflegt, jeder beteiligte Rechner startet einen Agent, der die Konfiguration des lokalen Systems mit den Vorgaben des Masters konsistent hält. Konfigurationen werden in Puppet als sogenannte „Manifeste" gepflegt. Ein Manifest ist eine deklarative Definition einer Konfiguration; sie beschreibt unter anderem, welche OS-Pakete installiert oder auch nicht installiert sein sollen oder welche Inhalte

Konfigurationsdateien in „/etc" oder in anderen Verzeichnissen haben sollen.

Puppet selbst ist prinzipiell auch durch Module erweiterbar, die die Konfiguration kompletter Sub-Systeme wie Jenkins oder Docker unterstützen. Leider kann Puppet die Abhängigkeiten seiner eigenen Module nicht verwalten. Daher nutzt die Demo-Implementierung die Shell-Skripte, um benötigte Puppet-Module bei Bedarf zunächst zu installieren. Jede Komponente, die auf der Plattform installiert werden soll, ebenso wie Integrationstests etc. liegen in der Demo ebenfalls als Modul vor. Ein Modul liegt im Unterverzeichnis „modules/" und hat einen einheitlichen Aufbau *(siehe Abbildung 5)*. Es gibt immer zwei Unterverzeichnisse mit jeweils mindestens einer init-Datei, etwa „jenkins-native/" mit „puppet/init.pp" und „scripts/init.sh" („scripts/test.pl" ist ein optionaler Modul-Test). Das Skript („scripts/init.sh") dient nur dazu, die erforderlichen Puppet-Module zu installieren und anschließend das enthaltene Puppet-Manifest („puppet/init.pp") auszuführen.

Die Module können auf einfache Weise komponiert werden. Dazu liegen im Verzeichnis „composites" zwei Unterverzeichnisse „scripts/" und „lists/" *(siehe Abbildung 6)*. In „lists/" stehen Dateien, die einfach Listen von Modulen enthalten, die installiert werden sollen. In „scripts/" liegt ein Shell-Skript („run.sh"), das als Parameter die Liste der Kompositionen erhält. Es liest alle zugehörigen Dateien (Listen) aus, iteriert über die darin aufgeführten Modul-Namen und führt diese aus, indem es jeweils in dem betreffenden Modul-Verzeichnis die Datei „scripts/init.sh" ausführt („current" ist ein Symbolic-Link auf die Default-Komposition, die von Vagrant benutzt wird).

```
Vagrant.configure("2 ") do |config|
  config.vm.box = "bento/ubuntu-16.04 "
  config.vm.hostname = "devopssquare-javaaktuell "

  config.vm.provider "virtualbox " do |vbox, override|
     vbox.name = config.vm.hostname
     vbox.memory = 2048
     override.vm.network "forwarded_port ",
        guest: 8080, host: 50080 # Jenkins
  end

  config.vm.provision "shell ",
     path: "composites/scripts/run.sh ",
     args: "current "
end
```

Listing 1

Abbildung 5: Jenkins-Module

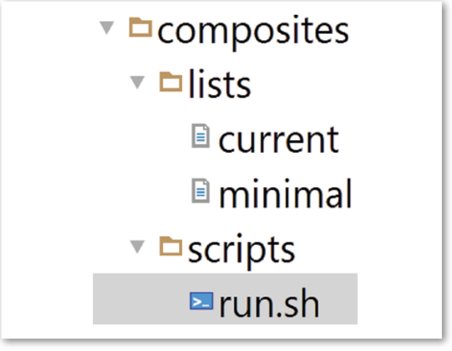


Abbildung 6: Simple Komposition von Modulen

Jenkins-Installation via Puppet

Das Jenkins-Modul in Puppet kann Jenkins als Betriebssystem-Paket installieren und konfigurieren, beispielsweise Jenkins-Plug-ins installieren. Die Plug-ins sind hier nur auszugsweise aufgeführt. Leider kann das Jenkins-Puppet-Modul auch kein Abhängigkeitsmanagement für die Jenkins-Plug-ins durchführen, sodass man eine vollständige Liste aller Plug-ins und ihrer Abhängigkeiten manuell pflegen muss. Das im *Listing 2* beispielhaft angelegte File „_jenkins.install.InstallUtil.lastExecVersion" sorgt dafür, dass Jenkins beim ersten Start nicht seinen interaktiven Initialisierungs-Wizard startet, der seit Version 2 obligatorisch ist – das wäre einem automatisierten Setup abträglich.

Das enthaltene Test-Skript („test.pl") führt abschließend einen Modul-Test aus, indem es prüft, ob Jenkins auf dem lokalen Port 8080 gestartet wurde (von außen ist er dann über den Port 50080 ansprechbar).

Nexus-Installation als Docker-Container

Viele Linux-Services liegen mittlerweile als Docker-Image vor, so auch typische CD-Komponenten wie Nexus oder Jenkins. Eine Installation via Docker *[13]* vereinfacht in der Regel den Betrieb beziehungsweise das Abstimmen von Bibliotheken und anderen Komponenten – im Zweifelsfall bringt der Dienst seine Sub-Komponenten in der genau richtigen Version in seinem Image einfach mit.

Im Jenkins gibt es sehr viel zu konfigurieren, daher empfiehlt sich eine direkte Installation als nativer OS-Service (Puppet installiert ein Debian-Paket) und die Konfiguration durch Puppet. Nexus ist dagegen in sich weitgehend abgeschlossen, hauptsächlich müssen die in den Repositories abgelegten Dateien in ein Volume auf dem Host ausgelagert werden. Alle weiteren Konfigurationen von Nexus können bei Bedarf über die REST-Schnittstelle vorgenommen werden. Ähnlich verhält es sich etwa mit SCM-Manager oder SonarQube. Die Installation von Nexus via Puppet ist daher deutlich einfacher, das Manifest beschränkt sich im Wesentlichen auf wenige Zeilen (siehe *Listing 3*, Anlegen des Daten-Volume ausgelassen).

Es wird ein Docker-Container basierend auf dem aktuellen („latest") Image „sonatype/nexus" gestartet, das Verzeichnis „/data/nexus/sonatype" wird unter „/sonatype-work" eingeblendet (hier legt Nexus im Docker-Container unter anderem die Reposi-

JavaLand-Schulungstag

Der Autor führt auf dem Schulungstag der nächsten JavaLand (30. März 2017) vertieft in das Thema ein. Die Teilnehmer realisieren in einem „Hands On"-Tagesworkshop eine eigene Continuous-Delivery-Plattform und lernen dabei viele Tipps und Tricks aus dem Erfahrungsschatz des Referenten kennen. Die Instanzen der Teilnehmer werden fortlaufend auf der Referenz-Plattform gebaut; so wird der praktische Nutzen der rekursiven CD-Strategie demonstriert.

tories ab). Der Nexus-HTTP-Port 8081 wird nach außen ebenfalls als Port 8081 freigegeben. Puppet legt den Container in Debian/Ubuntu als Linux-Service an, sodass er auch bei System-Start und -Stopp berücksichtigt wird. Hat man mehrere Services als Docker-Container vorliegen und beschreibt in Puppet deren Abhängigkeiten (etwa SonarQube plus Datenbank), werden diese Abhängigkeiten auch an das Abhängigkeitsmanagement des Betriebssystems weitergegeben und die Dienste beim Boot in der richtigen Reihenfolge gestartet.

Integrationstest

Continuous Delivery ist ohne hohe Test-Automatisierung nicht denkbar. Der automatische Aufbau der Plattform muss daher durch entsprechende Tests auf Funktionsfähigkeit geprüft werden. Im Falle der Kombination von Jenkins und Nexus kann ein Testfall etwa darin bestehen, ein Java-Projekt auf dem Jenkins mit Maven zu bauen und die entstehenden Artefakte in den Nexus zu laden (deploy). Der Test ist erfolgreich, wenn die Jenkins-Jobs durchlaufen und neue Artefakte in Nexus sichtbar werden. Gesteuert wird das Ganze durch einen Jenkins-Seed-Job, der den Maven-Build-Job inkl. Deployment nach Nexus über die Jenkins-Job-DSL *[14]* anlegt *(siehe Listing 4)*.

Hinzu kommt natürlich noch eine entsprechende Maven „settings.xml“, die ebenfalls via Puppet angelegt wird, sowie der Test im Rahmen des Aufbaus als Modul (siehe Demo-Beispiel).

