GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

März-Mai 1 / 2018

JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis

JAVAPRO

DIE ZUKUNFT VON JAVA ENTERPRISE

Aus Java EE wird Jakarta EE !

www.java-pro.de

Mit freundlicher Unterstützung unserer Partner.

JAVAPRO-PARTNER-NETWORK

Die JAVAPRO wird von den Mitgliedern des JAVAPRO-Partner-Network finanziert und aktiv unterstützt. Dadurch sind wir in der Lage, redaktionell unabhängig zu arbeiten und die JAVAPRO kostenlos für die gesamte Java-Community zu produzieren sowie die Java Konferenz JCON 2018 zu veranstalten.

JCON PARTNER 2017

GOLD PARTNER

ORACLE® eclipse XDEV™

SILBER PARTNER

redhat RAPID clipse™ jpro PEAKWORK THE PLAYER HUB NETWORK

BRONZE PARTNER

QFS Quality First Software viadee® IT-Unternehmensberatung ACCSO ACCELERATED SOLUTIONS GEBIT Solutions ecx.io an IBM Company PENTASYS Unser Maßstab ist der Mensch

Werden auch Sie Mitglied des JAVAPRO Partner Network und unterstützen Sie die JAVAPRO - Das kostenlose Fachmagazin für die Java Community.

www.java-pro.de/partner

JAVAPRO

Impressum

JAVAPRO

Verlag:
IT Press & Media GmbH
Mergenthalerallee 73-75
65760 Eschborn
Telefon: +49 (0) 61 96 - 20 48 010
Telefax: +49 (0) 61 96 - 20 48 019
E-Mail: info@java-pro.de
Website: http://www.java-pro.de

Chefredakteur:
Sebastian Dippold (V.i.S.d.P.)

Redaktion:
info@java-pro.de

Gestaltung, Layout, Produktion:
IT Press & Media GmbH
Mergenthalerallee 73-75
65760 Eschborn

Illustrationen:
Pixabay (Public Domain)
Erscheinungsweise: Vier mal jährlich
Gründungsjahr: 2017
Preis: kostenfrei
Die nächste Ausgabe erscheint am 15. Juli 2018.

#JAVAPRO #Editorial

Java ist tot – es lebe Jakarta!

Java EE gibt es nicht mehr. Oracle hat die Entwicklung der Java Enterprise Edition an die Eclipse Foundation übertragen. Als diese Ausgabe noch in Planung war, sollte Java Enterprise bei Eclipse als Top-Level-Projekt unter dem Namen Eclipse Enterprise for Java (EE4J) weiterentwickelt werden. Seit Ende Februar wissen wir aber: Es gibt einen neuen Namen. Einen festen neuen Namen. Java EE heißt jetzt Jakarta EE. Somit umgeht die Eclipse Foundation - wie auch schon mit dem „Arbeitstitel" EE4J - ein gravierendes Problem: das Namensrechtsproblem.

Die Rechte des Namen Java liegen bei Oracle. Java EE durfte somit nicht mehr Java EE heißen. In einem aufwändigen Community-Prozess mit Abstimmungen, Vorschlägen und Kommentaren wurde aus Java dann Jakarta. Namenspatron ist damit nicht mehr Indonesiens größte Insel, sondern Indonesiens größte Stadt – die Hauptstadt Jakarta. Und die liegt natürlich auf Java.

Ein weitaus gravierenderes Problem welches bei der Migration von Oracle zu Eclipse besteht: Wie die bis dato verwendeten Package-Namen (javax) bei der Namensdebatte behandelt werden sollen. Sollten auch diese von einer Um- bzw. Neubenennung betroffen sein, stünde damit die Abwärtskompatibilität von Java EE bzw. Jakarta EE auf dem Spiel. Oracle hat sich aber schon bereit erklärt hier einen Schritt auf die Community zuzugehen:
Bereits existierende javax-Packages sollen weiterhin verwendet werden dürfen, neue Packages müssen aber ab jetzt einen neuen Namen erhalten. Eine weitere große Forderung der Community, dass Java EE doch seinen Namen behalten solle, hat Oracle dagegen eine Absage erteilt.

Um die Community dennoch weiter mit einzubeziehen hat Eclipse sie in die Namenssuche mit eingebunden. In der letzten Abstimmung fiel dann die Entscheidung zwischen einem eher vorsichtig formuliertes Enterprise Profiles und Jakarta EE. Die fast 7.000 Teilnehmer an diesem Poll fällten eine deutliche Entscheidung: 64% waren für Jakarta EE.

Bleibt zu hoffen, dass die Mitwirkung an Jakarta EE genauso lebhaft bleibt und das „neue Java EE" eine starke Community bekommt.

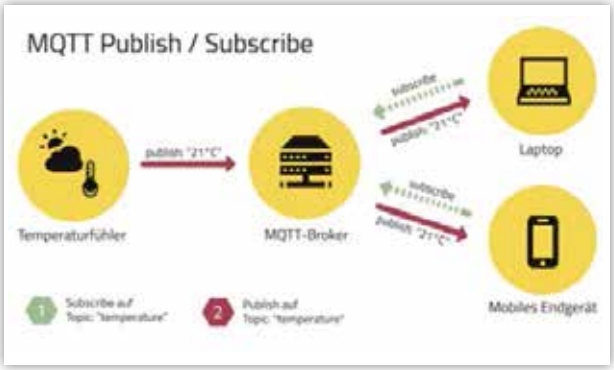
Viel Spaß beim Lesen.

Sebastian Dippold

Links:

1 https://www.eclipse.org/org/workinggroups/eclipse_ee_next_charter.php

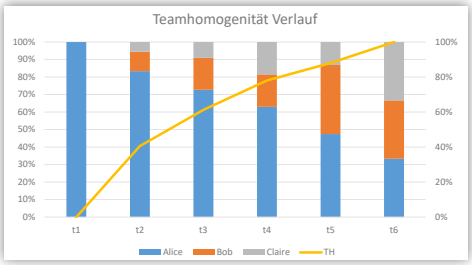
Sebastian Dippold
Chefredakteur
JAVAPRO



Publish-/Subscribe-Architektur von MQTT.

Von Richard Fichtner

Bei agiler Softwareentwicklung mit SCRUM stehen die Anpassung an veränderte Bedingungen bzw. die Anforderungen und die Optimierung des Entwicklungsprozesses im Mittelpunkt. Anpassungen und Optimierungen gehen dabei von den Entwicklern selbst aus. Um Entscheidungen besser beurteilen zu können, kommen i.d.R. Metriken zum Einsatz. Als Grundlage werden Daten zu Arbeitszeiten und Aufgaben benötigt. Die Metriken ermöglichen den Teammitgliedern die Metadaten ihrer eigenen Arbeit besser zu verstehen und seine Prozesse sinnvoll anzupassen.



Teamhomogenität Verlauf (Abb. 7)

#JAVAPRO #JCON

Starkes Duo

JAVAPRO und die Eclipse Foundation wollen bei der Vermarktung der JCON und der EclipseCon Europe in Zukunft zusammenarbeiten und eine Dual-Konferenz bilden. Beide Konferenzen ergänzen sich perfekt. Auch Teilnehmer und Aussteller dürfen sich über lukrative Angebote freuen und werden von der Kooperation profitieren.

Die JCON geht mit dem kürzlich gestarteten Call-for-Papers in ihr zweites Jahr und wird heuer vom 9. bis 12. Oktober 2018 erneut in Düsseldorf stattfinden. Die JCON ist die neue Java-Konferenz für die Java-Community, die von JAVAPRO und Mitgliedern des JAVAPRO-Partner-Network organisiert wird.

Call-for-Papers jetzt offen

Wer als Speaker dabei sein möchte, kann bis zum 10. April 2018 Vortragsvorschläge unter www.jcon.one einreichen. Das Mitmachen lohnt sich in jedem Fall, denn für die Einreichung gibt es ein Dreitagesticket für die JCON kostenlos. Wird der Vorschlag angenommen, erhält der Speaker zwei zusätzliche JCON-Tickets.

JCON und EclipseCon werden Partner

Die Eclipse Foundation und JAVAPRO, seit 2017 Mitglied der Eclipse Foundation, wollen in Zukunft eng zusammenarbeiten und sich bei der Organisation und Vermarktung beider Konferenzen gegenseitig unterstützen. Beide Konferenzen ergänzen sich perfekt. Auf der EclipseCon Europe, die vom 23. bis 25. Oktober 2018 und damit zwei Wochen nach der JCON in Ludwigsburg stattfinden wird, dreht sich alles um Open-Source-Projekte, die unter dem Dach der Eclipse Foundation entwickelt werden. Neben der populären Eclipse IDE zählen dazu weitere Projekte mit großer Bedeutung für die Java-Community, u.a. Eclipse MicroProfile, die von IBM spendierte Java-VM Eclipse

OpenJ9 und nun auch die Java-Enterprise-Edition, die Oracle im September 2017 zur Weiterentwicklung an die Eclipse Foundation übertragen hat. Die EclipseCon Europe ist traditionell eine Konferenz für Committer. Das Vortragsprogramm richtet sich überwiegend an Entwickler, die an Eclipse-Projekten mitarbeiten. Wer Frameworks wie Java Enterprise primär für die Entwicklung eigener Softwarelösungen nutzen möchte, ist auf der JCON richtig. Durch die Kooperation kann die Eclipse Foundation die gewohnte Ausrichtung der EclipseCon beibehalten und über die JCON können Eclipse-Committer Tuchfühlung mit den eigentlichen Anwendern ihrer Projekte aufnehmen. Auch die Teilnehmer und Aussteller profitieren von der Kooperation. Für Teilnehmer der EclipseCon Europe 2018 ist die Teilnahme an der JCON 2018 kostenlos. Aussteller, die bei beiden Konferenzen dabei sein möchten, erhalten Sonderkonditionen.

XDEVCON wieder Teil der JCON

Im ersten Jahr der JCON waren die XDEVCON-Vorträge besonders gut besucht, insbesondere die zahlreichen Live-Coding-Sessions. Deshalb ist die XDEVCON auch in diesem Jahr wieder Teil der JCON. Die XDEVCON ist seit nunmehr acht Jahren die Top-Konferenz schlechthin für Rapid-Cross-Platform-Development mit Java und Java Migration. Teilnehmer erfahren hier, wie man auf effiziente Weise datengetriebene Business-Anwendungen entwickelt, die man auf allen wichtigen Plattformen und in der Cloud betreiben kann. Auf der XDEVCON stehen die visuelle Entwicklung und Tools im Fokus. Die XDEVCON richtet sich folglich an professionelle Anwendungsentwickler, die möglichst schnell vorwärts kommen möchten, ohne sich zu sehr mit technischer Low-Level-Programmierung aufhalten zu müssen, wie es besonders bei Migrationsprojekten der Fall ist. Die XDEVCON zeichnet sich auch dadurch aus, dass hier noch viele Basics erklärt werden, die auf anderen Konferenzen (auch auf der JCON-Hauptkonferenz) vorausgesetzt werden. Deshalb ist die XDEVCON besonders bei Anwendungsentwickler, 4GL-Umsteiger (Oracle Forms, MS Access etc.), Projektleiter und IT-Entscheider beliebt und damit eine wichtige Ergänzung für das Konferenzprogramm der JCON 2018.

Über 800 Teilnehmer geplant

Obwohl im vergangenem Jahr die Planungen für die erste JCON erst sehr spät beginnen konnten, haben die JCON im ersten Jahr insgesamt 586 Teilnehmer besucht. Heuer starten Vorbereitungen drei Monate früher. Die Partnerschaft mit der EclipseCon wird der JCON weiteren Zulauf bringen, sodass für die JCON 2018 weit über 800 Teilnehmer erwartet werden.

Hochkarätige Fachvorträge und Schulungen

Auch in diesem Jahr wird sich die JCON durch ein hochkarätiges Vortragsprogramm auszeichnen. Geplant sind rund 100 Fachvorträge und bis zu vier Keynotes an insgesamt drei Konferenztagen. Im Fokus stehen in diesem Jahr erstmals gleich zwei neue Java-Versions-Updates - Java 9 und 10 – und das große Thema Serverless / Cloud-Native. Die JCON-Hauptkonferenz bietet nahezu alle Themen, die für die Java Entwickler wichtig sind, u.a. Core Java, APIs und Frameworks, Java Enterprise und Microservices. Auch die Themen Continuous-Delivery, Testing und Quality sowie Web- und Mobile-Development werden wieder hoch im Kurs stehen. Neben der Hauptkonferenz gibt es wieder 4 Special-Days zu den Themenschwerpunkten Cloud und DevOps, Big-Data, Architecture und Agile. Im Anschluss an die JCON-Konferenz gibt es heuer erstmals einen hochkarätigen Training-Day mit Tagesschulungen zu den Themen Java 9 und 10, Microservices und Kubernetes, Reaktive Programmierung, Hibernate Performance-Tuning, Entwicklung von Cloud-Native Applikationen, Entwicklung von Realtime-Applikationen mit Jetstream und Cross-Platform-Development.

Location Multiplex Kino

Die JCON ist die einzige Java-Konferenz in Deutschland, die in einem modernen Multiplex-Kino stattfindet. Der Grund dafür liegt auf der Hand. Für Entwickler ist der spannendste Teil eines Vortrags das Live-Coding. Wirklich zusehen können aber meist nur die Teilnehmer in den vordersten Reihen. Alle anderen Teilnehmer können beim Live-Coding nur zuhören. In einem Multiplex-Kino ist das ganz anders. Auf Kino-Großleinwänden kann man selbst auf den hintersten Reihen Programmcode noch sehr gut erkennen. Das hat enorme Vorteile für Teilnehmer und Referenten gleichermaßen. Die Teilnehmer bekommen deutlich mehr mit, sind aufmerksamer, können die Zusammenhänge besser verstehen und der Vortrag bleibt länger im Gedächtnis.

JAVAPRO startet Bildungsinitiative

Im Rahmen der Bildungsinitiative „Förderung für Informatikfachkräfte und Digitalisierung in Deutschland" ermöglicht JAVAPRO bis zu 200 Studentinnen und Studenten, Dozenten und Lehrkräften, kostenlos an der Java-Konferenz JCON 2018 teilzunehmen. Der Ablauf ist einfach: Interessierte Hochschulen können sich als Bildungspartner anmelden und erhalten dann ein Ticket-Kontingent für die JCON 2018.

Tickets und Preise

Ein Tag auf der JCON 2018 kostet 249 Euro zuzüglich Mehrwertsteuer. Bis 30. Juni 2018 gilt ein Earlybirdpreis von 199 Euro. Alle 3 Konferenztage kosten 499 Euro (Earlybird 399 Euro). Im EclipseCon-2018-Ticket ist die JCON 2018 kostenlos mit dabei. Die Teilnahmegebühr für den Schulungstag am 12. Oktober beträgt 599 Euro (Earlybird 449 Euro). Hier sind die Teilnehmerzahlen limitiert. Die reduzierten Earlybird-Preise gelten noch bis zum 30. Juni 2018.

DIE JCON 2018

JCON 2018

Die JCON ist die große Java-Community-Konferenz der JAVAPRO, mit Fokus auf Java-Entwicklung in der Praxis, APIs, Architektur, Projektmanagement und Cloud. In diesem Jahr dreht sich alles um die neuen Java-Updates 9 und 10, Java Enterprise, APIs, Microservices und vor allem um das Top-Thema Serverless / Cloud-Native. Parallel zur Hauptkonferenz gibt es wieder vier Special Days zu den Themen: Cloud, Serverless & DevOps, sowie zu Big-Data, Agile und Architecture. Am vierten Tag findet ein hochkarätiger Schulungstag statt.

Entwickler wollen Code sehen. Deshalb findet die JCON in einem Multiplex-Kino statt. Freuen Sie sich auf ein einzigartiges Live-Coding Erlebnis, das es in Deutschland nur auf der JCON gibt.

Organisiert wird die JCON 2018 von JAVAPRO zusammen mit Mitgliedern des JAVAPRO-Partner-Network.