Fazit und Ausblick

Der Artikel zeigt fragmentarisch, dass der automatisierte Aufbau einer Continuous Delivery Platform (CDP) als Platform as Code (PaC) möglich ist. Durch automatisierten, regelmäßigen Aufbau lassen sich unter anderem die Plattform kontinuierlich weiterentwickeln, neue Konzepte und Prozesse testen oder Schulungen durchführen. Der Autor hat ähnliche Umsetzungen bereits in verschiedenen Kundenprojekten realisiert. Auch das Open-Source-Projekt DukeCon mit der mobilen JavaLand-App und dazugehörigem Backend *[15]* enthält ein Unterprojekt („dukecon_infra“ *[16]*), das die Projekt-Infrastruktur inklusive CDP durch einen Build Server regelmäßig neu aufbaut – ultimativer Testfall ist hier das Erstellen der gesamten DukeCon-Plattform.

```
include jenkins

$plugins = [
  'ansicolor',
  'build-pipeline-plugin',
  'disk-usage',
…
  'workflow-scm-step',
  'workflow-step-api',
]

jenkins::plugin { $plugins : }

file { '/var/lib/jenkins/jenkins.install.InstallUtil.lastExecVersion':
  owner     => 'jenkins',
  group     => 'jenkins',
  mode      => 0644,
  content   => "2.8",
  require   => Class['jenkins'],
  replace   => 'no',
}
```

Listing 2

```
docker::run { 'nexus':
  image     => 'sonatype/nexus',
  volumes   => ['/data/nexus/sonatype:/sonatype-work'],
  ports     => ['8081:8081'],
}
```

Listing 3

```
mavenJob ("helloworld-deploy") {
  scm {
     github ("devopssquare/helloworld", "master")
  }
  triggers {
     scm("H/10 * * * *")
  }
  goals ('clean deploy')
}
```

Listing 4

Referenzen

[1] The History of Pets vs. Cattle and How to Use the Analogy Properly, Randy Bias: *http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle*
[2] *https://www.chef.io*
[3] *https://puppet.com*
[4] *https://www.ansible.com*
[5] *https://en.wikipedia.org/wiki/Infrastructure_as_Code*
[6] *https://jenkins.io*
[7] *https://github.com/devopssquare/demo-javaaktuell*
[8] Martin Fowler, Continuous Delivery: *http://martinfowler.com/bliki/ContinuousDelivery.html*
[9] *https://www.scm-manager.org*
[10] *http://www.sonatype.org/nexus*
[11] *https://www.vagrantup.com*
[12] *https://www.virtualbox.org*
[13] *https://www.docker.com*
[14] *https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin*
[15] DukeCon – das Innere der JavaLand-App, Gerd Aschemann, Java aktuell 02/2016
[16] *https://github.com/dukecon/dukecon_infra*

Gerd Aschemann
gerd@aschemann.net

Gerd Aschemann ist Automatisierungsfanatiker: Was geskriptet werden kann, wird auch geskriptet. Er ist als freiberuflicher Software-Entwickler und -Architekt unterwegs, in den letzten Jahren bevorzugt mit Aufgaben im Bereich „Continuous Integration und -Delivery“. Mit Java beschäftigt er sich schon seit dem Jahr 1995, als er als wissenschaftlicher Mitarbeiter im Fachgebiet „Verteilte Systeme“ der TU Darmstadt zum „Management verteilter Anwendungen“ geforscht hat.

Technische Schulden erkennen, beherrschen und reduzieren

Dr. Carola Lilienthal, WPS Workplace-Solutions GmbH

Das Leben von Entwicklern ist heute nicht die Neuentwicklung – es ist Wartung. Die typischen Programmierer entwickeln keine Software mehr auf der grünen Wiese, sondern sie reparieren, erweitern, verändern und bauen vorhandene Software aus.

Das größte Problem bei der täglichen Arbeit eines Entwicklers ist, dass sich Wartung mit der Zeit von strukturierter Programmierung hin zur defensiven Programmierung verändert. Die Entwickler fangen an, Code zu schreiben, von dem sie wissen, dass er aus Architektursicht schlecht ist. Aber er ist die einzige Lösung, die, wenn man Glück hat, funktioniert. Der Code wird immer komplexer und das Team häuft technische Schulden an. Die Kosten für die Wartung steigen und Erweiterungen führen zunehmend zu mehr Seiteneffekten.

Um dieser Abwärtsspirale auf Dauer entgegenwirken zu können, brauchen wir eine qualitativ hochwertige und flexible Architektur mit möglichst wenig technischen Schulden. Sind die technischen Schulden gering, dann finden sich die Wartungsentwickler gut im System zurecht. Sie können schnell und einfach Bugs fixen und haben keine Probleme, kostengünstig Erweiterungen zu machen. Wie kommen wir in dieses gelobte Land der Architekturen mit reduzierten technischen Schulden?

Technische Schulden

Der Begriff „technische Schulden" wurde im Jahr 1992 von Ward Cunningham geprägt *[1]*. Technische Schulden entstehen, wenn bewusst oder unbewusst falsche oder suboptimale technische Entscheidungen getroffen werden. Diese führen zu einem späteren Zeitpunkt zu Mehraufwand, der Wartung und Erweiterung teurer macht. Zu dem Zeitpunkt der falschen oder suboptimalen Entscheidung hat man also technische Schulden aufgenommen, die man mit ihren Zinsen irgendwann abbezahlen muss, wenn man nicht überschuldet enden will.

In der Literatur werden verschiedene Arten und Varianten von technischen Schulden aufgeführt. In diesem Artikel stehen die technischen Schulden im Fokus, die bei Architektur-Analysen gefunden werden können. Andere Problemfelder, die man auch als Schulden von Software-Projekten betrachten kann, wie fehlende Dokumentation, geringe Testabdeckung, schlechte Usability oder ungenügende Hardware bleiben hier außen vor. Hier geht es um:

- *Implementationsschulden*
  Im Sourcecode finden sich sogenannte „Code Smells" wie lange Methoden, leere Catch-Blöcke etc. Implementationsschulden sind heute weitgehend automatisiert mit einer Vielzahl von Tools im Sourcecode zu finden. Jedes Entwicklungsteam sollte diese Schulden in seiner täglichen Arbeit sukzessive beheben, ohne dass extra Budget dafür erforderlich ist.
- *Design- und Architekturschulden*
  Das Design der Klassen, Pakete, Subsysteme, Schichten und Module und die Ab-

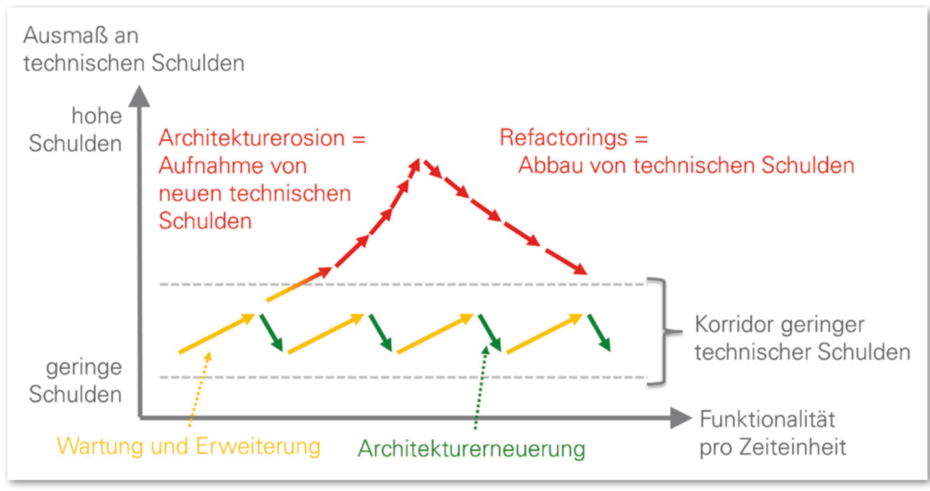


Abbildung 1: Entwicklung und Effekt von technischen Schulden

hängigkeiten zwischen ihnen sind uneinheitlich, komplex und passen nicht mit der geplanten Architektur zusammen. Diese Schulden sind durch einfaches Zählen und Messen nicht zu ermitteln und bedürfen einer umfassenden Analyse, die im dritten Abschnitt vorgestellt wird.

Entstehen von technischen Schulden

Wurde zu Beginn eines Software-Entwicklungsprojekts eine qualitativ hochwertige Architektur entworfen, dann kann man davon ausgehen, dass das Softwaresystem sich am Anfang gut warten lässt. In diesem Anfangsstadium befindet sich das Softwaresystem in einem Korridor hoher Architekturqualität ohne technische Schulden *(siehe Abbildung 1)*.

Erweitert man das System mehr und mehr, so entstehen zwangsläufig technische Schulden (gelbe Pfeile). Software-Entwicklung ist ein ständiger Lernprozess, bei dem der erste Wurf einer Lösung selten der endgültige ist. Die Überarbeitung der Architektur (Architektur-Erneuerung, grüne Pfeile) muss in regelmäßigen Abständen erfolgen. Es entsteht so eine stetige Folge von Erweiterung und Refactoring. Kann ein Team diesen Zyklus beibehalten, wird das System im Korridor geringer technischer Schulden bleiben. Gerade dieser Aspekt des frühzeitigen Refactoring ist vielen Projektleitern und Budgetverantwortlichen nicht so einfach zu vermitteln, obwohl er jedem Entwickler und Architekten absolut logisch erscheint.

Darf das Entwicklungsteam die technischen Schulden nicht ständig im Auge behalten, so setzt im Laufe der Zeit zwangsläufig Architektur-Erosion ein (rote Pfeile). Die Architektur des Systems erodiert, die Komplexität nimmt zu und es werden Schritt für Schritt Schulden aufgebaut. Sind erst einmal technische Schulden angehäuft, werden Wartung und Erweiterung der Software immer teurer, Folgefehler sind immer schwerer nachvollziehbar, bis zu dem Punkt, an dem jede Änderung zu einer schmerzhaften Anstrengung wird. Die *Abbildung 1* macht diesen langsamen Verfall dadurch deutlich, dass die roten Pfeile immer kürzer werden. Pro Zeiteinheit kann man bei steigenden Schulden immer weniger Funktionalität umsetzen.

Um aus diesem Dilemma der technischen Schulden herauszukommen, muss die Architektur rückwirkend verbessert werden. Auf diesem meist beschwerlichen Weg ist das System Schritt für Schritt wieder in den Korridor geringer technischer Schulden zurückzubringen (rote absteigende Pfeile).

Natürlich kann es auch passieren, dass zu Beginn der Entwicklung kein fähiges Team vor Ort war und das Softwaresystem ohne Architektur oder mit einer rudimentären Architektur-Vorstellung entwickelt wurde. In einem solchen Fall wächst die Architektur im Laufe der Zeit ohne Plan vor sich hin und technische Schulden werden gleich zu Beginn der Entwicklung aufgenommen und kontinuierlich erhöht. Über solche Softwaresysteme kann man wohl sagen: Sie sind unter schlechten Bedingungen aufgewachsen. Weder den Software-Entwicklern noch dem Management wird ein System in einem solchen Zustand auf Dauer Freude machen.

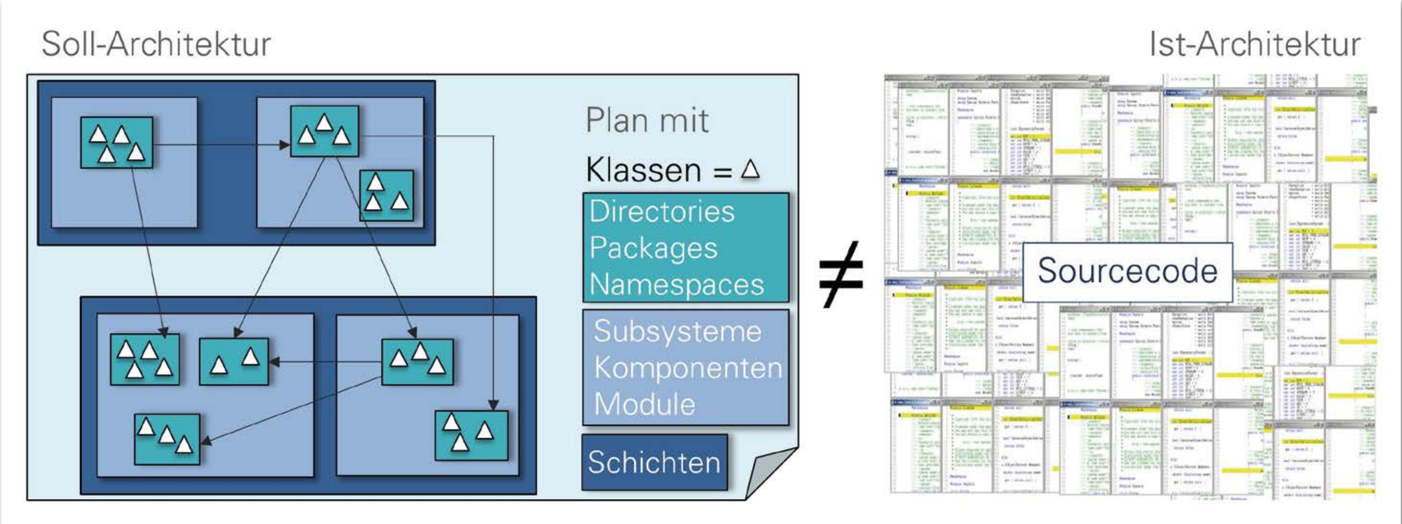


Abbildung 2: Soll-Ist-Vergleich der Architektur

Architektur-Analyse und -Verbesserung

Die meisten Entwicklungsteams können aus dem Stand eine Liste von Design- und Architektur-Schulden für das System aufzählen, an dem sie gerade entwickeln. Diese Liste lässt sich gut als Ausgangspunkt für die Analyse von technischen Schulden nutzen. Um den Design- und Architektur-Schulden weiter auf den Grund zu gehen, empfiehlt sich eine Architektur-Analyse *[2]*. Damit lässt sich überprüfen, inwieweit die geplante Architektur im Sourcecode tatsächlich umgesetzt worden ist und wie hoch der Verschuldungsgrad der Software ist. Für solche Soll-Ist-Vergleiche steht heute eine Reihe guter Tools zur Verfügung: Lattix, Sotograph/SotoArc, Sonargraph, Structure101 etc. Die Soll-Architektur ist der Plan für die Architektur, der auf Papier oder in den Köpfen der Architekten und Entwickler existiert *(siehe Abbildung 2)*.

Häufig wird dieser Plan bereits vor Beginn der Implementierung erstellt und im Laufe der Zeit an die Gegebenheiten angepasst. In der Soll-Architektur werden die Klassen und Pakete zu Subsystemen, Komponenten, Modulen (je nachdem, welchen Begriff man wählt) und Schichten zusammengefasst. Im weiteren Verlauf des Artikels sind alle diese Elemente als „Bausteine" bezeichnet.

Die Soll-Architektur wird bei der Architektur-Analyse mit dem echten Sourcecode abgeglichen. Dieser enthält die implementierte Ist-Architektur. In allen der Autorin bekannten Fällen weicht die Ist- von der Soll-Architektur ab. Die Ursachen dafür sind vielfältig: Abweichungen entstehen häufig unbemerkt, weil Entwicklungsumgebungen nur lokalen Einblick in den gerade bearbeiteten Sourcecode geben und keinen Überblick ermöglichen. Auch mangelndes Wissen über die Architektur im Entwicklungsteam führt zu diesem Effekt.

In anderen Fällen werden die Abweichungen zwischen Soll- und Ist-Architektur absichtlich eingegangen, weil das Team unter Zeitdruck steht und eine schnelle Lösung braucht. Das notwendige Refactoring wird dann auf unbestimmte Zeit in die Zukunft verschoben. Bei der Architektur-Analyse und -Verbesserung macht sich die Autorin gemeinsam mit den Architekten und Entwicklern auf die Suche nach einfachen Lösungen dafür, wie die Ist-Architektur an die Soll-Architektur angeglichen werden kann. Oder sie diskutieren die geplante Soll-Architektur und stellen fest, dass die im Sourcecode gewählte Lösung besser ist. In diesem Fall muss die Soll-Architektur angepasst werden.

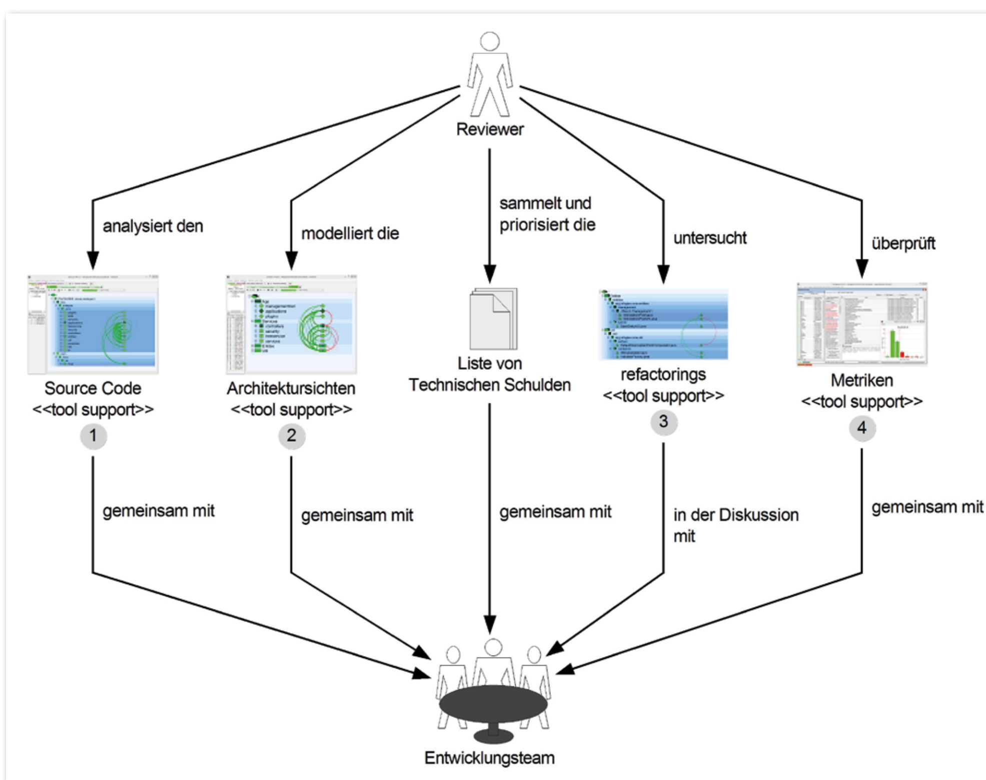


Abbildung 3: Identifizieren von technischen Schulden bei der Architektur-Analyse

Abbildung 3 zeigt den Ablauf einer Architektur-Analyse zur Identifizierung von technischen Schulden. Sie wird von einem Reviewer gemeinsam mit den Architekten und Entwicklern des Systems in einem Workshop durchgeführt. Zu Beginn des Workshops wird der Sourcecode des Systems mit dem Analysewerkzeug geparst (1) und so die Ist-Architektur erfasst. Auf die Ist-Architektur wird nun die Soll-Architektur modelliert, sodass der Vergleich von Soll und Ist möglich wird (2).