ECLIPSECON 2018 & XDEVCON 2018

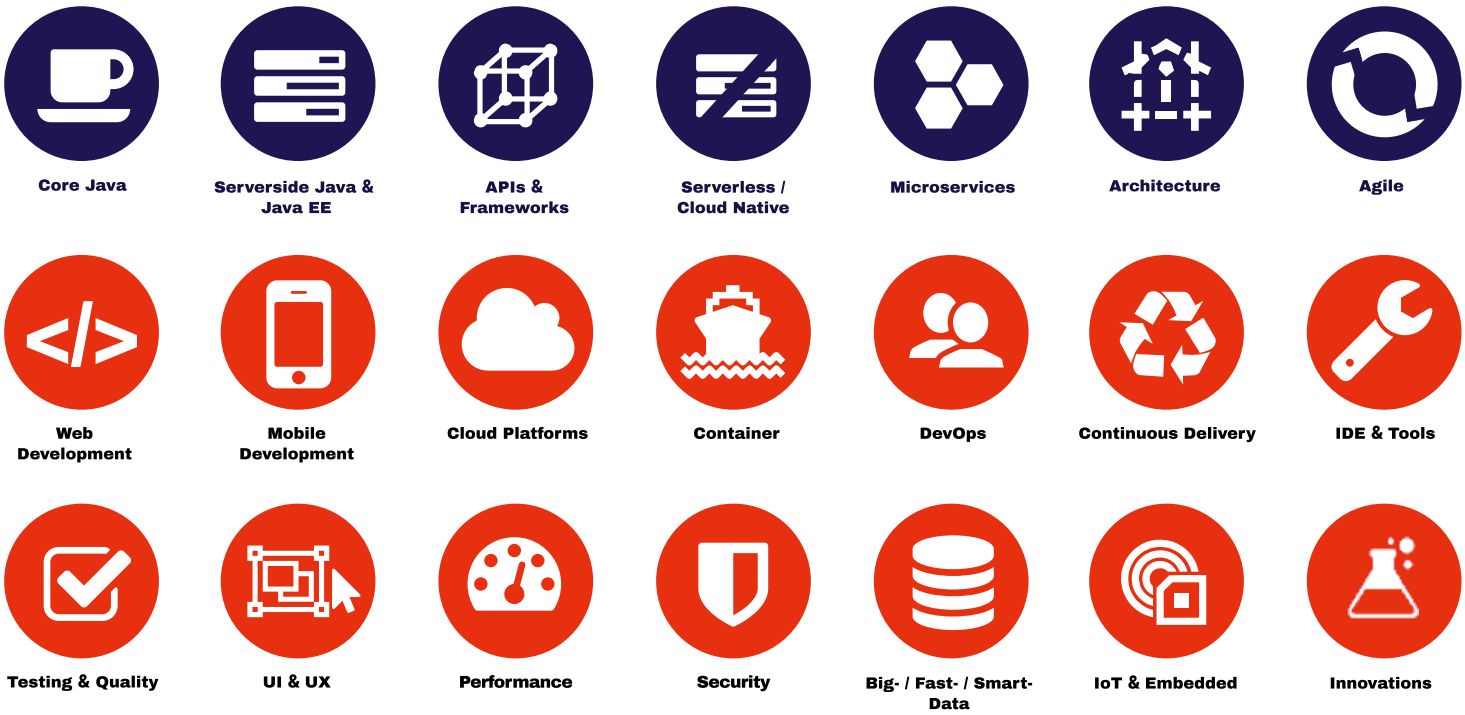
Die EclipseCon Europe 2018 und die XDEVCON 2018 sind unsere Partner-Konferenzen.

Die EclipseCon Europe 2018 findet zwei Wochen nach der JCON, vom 23. bis 25. Oktober in Ludwigsburg statt. Auf der EclipseCon dreht sich alles um Eclipse-Projekte. Hier treffen sich Committer, die an der Entwicklung von Eclipse-Projekten beteiligt sind. Die JCON richtet sich dagegen an die Anwender, die Eclipse-Projekte wie Java EE in ihren eigenen Projekten einsetzen.

Die XDEVCON ist seit nunmehr acht Jahren die Top-Konferenz für Rapid-Cross-Platform-Development und ist zum zweiten Mal Teil der JCON.

THEMEN

TOP-THEMEN 2018



#JCON2018

DIE GROSSE JAVA COMMUNITY KONFERENZ

9. - 12. Oktober 2018 in Düsseldorf

JCON 2018

AGENDA

Dienstag, 9. Oktober	Mittwoch, 10. Oktober	Donnerstag, 11. Oktober	Freitag, 12. Oktober
JCON 2018			
Haupt-Konferenz 8:45 - 19:00 Uhr			**Training Day** 9:00 - 17:00 Uhr
EXPO 8:45 - 19:00 Uhr			**Java 9 und 10 - Was ist neu?**
Special Days			**Microservices und Kubernetes**
Cloud & DevOps Day 8:45 - 19:00 Uhr	**Big-Data Day** 8:45 - 19:00 Uhr	**Architecture Day** 8:45 - 19:00 Uhr	**Reaktive Programmierung**
		Agile Day 8:45 - 19:00 Uhr	**Hibernate Performance-Tuning**
			Entwicklung von Cloud-Native Applications
XDEVCON 2018 - Rapid-Cross-Platform-Development			**Realtime-Applikationen mit Jetstream Java In-Memory**
XDEVCON 2018 8:45 - 19:00 Uhr	**XDEVCON 2018** 8:45 - 19:00 Uhr	**XDEVCON 2018** 8:45 - 19:00 Uhr	**Cross-Platform-Development mit Java & Eclipse**

TICKETS

1-TAGES-PASS	1-TAGES-PASS	2-TAGES-PASS	3-TAGES-PASS	4-TAGES-PASS
TAG 1 / 2 / 3 24. / 25. / 26. Oktober	**TAG 4 27. Oktober**	**TAG 1-2 / 2-3 24.-25. / 25.-26. Oktober**	**TAG 1-3 24.-26. Oktober**	**TAG 1-4 24.-27. Oktober**
Haupt-Konferenz Tag 1: Cloud & DevOps Day Tag 2: Big-Data Day Tag 3: Architecture Day Tag 3: Agile Day XDEVCON 2018	Training Day	Haupt-Konferenz Tag 1: Cloud & DevOps Day Tag 2: Big-Data Day Tag 3: Architecture Day Tag 3: Agile Day XDEVCON 2018	Alle Sessions der JCON 2018 XDEVCON 2018	Alle Sessions der JCON 2018 XDEVCON 2018 Training Day
Normalpreis **249 €**	Normalpreis **599 €**	Normalpreis **449 €**	Normalpreis **499 €**	Normalpreis **1.098 €**
EarlyBird bis 30. Juni 2018 **199 €**	EarlyBird bis 30. Juni 2018 **449 €**	EarlyBird bis 30. Juni 2018 **349 €**	EarlyBird bis 30. Juni 2018 **399 €**	EarlyBird bis 30. Juni 2018 **799 €**
JCON 2017 Teilnehmer Eclipse Member **149 €**	JCON 2017 Teilnehmer Eclipse Member **399 €**	JCON 2017 Teilnehmer Eclipse Member **299 €**	JCON 2017 Teilnehmer Eclipse Member **349 €**	JCON 2017 Teilnehmer Eclipse Member **749 €**

Alle auf dieser Seite aufgeführten Preise sind Nettopreise zzg. 19% MwSt.!

Jetzt anmelden unter: www.jcon.one

EarlyBird bis 30. Juni 2018 bis zu 299 € sparen !

Haben Sie Fragen zur JCON 2018 ? Wir beraten Sie gerne!

Fon: +49 (0)6196 – 204801 – 0 | E-Mail: info@jcon.one

Veranstalter: JAVAPRO - IT Press & Media GmbH

#JAVAPRO #JakartaEE

Java EE ist tot, es lebe Jakarta EE

Oracle hat im September 2017 die Alleinherrschaft über Java Enterprise Edition aufgegeben und an die Eclipse Foundation übertragen, wo Java EE unter dem neuen Namen Jakarta EE weiterentwickelt werden soll. Wir haben mit Mike Milinkovic über den aktuellen Stand und die Zukunft von Java Enterprise gesprochen.

JAVAPRO: Mike, wir freuen uns, dass wir mit Ihnen als Executive Director der Eclipse Foundation über das neue und wohl eines der wichtigsten Eclipse-Projekt sprechen können. Was verbirgt sich hinter Jakarta EE?

Mike Milinkovic: Im September vergangenen Jahres hat der Java Hersteller Oracle die Weiterentwicklung der Java Platform, Enterprise Edition (Java EE) an die Eclipse Foundation übergeben. Java Enterprise wird nun als Eclipse Top-Level-Projekt mit dem Namen Jakarta EE, weiterentwickelt..

JAVAPRO: Warum hat Oracle die Eclipse Foundation ausgewählt? Auch die Apache Foundation wurde als heißer Kandidat gehandelt.

Mike Milinkovic: Es freut uns natürlich sehr, dass Oracle sich für die Eclipse Foundation als neue Heimat für Java Enterprise entschieden hat. Alle großen Unternehmen, die Java aktiv vorantreiben, sind bereits Member der Eclipse Foundation. Neben Oracle selbst sind das vor allem IBM, Red Hat, Tomitribe und Payara. Zudem gibt es mit Eclipse MicroProfile bereits ein Java-Projekt, das insbesondere von diesen Unternehmen stark vorangetrieben wird. Ich denke damit war es naheliegend, Java Enterprise an die Eclipse Foundation zu übergeben.

JAVAPRO: Java Enterprise endlich als Open Source frei zu geben, wurde von der Community schon sehr lange gefordert. Was hat Oracle dazu bewogen, die Kontrolle über die Java Enterprise Edition jetzt tatsächlich aufzugeben und welche Vorteile ergeben sich durch die Übergabe des Projekts an die Eclipse Foundation für die Community?

Mike Milinkovic: Auch für uns kam das etwas überraschend. Der Schritt ist aber nachvollziehbar. Die Weiterentwicklung von Java Enterprise war vielen Anwendern zu träge und nicht offen genug. Lösungsansätze, die sich zwischenzeitlich in der Praxis etabliert und bewährt hatten, wurden zudem aus Sicht der Community zu wenig berücksichtigt. Oracle hat eine gute Entscheidung getroffen, die Weiterentwicklung der Community zu übergeben. Die Unternehmen, die in Java Enterprise investieren, können nun die Inhalte und das Tempo selbst mitbestimmen, was sich sicher positiv auf das Projekt auswirken wird.

Neben Unternehmen kann sich natürlich jeder Einzelne an der Entwicklung beteiligen und ich möchte natürlich alle Nutzer dazu einladen das zu tun.

JAVAPRO: Ist die Übergabe nicht auch ein Risiko? Wird Oracle sich nun von Java Enterprise genauso zurückziehen wie in der Vergangenheit von anderen Projekten, die man Open-Source-Communities übergeben hat, insbesondere OpenOffice.org oder das Continuous-Integration-Tool Hudson. Besonders skeptische Stimmen befürchten sogar das Ende von Java EE.

Mike Milinkovic: Nein, das sehe ich nicht so. Ich kann nicht für Oracle sprechen, aber ich gehe davon aus, dass Oracle auch weiterhin in die Java Enterprise Edition investieren wird und ein großes Interesse am Erfolg von Jakarta EE hat. Zudem erwarten wir nun sehr viel Unterstützung seitens der Community, welche die Ausgliederung ja gefordert hat, und nun erstmals aktiv an der Entwicklung von Jakarta EE mitwirken kann. Das ist ein großer Vorteil und gibt uns die Möglichkeit, Java EE sehr viel schneller als bisher weiterentwickeln zu können. Insbesondere Anbieter von Java EE Application-Server wie IBM, Red Hat, Payara, Tomitribe und SAP haben sicher ein Interesse daran Jakarta EE schneller als Java EE bisher voranzutreiben. Aber auch andere Unternehmen, deren Lösungen auf Java EE basieren und für die der Erfolg von Java Enterprise strategisch wichtig ist, können nun Entwickler abstellen und eine solche Lücke gegebenenfalls schließen. Oracle hat inzwischen aber auch sein Engagement für die Zukunft bekräftigt. Die Community wird also in jedem Fall von der Übergabe profitieren.

JAVAPRO: Warum wird die vertraute Bezeichnung Java Platform, Enterprise Edition, wie Java EE offiziell bislang hieß, überhaupt in Jakarta EE umbenannt?

Mike Milinkovic: Das ist eine rein namensrechtliche Sache. Java ist ein von Oracle geschützter Markenname. Oracle behält sich das alleinige Recht vor, den Markennamen Java in Produktbezeichnungen zu verwenden. Laut Oracle entstünde durch die Verwendung des Namens Java der Eindruck einer Kompatibilität und Evolution durch Oracle, was jedoch nicht mehr der Fall ist. Deshalb mussten wir für Java EE eine neue Bezeichnung finden. Die Community konnte sich bei der Findung des endgültigen Namens beteiligen. Knapp 7.000 Mitglieder haben abgestimmt, über 64 Prozent stimmten für Jakarta EE. Zuvor wurde eine Liste potentieller Namen rechtlich überprüft. Selbstverständlich muss ein hoher Grad an Sicherheit gewährleistet werden, dass der neue Name für eine exponierte Technologie wie Java EE weltweit problemlos verwendet werden darf

JAVAPRO: Was genau beinhaltet das Jakarta EE Projekt? Gibt es Teile von Java Enterprise, die (noch) bei Oracle bleiben?

Mike Milinkovic: Jakarta EE umfasst die gesamte Weiterentwicklung von Java Enterprise, u.a. die Spezifikation, APIs und die Implementierungen dafür, inklusive der Referenzimplementierung GlassFish (jetzt Eclipse GlassFish) sowie auch die Technology-Compatibility-Kits (TCKs) für Java Laufzeitumgebungen.

JAVAPRO: Seit 1998 wird die Programmiersprache Java sowie die Platform, Enterprise Edition im Rahmen des Java-Community-Process (JCP) weiterentwickelt. Es gab viele Befürworter für die Übernahme der Weiterentwicklung von Java EE durch den bereits existierenden JCP.

Mike Milinkovic: Ich sehe darin kein großes Problem. Wir haben jetzt mit Eclipse EE.next Working Group (EE.next) einen neuen Spezifikationsprozess eingeführt, der den Java-Community-Process ersetzen wird. Mit IBM, Red Hat, Tomitribe und natürlich Oracle werden vier Mitglieder des bisherigen Java-Community-Process weiterhin federführend bleiben. Die Project-Leads werden gewählt und wechseln sich regelmäßig ab. Ich kann jedoch garantieren, dass es keine Sonderrechte für Oracle oder andere spezielle Unternehmen geben wird. Jeder kann sich an diesem Prozess beteiligen und ich lade erneut alle interessierten Firmen, Organisationen und Privatpersonen dazu ein, sich aktiv zu beteiligen.

JAVAPRO: Gibt es schon eine Roadmap und Pläne für die ersten neuen Features? Wann wird es die erste Version von Jakarta EE geben? Wird die Versionsnummer von Java Enterprise fortgeführt oder wird es eine Version 1.0?

Mike Milinkovic: Dafür ist es immer noch etwas zu früh. Wir mussten uns zuerst einmal um den Aufbau einer Organisation für das Jakarta-EE-Projekt kümmern. Dann musste der gesamte Java EE Code übertragen und auf GitHub veröffentlicht werden. Sobald die organisatorischen Arbeiten abgeschlossen sind, werden wir zügig die Arbeit an Jakarta EE aufnehmen und wollen möglichst schnell ein Java EE 8 kompatibles Release zur Verfügung stellen. Wir bitten darum, bis dahin noch keine Änderungen an den bereits migrierten APIs in Jakarta-EE-Projekten vorzunehmen. Wer trotzdem bereits an neuen API-Prototypen arbeiten möchte, sollte am besten die GitHub-Repositories forken.

JAVAPRO: Gibt es schon+# Details zur Lizenzierung von Jakarta-EE-Code?

Mike Milinkovic: Alles, was im Rahmen des Jakarta-EE-Projektes entsteht, wird als Open Source unter der für Eclipse-Projekte üblichen EPL-Lizenz Version 2.0 (Eclipse-Public-License) sowie unter der für Java üblichen GPL 2.0 (General-Public-License) mit Classpath-Exception lizenziert.

JAVAPRO: Mike, herzlichen Dank für das Interview und viel Erfolg mit Jakarta EE.

Mike Milinkovic: Vielen Dank und viele Grüße an die Leser der JAVAPRO !

#JAVAPRO #JavaEE #EE4J #Eclipse

Zu neuen Ufern

Die bislang unter dem Namen „Java EE“ bekannte Entwicklungsplattform ist wieder im Aufwind. Nach einer längeren Phase der Ungewissheit, während der sogar das Ende der Plattform befürchtet wurde, erschien im September endlich das neue Release Java EE 8. Die ersten kompatiblen Application Server stehen so langsam in den Startlöchern – höchste Zeit also für einen detaillierten Blick auf die Neuerungen. Ebenfalls im Herbst gab Oracle bekannt, die alleinige Kontrolle über Java EE aufzugeben und an die Eclipse Foundation zu übertragen. Ein Schritt, der allseits für große Zuversicht sorgt, der aber auch viele Änderungen mit sich bringt, unter anderem eine Namensänderung.

Autor:

Thilo Frotscher arbeitet als freiberuflicher Softwarearchitekt und Trainer. Als Experte für Enterprise Java, APIs und Systemintegration unterstützt er seine Kunden überwiegend durch Projektarbeit, Reviews oder Schulungen. Thilo ist (Co-) Autor mehrerer Bücher zu Java EE, (Web) Services und Systemintegration, hat zahlreiche Fachartikel veröffentlicht und spricht regelmäßig auf Fachkonferenzen, auf Schulungsveranstaltungen oder bei Java User Groups.

http://www.frotscher.com
https://twitter.com/thfro
feedback@frotscher.com

Die Fertigstellung einer Java-EE-Spezifikation geht einher mit der Bereitstellung einer Referenz-implementierung in Form eines Application Servers. Im Falle von Java EE 8 handelt es sich dabei um GlassFish 5, der in den Varianten Web-Profile und Full-Platform seit Herbst zum Download bereitsteht und für erste Experimente mit dem neuen Release eingesetzt werden kann. Zusätzlich sind entsprechende Docker-Images verfügbar. Wie gewohnt benötigten die Hersteller der anderen Application-Server im Anschluss einige Zeit, um ihre mit Java EE 8 kompatiblen Releases fertig zu stellen. Inzwischen sind aber große Fortschritte gemacht worden, und die ersten Release Candidates wurden veröffentlicht. Grundsätzlich besteht daneben aber natürlich die Möglichkeit, auch ohne Application-Server einzelne Bestandteile von Java EE 8 einzusetzen. Dies geschieht indem die Implementierungen der jeweiligen APIs einer Anwendung

hinzugefügt werden, die beispielsweise in Tomcat oder Jetty betrieben wird. Um etwa die neuen Features von JAX-RS 2.1 nutzen zu können, muss lediglich Jersey 2.26 Bestandteil der Anwendung sein.