Dabei werden technische Schulden sichtbar und der Reviewer macht sich gemeinsam mit dem Entwicklungsteam auf die Suche nach einfachen Lösungen, wie die Ist-Architektur durch ein Refactoring an die Soll-Architektur angeglichen werden kann (3). Oder aber Reviewer und Entwicklungsteam stellen in der Diskussion fest, dass die im Sourcecode gewählte Lösung besser ist als der ursprüngliche Plan. Manchmal ist allerdings weder die Soll- noch die abweichende Ist-Architektur die beste Lösung; Reviewer und Entwicklungsteam müssen dann gemeinsam ein neues Zielbild für die Architektur entwerfen.

Insbesondere, wenn ein System vor einer größeren Erweiterung steht, tritt dieser Fall ein. Die geplante Architektur war für die anstehende Erweiterung nicht ausgelegt, sodass eine grundlegende Weiterentwicklung der Architektur notwendig wird. Dabei werden Fragen beantwortet wie: Ist die vorhandene Architektur flexibel genug für die Erweiterung? Muss die Kopplung an einigen Stellen reduziert werden, damit eine Umstrukturierung möglich wird? Ist der Schnitt der Module und Schichten richtig gewählt, um die neuen Module mit der neuen Funktionalität konsistent zu integrieren?

Hat man ein Software-System übernommen, das man nicht selbst mitentwickelt hat, dann lohnt sich auf jeden Fall eine tiefgehende Architektur-Analyse, um das System überhaupt erst einmal kennenzulernen. Ist die Sourcecode-Basis dem Entwicklungsteam unbekannt, so können auf diese Weise die ursprünglich geplanten Strukturen überhaupt erst sichtbar gemacht werden.

Der Vollständigkeit halber ist in *Abbildung 3* auch die Überprüfung von Metriken erwähnt (4). Metriken fallen in den Bereich der Implementationsschulden, weshalb wir sie hier nicht weiter betrachten.

Neben diesem Abgleich zwischen Soll- und Ist-Architektur besteht ein wichtiger Teil der Architektur-Analyse darin, dass das Entwicklungsteam oder auch das Manage-

ment wissen will, ob die gewählte Architektur gut wartbar ist. Um diese Frage zu beantworten, bedient sich die Autorin bei ihren Analysen eines Modells, das sie auf Basis von Erkenntnissen aus der kognitiven Psychologie entwickelt hat.

Kognitive Psychologie als Basis der Bewertung

Das menschliche Gehirn hat sich im Laufe der Evolution einige beeindruckende Mechanismen angeeignet, die uns beim Umgang mit komplexen Strukturen helfen. Diese gilt es in Software-Systemen zu nutzen, damit Wartung und Erweiterung schnell und ohne viele Fehler von der Hand gehen. Das Ziel ist, dass wir unsere Software-Systeme auch mit sich verändernden Entwicklungsteams lange bei gleichbleibender Qualität weiterentwickeln können. Die drei Mechanismen, die unser Gehirn für komplexe Strukturen entwickelt hat, sind „Chunking", „Bildung von Hierarchien" und „Aufbau von Schemata". Sie haben direkte Abbilder in Kriterien für die Architektur *(siehe Abbildung 4)*.

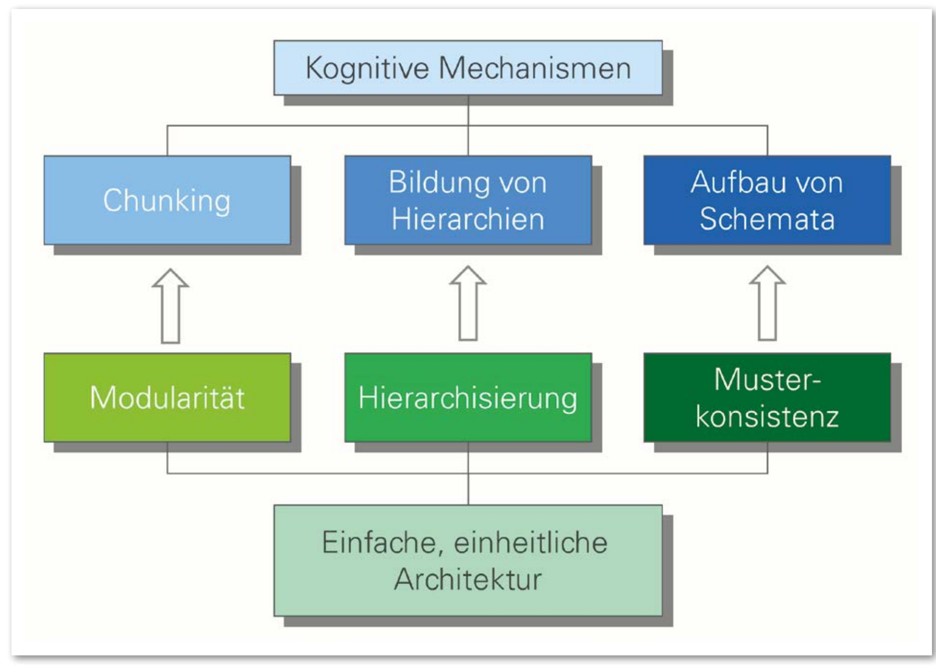


Abbildung 4: Kognitive Mechanismen und ihr Abbild in der Architektur

Aufbau von Schemata und Muster-Konsistenz

Der effizienteste Mechanismus, den Menschen einsetzen, um komplexe Zusammenhänge zu strukturieren, sind sogenannte „Schemata". Ein Schema besteht auf der abstrakten Ebene aus den typischen Eigenschaften der von ihm schematisch abgebildeten Zusammenhänge. Auf der konkreten Ebene beinhaltet ein Schema eine Reihe von Exemplaren, die prototypische Ausprägungen des Schemas darstellen.

Haben wir für einen Zusammenhang in unserem Leben ein Schema, so können wir die Fragen und Probleme, mit denen wir uns gerade beschäftigen, sehr viel schneller verarbeiten als ohne Schemata. Die in der Software-Entwicklung vielfältig eingesetzten Entwurfsmuster nutzen die Stärke des menschlichen Gehirns, um mit Schemata zu arbeiten. Haben Entwickler bereits mit einem Entwurfsmuster gearbeitet und daraus ein Schema gebildet, so können sie Programmtexte und Strukturen, die dieses Entwurfsmuster einsetzen, schneller erkennen und verstehen. Der Aufbau von Schemata liefert für das Verständnis von komplexen Strukturen also entscheidende Geschwindigkeitsvorteile. Das ist auch der Grund, warum Muster in der Software-Entwicklung bereits vor Jahren Einzug gefunden haben.

Muster kann man bei der Architektur-Analyse nicht messen, man kann sie aber sichtbar machen und ihre Umsetzung qualitativ überprüfen. Für die Entwickler und Architekten ist es wichtig, dass es Muster gibt, dass sie im Sourcecode wiederzufinden sind und dass sie einheitlich und durchgängig eingesetzt werden. Deshalb verwendet die Autorin für diesen Bereich den Begriff „Muster-Konsistenz".

Abbildung 5 zeigt ein anonymisiertes Tafelbild, das sie mit einem Team entwickelt hat, um seine Muster aufzunehmen. Auf der rech-

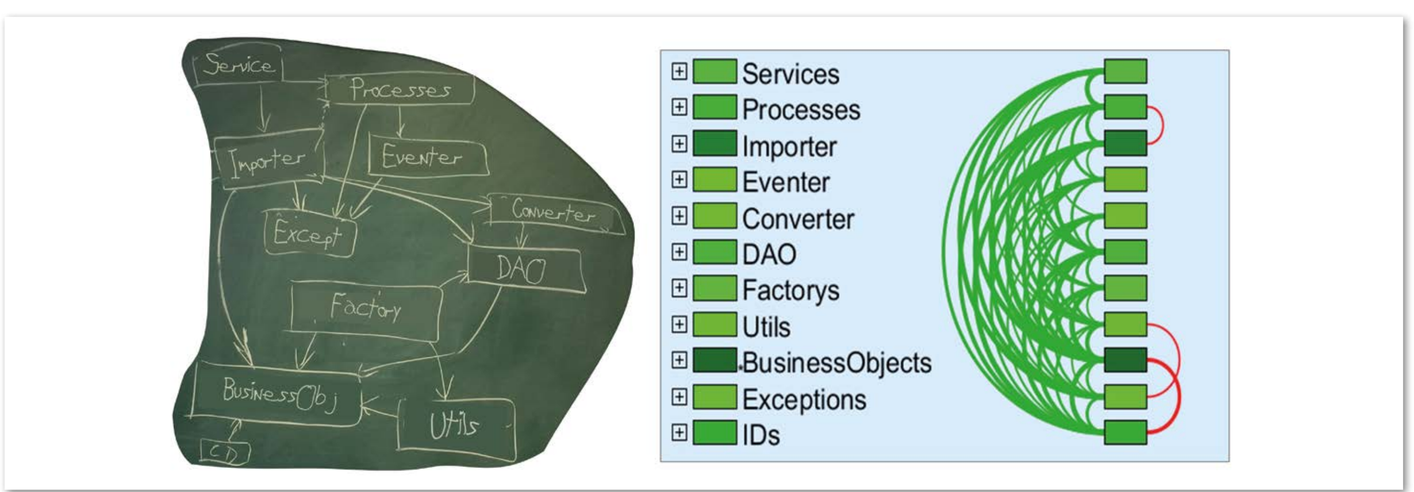


Abbildung 5: Muster auf Klassenebene = Mustersprache

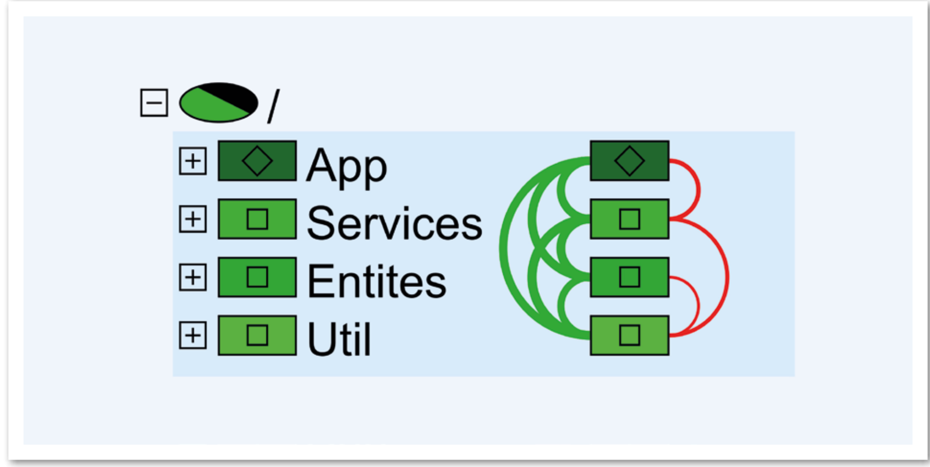


Abbildung 6: Zyklen auf Architektur-Ebene

ten Seite ist der Sourcecode in diese Muster-Kategorien eingeteilt und man sieht sehr viele grüne und einige wenige rote Beziehungen. Die roten Beziehungen gehen von unten nach oben gegen die durch die Muster entstehende Schichtung. Die geringe Anzahl der roten Beziehungen ist ein sehr gutes Ergebnis und zeugt davon, dass das Entwicklungsteam seine Muster sehr konsistent einsetzt.

Spannend ist bei der Analyse auch, welchen Anteil des Sourcecodes man Mustern zuordnen kann und wie viele Muster das System schlussendlich enthält. Lassen sich 80 Prozent oder mehr des Sourcecodes Mustern zuordnen, so spricht die Autorin davon, dass dieses System eine Mustersprache hat. Für die richtige Anzahl von Mustern hat sie keine exakte Zahl. Wichtig ist vielmehr, dass die vorhandenen Muster tatsächlich unterschiedliche Konzepte darstellen und nicht Varianten eines Konzeptes sind. Beispiele für solche Varianten könnten zum Beispiel zwei Muster sein, die „Service" und „Manager" heißen. Hier wäre zu klären, was den Manager von einem Service unterscheidet und in welchem Verhältnis sie zueinander stehen.

Die Untersuchung der Muster im Sourcecode ist in der Regel der spannendste Teil einer Architektur-Analyse. Hier hat man die Ebene zu fassen, auf der das Entwicklungsteam wirklich arbeitet. Die Klassen, die die einzelnen Muster umsetzen, liegen oft über die Packages oder Directories verteilt. Mit einer Modellierung der Muster wie in *Abbildung 5* rechts kann man diese Ebene der Architektur sichtbar und analysierbar machen.

Chunking und Modularität

Damit Menschen in der Menge der Informationen zurechtkommen, mit denen sie konfrontiert sind, müssen sie auswählen und Teil-Informationen zu größeren Einheiten gruppieren. Dieses Bilden von höherwertigen Abstraktionen, die immer weiter zusammengefasst werden, nennt man „Chunking". Dadurch, dass Teil-Informationen als höherwertige Wissenseinheiten abgespeichert sind, wird das Kurzzeitgedächtnis entlastet und weitere Informationen können aufgenommen werden. Chunking kann unser Gehirn allerdings nur dann anwenden, wenn die Teil-Informationen eine sinnvoll zusammenhängende Einheit bilden. Bei unzusammenhängenden Informationen gelingt das Chunking nicht.

Entwickler wenden Chunking automatisch an, wenn sie sich unbekannte Programme erschließen müssen. Der Programmtext wird im Detail gelesen und die gelesenen Zeilen werden zu Wissenseinheiten gruppiert und so behalten. Schritt für Schritt werden die Wissenseinheiten immer weiter zusammengefasst, bis ein Verständnis des benötigten Programmtextes erreicht ist.

Allerdings funktioniert auch bei Software-Systemen das Chunking nur dann, wenn die Struktur des Software-Systems sinnvoll zusammenhängende Einheiten darstellt. Programm-Einheiten, die beliebige Operationen zusammenfassen, sodass für die Entwickler nicht erkennbar ist, warum sie zusammengehören, lassen sich nicht in Wissenseinheiten codieren. Für unsere wartbaren Software-Architekturen ist es also essenziell, dass sie Bausteine wie Klassen, Komponenten, Module oder Schichten enthalten, die sinnvoll zusammenhängende Elemente gruppieren.

Ob die Bausteine in einer Software-Architektur zusammenhängende Elemente darstellen, lässt sich nicht messen, aber qualitativ überprüfen. Wichtige Hinweise geben uns hier die Größenverhältnisse auf allen Ebenen. Bausteine, die auf einer Ebene liege, also die Schichten, die fachlichen Module, die Packages, die Klassen oder die Methoden, sollten untereinander ausgewogene Größenverhältnisse haben. Hier lohnt es sich, die sehr großen Bausteine zu untersuchen, um festzustellen, ob sie Kandidaten für eine Zerlegung sind. Solche großen Bausteine haben in der Regel mehrere Zuständigkeiten beziehungsweise Aufgaben und arbeiten mit zu vielen anderen Bausteinen zusammen. Ihnen fehlt es an Kohäsion und Kopplung.

Bildung von Hierarchien und Hierarchisierung

Hierarchien spielen beim Wahrnehmen und Verstehen von komplexen Strukturen sowie beim Abspeichern von Wissen eine wichtige Rolle. Menschen können Wissen dann gut aufnehmen, es wiedergeben und sich darin zurechtfinden, wenn es in hierarchischen Strukturen vorliegt. Untersuchungen zum Lernen von zusammengehörenden Wort-Kategorien, zur Organisation von Lernmaterialien, zum Verstehen, zur Analyse und zur Wiedergabe von Texten haben gezeigt, dass hierarchisch geordnete Inhalte für Menschen leichter zu erlernen und zu verarbeiten sind und dass aus einer hierarchischen Struktur Inhalte effizienter abgerufen werden können.

Die Bildung von Hierarchien wird in Programmiersprachen bei den Enthaltenseinsbeziehungen unterstützt: Klassen sind in Packages, Packages wiederum in Packages und schließlich in Projekten beziehungsweise Build-Artefakten enthalten. Diese Hierarchien passen zu unseren kognitiven Mechanismen. Sind die Hierarchien an die Muster der Architektur angelehnt, so unterstützen sie uns nicht nur durch ihre hierarchische Strukturierung, sondern sogar auch noch durch Architektur-Muster.

Für alle anderen Arten von Beziehungen gilt das nicht: Wir können beliebige Klassen und Interfaces in einer Sourcecode-Basis per Benutzt- oder/und per Vererbungs-Beziehung miteinander verknüpfen. Dadurch erschaffen wir verflochtene Strukturen (Zyklen), die in keiner Weise hierarchisch sind. Es bedarf einiges an Disziplin und Anstrengung, Benutzt- und Vererbungs-Beziehung hierarchisch zu verwenden. Verfolgen die Entwickler und Architekten von Anfang an dieses Ziel, so sind die Ergebnisse in der Regel nahezu zyklenfrei.