Unterstützung für HTTP/2

Java EE 8 führt keine bahnbrechenden neuen Konzepte ein, beinhaltet jedoch eine ganze Reihe sinnvoller Ergänzungen und Modernisierungen, die bislang teils schmerzlich vermisst wurden. Eine dieser wichtigen Modernisierungen wird von Servlet 4.0 umgesetzt, das nun Unterstützung für HTTP/2 bietet. HTTP/2 wurde 2015 standardisiert und wird mittlerweile von allen gängigen Browsern beherrscht. Hauptziel der neuen Protokollversion sind eine geringere Latenz und eine damit einhergehende Beschleunigung beim Laden von Webseiten. Dies wird durch eine ganze Reihe von Maßnahmen erreicht, wie etwa der Kompression von HTTP Headern oder dem Multiplexing mehrerer Requests über die gleiche TCP-Verbindung. Gleichzeitig ist HTTP/2 vollständig rückwärtskompatibel zu HTTP/1.1: Bekannte Konzepte wie Methoden oder Statuscodes wurden erhalten; Kommunikationspartner wie beispielsweise Browser und Web-Server können sich über ein standardisiertes Verhandlungsprotokoll auf eine Protokollversion verständigen. Da alle diese Veränderungen letztlich die Protokollebene betreffen, sind sie für Anwendungsentwickler überwiegend transparent. Daher fallen die für HTTP/2 relevanten API-Änderungen in Servlet 4.0 sehr gering aus. Die wesentliche Änderung ist ein neuer **`PushBuilder`**, mit dessen Hilfe Anwendungen das neue HTTP/2 Server Push verwenden können. Mittels Server-Push kann ein Server zusätzlich zu einer vom Client angeforderten Ressource noch weitere Ressourcen zurücksenden, und zwar schon bevor der Client diese explizit angefordert hat. So könnten etwa bei einer Anforderung von **`index.html`** neben dieser Seite selbst gleich noch die verknüpften Stylesheets oder Grafiken an den Client gesendet werden. Es wäre somit nur ein einziger Request notwendig, um all diese Ressourcen zu erhalten.

Neues für die Arbeit mit JSON

Auch hinsichtlich der Unterstützung für JSON bietet Java EE 8 einige wichtige Erweiterungen. So unterstützt JSON-P 1.1 nun JSON Pointer (RFC 6901), JSON Patch (RFC 6902) und JSON Merge Patch (RFC 7396). Im Zusammenspiel mit der im ebenfalls aktualisierten JAX-RS 2.1 eingeführten serverseitigen Unterstützung für HTTP PATCH lassen sich PATCH-Operationen für REST APIs nun deutlich einfacher implementieren als bisher.

Neben diesen Basisfunktionalitäten für die Arbeit mit JSON-Formaten stellt insbesondere auch das neue JSON-B 1.0 (JSON-Binding) eine wichtige Ergänzung der Plattform dar. Dabei handelt es sich in etwa um das Äquivalent von JAXB für die JSON-Welt, also einen standardisierten Mechanismus zur automatischen Umwandlung zwischen JSON-Strukturen und Java-Objekten. Hierzu definiert JSON-B eine leicht zu verwendende API, sowie umfangreiche Regeln für ein Default-Mapping, so dass im einfachsten Fall keinerlei Anpassungen an den Modellklassen notwendig sind. Das Default-Mapping umfasst Konventionen für die Umwandlung der gängigsten Datentypen, von Klassen und Collections, aber auch für die Behandlung von NULL-Werten oder die Attributreihenfolge.

Für solche Fälle, in denen das Default-Mapping entweder nicht ausreicht oder schlicht von den Anforderungen einer konkreten Anwendung abweicht, können Entwickler entsprechende Custom-Mappings einsetzen, die beschreiben in welcher Art von den Konventionen des Default-Mappings abzuweichen ist. Dies geschieht typischerweise mithilfe von Annotationen wie **`@Jsonb Property`** (zur Umbenennung von Properties und zur Festlegung der Behandlung von NULL-Werten) oder **`@JsonbPropertyOrder`** (Attributreihenfolge). Diese Annotationen können an Attributen, Properties, Typen und in manchen Fällen sogar Packages angebracht werden. Alternativ dazu können über die JSON-B-API auch so genannte Strategien konfiguriert werden, die ein bestimmtes, vom Default-Mapping abweichendes Verhalten dann global für alle Modellklassen umsetzen. Für besonders schwierige Fälle können zudem mit Hilfe der generischen Schnittstelle **`Jsonb Adapter`** auf einfache Weise auch eigene Umwandlungsregeln implementiert werden. Hierzu sind lediglich die Methoden **`adaptToJson`** und **`adaptFromJson`** zu implementieren. Dies ist beispielsweise dann notwendig, wenn mehrere Attribute eines Objektes auf nur eine JSON-Property abgebildet werden sollen, oder umgekehrt. Die Einführung einer standardisierten API für das JSON-Binding bedeutet für Entwickler im Wesentlichen, dass künftig die Abhängigkeit auf eine proprietäre Lösung, wie etwa das beliebte Jackson Framework, entfallen kann.

Weiterentwicklung von JAX-RS

Das bereits erwähnte JAX-RS 2.1 bietet neben der Unterstützung für http-PATCH und der engen Integration mit JSON-B zur Umwandlung eingehender und ausgehender Nachrichten noch einige weitere spannende Neuigkeiten. Hier sind insbesondere die Unterstützung von Server-Sent-Events (SSE) und eine neue reaktive Client-API zu nennen. Bei SSE handelt es sich um einen Standard, der bereits 2015 vom W3C verabschiedet wurde und eine Alternative zu den verbreiteten Polling-Szenarien bietet. Beim Einsatz von Polling senden Clients periodische Requests an ein Backend, um den aktuellen Status einer Ressource abzufragen. Vor allem in solchen Fällen, in denen sich dieser Status eher selten ändert, bedeutet dies eine erhebliche Last für die betreffenden Systeme und das Netzwerk, die gleichzeitig überwiegend nutzlos ist. Beim Einsatz von SSE können Clients sich dagegen mit einem einzigen Request für Ereignisse eines bestimmten Typs (Event-Stream) registrieren, also beispielsweise für Änderungen an einer bestimmten Ressource. Im Anschluss an diese Registrierung kann das jeweilige Backend dann im Falle einer tatsächlichen Datenänderung ein entsprechendes

Event an den Client senden. Haben sich mehrere Clients für das gleiche Ereignis registriert, kann dessen Versand als Broadcast geschehen. Somit werden nach der initialen Registrierung der Clients nur noch dann Nachrichten über das Netzwerk versendet, wenn tatsächliche Datenänderungen vorliegen. Gegenüber der Polling-Variante ergibt sich somit eine deutliche Reduzierung des Kommunikationsaufkommens. Ein mögliches Einsatzszenario im Zusammenhang mit JAX-RS könnte eine Single-Page-Anwendung (SPA) sein. Jede Instanz dieser Anwendung registriert sich als SSE-Client bei einem JAX-RS Backend. Sobald das Backend beispielsweise einen PUT- oder PATCH-Request empfängt, welcher bestimmte Ressourcen verändert, werden die einzelnen SPA-Clients mittels eines entsprechenden Events darüber informiert. JAX-RS 2.1 bietet hierfür entsprechende API-Erweiterungen für das Backend, die API für eventuelle JavaScript-Clients wird dagegen vom SSE-Standard selbst definiert. Aber auch in Java implementierte SSE-Clients werden von JAX-RS 2.1 unterstützt. Somit sind auch serverseitige Anwendungen denkbar, die sich als SSE-Clients bei anderen Systemen registrieren.

Eine zweite interessante Erweiterung in JAX-RS 2.1 ist die neue reaktive Client-API. Schon bisher war es möglich, REST-Requests asynchron auszuführen, das heißt den Versand des HTTP Requests in einen separaten Thread auszulagern. Dabei hatten Entwickler die Wahl, ob sie zur Behandlung der Response entweder eine von der API zurückgelieferte Instanz von **`Future`** einsetzen wollten, oder ob ein **`InvocationCallback`** übergeben wurde. Beide Ansätze haben jedoch ihre Schwächen. Im Falle von **`Future`** muss das Resultat irgendwann mittels eines blockierenden Aufrufs abgeholt werden. Der Einsatz eines **`InvocationCallback`** führt dagegen zwar zu nicht-blockierendem Code. Dafür ist die Implementierung mehrerer zusammenhängender Requests jedoch aufwendig und führt zu schlecht wartbarem Code, da in diesem Fall mehrfach verschachtelte **`InvocationCallbacks`** zu implementieren sind. In Java SE 8 wurden für genau solche Fälle **`CompletionStage`** und **`CompletableFuture`** eingeführt. Sie erlauben die einfache Implementierung mehrstufiger Abläufe, deren einzelne Stufen asynchron ausgeführt werden, und die jeweils von den Resultaten der vorherigen Stufen abhängen. Der Einsatz von **`CompletionStage`** führt in solchen Fällen zu deutlich besser lesbarem Code, bei dem die implementierte Fachlichkeit klarer erkennbar wird. Die neue reaktive Client-API von JAX-RS 2.1 verwendet nun **`CompletionStage`** als Rückgabewert beim asynchronen Versand von REST-Requests. Zudem wurde die API erweiterbar ausgelegt, so dass JAX-RS Implementierungen optional auch andere reaktive Frameworks wie beispielsweise RxJava unterstützen können.

Asynchrone Events für CDI

Auch die beliebte Komponententechnologie CDI wurde für Java EE 8 überarbeitet. Eine wesentliche Neuigkeit besteht darin, dass CDI 2.0 nun auch Java SE unterstützt. Zu diesem Zweck wurde die Spezifikation in die Abschnitte Core-CDI, CDI in Java SE und CDI in Java EE unterteilt. Der erste Abschnitt definiert dabei die zentralen Konzepte wie Qualifiers, Scopes, Alternatives, Stereotypes oder das Programmiermodell, während die anderen beiden Teile die Details beschreiben, welche für die jeweilige Umgebung relevant sind. Für den Einsatz von CDI in Java SE wurden dabei im Wesentlichen das Bootstrapping des CDI-Containers, Scopes, sowie Paketierung und Deployment definiert.

Eine weitere wichtige Neuerung betrifft die CDI-Events. Diese konnten bislang nur synchron abgefeuert werden, und auch die jeweiligen Observer wurden synchron ausgeführt. Dies konnte zu längeren Blockaden führen, zudem war die Ausführungsreihenfolge der Observer nicht definiert. Und schließlich werden im Falle einer auftretenden Exception in einem der Observer alle eigentlich noch folgenden Observer für das jeweilige Event nicht mehr aufgerufen. Mit CDI 2.0 wurde daher nun die Möglichkeit geschaffen, synchronen Observern mit Hilfe der Annotation **`@Priority`** eine Priorität zuzuweisen, um so eine bestimmte Reihenfolge zu erzwingen. Zusätzlich wurden asynchrone Events eingeführt. Da die Observer asynchroner Events in separaten Threads ausgeführt werden, muss der Erzeuger eines Events nicht mehr auf die Ausführung sämtlicher Observer warten. Zudem haben Fehler in asynchronen Observern keine Auswirkung auf den Aufruf der anderen asynchronen Observer des gleichen Events.

Endlich: Ein Standard für Security

Nicht zuletzt verdient auch die komplett neue Security 1.0 API einen genaueren Blick. Sie zielt auf die Beseitigung des Umstandes, dass Sicherheitsaspekte wie Authentifizierung oder die Überprüfung von Zugangsdaten bislang überwiegend durch die Hersteller der Application-Server zu implementieren waren. Hierdurch entstanden unterschiedliche proprietäre Lösungen, deren Anbindung nicht unerhebliche Aufwände bedeutete, falls im Unternehmen kein einfacher LDAP-Server, sondern beispielsweise eine proprietäre Benutzerverwaltung oder ein Authentication-Store eines Drittanbieters im Einsatz waren. Zwar wurde in Java EE 6 mit JASPIC eine Lösung zur Implementierung und Anbindung eigener Authentifizierungsmodule eingeführt. Diese erwies sich jedoch als recht komplex, war daher recht unbeliebt und wurde entsprechend selten eingesetzt. Zudem gilt sie inzwischen als veraltet.

Security 1.0 definiert nun standardisierte APIs für drei sicherheitsrelevante Grundkonzepte: Authentifizierungs¬mechanismus, Anbindung von Identity-Stores und Security-Context. Somit können diese Aspekte künftig von den Anwendungen selbst implementiert werden, anstelle auf eine Unterstützung und Konfiguration seitens des Application-Servers angewiesen zu sein. Dies bedeutet einen wichtigen Fortschritt für den Lebenszyklus von Anwendungen, was etwa am Beispiel von Cloud-Deployments oder in Docker betriebenen Anwendungen deutlich wird. Für Entwickler, die erst heute in die Welt von Java EE einsteigen,

mag es erstaunlich erscheinen, dass so zentrale Aspekte erst in Java EE 8, und nicht schon vor langer Zeit standardisiert wurden.

Bislang unterstützten Servlet-Container zur Authentifizierung lediglich die Varianten Basic-, Digest-, Form- und Certificate-Authentication. Jenseits von JASPIC bestand für Anwendungsentwickler kaum eine Möglichkeit, in den Authentifizierungsprozess einzugreifen oder weitere Mechanismen zur Authentifizierung zu unterstützen. Die neue Schnittstelle **`HttpAuthenticationMechanism`** ermöglicht es Entwicklern nun künftig beliebige Mechanismen zu implementieren. Die Installation eines solchen Mechanismus erfolgt dabei auf einfache Weise durch die Bereitstellung der jeweiligen Implementierung als CDI-Bean. Auf dieser Basis könnte beispielsweise eine tokenbasierte Authentifizierung mit JSON-Web-Token oder OAuth leicht umgesetzt werden. Spätere Releases der Security-API werden möglicherweise vorschreiben, dass Container selbst Implementierungen für solche populären Mechanismen verpflichtend mitbringen müssen. Security 1.0 schreibt jedoch zunächst lediglich eingebaute Mechanismen für Basic-Authenticaton und formularbasierte Authentifizierung vor.

Auf ganz ähnliche Weise wie im Falle des Authentifizierungsmechanismus führt Security 1.0 mit **`IdentityStore`** auch eine standardisierte Schnittstelle für die Anbindung von Systemen zur Benutzerverwaltung ein. Auch hier gilt, dass Anwendungsentwickler auf Basis der Schnittstelle die Anbindung beliebiger Systeme zur Prüfung von Zugangsdaten implementieren und dem Container als CDI-Bean bereitstellen können.

Als dritten Grundbaustein führt Security 1.0 schließlich auch einen (weiteren) **`SecurityContext`** ein. An dieser Stelle gibt es Überschneidungen mit diversen anderen APIs, wie etwa Servlet, EJB, JAX-RS oder JSF, die bereits über ähnliche Kontexte verfügen. Da nun jedoch eine dedizierte Security-API existiert, wird angestrebt diese bestehenden Kontexte mittelfristig aufzugeben und den neuen **`SecurityContext`** für die gesamte Plattform als zentralen, einheitlichen Zugang zu allen sicherheitsrelevanten Laufzeitinformationen zu nutzen.

Neben den hier besprochenen Features enthält Java EE 8 weitere Neuigkeiten in Bean-Validation 2.0, JPA 2.2, JSF 2.3, Web-Socket 1.1, Common-Annotations 1.3 und Java-Mail 1.6, die im Rahmen dieses Artikels jedoch nicht besprochen werden können.

Java EE wird Jakarta EE

Beinahe zeitgleich mit dem Release von Java EE 8 wurde bekannt, dass Oracle beabsichtigt seine federführende Rolle in der Entwicklung der Plattform aufzugeben und diese an die Eclipse Foundation zu übertragen. Dieser Schritt war zwar über längere Zeit von zahlreichen Vertretern der Community gefordert oder erhofft worden, kam für viele zu diesem Zeitpunkt dann aber doch recht überraschend. In jedem Fall bedeutet die Übertragung an die Eclipse Foundation eine Reihe signifikanter Veränderungen. So sollen unter anderem die bislang nur eingeschränkt zugänglichen TCKs (Test Compatibility Kit) künftig als Open Source zur Verfügung stehen. Dies wird praktisch jedermann erlauben, Implementierungen für die Plattform bereitzustellen und deren Kompatibilität nachzuweisen. Hierdurch erhofft man sich eine stärkere Beteiligung der Community und mehr Wettbewerb unter den Implementierungen. Auch GlassFish, die bisherige Referenzimplementierung von Java EE, ist bereits an die Eclipse Foundation übergeben worden und heißt nun Eclipse Glassfish. Zudem soll es künftig nicht mehr nur eine einzige Referenzimplementierung der Plattform geben, sondern voraussichtlich mehrere. Die künftige Ausrichtung der Plattform, welche neuen Features entwickelt werden, sowie verschiedene organisatorische Einzelheiten werden fortan nicht mehr ausschließlich durch einzelne Hersteller bestimmt oder vereinbart. Stattdessen bekommt die Community ein gewichtiges Mitspracherecht. Insgesamt wird somit die ganze Plattform viel offener und ermöglicht deutlich mehr Beteiligung der Community. Ein neuer Prozess hierfür wird gerade innerhalb der Eclipse Foundation beraten. Eine Working Group namens "EE.next" wird den "Java Community Process for Java EE" ersetzen.