In ihren Analysen bekommt die Autorin die ganze Bandbreite von sehr wenigen zyk-

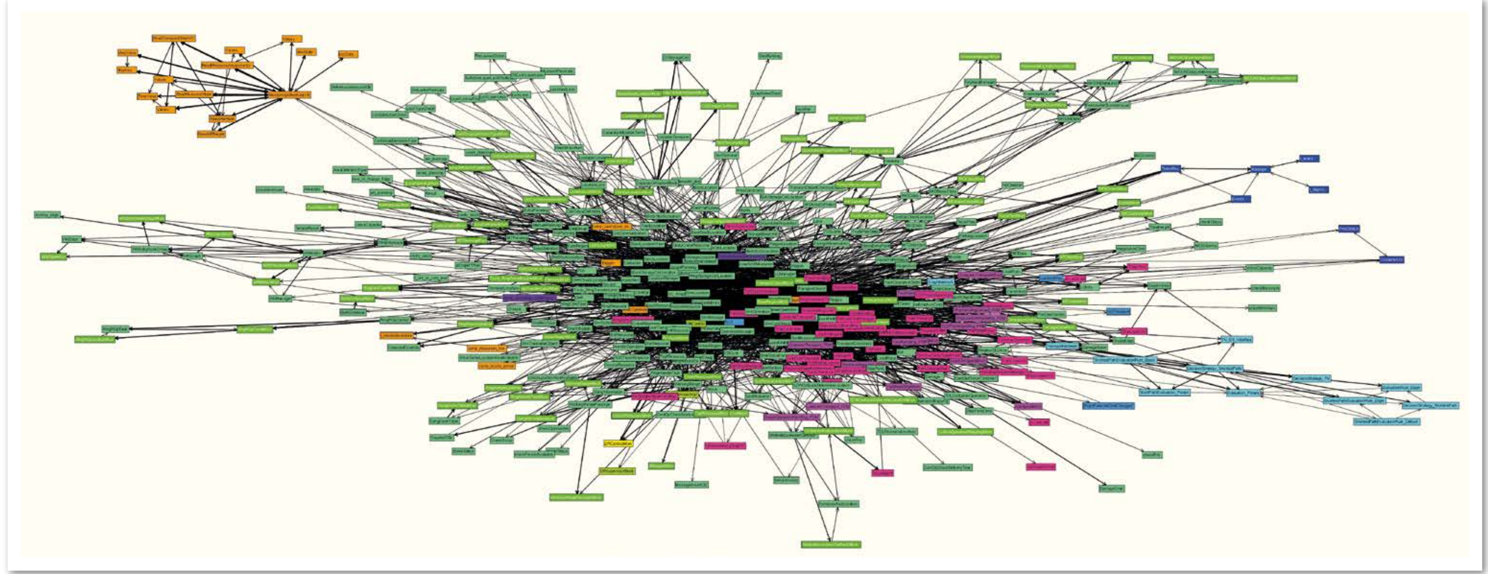


Abbildung 7: Zyklus aus 327 Klassen

lischen Strukturen bis zu großen zyklischen Monstern zu Gesicht. Ähnlich wie bei den Mustern und der Modularität kann man Zyklen auf Architektur-Ebene und auf Klassenebene untersuchen.

Abbildung 6 zeigt vier technische Schichten eines kleinen Anwendungssystems (80.000 Lines of Code). Zwischen den Schichten haben sich einige Rückreferenzen (rote Bögen) eingeschlichen, die zu Zyklen führen. Die Zyklen in der Abbildung sind nur durch 16 Klassen hervorgerufen und lassen sich in diesem Fall leicht ausbauen. Die Abbildung stellt also eine gut gelungene Schichten-Architektur dar.

Der Klassenzyklus in Abbildung 7 stammt von einem anderen System. Die 242 Klassen in diesem Zyklus sind über 18 Verzeichnisse verteilt. Jedes Verzeichnis ist in der Abbildung mit einer anderen Farbe vertreten. Insgesamt hat das System 479 Klassen. Hier brauchen sich also über die Hälfte aller Klassen (327) direkt oder indirekt. Noch dazu hat dieser Zyklus eine starke Konzentration im Zentrum und wenige Satelliten. Eine natürliche Möglichkeit, ihn anhand von Kopplungszentren zu zerlegen, bietet sich also nicht an. Zum Glück finden sich in den meisten Systemen kleinere und weniger konzentrierte Zyklen, die man mit wenigen Refactorings zerlegen kann.

Fazit

Dieser Artikel hat einen ersten Eindruck davon vermittelt, wie technische Schulden in Architekturen entstehen und wie man sie reduzieren kann. Technische Schulden lassen sich abbauen, indem man Strukturen schafft, die unser Gehirn leicht verarbeiten kann. Hat man die Architektur in diese Richtung verbessert, so gehen Wartung und Erweiterung effizienter und schneller von der Hand.

Weitere Informationen

[1] Cunningham, W.: The WyCash Portfolio Management System. Experience Report, OOPSLA ,92, 1992.
[2] Carola Lilienthal: Langlebige Software-Architekturen, Technische Schulden analysieren, begrenzen und abbauen. Dpunkt.verlag, 2015.

Dr. Carola Lilienthal
cl@wps.de

Dr. Carola Lilienthal ist Senior Software-Architektin bei der WPS – Workplace Solutions GmbH. Sie analysiert seit dem Jahr 2003 regelmäßig im Auftrag ihrer Kunden die Zukunftsfähigkeit von Software-Architekturen und spricht auf Konferenzen zu diesem Thema. 2015 hat sie ihre Erfahrungen aus mehr als 100 Analysen von 20.000 Projekten und 15 Millionen Lines of Code in dem Buch „Langlebige Software-Architekturen" zusammengefasst.

Topaktuell: *Das Wahlergebnis zum JCP Executive Committee*

Leider hat es für den iJUG nicht gereicht. Bei der Wahl für einen der sechs Sitze des Executive Committee im Java Community Process (JCP) kam der iJUG auf den achten Platz. Unter den gegebenen Umständen ist das Ergebnis nicht so schlecht, wenn man bedenkt, dass der iJUG international noch keinen großen Bekanntheitsgrad hat. Der IJUG wird die Weiterentwicklung des JCP auch von außen mit voller Kraft unterstützen und sein Community-Aktivitäten weiter verstärken – mal sehen, was bei der Wahl in zwei Jahren wird.

Das Ergebnis der Wahl finden Sie hier: *„https://jcp.org/aboutJava/communityprocess/elections/2016.html".*

„Eine Plattform für den Austausch in der Region …“

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 der Java aktuell, sprach darüber mit Stefan Hildebrandt von der Java User Group Bremen.

Wie ist die Java User Group Bremen organisiert?

Stefan Hildebrandt: An der Organisation sind neben mir primär Rabea Gransberger als Initiatorin der JUG, Dennis Kieselhorst und Torsten Mohrbach beteiligt. Hinzu kommen einige Ansprechpartner für Räume. Da regelmäßige Treffen für die Organisation zusätzlich zu den Vorträgen nicht gut funktioniert haben, findet die Organisation mittlerweile über einen dauerhaften Hangout-Chat – teilweise als Videokonferenz – statt. Da die meisten Leute aus dem Orga-Team und viele der Teilnehmer aus Oldenburg kommen, finden nun auch Veranstaltungen in Oldenburg statt und wir haben uns in „JUG Bremen/Oldenburg“ umbenannt.

Was sind die Ziele der Java User Group Bremen/Oldenburg?

Stefan Hildebrandt: Neben der Veranstaltung von klassischen Vorträgen geht es uns darum, eine Plattform für den Austausch in der Region rund um Java und Software-Entwicklung im Allgemeinen zu sein. Da wir zeitweise sehr wenig Teilnehmer bei den Vorträgen hatten und einige Termine absagen mussten, haben wir mehr auf lokale Sprecher gesetzt. Aus der Not ist eine Tugend entstanden und wir setzten auch bei deutlich höheren und stabileren Teilnehmerzahlen weiter auf Sprecher aus der Region. Vorträge von Sprechern von außerhalb sind trotzdem gern gesehen und lockern das Programm auf. Weiterhin wollen wir Leuten, die von ihren Unternehmen nicht die nötige Zeit bekommen, um sich kontinuierlich fortzubilden, einen einfachen Zugang zu aktuellen Trends bieten. Deshalb engagieren wir uns auch beim Java-Forum-Nord, das als Ein-Tages-Veranstaltung ohne Übernachtung und mit moderatem Ticketpreis für viele Angestellte im Norden bei ihren Arbeitgebern leichter zu vermitteln ist.

Wie viele Veranstaltungen gibt es pro Jahr?

Stefan Hildebrandt: Ziel ist es, jeden Monat etwas anzubieten. In den Sommerferien und um Weihnachten klappt es nicht immer, dafür haben wir über die zwei Standorte auch mal mehrere Vorträge in einem Monat.

Was bedeutet Java für euch?

Stefan Hildebrandt: Für viele ist es das zentrale Werkzeug im beruflichen Alltag, für einige auch darüber hinaus in privaten oder Open-Source-Projekten. Die Unternehmen, für die wir arbeiten, haben beträchtliche Investitionen in Projekte getätigt, die auf der Java-Plattform basieren. So ist es für deren Zukunft von essenzieller Bedeutung, da aktuell auch kaum reelle Alternativen existieren. Das ist leider nicht allen Unternehmen bewusst.

Welchen Stellenwert besitzt die Java-Community für euch?

Stefan Hildebrandt: Die Organisatoren sind über unterschiedliche Kanäle seit Jahren in der überregionalen Java-Community tätig und haben viele Kontakte. Diese lassen sich mit der Arbeit in der JUG gut verbinden. Für viele Teilnehmer ist die JUG der erste Berührungspunkt mit einer überbetrieblichen Software-Entwicklungs-Community, bei dem man deutliche Unterschiede zwischen den Teilnehmern aus Bremen und Oldenburg sieht. Hier hoffen wir auf Dauer den Austausch verbessern und mehr Interaktivität in die Vorträge und in die Diskussionen nach den Vorträgen bringen zu können. Auch alternative Formate wollen wir weiter ausprobieren.

Wie sollte sich Java weiterentwickeln?

Stefan Hildebrandt: Hier sehe ich drei Bereiche, die ich getrennt betrachten möchte: Erstens die Sprache an sich: Sie sollte mit der Zeit gehen, aber eher evolutionär, da viele Projekte mit großen Code-Basen, aber wenig Entwicklern existieren und große Änderungen die Adaption einer neuen Version verhindern würden. Für meinen Geschmack könnte es trotzdem mehr Releases mit kleinerem Feature-Umfang geben. Zweitens: Die JVM ist Basis für eine Vielzahl von Sprachen, die sich auch in einer Anwendung vereinen lassen. Somit ist das eine oder andere Feature, das Java als Sprache fehlt, zu verschmerzen, da es eine der anderen Sprachen bietet, wenn ich es wirklich benötige oder es mir viele Erleichterungen bietet. Ich bevorzuge für Tests zum Beispiel Groovy, da sich damit deutlich sprechendere Tests erstellen lassen. Die JVM sollte eine ausgereifte, stabile, sichere und zuverlässige Basis für dieses Ökosystem bilden. Drittens, Java EE: Dies ist sicher der politischste Teil, da für Oracle hier das meiste – oder überhaupt – Geld zu verdienen ist: mit dem WebLogic Application Server und in Zukunft vermehrt mit Java-EE-Cloud-Lösungen. An dieser Zukunft arbeiten sie und sind dabei, die Investitionen zu sichern. Dass sie es über Standards machen wollen, ist löblich, scheint ihrer Meinung nach aber besser zu funktionieren, wenn nicht so viel Community mitspielt. Das führt leider zu Verstimmungen und zu Verzögerungen, insbesondere bei ein paar zwingenden Spec-Updates rund um REST-Services und bei der Einführung von MVC.

Zur Person: *Stefan Hildebrandt*

Stefan Hildebrandt ist als freier Software-Entwickler und Berater seit zehn Jahren in größeren Projekten bei Kunden aus unterschiedlichen Branchen tätig. Seine Schwerpunkte sind Web- und Backend-Entwicklung mit Java und JavaScript sowie Werkzeuge zur Test- und Deployment-Automatisierung. Sein Interesse gilt vermehrt der ganzheitlichen Betrachtung des Software-Entwicklungsprozesses und der Potenziale, die außerhalb der eigentlichen Entwicklung schlummern.

Wie sollte Oracle eurer Meinung nach mit Java umgehen?

Stefan Hildebrandt: Oracle sollte in Bezug auf die Sprache Java und die JVM als Product Owner (zum Beispiel nach Scrum) im positiven Sinne vorgehen: Feedback sammeln, Ideen entwickeln und in neue Features umwandeln, die Mehrwert für die Nutzer und damit für sich schaffen. Wenn sie das gut hinbekommen, können sie gegebenenfalls mehr Support und Werkzeuge rund um die VM verkaufen und die Weiterentwicklung auf Dauer sicherstellen. Leider funktioniert es in großen, bisher wenig agilen Unternehmen nicht so einfach. Im Java-EE-Umfeld ist es noch schwieriger, da die Anforderungen noch heterogener sind und die eigenen Interessen vermutlich stärker abweichen.

Wie sollte sich die Community gegenüber Oracle verhalten?

Stefan Hildebrandt: Die Community muss respektieren, dass Oracle ein Unternehmen ist, das einen eigenen Kurs verfolgt, um seine Ziele zu erreichen. Auf der anderen Seite muss Oracle auch sehen, dass aus Java EE nur durch die Community und deren Unterstützung ein brauchbarer Standard geworden ist und nicht nur sie dort Geld investieren. Daher sollten beide Seiten einen respektvollen Umgang miteinander pflegen. Diese Kommunikation sollte kontinuierlich stattfinden, damit nicht zu große und ungeklärte Sendepausen entstehen. Wenn Oracle sich neu organisieren muss, sollten sie es frühzeitig und offen kommunizieren. Dazu gehören Termine, der aktuelle Stand der Dinge und wann die nächsten Informationen zu erwarten sind. Nur so können die Leute und Unternehmen, die mit viel Einsatz und Herzblut bei der Weiterentwicklung mitarbeiten, für sich ein wenig Planungssicherheit gewinnen. Daher ist auch das Einfordern von Antworten seitens Oracle verständlich. Da die Kommunikation bei vielen JSRs quasi vollständig offen ist, ist das Ganze schnell ein öffentliches Politikum.

Stefan Hildebrandt
stefan.hildebrandt@jug-hb-ol.de
http://www.meetup.com/de-DE/jugbremen

JUG Saxony Day 2016 mit 400 Teilnehmern

Bei Sonnenschein und bester Laune hat die JUG Saxony am Fuß der Radebeuler Weinberge mehr als doppelt so viele Java-Begeisterte als im Vorjahr begrüßen dürfen.

Die Eröffnung der Konferenz durch Staatsminister für Wirtschaft, Arbeit und Verkehr Martin Dulig hat Java User Group darin bestätigt, dass sie im Software-Bereich eine hohe Sichtbarkeit erreicht hat. Im Anschluss stimmte Dalibor Topic (Oracle Deutschland) alle Teilnehmer mit seiner Keynote „Prepare for JDK 9!" auf die kommende Java-Version ein. Danach war es an den Teilnehmern, aus 25 hochkarätigen Sessions das eigene Tagesprogramm auszuwählen.

Zwischen den Vorträgen konnten die Konferenzbesucher mit den 19 Ausstellern ins Gespräch kommen oder sich am Bücher-Tisch über die neuesten Bücher der IT- Welt informieren. Dabei blieb auch genügend Zeit für Networking und angeregte Diskussionen.

Die Digitale Transformation hat auch vor der Konferenz- und Vortragsbewertung nicht Halt gemacht. Mithilfe der JUG Saxony Day App konnten die Teilnehmer schon zur Abschlussveranstaltung den besten Speaker der Konferenz küren. Unter vielen interessanten und lehrreichen Vorträgen konnte Stefan Zörner zum Thema „Architektur-Dokumentation" das Publikum am stärksten mitreißen.

Am Abend haben die Teilnehmer noch einmal in entspannter Atmosphäre die Gelegenheit genutzt, Netzwerke zu erweitern, Ideen voranzutreiben oder einfach nur der wundervollen Abendbegleitung durch die Band „Elephants Crossing" zu lauschen.

Meinungsbild der Java-Community zu den aktuellen Entwicklungen auf der JavaOne 2016

Marina Fischer, DOAG Dienstleistungen GmbH

Wie auf der JavaOne jüngst bestätigt, werden sich die Releases von Java EE 8 und Java 9 noch bis 2017 verzögern. Die gute Nachricht: Es geht weiter. Doch reicht das, um die Gunst der Community zurückzugewinnen? Fünf Vertreter der Community berichten von ihren aktuellen Eindrücken der Java-Konferenz.

In den letzten Monaten hatte es für Oracle viel Kritik gehagelt: Die Community beklagte eine lange Phase des Stillstands, nachdem Oracle im Juni 2015 eine weitere Aufschiebung des „Java EE 8"-Releases angekündigt hatte. Nach vielen Protesten und Forderungen der Community versprach Oracle zuletzt, dass es mit der Entwicklung und dem Support von Java EE weitergehen wird. Das bestätigte die von Oracles Group Vice President Anil Gaur auf der JavaOne 2016 vorgestellte aktualisierte „Java EE"-Roadmap. Mark Reinhold, Chief Architect der Java Platform Group bei Oracle, hatte ähnlich unspezifische Nachrichten zu verkünden: „Java 9 wird bald kommen, aber nicht ganz so schnell, wie es viele erhofft haben", sagte er während seiner Keynote.

Was sagt die Community zu den neuesten Entwicklungen?