Während all dies offensichtlich zahlreiche Chancen bietet, ergeben sich aus diesen Veränderungen jedoch auch Risiken. So hat Oracle zwar offiziell angekündigt, sich auch weiterhin an der Entwicklung der Plattform zu beteiligen, gleichzeitig ist jedoch damit zu rechnen, dass das Unternehmen den Umfang seines Engagements reduzieren wird. Hierdurch wird es notwendig werden, dass andere Hersteller, oder eben die Community, diese Lücke füllen. In diesem Zusammenhang bleibt abzuwarten, wie stark sich die Community hier tatsächlich einbringen wird. Unter anderem besteht die Hoffnung, dass Großunternehmen mit umfangreichen Entwicklerteams, die Java EE als Basis für zahlreiche Systeme einsetzen, einzelne Mitarbeiter abstellen, um die Weiterentwicklung der Plattform voranzutreiben.

Eine recht kontroverse Debatte verursachte die Namensgebung. Aufgrund der bei Oracle liegenden Namensrechte konnte der bisherige Name „Java EE" nicht weitergeführt werden. Nach einigen Diskussionen über einen möglichen neuen Namen für die Plattform, und nachdem in der Community bereits zahlreiche Vorschläge gesammelt worden waren, wurde schließlich - erneut recht überraschend - angekündigt, dass die Plattform künftig unter einem neuen Top-Level-Project namens „Eclipse Enterprise for Java" (EE4J) beheimatet sein wird. Dieser neue Name sorgte für sehr gemischte Reaktionen, insbesondere weil von vielen befürchtet wurde, EE4J würde auch der neue offizielle Name der Plattform werden. Dies ist jedoch nicht der Fall. Denn inzwischen wurde, dieses Mal unter Beteiligung der Community, ein neuer „Brand Name" gefunden: Die Plattform wird künftig Jakarta EE heißen.

Die Problematik der Namensrechte betrifft auch die bislang in Java EE verwendeten Package-Namen, die mit javax beginnen.

So ist bislang noch unklar, ob diese in EE4J bestehen bleiben können. Sollte dies nicht der Fall sein, müssten alle APIs in ein neues Package überführt werden, was die Rückwärtskompatibilität der Plattform vollständig zerstören würde. Dabei war gerade diese Rückwärtskompatibilität eine bislang ganz besonders gepflegte Eigenschaft von Java EE, die von vielen Unternehmen als eine gewisse Sicherheit oder Bestandsschutz empfunden wurde. Es ist daher zu hoffen, dass in diesem Punkt eine Einigkeit zwischen Oracle und der Eclipse Foundation erzielt werden kann, so dass die bisherigen Package-Namen weitergeführt werden können, möglicherweise durch eine Lizensierung.

Eine interessante Entwicklung stellt derweil die Ernennung von Ivar Grimstad zum Leiter des Project Management Committees (PMC) dar. Grimstad ist gleichzeitig Specification-Lead von MVC 1.0 (JSR-371), das ursprünglich Bestandteil von Java EE 8 sein sollte, dann jedoch kurzfristig von Oracle fallen gelassen wurde. Da MVC 1.0 ebenfalls zur Eclipse Foundation transferiert wurde, liegt somit nahe, dass es zukünftig in Jakarta EE integriert, und somit doch noch fester Bestandteil der Plattform werden wird.

MicroProfile als Innovationsmotor

Aktuell sind also noch zahlreiche organisatorische Fragen zu klären, so dass es mit der tatsächlichen Weiterentwicklung und einem ersten Release von Jakarta EE sicherlich noch eine Weile dauern wird. Gleichzeit entwickelt sich jedoch die MicroProfile-Initiative zu einem Innovationsmotor für die Plattform. Die Initiative entstand ursprünglich im Spätsommer 2016 aus der Situation heraus, dass allgemein große Unsicherheit über die Zukunft und Weiterentwicklung von Java EE herrschten, da Oracle die Community über einen längeren Zeitraum im Unklaren über seine weiteren Pläne gelassen hatte. Gleichzeitig meldeten sich viele Stimmen, die forderten die Plattform müsse bessere Unterstützung für aktuelle Trends, wie Microservices oder Cloud anbieten. Als Reaktion hierauf vereinbarten einige Hersteller wie Red Hat, IBM, Payara und Tomitribe die Plattform gemeinsam, jedoch zunächst außerhalb des offiziellen Standardisierungsprozesses voranzutreiben, Ideen für die Weiterentwicklung zu sammeln, entsprechende Implementierungen kurzfristig bereitzustellen, und eine Standardisierung erst anschließend anzustreben. Erster Ansatzpunkt war dabei der Vorschlag eines neuen Java-EE-Profiles, das speziell für die Entwicklung von Microservices geeignet sein sollte und lediglich die Technologien JAX-RS, CDI und JSON-P umfasst. Für dieses Profile wurden unter dem Namen „MicroProfile 1.0" eine ganze Reihe unterschiedlicher Implementierungen bereitgestellt, unter anderem Wildfly Swarn, Payara Micro, TomEE und Open Liberty, die Basis des WebSphere Liberty Application-Servers.

In der Zwischenzeit hat sich auch die MicroProfile Initiative der Eclipse Foundation angeschlossen, was eine enge Zusammenarbeit mit Jakarta EE ermöglicht. Zudem wurden in rascher Abfolge weitere Releases des MicroProfile vorgestellt und eine ambitionierte Roadmap veröffentlicht. Das im Herbst 2017 veröffentlichte MicroProfile 1.2 beinhaltet APIs für lange geforderte Erweiterungen, wie einen standardisierten Zugang zu Konfigurationsinformationen aus unterschiedlichen Quellen, sowie Unterstützung für Fault-Tolerance, Health-Checks, Metriken oder JSON Web-Tokens. Bereits Anfang 2018 folgte schon MicroProfile 1.3, das zusätzlich Unterstützung für Open-API (früher bekannt unter dem Namen Swagger) und Open-Tracing beinhaltet. Die Community hat nun die Möglichkeit, die vorgestellten Implementierungen des MicroProfile zu testen und entsprechendes Feedback zu geben. Stellen sich die vorgeschlagenen Erweiterungen als hilfreich heraus, könnten diese möglicherweise in eines der ersten Releases von Jakarta EE einfließen.

Fazit

Unterm Strich ergibt sich eine zwiespältige Bewertung von Java EE 8. Einerseits zeigen sich manche Entwickler enttäuscht, die sich nach einer Wartezeit von immerhin vier Jahren (seit dem Release von Java EE 7) mehr Innovation erhofft hatten. Dies gilt insbesondere bezüglich einer besseren Unterstützung von Cloud-Szenarien und Microservices. Andererseits bringt Java EE 8 durchaus einige wichtige neue Features und Modernisierungen mit. An erster Stelle sind hier sicherlich die Unterstützung von HTTP/2, JSON Pointer, JSON Patch, HTTP Patch oder Server-Sent-Events zu nennen. Auch die Einführung einheitlicher APIs zur Umsetzung von Sicherheitsmechanismen war überfällig und ist sehr zu begrüßen. Nicht zuletzt wurden zahlreiche APIs auf sprachlicher Ebene modernisiert, indem sie nun Sprachfeatures unterstützen, die in Java SE 8 eingeführt wurden. Zudem ist positiv zu bewerten, dass nach der im Jahr 2016 herrschenden Verunsicherung bezüglich der Zukunftsperspektive der Plattform mit Java EE 8 nun letztlich doch ein weiteres Release zustande gekommen ist, welches zudem einen ausgezeichneten Startpunkt für die Weiterentwicklung durch die Eclipse Foundation darstellt.

Auf genau diese Weiterentwicklung werden Entwickler jedoch noch ein wenig warten müssen. Denn bevor ein erstes Release von Jakarta EE zu erwarten ist, müssen noch einige organisatorische und rechtliche Dinge geklärt werden. Zudem ist zu erwarten, dass Jakarta EE 1.0 mehr oder weniger identisch mit Java EE 8 sein wird. Ziel dieses Releases dürfte es sein, den vollständigen und erfolgreichen Übergang zur Eclipse Foundation nachzuweisen. Gleichzeitig arbeitet jedoch die MicroProfile Initiative in bisher ungekanntem Tempo an der Entwicklung möglicher neuer Features. Es wird daher nun ganz besonders auch auf die Community ankommen, hierzu Feedback zu geben und das über lange Zeit teils heftig geforderte Mitspracherecht nun auch tatsächlich zu nutzen, um Einfluss auf die Ausrichtung der kommenden Releases zu nehmen. Auf diese Weise könnte die bewährte, auf breiter Basis eingesetzt Java EE Plattform unter neuem Namen - und in rascheren Zyklen als bisher - an die Anforderungen zeitgemäßer Enterprise-Anwendungen angepasst werden, und somit einer rosigen Zukunft entgegensehen.

#JAVAPRO #JavaEE #JPA

Was gibt's Neues in JPA 2.2

Für die beliebte Persistenzspezifikation gab es im Rahmen von Java EE 8 bislang erst ein Maintenance-Release. Aber auch darin verbergen sich ein paar interessante Features, auf die viele Entwickler bereits gewartet haben.

Java EE 8 brachte einige Neuerungen in die Java-Enterprise-Welt. Für die Java-Persistence-API -Spezifikation (JSR 338) blieb allerdings nicht viel Zeit übrig und es reichte nur für ein kleines Maintenance-Release. Darin wurde die Spezifikation vor allem an Java 8 angepasst. Dies war erforderlich, da die Java-Persistence-API (JPA) 2.1 bereits vor der Veröffentlichung von Java 8 released wurde und somit die neuen Datentypen und Programmierkonzepte nicht unterstützt.

Mit JPA 2.2 können nun einige Klassen des Date-and-Time-API verwendet und Abfrageergebnisse als **`Stream`** verarbeitet werden. Einige Annotationen wurden als **`@Repeatable`** deklariert, sodass die Verwendung von Container-Annotationen nun nicht mehr erforderlich ist. Außerdem wird nun CDI (Injection in AttributeConverter) unterstützt, wodurch u.a. Konvertierungsalgorithmen leichter wiederverwendet werden können.

Die meisten Anwendungsentwickler haben auf die Anpassungen an Java 8 bereits seit einiger Zeit gewartet und diese werden voraussichtlich auch die größten Auswirkungen auf bestehende und zukünftige Anwendungen haben.

Wiederholbare Annotationen

Bis zur Einführung von **`@Repeatable`** Annotationen in Java 8 konnte jede Klasse, Methode und Eigenschaft zwar mit verschiedenen Annotationen, jedoch mit jeder Annotation nur einmal annotiert werden.

In allen JPA-basierten Anwendungen gibt es allerdings zahlreiche Fälle in denen dies nicht ausreicht. Ein typisches Beispiel ist die **`@NamedQuery`** Annotation, mit der Datenbankabfragen definiert werden können. Für die meisten Entitäten werden mehrere Abfragen definiert, sodass auch mehrere **`@NamedQuery`** Annotationen benötigt werden.

Um die in älteren Java-Versionen bestehende Einschränkung zu umgehen, definiert die JPA-Spezifikation eine Reihe von Container-Annotationen, wie die bekannte **`@NamedQueries`** Annotation. Die einzige Aufgabe dieser Annotationen ist es, ein Array anderer Annotationen als Parameter zu verwalten und somit das mehrfache Annotieren mit derselben Annotation zu ermöglichen.

(Listing 1)

```
@NamedQueries({
  @NamedQuery(name = MyEntity.findByUser, query = "SELECT e FROM
MyEntity e WHERE e.user = :user"),
  @NamedQuery(name = MyEntity.findByPublishingDate, query = "SELECT e
FROM MyEntity e WHERE e.date = :date")})
public class MyEntity { … }
```

Seit Java 8 besteht diese Einschränkung nicht mehr. Als **`@Repeatable`** gekennzeichnete Annotationen können nun mehrfach für dieselbe Klasse, Methode oder Eigenschaft verwendet werden. Der Compiler fügt dabei automatisch die umschließende Container-Annotation hinzu. Container-Annotationen werden also weiterhin benötigt. Allerdings müssen sie nur noch bei der Implementierung der Annotation deklariert und nicht mehr bei der Anwendungsentwicklung verwendet werden.

Autor:

Thorben Janssen ist freiberuflicher Trainer und Autor des Buchs Hibernate Tips - "More than 70 solutions to common Hibernate problems". Er entwickelt seit mehr als 15 Jahren Anwendungen auf Basis von Java EE und ist Mitglied der CDI 2.0 Expert Group. Auf seinem Blog www.thoughts-on-java.org schreibt er mehrmals wöchentlich über JPA und Hibernate.

https://www.thoughts-on-java.org
Twitter @thjanssen123 – Xing https://www.xing.com/profile/Thorben_Janssen – LinkedIn https://www.linkedin.com/in/thorbenjanssen – Github https://github.com/thjanssen
Email thorben@thoughts-on-java.org

Mit JPA 2.2 wurden nun alle Annotationen, für die es in der Spezifikation eine Container-Annotationen gibt, als **@Repeatable** gekennzeichnet. Dies sind:

- **javax.persistence.AssociationOverride**,
- **javax.persistence.AttributeOverride**,
- **javax.persistence.JoinColumn**,
- **javax.persistence.MapKeyJoinColumn**,
- **javax.persistence.NamedEntityGraph**,
- **javax.persistence.NamedNativeQuery**,
- **javax.persistence.NamedQuery**,
- **javax.persistence.NamedStoredProcedureQuery**,
- **javax.persistence.PersistenceContext**,
- **javax.persistence.PersistenceUnit**,
- **javax.persistence.PrimaryKeyJoinColumn**,
- **javax.persistence.SecondaryTable**,
- **javax.persistence.SqlResultSetMapping**,
- **javax.persistence.SequenceGenerator**,
- **javax.persistence.TableGenerator**.

Somit können die genannten Annotationen nun ohne Container-Annotationen verwendet werden.

(Listing 2)

```
@NamedQuery(name = MyEntity.findByUser, query = "SELECT e FROM
MyEntity e WHERE e.user = :user")
@NamedQuery(name = MyEntity.findByPublishingDate, query = "SELECT
e FROM MyEntity e WHERE e.date = :date")
public class MyEntity { … }
```

Abfrageergebnisse als Stream verarbeiten

Mit JPA wurde das **Query** Interface um die **getResultStream** Methode erweitert. Die neue Methode stellt das Ergebnis einer Abfrage als Stream zur Verfügung.

Die Methode mag auf den ersten Blick überflüssig erscheinen, da dies über einen kleinen Umweg auch bereits mit JPA 2.1 möglich war **(Listing 3)**. Es musste lediglich das Abfrageergebnis mit Hilfe der **getResultList** Methode als **List** abgerufen werden. Das **List** Interface bietet mit der **stream** Methode eine Möglichkeit, einen Stream auf Basis der Liste zu erstellen. Anschließend kann dann mit der Stream-API durch die Datensätze des Abfrageergebnisses iteriert und diese verarbeitet werden.

(Listing 3)

```
TypedQuery<MyEntity> q = em.createQuery("SELECT e FROM MyEntity
e", MyEntity.class);
Stream<MyEntity> s = q.getResultList().stream();
```

Dieser Ansatz hat bei größeren Ergebnismengen jedoch einen Nachteil. Beim Aufruf der **getResultList** Methode müssen alle Datensätze des Abfrageergebnisses gelesen und in die entsprechende Datenstruktur überführt werden. In **(Listing 3)** bedeutet dies, dass zuerst alle Datensätze der durch die **MyEntity** abgebildeten Datenbanktabelle ausgelesen und anschließend in managed Entitäten überführt werden müssen. Im Anschluss werden die Entitäten dann als Stream zur Verfügung gestellt und einzeln verarbeitet.

Bei größeren Datenmengen ist es häufig allerdings sinnvoller durch die Ergebnismenge zu iterieren und nicht alle Datensätze auf einmal zu laden. Dadurch können bereits verarbeitete und somit nicht mehr benötigte Datensätze vom aktuellen Persistenzkontext detacht und dem Garbage-Collector übergeben werden. Das reduziert den Speicherbedarf der Anwendung und den internen Verwaltungsaufwand der durch den Persistenzkontext verwalteten Entitäten.

Die JDBC-API bietet bereits die Möglichkeit durch das **ResultSet** zu iterieren. Mit der neuen **getResultStream -Methode** haben JPA-Implementierungen nun die Möglichkeit eine für die Stream-Verarbeitung optimierte Methode zu implementieren und dieses JDBC-Feature zu nutzen.

(Listing 4)

```
TypedQuery<MyEntity> q = em.createQuery("SELECT e FROM MyEntity
e", MyEntity.class);
Stream<MyEntity> s = q.getResultStream();
```

Das bedeutet allerdings nicht, dass alle JPA-Implementierungen eine auf die Stream-Verarbeitung optimierte Implementierung der **getResultStream** Methode bieten müssen. Die Spezifikation stellt eine Default-Implementierung der Methode bereit, die den in **(Listing 3)** gezeigten Ansatz implementiert. Wenn diese nicht überschrieben wird, bietet die **getResultStream** Methode lediglich einen einfacheren Zugriff auf den **Stream**. Es hängt daher von der jeweiligen JPA-Implementierung ab, ob die neue **getResultStream** Methode einen echten Mehrwert oder lediglich eine Vereinfachung der API bietet.

Unabhängig von der Implementierung der **getResultStream** Methode sollte außerdem bedacht werden, dass die Datenbankabfrage weiterhin die effizienteste Möglichkeit zur Auswahl und Transformation von Datensätzen bietet. Auch wenn mit Hilfe der Stream-API die Sortierung des Streams geändert, Elemente gefiltert und die Verarbeitung vorzeitig beendet werden kann, sollte dies nur in Ausnahmefällen genutzt werden. Datenbanken sind für diese Anwendungsfälle optimiert und erzielen eine deutlich bessere Performanz als ein selbstentwickelter Java-Algorithmus.