André Sept, Leiter der DOAG Java Community:

„Nach den Versprechungen seitens Oracle, zur JavaOne eine Lösung zum Stillstand der Entwicklung von JavaEE 8 und 9 zu liefern, hatte ich nach der Keynote nicht das Gefühl, das Oracle ernsthaft vorhatte, die Community zu beruhigen. Meine Bilanz am Montagabend: Oracle hat eingesehen, dass sich die Welt mit neuen Programmiermodellen und Microservice-Architekturen sehr stark geändert hat. Dies ist für mich der einzig richtige, vernünftige, aber auch mutige Weg, den Oracle eingeschlagen hat. Bei der ganzen Diskussion der letzten Monate war mir persönlich die Erhal-

tung des JCPs und der JavaEE-Standards am wichtigsten, da auch hier nicht zu 100 Prozent klar war, wohin die Reise durch den Stillstand geht. Jetzt gilt es seitens der Community, mitanzupacken und sich in den Expert Groups zu beteiligen, um die Zukunft im konstruktiven Dialog mit Oracle mitzugestalten."

Werner Keil, Specification Lead:
„Ich habe den Eindruck, dass Java EE nach langer Unsicherheit deutlich an Gewicht gewonnen hat, auch was die Länge der Keynote-Sessions angeht. Mark Reinhold schien auch ob der erneuten Verzögerung von JDK9 etwas weniger großspurig als sonst, obwohl gerade seine JShell- oder JLink-Demos sehr informativ waren, und auch für Java EE spätestens mit EE 9 viel Potential bieten. Ich sage nur Profiles mehr oder weniger nach Bedarf und Belieben.

Die Abkehr von einstigen „Darling"-JSRs wie MVC mag überraschen, aber mit Fokus auf Cloud, Internet of Things, DevOps oder Big Data wurden andere JSRs wie JSON einfach wichtiger. Insofern dürfte Oracle mit den jüngsten Entscheidungen durchaus auf die Community gehört haben.

Konstruktive Gespräche mit Projekten wie Tamaya oder Archaius zum Thema Configuration Standard bieten Anlass zur Hoffnung. Wenn man auch deren Hilfe annimmt, dann wirkt auch ein etwas „sportlicher" Zeitplan nicht völlig unrealistisch."

Niko Köbler, JUG-Leader Darmstadt:
„Wirkliche Neuerungen wurden in der JavaOne-Keynote nicht verkündet. Das groß erwartete Thema, wie Oracle mit Java EE im Allgemeinen und mit der Version 8 im Speziellen umgeht, wurde zwar behandelt – jedoch wie? Anil Gaur verkündete zwar stolz, man habe auf die Community gehört und deshalb eine neue Ausrichtung von Java EE 8 vorgenommen, aber mir (und vielen anderen) ist das zu unengagiert. Projekte und APIs dürfen nicht einfach fallen gelassen werden, ohne darüber ein Wort zu verlieren. Im Speziellen meine ich den Umgang mit JSR-371, der MVC API. Dieser JSR erschien weder auf Oracles Folien und in der neu angelegten Community-Umfrage wird er nur am Rande behandelt. Oracle sollte langsam verstehen, dass ein serverseitiges, zustandsloses und Action-basiertes Framework im Standard seit Jahren überfällig ist und der Bedarf für serverseitig gerenderte Seiten nach wie vor hoch ist. Neben dem MVC-JSR wurden aber auch andere JSRs einfach stillgeschwiegen."

Andreas Badelt, Stellvertretender Leiter der DOAG Java Community:
„Die JavaOne-Keynote am Vortag der eigentlichen Konferenz hätte besser laufen können. Nach den ganzen, teilweise sehr emotional geführten Diskussionen um den Stillstand von Java EE 8, und dem Vertrösten der Community auf die JavaOne, war es sicher nicht gelungen, das Publikum mit off-topic-Präsentationen hinzuhalten, und am Ende den eigentlich spannenden Teil der Präsentation aus Zeitmangel frühzeitig zu beenden. Die vielen kritischen Kommentare auf Twitter waren da schon berechtigt. Mit den detaillierten Sessions am Montag sollte Oracle aber ein erster Schritt gelungen sein, die Community wieder einzufangen. Zumindest wurden vernünftige Gründe für strittige Punkte wie das Stoppen des MVC JSRs geliefert. Außerdem wurden viele Details zu den geplanten Änderungen präsentiert, wenn auch vieles noch im Anfangsstadium steckt. Das von Oracle angekündigte „Community Survey" wird hoffentlich auf große Resonanz stoßen und von Oracle dann auch vernünftig genutzt. Die vielen offenen Fragen sehe ich aber auch als Chance bzw. Aufforderung insbesondere an die Java-User-Gruppen, sich verstärkt zu engagieren. Persönlich hoffe ich, dass in den nächsten Jahren nicht ausschließlich auf die Hype-Themen Cloud, Microservices und „Distributed Processing" im Allgemeinen gesetzt wird, weil sie auch nicht die Lösung für alle Probleme sind."

Sebastian Daschner, DOAG-Themenverantwortlicher JCP und Expert Group Member für JAX-RS, schreibt auf seinem Blog:
„Zunächst einmal ist es sehr positiv, von Oracle eine Äußerung zur Zukunft von Java EE zu bekommen. Die Neuzugänge und Updates der Plattform scheinen für mich auch sehr vernünftig, vor allem in Hinblick auf Belastbarkeit, Reaktionsfähigkeit, Eventual Consistency, Health Checks und Configuration. Nicht wenige Open-Source-Beiträge sind entstanden, zum Beispiel Deltaspike, Adam Biens Breakr oder Porcupine-Projekte sowie herstellerspezifische Funktionalitäten wie der reaktive Support in Jersey. Das Hinzufügen solcher Features zur Plattform begrüße ich sehr. (...) Im Gegensatz dazu macht die Entfernung von MVC und deren Rechtfertigung keinen Sinn für mich. (...) Eine weitere JSR, die in EE 8 aufgenommen werden sollte, ist JCache. (...) Was in Zukunft auch verbessert werden sollte, ist die Kommunikation mit der Community von Oracle-Seite."

Die iJUG-Mitglieder auf einen Blick

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

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

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

Java User Group Köln
www.jugcologne.eu

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

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

Java User Group Nürnberg
www.meetup.com/de-DE/JUG-Nurnberg

Java User Group Ostfalen
www.jug-ostfalen.de

Java User Group Saxony
www.jugsaxony.org

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

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

Berlin Expert Days e.V.
www.bed-con.org

Java Student User Group Wien
www.jsug.at

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

Java User Group Hannover
www.jug-h.de

Java User Group Augsburg
www.jug-augsburg.de

Java User Group Bremen
www.jugbremen.de

Java User Group Münster
www.jug-muenster.de

Java User Group Hessen
www.jugh.de

Java User Group Dortmund
www.jugdo.de

Java User Group Hamburg
www.jughh.de

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

Java User Group Kaiserslautern
www.jug-kl.de

Java User Group Switzerland
www.jug.ch

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

Java User Group Görlitz
www.jug-gr.de

Java User Group Mannheim
www.majug.de

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

Java User Group Düsseldorf rheinjug
www.rheinjug.de

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

Java User Group Bielefeld
www.meetup.com/de-DE/Java-User-Group-Bielefeld

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

www.ijug.eu

Impressum

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

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

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

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

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

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

Fotonachweis:
Titel: © ra2 studio/Fotolia
Foto S. 11 © www.raspberrypi.org
Foto S. 16 © d3js.org
Foto S. 21 © Kotlin/www.kotlinlang.org
Foto S. 30 © Sergey Nivens/123 RF
Foto S. 34 © Sergey Nivens/Fotolia
Foto S. 41 © Artem Egorov/123 RF
Foto S. 46 © Olga Sokolova/123 RF
Foto S. 51 © Nytoprod/Fotolia
Foto S. 56 © Kaspars Grinvalds/123 RF
Foto S. 64 © alphaspirit/Fotolia

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

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

Alle Rechte vorbehalten. Jegliche Vervielfältigung oder Weiterverbreitung in jedem Medium als Ganzes oder in Teilen bedarf der schriftlichen Zustimmung des Verlags.
Die Informationen und Angaben in dieser Publikation wurden nach bestem Wissen und Gewissen recherchiert. Die Nutzung dieser Informationen und Angaben geschieht allein auf eigene Verantwortung. Eine Haftung für die Richtigkeit der Informationen und Angaben, insbesondere für die Anwendbarkeit im Einzelfall, wird nicht übernommen. Meinungen stellen die Ansichten der jeweiligen Autoren dar und geben nicht notwendigerweise die Ansicht der Herausgeber wieder.

Inserentenverzeichnis

cellent AG — S. 15
www.cellent.de

DOAG e.V. — U 2, U 3, U 4
www.doag.org

Werden Sie DOAG-Mitglied!

Ab 105 EUR/Jahr (zzgl. MwSt)

„Gemeinsame Interessen gemeinsam vertreten“

+ 20 % Rabatt auf Veranstaltungen
+ Bezug der Zeitschriften
Red Stack Magazin, Business News, Java aktuell

DevCamp 2017

8. Februar in Hannover

Development by choice

Moderne Software-Entwicklung mit Oracle

devcamp.doag.org