Klassen der Date and Time API persistieren

Seit der Einführung des Date-and-Time-API werden in vielen Projekten **LocalDate** und **LocalDateTime** anstelle von **java.util.Date** verwendet. Mit JPA 2.1 wurden diese Klassen allerdings noch nicht als Typ für Entitätseigenschaft unterstützt. Entwickler mussten sich daher entscheiden, ob sie:

- auf die Verwendung dieser Klassen in Entitäten verzichten oder
- die proprietäre Funktionalität einer JPA-Implementierung oder eines anderen Frameworks wie Spring Data einsetzen, oder
- eine eigene Datentypkonvertierung mit Hilfe eines Attribute-Converter bereitstellen wollen.

Ab JPA 2.2 ist diese Entscheidung nicht mehr notwendig. Die Spezifikation unterstützt die folgenden Klassen des Date-and-Time-API als Basistypen:

Java Type	Beschreibung JDBC Type
java.time.LocalDate	DATE
java.time.LocalTime	TIME
java.time.LocalDateTime	TIMESTAMP
java.time.OffsetTime	TIME_WITH_TIMEZONE
java.time.OffsetDateTime	TIMESTAMP_WITH_TIME ZONE

Eigenschaften dieser Typen können nun ohne zusätzliche Annotationen auf Datenbankspalten abgebildet werden **(Listing 5)**. Da die Date-and-Time-API unterschiedliche Klassen zur Speicherung von Datum und Datum mit Uhrzeit bietet, wird auch die früher häufig verwendete **`@Temporal`** Annotation nicht mehr benötigt. Die neuen Klassen liefern dem Persistenzprovider bereits alle benötigten Informationen.

Einige JPA-Implementierung, z.B. Hibernate, können auch anderen Klassen des Date-and-Time-API abbilden . Diese werden in nachfolgenden JPA-Versionen hoffentlich auch durch die Spezifikation unterstützt werden.

(Listing 5)

```
@Entity
public class MyEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;

    private LocalDate localDate;

    private LocalDateTime localDateTime;

    public LocalDate getLocalDate() {
        return localDate;
      }

    public void setLocalDate(LocalDate localDate) {
        this.localDate = localDate;
      }

    public LocalDateTime getLocalDateTime() {
        return localDateTime;
      }

    public void setLocalDateTime(LocalDateTime localDateTime) {
        this.localDateTime = localDateTime;
      }

    public Long getId() {
        return id;
      }

}
```

Die Verwendung von Eigenschaften der unterstützten Typen der Date-and-Time-API unterscheiden sich nicht von der Verwendung anderen Eigenschaften.

(Listing 6)

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

MyEntity e = new MyEntity();
e.setLocalDate(LocalDate.now());
e.setLocalDateTime(LocalDateTime.now());
em.persist(e);

em.getTransaction().commit();
em.close();
```

Fazit

Wie von einem Maintenance-Release zu erwarten, bringt JPA 2.2 nur eine sehr geringe Anzahl an Änderungen. Diese dienen vor allem zur Unterstützung der mit Java 8 eingeführten Datentypen und Programmierkonzepten. Da die vorherige Version der Spezifikation allerdings schon sehr ausgereift war und für die meisten Anwendungsfälle gute Lösungsmöglichkeiten bot, adressiert die neue Version trotzdem die häufigsten Fragestellungen vieler Anwendungsentwickler.

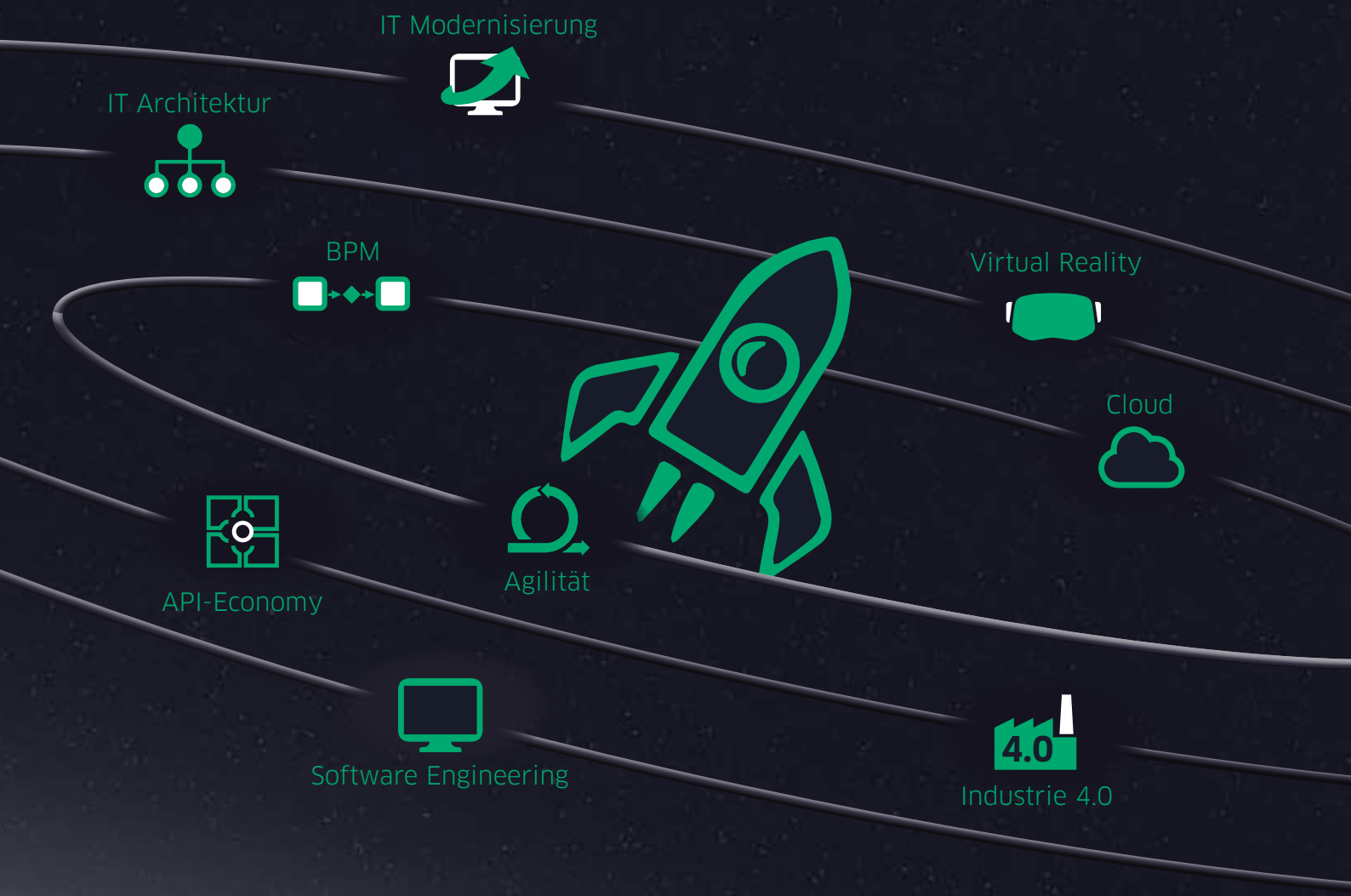
Aktuell wird die JPA-Spezifikation, wie auch alle anderen Java-EE-Spezifikationen, an die Eclipse Foundation übergeben. Es bleibt abzuwarten, in welche Richtung und mit welcher Geschwindigkeit die Entwicklung danach weitergeht. Der JPA-Bugtracker und die sehr aktive Diskussion über eine asynchrone JDBC-API dürften ausreichend Anregungen für weitere Verbesserungen liefern. Und mit der MicroProfile-Initiative gibt es in der Eclipse Foundation auch bereits ein positives Beispiel, wie die Community ein Projekt ohne Oracles Schirmherrschaft erfolgreich weiterentwickeln kann.

Links:

1 Java Persistence 2.2, JSR 338: https://goo.gl/xfKuHx
2 How to persist LocalDate and LocalDateTime with JPA: https://goo.gl/299LPM
3 How to persist LocalDateTime & Co with Hibernate: https://goo.gl/nuM5d9

ACCSO

ACCELERATED SOLUTIONS



Mit Accso spannende Themen vorantreiben und zu neuen Sphären aufbrechen

Werde Accsonaut!

www.accso.de

#JAVAPRO #Microservices

Microservices – The Dark Side

Microservices erfreuen sich als Architekturparadigma großer Beliebtheit. Dabei werden zwei wichtige Tatsachen weitgehend ausgeblendet: Erstens verursachen Microservices sowohl bei der Entwicklung als auch im Betrieb zusätzliche Kosten, die gerechtfertigt werden müssen. Zweitens sind viele der angestrebten Vorteile allein durch saubere Modularisierung erreichbar. Aus der Diskussion dieser Aspekte wird eine Liste von Fragen entwickelt, die sich jeder vor der Entscheidung für eine auf Microservice basierende Anwendung ehrlich beantworten sollte.

Jede Medaille hat bekanntlich zwei Seiten: eine glänzende Vorderseite und die im Schatten liegende Rückseite. Als Metapher steht die Medaille dabei für eigentlich alle Dinge im Leben. Doch so trivial und bekannt das auch ist, es wird immer wieder übersehen oder ignoriert. Das verhält sich im Bereich der IT-Technologien nicht anders. Die Konzentration auf versprochene positive Effekte und das Verdrängen möglicher Nachteile ist infolge des Wettlaufs um Innovationen eher noch ausgeprägter. Und nicht zu vergessen: Es macht natürlich auch Spaß, Neues auszuprobieren und auf alte Fragestellungen anzuwenden.

Autor:

Dr. Jürgen Lampe ist Principal Consultant bei der Agon Solutions GmbH in Frankfurt. Seit rund zwanzig Jahren berät er Kunden, vorrangig aus dem Bankenbereich, bei der Konzeption und Implementation von Java-Anwendungen. Sein besonderes Interesse gilt dem Erreichen hoher Performance durch gute Strukturierung und die Auswahl geeigneter Algorithmen. Zu diesen Themen veröffentlicht er regelmäßig Fachartikel.

Ziel dieses Artikels ist es, einen Beitrag zu einer verantwortungsvollen Bewertung und Einordnung zu leisten. Zur Verdeutlichung dieser Zielsetzung sei ein kleiner Exkurs erlaubt. Per Zufall entdeckt, gibt es im Internet eine Liste von Fragen, die sich jeder Interessent stellen sollte, bevor ein Hund angeschafft wird.[1] Ganz sicher geht es dabei nicht darum, Hunde als Haustiere abzulehnen, aber es gibt einfach eine Reihe von Voraussetzungen, die erfüllt sein müssen, um vorhersehbaren nachfolgenden Problemen vorzubeugen. Natürlich verhindert so eine Checkliste nicht, dass Hunde in ungeeignete Verhältnisse kommen, aber sie gibt verantwortungsvollen Menschen nützliche Hinweise.

In der IT, wo die Probleme nicht im Tierheim akkumuliert, sondern eher mit einem Achselzucken hingenommen werden, sind fundierte Entscheidungshilfen für die Auswahl bestimmter Technologien und dabei insbesondere Ausschlusskriterien leider selten. Kostenüberschreitungen oder gar Fehlschläge sind so häufig, dass nur in Ausnahmefällen eine genaue Ursachenanalyse erfolgt.

Es wäre vermessen, hier eine Liste von Kriterien vorzulegen. Vielmehr sollen einige Gedanken und Erfahrungen präsentiert werden, die als Materialsammlung für die Extraktion von Kriterien dienen können. Der Schwerpunkt auf den Aspekten, die sich aus der praktischen Arbeit in konkreten Projekten ergeben haben.

Zur Einführung noch eine allgemeine Bemerkung: Microservices sind ja nicht der erste Ansatz zur Strukturierung verteilter Anwendungen. Der eine oder andere erinnert sich vielleicht noch an die Common-Object-Request-Broker-Architecture (CORBA). In einigen Punkten hat der Umgang mit Microservices bemerkenswerte Parallelen. Konzepte wie den Service-Locator gab es bereits damals. Auch die gängige Praxis, Services in YAML mit Hilfe von Swagger[2] zu beschreiben und gegebenenfalls daraus automatisch Client- und Service-Stubs (in unterschiedlichen Programmiersprachen) generieren zu lassen, entspricht letztlich der Rolle der IDL (Interface-Definition-Language) von CORBA. Nun geht es nicht darum, in ein „Alles-schon-mal-da-gewesen" zu verfallen. Stattdessen lautet die Frage: Was können wir aus den Erfahrungen lernen? Leider gibt es keine fertige Antwort darauf. Sobald eine Technologie aus dem Fokus gerät, findet sich viel zu selten jemand, der die gewonnenen Erkenntnisse sammelt und aufbereitet. Die zwangsläufige Folge ist, dass bestimmte Fehler wieder und wieder gemacht werden. In Bezug auf CORBA lässt sich sagen, dass es ganz sicher nicht nur an der Schwergewichtigkeit und der Vorherrschaft eines großen Anbieters gelegen hat. Ein wichtiges Problem bestand beispielsweise in den Kosten, die jede Schnittstellenänderung verursachte.[3] Kann man wirklich hoffen, dass Microservices diese Falle – auf Dauer – vermeiden können?

Begrifflichkeit

In diesem Beitrag werden die Begriffe Microservice-Architektur und monolithische Architektur zur Gegenüberstellung verwendet. Letztere ist dadurch gekennzeichnet, dass in der Regel die Anwendung als ein Artefakt betrachtet und in Betrieb genommen wird. Das schließt ausdrücklich nicht aus, dass auch eine monolithische Anwendung wartungsfreundlich modular aufgebaut sein kann. Der Fokus dieser Betrachtungen liegt auf den Konsequenzen, die sich aus den für Microservice-Architekturen typischen unabhängigen Deployments und der Kommunikation über Netzwerkprotokolle ergeben.

Microservices verursachen zusätzliche Kosten, und zwar sowohl in der Entwicklung als auch zur Laufzeit. Diese Tatsache wird gern übersehen. Deshalb werden die entstehenden Aufwände im Folgenden genauer betrachtet.

Laufzeitkosten

Als Laufzeitkosten werden all die Kosten zusammengefasst, die während der produktiven Phase einer Anwendung einzig und allein aus dem Architekturparadigma entstehen. Hier beschränken wir uns auf den Anteil, der unmittelbar bei der Code-Ausführung selbst entsteht. Mittelbare Kosten, die sich beispielsweise aus dem Betrieb einer verteilten Anwendung etwa für Zertifikatsverwaltung, Netzwerkkonfiguration usw. ergeben, bleiben unberücksichtigt. Das heißt jedoch ausdrücklich nicht, dass dieser Kostenblock vernachlässigbar wäre.

Der Aufruf eines Microservices ist um einige Größenordnungen teurer als ein Methodenaufruf. Die Daten müssen

1. konvertiert und in ein textuelles Standardformat gebracht werden (JSON, XML),
2. über eine zu etablierende Netzwerkverbindung an den Empfänger übertragen werden und
3. schließlich beim Empfänger geparst und in ein passendes Datenformat zurück übertragen werden.

Der erste und der letzte Schritt erfordern umfangreiche Zeichenketten-Operationen, die nicht nur in Java als ressourcenintensiv bekannt sind. Bei einfachen Funktionen kann dieser Overhead spürbar ins Gewicht fallen.

Dass Netzwerkverbindungen aufwendig sind, ist ebenfalls nicht neu. Wie aufwendig, hängt u. a. vom verwendeten Protokoll (z. B. HTTP oder HTTPS) ab. Durch die zusätzlich benötigten Aktivitäten zur Absicherung gegen unbefugte Zugriffe steigen die Kosten weiter.

Sowohl der Rechenaufwand als auch die Netzwerkoperationen sollten im Hinblick auf Cloud-Computing, für das viele Microservice-Anwendungen gebaut werden, nicht unterschätzt werden. Wenn, wie häufig üblich, die Abrechnung auf der Basis von abgerufener Prozessorleistung und genutztem Übertragungsvolumen erfolgt, hat der beschriebene Overhead unter Umständen auch messbare finanzielle Konsequenzen.
Außer bei großen verteilten Anwendungen ist die wirtschaftliche

Bedeutung der zusätzlichen Laufzeitkosten im Vergleich zum folgenden Kostenblock jedoch weniger wichtig. Man sollte aber nie vergessen, dass es diese Laufzeitkosten gibt.

Entwicklungskosten

Zusätzliche Kosten in der Entwicklung entstehen an verschiedenen Stellen und sind nicht immer klar abgrenzbar. Beispielsweise können Aufwände, die sich aus der konsequenten Modularisierung ergeben, auch bei einer monolithischen Lösung anfallen. Diese Modularisierungskosten finden ganz allgemein viel zu wenig Beachtung, wie in einer Analyse derartiger Projekte gezeigt werden konnte.[4] Darüber hinausgehende Kosten können sich daraus ergeben, dass die Modularisierungsanforderungen für Microservices tendenziell höher oder fein granularer sind.

Die Implementierung hochgradig verteilter Anwendungen ist technisch kein Problem. Auf den ersten Blick erscheint es sogar relativ einfach und schnell möglich zu sein, einen Microservice zu erzeugen. Dank frei verfügbarer Bibliotheken und CDI (Contexts and Dependency-Injection) ist das tatsächlich so. Die nicht unerheblichen Kosten entstehen vorrangig in Bereichen, die zunächst weniger Aufmerksamkeit finden.

Sicherheitsaufwand

Jeder Microservice ist ein potentielles Einfallstor für Schadprogramme. Kommunikation ausschließlich über gesicherte Verbindungen, z. B. HTTPS, als ein erster nützlicher Schutz ist jedoch nicht gratis zu realisieren. Überdies hat die Praxis gezeigt, dass bisweilen ganze Netzwerkstrukturen infiltriert werden konnten oder Protokolle Implementierungsfehler enthielten, sodass es leichtfertig wäre, sich nur auf diesen Schutz zu verlassen. Deshalb müssen alle empfangenen Daten umfassend validiert werden. Insbesondere auf Injektion bösartiger Daten beruhende Angriffsvektoren sind dabei ins Auge zu fassen. In manchen Fällen kann es sinnvoll oder sogar zwingend erforderlich sein, Daten zusätzlich verschlüsselt oder zumindest signiert zu übertragen.

Intern sollte jeder Service durch einen Wrapper geschützt sein, der sicherstellt, dass wirklich nur die geforderten Funktionen von außen erreichbar sind. In der entgegengesetzten Richtung, beim Aufruf von Services, sollte auf diese nur über einen Adapter zugegriffen werden, der die externe Funktion kapselt und verhindert, dass irrtümlich Informationen gesendet werden, die ein Angreifer nutzen könnte.[5] Das alles ist zusätzlicher Code, der geschrieben, dokumentiert und gewartet werden muss und der sehr wahrscheinlich zusätzliche Fehler enthalten kann, die gefunden und beseitigt werden müssen.

Testaufwand

Nur ganz selten findet man Hinweise darauf, dass Microservices einen erheblich höheren Aufwand beim Testen verursachen. Das gilt für alle Ebenen, angefangen vom Entwicklertest bis hin zum Abnahmetest.

Einerseits hat das die eher triviale Ursache, dass im Hinblick auf die angestrebten unabhängigen Deployments eine sehr hohe Testabdeckung erreicht werden muss. Schließlich dürfen auch momentan noch gar nicht verwendete Facetten nicht vernachlässigt werden, da sie möglicherweise in Zukunft von einem Service-Kunden benutzt werden. Testfälle wachsen dabei in die Rolle von ausführbaren Verträgen zwischen Service-Anbieter und –Kunden hinein. Nur Funktionen, die durch Testfälle garantiert sind, können sicher verwendet werden. Das stellt nicht nur an die Definition der Tests hohe Anforderungen, immerhin soll der Service möglichst vollständig repräsentiert werden, sondern auch an die verständliche Formulierung (Implementation und Dokumentation).

Andererseits müssen Services auch über ihre Netzwerkschnittstellen getestet werden, z. B. die REST-Schnittstellen. Zum Testen gibt es zwar recht gute Werkzeuge, beispielsweise Wiremock[6], aber der Aufwand für die Erstellung der Testfälle ist trotzdem deutlich größer als bei einfachen Unit-Tests. Außerdem sind die Test-Frameworks selbst komplex und nicht immer einfach zu verstehen. Es gibt viele Möglichkeiten Fehler zu machen und da die interne Funktionsweise der Mocks verborgen bleibt, kommt es bisweilen zu schwer ergründbaren unerwünschten Effekten, z. B. gegenseitige Beeinflussung und sporadische Fehlschläge. Erfahrungsgemäß erfordert das Erstellen eines Microservice-Testfalls häufig ein Vielfaches des Aufwands, den der Unit-Test der zugrundeliegenden Funktion verursacht. Dass solche Tests dann auch deutlich länger laufen, fällt dagegen eher weniger ins Gewicht.

Nutzen

Letztlich ist es das Ziel, wie bei allen wirtschaftlichen Aktivitäten, aus dem Einsatz von Microservice-Architekturen einen finanziellen Nutzen zu ziehen. Einige der häufig vorgebrachten Vorteile werden nun diesbezüglich genauer betrachtet.

Flexibilität und bessere Wartbarkeit

Es wird allgemein angenommen, dass Microservices einzeln verteilt und entwickelt werden können. Dadurch könnten Teams weitgehend unabhängig voneinander arbeiten, was die Skalierung agiler Entwicklungsprozesse mit wenig Kommunikations- und Koordinationsaufwand ermöglichen soll. Darüber hinaus wird erwartet, dass Microservices, da sie klein sind, übersichtlich bleiben und leicht weiterentwickelbar, bzw. durch Neuimplementierungen ersetzbar sind.

Genau genommen sind das jedoch Vorteile, die größtenteils bereits durch konsequente Modularisierung ohne die beschriebenen Mehrkosten erreicht werden können. Der Zusatzaufwand,

der dadurch entsteht, dass beim Deployment ein größeres Artefakt (der Monolith) gebaut und verteilt werden muss, fällt erst bei sehr großen Projekten ins Gewicht.

Für den Testaufwand lässt sich das nicht so einfach sagen. In der Zielvorstellung mit häufigen automatischen Service-Updates werden alle Tests automatisch als Teil der Deployment-Pipeline ausgeführt. Das setzt nicht nur den erwähnten hohen Testabdeckungsgrad voraus, sondern auch ein entsprechendes Geschäftsumfeld, indem die in einem solchen Szenario nie ausschließbaren Fehlfunktionen beherrschbar und tolerierbar sind. In so einem Fall sind echte Einsparungen möglich. Wenn die erforderliche Elastizität gegenüber Fehlern aufgrund geschäftspolitischer, gesetzlicher oder regulatorischer Vorgaben nicht gegeben ist, können die entsprechenden Potentiale hingegen nicht realisiert werden. Dann muss, wie bei einem Monolithen und ohne Kostenersparnis, weiterhin immer die gesamte Anwendung vor der Freigabe überprüft werden.

Skalierbarkeit

Microservices können unabhängig voneinander skaliert werden. Welcher Nutzen lässt sich daraus gewinnen? Die Antwort hängt ganz von der Anwendung ab. Wenn die Geschäftsvorfälle so strukturiert sind, dass sie zu stark variierender Frequenz bei der Nutzung der verschiedenen Services führen können, ist die unabhängige Skalierbarkeit zweifellos ein Vorteil. Das gilt umso mehr, wenn durch die Beschränkung auf jeweils einige wenige Services neue Processing-Knoten bei volatiler Last schneller hochgefahren werden können.

Wenn die Last allerdings weniger stark schwankt und wenn vor allem die Nutzungsfrequenzen der diversen Services in relativ festen Verhältnissen zueinanderstehen, ergeben sich keine nennenswerten Vorteile gegenüber einer kompakteren Anwendung, vorausgesetzt letztere ist ebenfalls so ausgelegt, dass sie den Erfordernissen entsprechend durch Starten weiterer Knoten skaliert werden kann. Einsparungen sind vor allem dann möglich, wenn nur relativ kurze aber hohe Lastspitzen abgedeckt werden müssen.

Fehlertoleranz

Verteilte Systeme können gut gegen den Ausfall einzelner Services abgesichert werden, sodass das Gesamtsystem robust ist. Allerdings ist auch dieser Vorteil nicht ausschließlich durch Microservices realisierbar. Im Gegenteil, es ist eher umgekehrt: Weil massiv verteilte Systeme viel häufiger von Fehlern bedroht sind, z. B. Verbindungsabbrüchen, müssen sie von vornherein eine höhere Resilienz aufweisen. Diese Fehlertoleranz muss natürlich organisiert und verwaltet werden. Auf IT-Seite entstehen dabei eher Mehrkosten. Der etwaige Nutzen ergibt sich vorrangig auf der geschäftlichen Seite aus höherer Verfügbarkeit und der daraus hoffentlich resultierenden größeren Kundenzufriedenheit.

Leistungsfähigkeit

Dieser ganz wichtige Aspekt spielt in der allgemeinen Diskussion eine viel zu kleine Rolle. Viele der großen weltweit agierenden Systeme sind in einer anderen Architektur gar nicht vorstellbar. Für diese Anwendungen ist aber auch typisch, dass sie stark aus der IT-Architektur heraus entwickelt worden sind und weiterentwickelt werden.

Obwohl Anwendungen dieser Kategorie nicht die Regel sind, lässt sich aus ihnen eine wichtige Erkenntnis gewinnen. Auf die Dauer sind derartige Architekturen nur profitabel, wenn die Geschäftsvorfälle zur technischen Struktur passen. Das bedeutet einmal, dass Geschäftsmodelle, die sich nicht entsprechend aufgliedern lassen, besser nicht so abgebildet werden sollten. Zum anderen heißt das, dass die Weiterentwicklung der Geschäftstätigkeit auf die technischen Gegebenheiten Rücksicht nehmen muss, um Modifikationen der Schnittstellen zwischen den Service-Einheiten zu vermeiden, weil die besonders teuer umzusetzen sind.

Kosten-Nutzen-Vergleich

Alles hat seinen Preis. Diese einfache Wahrheit wird angesichts begeisternder neuer Technologien gern übersehen. Die Kunst besteht darin, für die jeweilige Situation einen akzeptablen Maximalbetrag auszumachen und dann zu prüfen, ob die Lösung innerhalb des so gegebenen Kostenrahmens möglich ist.

In den vorangegangenen Abschnitten wurden die Kosten und der Nutzen von Microservice-Architekturen betrachtet. Für eine abschließende Bewertung müssen beide Größen gegenüber gestellt werden. Dabei steht man vor dem Dilemma, dass unterschiedliche Qualitäten zu vergleichen sind:

- Die aufgeführten Kosten entstehen in jedem Fall und sind – mit allen Unsicherheiten – relativ gut quantifizierbar.
- Der Nutzen entsteht hingegen fast ausschließlich aus weichen Faktoren, z. B. Time-to-Market, höhere Verfügbarkeit, bessere Skalierbarkeit, die sich schwieriger in Zahlen fassen lassen und überdies noch stark von ungewissen Entwicklungen in der Zukunft abhängen.

Trotzdem ist es möglich sinnvolle Schlussfolgerungen zu ziehen. Der erste wichtige Punkt besteht darin, sich der zusätzlichen Kosten überhaupt einmal bewusst zu werden. Danach kann man versuchen, anhand der dargestellten Überlegungen, den Umfang des Mehraufwands sowohl für die Entwicklung als auch den Betrieb abzuschätzen. Es wäre nicht überraschend, wenn bei manchem Projekt bereits an dieser Stelle klar würde, dass es wirtschaftlich nicht sinnvoll ist.

Falls die erwarteten Mehraufwände vertretbar sind, sollte im nächsten Schritt der potentielle Nutzen ermittelt werden. Zweckmäßigerweise werden dazu mindestens zwei Szenarien

betrachtet: ein optimales und eines für den wahrscheinlichsten oder ungünstigsten Fall. Auch dabei können wieder die oben beschriebenen Aspekte zur Anwendung kommen. Wenn der Nutzen selbst im optimalen Fall die Kosten nicht rechtfertigt, ist die Entscheidung klar. Andernfalls ist es die ureigenste Aufgabe der Geschäftsführung, auf der Basis der aufbereiteten Zahlen über das weitere Vorgehen zu entscheiden.

Fragenliste

Zum Abschluss sollen die aufgeworfenen Probleme in einer Liste von Fragen zusammengefasst werden, die sich jeder vorlegen sollte, bevor eine Entscheidung für oder gegen eine Microservice-Architektur erfolgt.

Ausgangspunkt ist eine ganz generelle Frage, die dabei helfen soll zu erkennen, ob es wirklich essentielle Gründe für die Wahl der Technologie gibt, oder ob man nicht vielleicht doch nur einem aktuellen Trend folgt:

- Welche konkreten Anforderungen lassen sich nur mit der Microservice-Architektur und nicht mit einem sauber modularisierten Monolithen erreichen?

Microservices bringen in gewissem Sinne die Silostruktur zurück.

- Lassen sich die Geschäftsprozesse in klar abgegrenzte Stränge/Geschäftsvorfälle zerlegen, die isoliert bearbeitet werden können?
- Änderungen dieser Zerlegungsstruktur sind im Allgemeinen sehr kostspielig. Ist die gewählte Struktur flexibel genug, um den wahrscheinlichen Änderungen des geschäftlichen Umfelds ohne substantielle Umbauten der Service-Struktur entsprechen zu können?
- Sind die Beziehungen zwischen diesen Strängen hinreichend exakt und vollständig definiert, sodass langwierige und konfliktträchtige Abstimmungsrunden wahrscheinlich nicht nötig sein werden?
- Viele Vorteile erschließen sich erst auf längere Sicht. Haben die einzelnen Stränge hinreichendes Entwicklungspotential, um über einen absehbaren Zeitraum eigenständig neue Features liefern zu können?

Abhängigkeiten können auch entstehen, wenn Entwicklerteams mehr als einen Service entwickeln oder einzelne Personen in mehrere Teams integriert sind.

- Sind ausreichende Entwicklungskapazitäten vorhanden, um wirklich parallel – und damit schneller - entwickeln zu können?
- Der Aufbau und das Zusammenfinden agiler Teams kosten Zeit und Geld. Können die eingespielten Teams dann auch über einen wirtschaftlich sinnvollen Zeitraum hinweg ausgelastet werden?

Der unmittelbare Nutzen, der sich aus der Microservice-Architektur ergibt, ist betragsmäßig in der Regel überschaubar. Erst über Skaleneffekte lassen sich daraus ansehnliche Gewinne generieren.

- Kann realistischerweise mit einem so starken Wachstum gerechnet werden oder ist der Bedarf bereits so groß, dass sich die in die Entwicklung investierten Mehraufwände in überschaubarer Zeit amortisieren?
- Schwanken die Leistungsanforderungen tatsächlich so stark, dass der organisatorische Aufwand für das bedarfsgesteuerte Zu- und Abschalten von Rechnerinstanzen durch eingesparte Nutzungsentgelte gerechtfertigt werden kann?

Eine besonders tückische Kostenfalle lauert in den Verwaltungsfunktionen. Häufig beschränkt man sich bei Pilotanwendungen auf die kostengünstig bereitgestellten Paketlösungen. Bei der Realisierung des echten Projekts stellt sich dann heraus, dass teure Anpassungen zwingend erforderlich sind.

- Sind beispielsweise die von Cloud-Anbietern standardmäßig bereitgestellten Verwaltungsfunktionen wirklich ausreichend?
- Rechtfertigt der Umfang der voraussichtlichen Nutzung die gegebenenfalls anfallenden Anpassungsaufwände?

Letzten Endes läuft alles auf die einfache Frage hinaus: Welcher Nutzen bleibt unter dem Strich übrig?

Links:

1 Landestierschutzbeauftragte Hessen. Fragen an Tierfreundinnen/Tierfreunde, die sich einen Hund anschaffen wollen: goo.gl/1puv3S
Tierschutz Hessen. [Online] [Zitat vom: 16. Oktober 2017.] https://tierschutz.hessen.de/heimtiere.

2 Smartbear. Swagger – The World's Most Popular API Tooling. [Online] [Zitat vom: 27. Oktober 2017.] https://swagger.io/

3 Turnquist, Greg. REST, SOAP, and CORBA, i.e. how we got here: https://goo.gl/7CEsKm [Online] [Zitat vom: 27. Oktober 2017.] http://greglturnquist.com/2016/05/rest-soap-corba-e-got.

4 Knodel, Jens, Naab, Matthias und Weitzel, Balthasar. Modularity – Often Desired, but Rarely Achieved. Softwaretechnik-Trends. 2015, Bd. 35, 2, S. 37–38

5 Anton, Katy. The Path of Secure Software. [Online] 2017. Nov 2017. [Zitat vom: 16. Nov 2017.] https://www.youtube.com/watch?v=IjKIK1GWKKY.

6 WireMock: http://wiremock.org [Online] [Zitat vom: 16. November 2017.]

#JAVAPRO #SpringRactorCore #ReaktiveProgrammierung

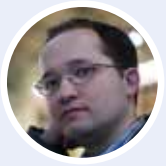
Reaktive Programmierung - Teil 1

Reaktive Architekturen werden immer beliebter. Beim Einzug in das Big-Data- und Cloud-Zeitalter entfalten sie ihre Stärke. Neue syntaktische Elemente von JDK8 wie Lambda und Streams erhöhen die Akzeptanz und glätten die Lernkurve aus. Der 1. Teil der 3-teiligen Serie bietet eine praktische Einführung in das reaktive Framework Spring Reactor Core 3.

Die formale Definition aus Wikipedia beschreibt reaktive Programmierung als ein Programmierparadigma, welches sich an Datenflüssen orientiert. Darunter fällt eine Reihe von Implementierungen und Konzepten. Es sind sowohl verteilte Architekturen, wie JMS (Java Message Service) oder SOA (Service-Oriented-Architecture), als auch Einsätze innerhalb einer Anwendung, wie z.B. die Berechnung von Formeln einer Excel-Tabelle.

Autor:

Vadym Kazulkin ist Chief-Software-Architekt bei ip.labs GmbH, einer 100% Tochter der FUJIFLM Gruppe mit Sitz in Bonn. Ip.labs ist das weltweit führende White-Label-E-Commerce-Unternehmen im Bereich Softwareanwendungen für den digitalen Fotoservice, mit einem modernen und motivierenden Arbeitsumfeld, das technologisch immer einen Schritt voraus ist. Vadyms Schwerpunkte sind derzeit die AWS-Cloud und Konzeption und Implementierung der hochskalierbaren und hochverfügbaren Anwendungen.



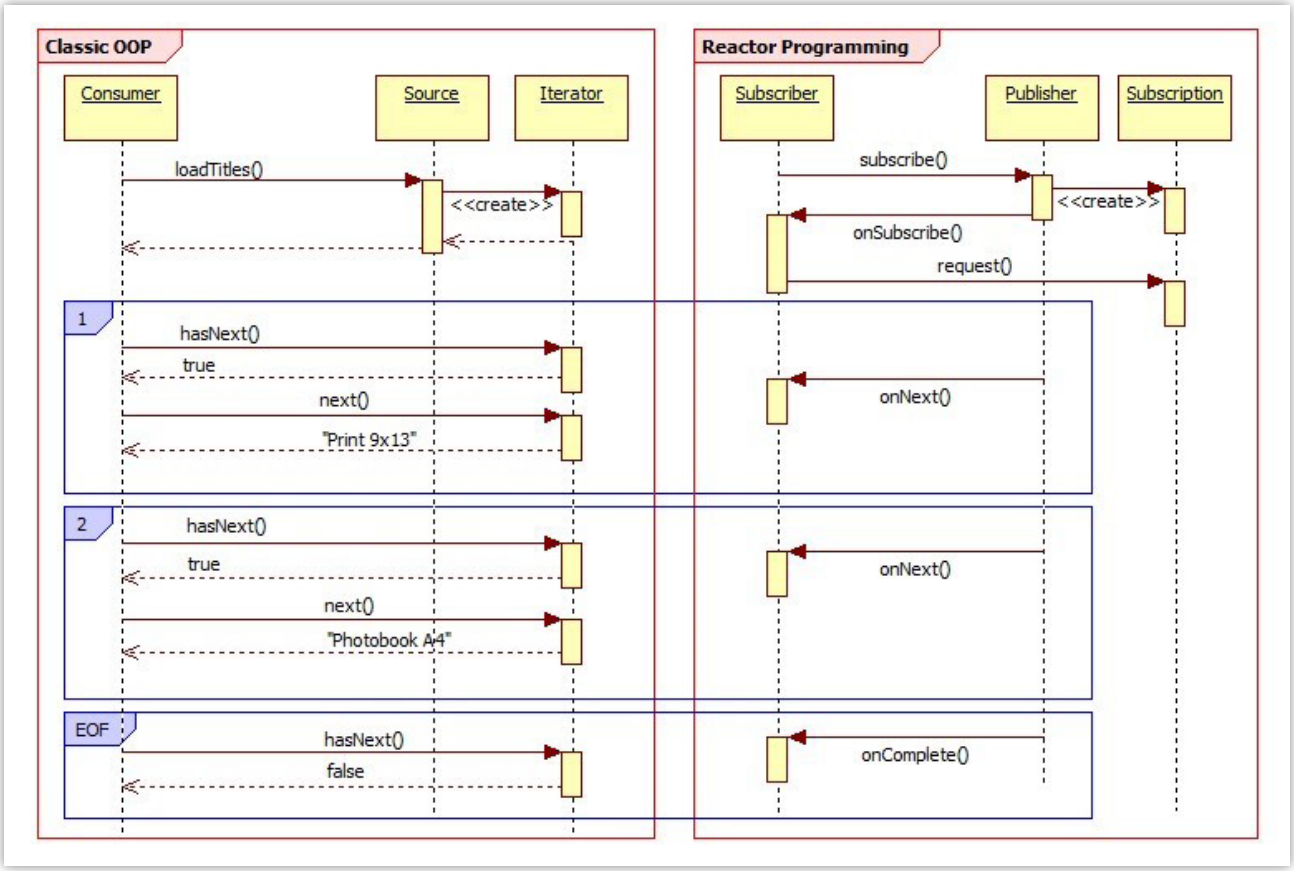
Rodion Alukhanov ist Senior-Software-Entwickler bei ip.labs GmbH mit Sitz in Bonn. Seine mehrjährige Erfahrung streckt sich von MS-DOS über die Hausautomation bis in die AWS Cloud. Seine Schwerpunkte sind Cloud Migration, Big Data und Integration Tests.

Alexander Peters ist Senior-Software-Entwickler bei ip.labs GmbH mit Sitz in Bonn. Er hat mehr als zehn Jahre Berufserfahrung in der webbasierten Softwareentwicklung mit Java(EE) u. Spring Framework. Seine Schwerpunkte liegen bei der serverseitigen Anwendungsentwicklung sowie bei der Definition und Implementierung von externen u. internen Schnittstellen

Das Gemeinsame an den Konzepten ist, dass der Softwareentwickler nur eine eingeschränkte Kontrolle über die ausführenden Threads oder Prozesse hat. Der Programmcode besteht daher aus zustandslosen Funktionen, die miteinander über Data-Streams verbunden sind.

Dieser Artikel beleuchtet praktische Aspekte der aktuellen Implementierung des Standards Reactive-Streams[1] am Beispiel von Spring Reactor 3. Der reaktive Einsatz bietet zahlreiche Vorteile: unter anderem bessere Hardwareausnutzung, Performance und Wartbarkeit des Codes. Hier muss man unbedingt anmerken, dass der reaktive Einsatz keine allgemein bessere Vorgehensweise ist. Es ist lediglich eine andere Art von der Gestaltung des Codes und der Programmlogik. Je nach Use-Case und zu entwickelnder Logik kann der reaktive Einsatz sowohl deutliche Vorteile, als auch große Nachteile mit sich ziehen.

Die allgemeine Vorgehensweise ist am besten durch ein UML-Sequenzdiagramm abbildbar. In **(Abb. 1)** sind zwei Einsätze nebeneinander dargestellt. In beiden Diagrammen werden zwei Produkttitel gelesen (z.B. aus der Datenbank). Während es im klassischen Einsatz darum geht, eine Methode (in dem Beispiel **`loadTitles`**) zu implementieren, schreibt man in dem reaktiven



Java Iterator vs. Reactor (Abb. 1)

Code einen Publisher. Zum Konsumieren von Elementen benötigt man einen Subscriber.

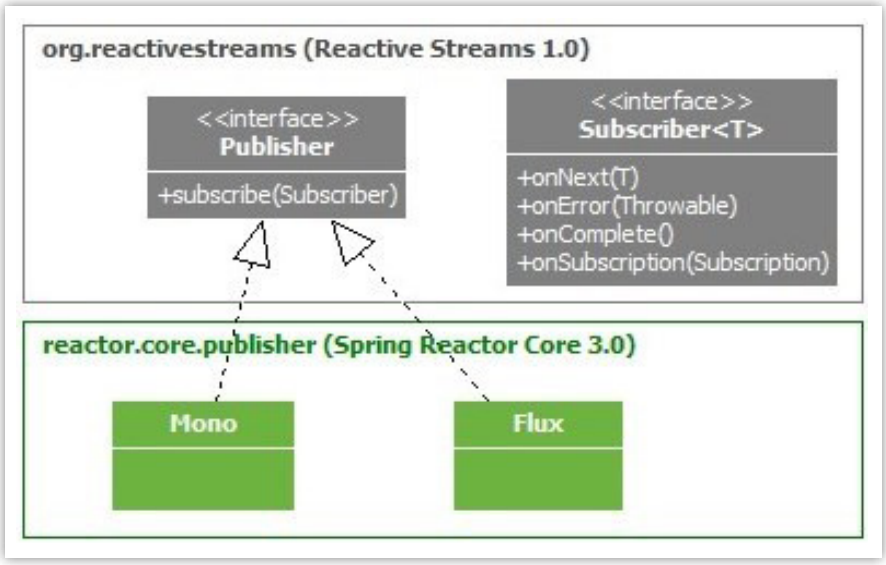
Für beide Klassen bietet der Reactive-Streams-Standard[2] die gleichnamigen Interfaces mit an. Die genannten Interfaces werden selten direkt implementiert. Spring Reactor bietet dafür eine Reihe von Implementierungen und Hilfsklassen. Die wichtigsten davon sind **`Flux`** und **`Mono`** (Abb. 2).

In dem folgenden Code ist ein einfacher Publisher dargestellt. Er holt die Daten aus einer klassischen, nicht reaktiven Schnittstelle (z.B. JDBC) und leitet sie an einen Subscriber weiter. So ein Publisher bildet einen typischen Baustein einer reaktiven Datenverarbeitung.

(Listing 1)

```
int[] ids = {1, 2, 3, 4, 5};

Flux<String> titles = Flux.create(s -> {
    try {
        for (String title :
        dao.loadTitles(ids)) {
            s.next(title);
        }
        s.complete();
    } catch (SQLException e) {
        s.error(e);
    }
});
```

Oft geht es auch einfacher. Ein Publisher kann von einer Collection oder einem Array erzeugt werden. Dazu kommen ein paar Sonderfälle, wie ein leerer Publisher oder ein Publisher, der immer einen Fehler weitergibt.



Flux und Mono (Abb. 2)

(Listing 2)

```
Flux<String> titles =
    Flux.just("Print 9x13", "Book A4", "Calendar A4");
Flux<String> noElements = Flux.empty();
Flux<String> alwaysError =
    Flux.error(new IOException("Host not found"));
```

Hier und im weiteren Verlauf wird eine Untermenge von Publishers dargestellt. Es sind sogenannte Cold-Publishers. Sie erzeugen die gleiche Kette von Elementen für jeden Subscriber und beenden die Erzeugung mit einem Aufruf der Methode **onSuccess** oder **onError**. Ein solcher Publisher erzeugt, genauso wie eine Methode oder eine Funktion, durch die Definition noch keine Verarbeitung. Zum Zug kommt der Code nur, wenn ein Subscriber sich den Publisher abonniert.

Die andere Gruppe sind die Hot-Publishers. Es sind meistens Elemente einer verteilten Architektur, z.B eines JMS-Systems. Sie generieren ununterbrochen Elemente im Hintergrund und speisen diese in Subscribers ein, sobald sich welche bei dem Publisher melden. Eine Beschreibung der Hot-Publishers würde allerdings den Rahmen dieses Artikels sprengen.

Analog geht es bei der Verwertung von erzeugten Streams durch einen Publisher. Eine direkte Lösung wäre der Subscriber in dem folgenden Listing.

(Listing 3)

```
imagePublisher.subscribe(new Subscriber<Image>() {

    public void onSubscribe(Subscription subsc) {
        subsc.request(Long.MAX_VALUE);
    }
    public void onComplete() {
        printerService.printSeparator();
    }
    public void onError(Throwable error) {
        printerService.printErrorPage(error);
    }
    public void onNext(Image image) {
        printerService.print(image);
    }
});
```

Auch beim Erzeugen eines Subscribers ist eine direkte Implementierung des gleichnamigen Interfaces nicht notwendig. Zahlreiche statische Hilfsmethoden der Spring-Reactor-Bibliothek stehen dem Entwickler zur Verfügung.

(Listing 4)

```
imagePublisher.subscribe(
    picture->printerService.print(picture),
    error->printerService.printErrorPage(error));
```

Eine Besonderheit des Spring-Frameworks ist ein spezieller Publisher vom Typ **Mono**. Es ist ein Publisher, der höchstens ein Element generieren kann. Während ein Flux einem JDK8-Stream ähnlich ist, wäre ein **Mono** eine Alternative zu **Optional**. Die Mono-Klasse bietet erwartungsgemäß keine zusätzliche Funktionalität, sorgt aber durch einen anderen Satz von Hilfsmethoden für mehr Übersicht. Der folgende Code lädt zum Beispiel genau ein Bild aus dem Netz herunter.

(Listing 5)

```
Mono<Integer> mid = Mono.just(imageId);
CompletableFuture<Image> loadingFuture =
    mid.map(id->loadFromInstagram(id)).toFuture();
legacyService.processPicture(loadingFuture);
```

Die Stärke des reaktiven Konzepts entfaltet sich bei der Nutzung von komplexeren Ausführungsplänen. Zusätzliche Kontrolle der Ausführung durch die Reactor-Bibliothek ermöglicht eine deklarative Multithread-Ausführung. Der Code in **(Listing 6)** erlaubt das parallele Laden von beliebig vielen Bildern. Der gleiche Code wäre in der klassischen OOP bei weitem nicht so übersichtlich gewesen. Möglichkeiten einer parallelen Ausführung werden im weiteren Verlauf der Artikelserie ausführlicher beleuchtet.

(Listing 6)

```
Scheduler scheduler = Schedulers.elastic();
Flux<Integer> ids = Flux.just(1, 2, 3, 4, 5);
Function<Integer, Mono<Image>> loadPicture =
    id->Mono.create(s -> {
        s.success(loadFromInstagram(id));
    });
Flux<Picture> pictures =
    ids.flatMap(loadPicture).subscribeOn(scheduler);
```

Fehlerbehandlung

Die Fehlerbehandlung im reaktiven Code ist nicht so gut strukturiert, wie in klassischem Java-Code. Es gibt zwei Möglichkeiten eine Ausnahmesituation zu melden: eine Exception zu werfen, oder eine Exception-Klasse einer **error** Methode zu übergeben.

(Listing 7)

```
Mono<String> publisher =  Mono.create(s -> {
    if (somethingWrong1) {
        throw new RuntimeException("Problem 1");
    }
    if (somethingWrong2) {
        s.error(new Exception("Problem 2"));
    } else {
        s.success("No Problems");
    }
});
```

Bei der Fehlerbehandlung muss man sicherstellen, dass die Regeln der Reactive-Streams-Specification nicht verletzt werden. Es ist z.B. nicht zulässig, nach einem Error einen Success zu

melden. In den meisten Fällen wird die Spring-Reactor-Bibliothek eine Verletzung der Spezifikation unterbinden. Je nach Situation kann es zur Exception oder zum Ignorieren eines Events führen. Besonders letzteres kann zu schwer zu findenden Unregelmäßigkeiten bei der Ausführung führen.

Bei der Ausnahmebehandlung ist die reaktive Programmierung ein gewisser Rückschritt gegenüber dem klassischen Java-Modell. Es besteht keine Möglichkeit, Exceptions durch separate Catch-Blöcke zu behandeln. Die Compiler-Kontrolle über die Behandlung von Checked-Exceptions geht hier leider ebenfalls verloren. Es erfordert mehr Aufmerksamkeit vom Entwickler: man muss dafür sorgen, dass die nach der Businesslogik vorgesehenen Ausnahmefälle richtig behandelt werden.

Logging

Bei einer komplexeren Verkettung von Publishern ist es oft notwendig, den Datenfluss zu verfolgen. In einem reaktiven Code erweist sich dies oft als umständlich. Insbesondere trifft es zu, wenn kurze Lambda-Ausdrücke und Hilfsfunktionen von Spring benutzt werden. Diese Arbeit kann durch einen loggenden Zwischen-Publisher erleichtert werden, was mithilfe der Log-Methode erfolgt.

(Listing 8)

```
Flux<String> titles =
    Flux.just("Photobook", "Calendar").log();
Flux<String> format = titles.flatMap(
    t->Flux.just(t + " A5", t + " A4")).log();
format.subscribe(System.err::println);

--- AUSGABE (verkürzt) ---
[INFO] (main) | onSubscribe(FluxArray)
[INFO] (main) onSubscribe(FluxFlatMap)
[INFO] (main) | onNext(Photobook)
[INFO] (main) onNext(Photobook A5)
[INFO] (main) onNext(Photobook A4)
[INFO] (main) | onNext(Calendar)
[INFO] (main) onNext(Calendar A5)
[INFO] (main) onNext(Calendar A4)
[INFO] (main) | onComplete()
[INFO] (main) onComplete()
```

Durch eine Anbindung der SLF4J-Bibliothek werden alle gängigen Log-Formate unterstützt.

Publisher-Operationen

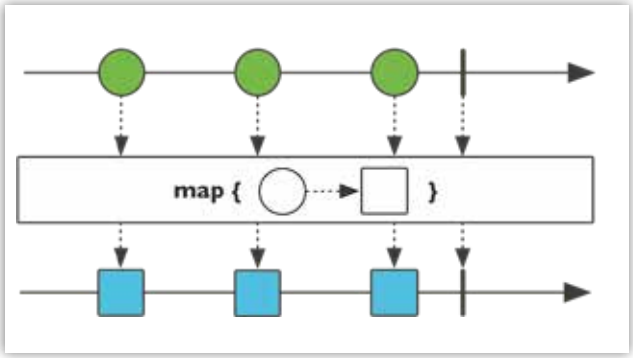
Wie in der funktionalen Welt üblich, haben **`Flux`** und **`Mono`** viele Operationen, die deren Werte manipulieren können. Fast alle diese Operationen liefern wiederum einen neuen Publisher (**`Flux`** oder **`Mono`**) zurück. Somit ist das Prinzip einer nicht blockierenden Anwendung gegeben, und es kann auf die Elemente zugegriffen werden, sobald diese verfügbar sind.

Operationen kann man grob in folgende Gruppen zusammenfassen[3]:

- Transformationen (**`map, flatMap, scan ...`**),
- Kombinationen (Verknüpfungen) (**`zip, zipWith, merge, mergeWith, concat, concatWith ...`**),
- Filterung (**`filter, first, firstEmmiting, last, skip, take ...`**),
- Mathematische Operationen (**`count, average, max ...`**),
- Boolesche Ausdrücke (**`all, every, some, includes ...`**).

Transformationen

Um die Werte eines Publishers mit einer synchronen Funktion (Lambda-Ausdruck) zu transformieren, kann die **`map`** Operation benutzt werden **(Abb. 3)**. In dem folgenden Beispiel wird der Titel eines Produktes zu Großbuchstaben umgewandelt.



Map Operation für Flux (Abb.3)

(Listing 9)

```
Mono.just("Print 9x13").
    map(p -> p.toUpperCase()).
    subscribe(System.out::print);
--- AUSGABE ---
PRINT 9X13
```

Für **`Flux`** kann man die gleiche Vorgehensweise wählen, mit dem Unterschied, dass die Transformation für jedes Element nacheinander angewendet wird:

(Listing 10)

```
Flux.just("Print 9x13", "Photobook A4").
    map(p -> p.toUpperCase()).
    subscribe(System.out::print);
--- AUSGABE ---
PRINT 9X13PHOTOBOOK A4
```

Soll für die Transformation eine Methode verwendet werden, die eine unbestimmte Zeit für die Bearbeitung in Anspruch nimmt, zum Beispiel ein Aufruf über Remote-API, ist die obere

Vorgehensweise nicht optimal. Die Methode **flatMap** bietet die Möglichkeit an, die Verarbeitung in einen anderen Publisher auszulagern[4]. Der zweite Publisher (siehe Beispiel mit Typ **Mono**) kann die Arbeit asynchron erledigen:

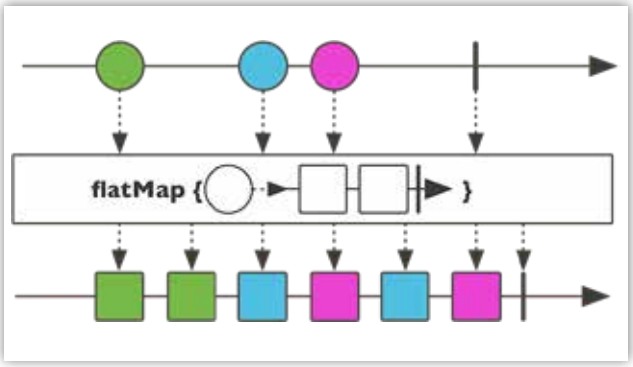
(Listing 11)

```
Flux.just("Print 9x13", "Photobook A4").
    flatMap(p -> asyncCapitalise(p)).
subscribe(System.out::print);

Mono<String> asyncCapitalise(String x) {
    Mono<String> mono = Mono.create(s->{
        s.success(title.toUpperCase());
    });
    return mono.subscribeOn(Schedulers.elastic());
}
--- AUSGABE ---
PHOTOBOOK A4PRINT 9X13
```

Ein weiterer Vorteil der **flatMap** Operation ist, dass die transformierende Methode zu einer Eingabe keine oder mehrere Ausgaben liefern darf **(Abb. 4)**:

(Listing 12)

```
Flux.just("Print 9x13", "Photobook A4").
    flatMap(p -> capitaliseAndUnderline(p)).
    subscribe(System.out::print);

Flux<String> capitaliseAndUnderline(String x) {
    return Flux.just(x.toUpperCase(), x.replace(" ", "_"));
}
--- AUSGABE ---
PRINT 9X13Print_9x13PHOTOBOOK A4Photobook_A4
```



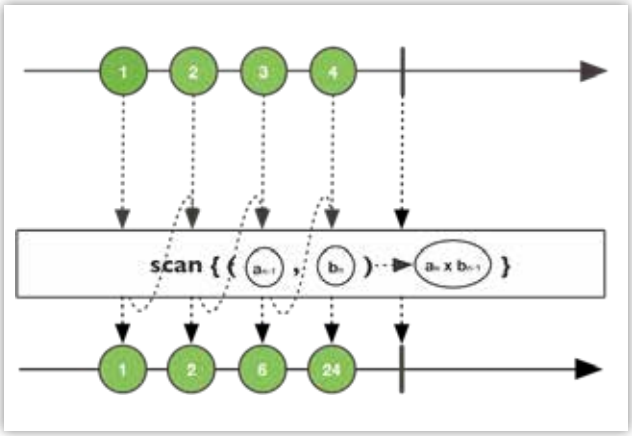
FlatMap Operation (Abb. 4)

scan verändert die Werte eines Publishers mit einer Accumulator-Funktion. Dabei werden die Ergebnisse als Eingabeparameter für die weiteren Werte-Veränderungen benutzt **(Abb. 5)**:

(Listing 13)

```
Flux.just(1,2,3,4,5).
    scan((x, y) -> x + y).
    subscribe(System.out::print);
```

```
--- AUSGABE (mit Leerzeichen getrennt) ---
1 3 6 10 15
```



Scan Operation (Abb. 5)

Man kann die Vorgehensweise mit folgenden Ausdrücken beschreiben:

(Listing 14)

```
result[0] = source[0]
result[1] = accumulator(result[0], source[1])
result[2] = accumulator(result[1], source[2])
result[3] = accumulator(result[2], source[3])
...
```

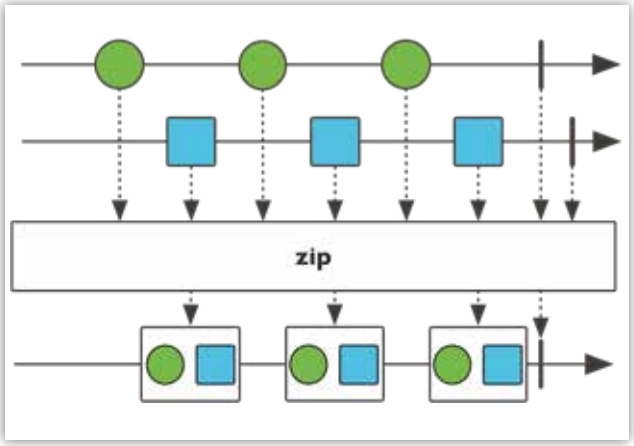
Kombinationen

Mit der statischen **zip** bzw. **zipWith** Operation des Publishers ist es möglich, Werte eines Publishers mit den Werten eines anderen Publishers zusammenzuführen. Dabei wird das erste Element des ersten Publishers mit dem ersten Element des zweiten Publishers, das zweite Element mit dem zweiten Element und so weiter zusammen gruppiert. Als Rückgabewert bekommt man einen Publisher mit dem Tupel von zwei Elementen zurück. **(Abb. 6)**

Diese Operation ist hilfreich, wenn die Elemente aus unterschiedlichen Quellen stammen, zum Beispiel aus unterschiedlichen Microservices geliefert werden. So können die in Gruppen zusammengefasst werden, sobald die Ergebnisse ankommen. Falls ein Publisher mehr Elemente als ein anderer Publisher besitzt, werden die überflüssigen Elemente nicht berücksichtigt:

(Listing 15)

```
Flux<String> titles =
    Flux.just("Print 9x13", "Photobook A4");
Flux<Double> prices =
```



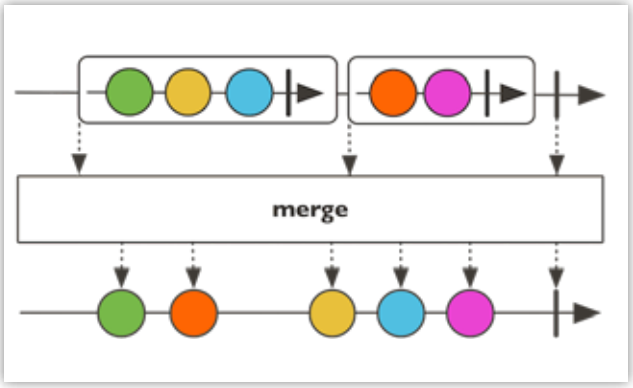
Zip Operation (Abb. 6)

```
  Flux.just(0.09, 29.99, 39.99);
Flux<Tuple2<String, Double>> zipped =
    Flux.zip(titles, prices);
zipped.subscribe(System.out::print);
--- AUSGABE ---
[Print 9x13,0.09][Photobook A4,29.99]
```

Die statische **merge** bzw. **mergeWith** Operation des Publishers fügt Elemente aus zwei Publisher in einem neuen **Flux** zusammen. Die Werte kommen dabei in der Reihenfolge an, in der sie verfügbar werden (Abb. 7):

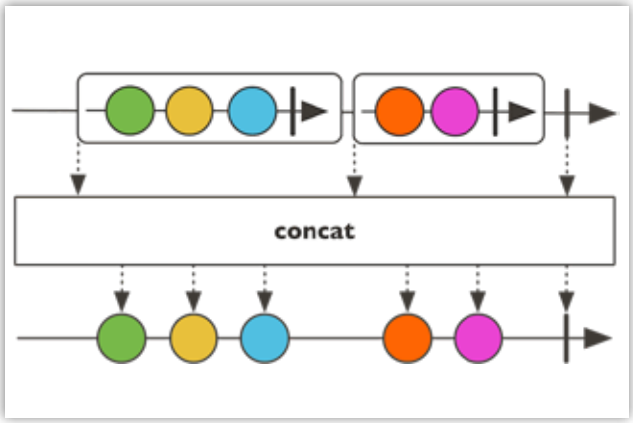
(Listing 16)

```
Flux<String> flux1 = Flux.just("a", "b", "c", "d");
// Flux emits item each 500ms:
Flux<String> flux1Delayed =
    Flux.interval(Duration.ofMillis(500)).
    zipWith(flux1, (i, string) -> string);
Flux<String> flux2 = Flux.just("1", "2", "3");
// Flux emits item each 700ms:
Flux<String> flux2Delayed =
    Flux.interval(Duration.ofMillis(700)).
    zipWith(flux2, (i, string) -> string);
Flux.merge(flux1Delayed, flux2Delayed).
    subscribe(System.out::print);
--- AUSGABE ---
a1b2cd3
```



Merge Operation (Abb. 7)

Im Unterschied zu **merge** wartet die **concat** bzw. **concatWith** Operation, bis alle Werte des ersten Publishers angekommen sind. Anschließend fängt sie mit dem Hinzufügen der Werte aus dem zweiten Publisher an (Abb. 8):



Concat Operation (Abb. 8)

(Listing 17)

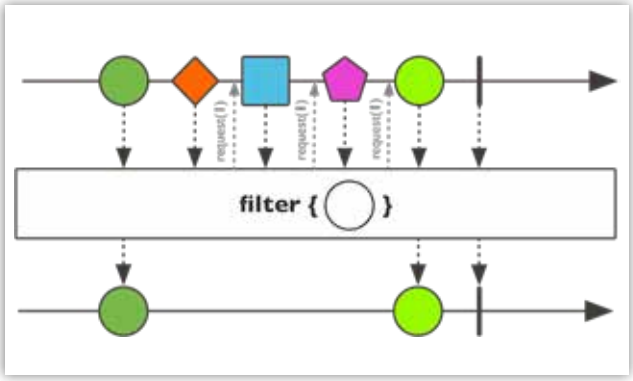
```
Flux.concat(flux1Delayed, flux2Delayed).
    subscribe(System.out::print);
--- AUSGABE ---
abcd123
```

Filterung

Mit der **filter** Operation besteht die Möglichkeit, Publisher-Werte zu filtern, in dem der Methode ein Kriterium in Form eines Lambda-Ausdruckes übergeben wird (Abb. 9):

(Listing 18)

```
Flux<String> titles =
    Flux.just("Print 9x13", "Photobook", "Calendar");
Flux<String> titlesStartingWithP =
    titles.filter(title -> title.startsWith("P"));
titlesStartingWithP.subscribe(System.out::print);
--- AUSGABE ---
Print 9x13Photobook A4
```



Filter Operation (Abb. 9)

Mathematische Operationen

Um zu ermitteln, wie viele Elemente ein Publisher liefert, wird die **count** Methode verwendet. Weil die Methode nur einen Wert zurückliefert, ist der Rückgabetyp ein **Mono<Long>**. Damit erhält man eine Anzahl der Elemente als **Long** zurück, sobald der Publisher ein **onComplete** Event sendet:

(Listing 19)

```
Mono<Long> count =
    Flux.just("Print 9x13", "Photobook A4").
    count();
count.subscribe(System.out::print);
--- AUSGABE ---
2
```

Boolesche Ausdrücke

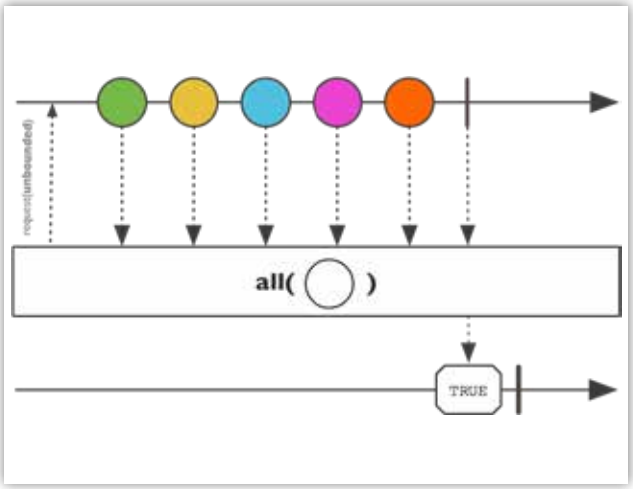
Für die Prüfung, ob alle Elemente eine Bedingung erfüllen, wird die Methode **all** eingesetzt. Als Parameter übergibt man ein Kriterium in Form eines Lambda-Ausdruckes. Als Rückgabewert wird ein **Mono<Boolean>** geliefert (Abb. 10):

(Listing 20)

```
Mono<Boolean> titlesBiggerThen5 =
    Flux.just("Print 9x13", "Photobook A4").
    all(title -> title.length() > 5);
titlesBiggerThen5.subscribe(System.out::print);
--- AUSGABE ---
true
```

Kompatibilität mit Java 9 Flow-API

Spring-Reactor-Core-3-Projekt implementiert die vier Interfaces aus dem Projekt Reactive-Streams. Die gleichen Interfaces hat Oracle in Java 9 als Flow-API in das Package **java.util.concurrent** übernommen. Wenn im Projekt Java 9 zum Einsatz kommt, kann man die Flow-API-Interfaces anstatt Interfaces von Reactive-Streams verwenden. Dafür gibt es im Spring-Reactor-Core-3-Projekt die **JdkFlowAdapter** Klasse. Diese stellt zu diesem Zweck die Konvertierungsmethoden **flowPublisherToFlux** und **publisherToFlowPublisher** bereit. Da Java zur Zeit keine eigene Implementierung der Reactive-Streams bereitstellt, sind die Entwickler eher dazu gezwungen, eine der bereits existierenden Implementierungen aus der Open-Source-Welt zu verwenden. Alleine wegen der vier genannten Interfaces auf Java 9 umzusteigen, bringt keine nennenswerten Vorteile. Es wird nur dann in Frage kommen, falls Java in den kommenden Releases eine eigene konkurrenzfähige Implementierung bereitstellt.



All Operation (Abb. 10)

Fazit und Ausblick

Die Standardisierung unterschiedlicher Einsätze durch Reactive-Streams-Initiative ermöglicht in der absehbaren Zukunft die nahtlose Integration von externen Bibliotheken und Schnittstellen. Durch Einbinden der reaktiven Komponenten für Web-Client, Datenbank und View bekommt die Software einen neuen Schwung.

In der nächsten JAVAPRO Ausgabe geht es im zweiten Teil dieser Serie um die fortgeschrittenen Themen und Konzepte rund um das Spring Reactor Core 3 Framework, wie Cold- und Hot-Publisher, Backpressure, parallele Verarbeitung, blockierende Operationen und Testen von Publishern. Außerdem widmen wir uns der Zukunft der reaktiven APIs und dem Vergleich zwischen reaktiven und streaming APIs.

Links:

1 Reactive Streams: http://www.reactive-streams.org
2 Reaktive Stream Specification: https://goo.gl/iwwLBh
3 Interactive diagrams of Rx Observables:
  http://rxmarbles.com
4 Lite Rx API Hands-On with Reactor Core 3:
  https://goo.gl/xmyavu

Reactive Streams:
http://www.reactive-streams.org

Reaktive Stream Specification
https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.1/README.md#specification

Interactive diagrams of Rx Observables
http://rxmarbles.com/

Lite Rx API Hands-On with Reactor Core 3
https://github.com/reactor/lite-rx-api-hands-on

Reactor - How to Combine Publishers (Flux/Mono)
http://javasampleapproach.com/reactive-programming/reactor/reactor-how-to-combine-flux-mono-reactive-programming

PEAKWORK
THE PLAYER HUB NETWORK

Join the digital travel pioneers

Du bist zielorientiert, hast Spaß daran eigenständig zu arbeiten? Du hast das richtige Unternehmen noch nicht gefunden und bist auf der Suche nach einer neuen Herausforderung im Bereich **Software Development**, **Projekt Management**, **IT Administration** oder **Sales**? Dann sollten wir schnellstens über Deine Karriere bei Peakwork sprechen, denn in unserem multikulturellen Team kannst Du Deine Karriere vorantreiben.
Informationen über Peakwork und offene Stellen findest Du unter **www.peakwork.com**.

Follow us on

#jointhepeakworkcrew

#JAVAPRO #Kotlin #Spring #TornadoFX #JavaFX

Kotlin DSLs mit Extension-Functions

In Kotlin lässt sich Code mit Hilfe von domänenspezifischen Erweiterungen stark vereinfachen. Inzwischen gibt es zahlreiche solcher DSLs in Kotlin, z.B. für SQL, Gradle und Vaadin. In diesem Workshop beleuchten wir die Extension-Functions von Kotlin am Beispiel einer JavaFX-Anwendung und werden anschließend selbst eine kleine domänenspezifische Erweiterung konstruieren.

Autor:

Thomas ist unabhängiger Trainer und Softwareentwickler aus Hamburg. Er beschäftigt sich seit über 20 Jahren mit UI und Datenbankentwicklung und gibt sein Wissen in lebendigen Trainings- und Coachingeinsätzen an Entwicklerteams weiter. Seine Lieblingsthemen sind aktuell Kotlin, Vaadin Flow und jooq mit Postgres.

In diesem Workshop werden wir eine wirklich sehr simple Newsletter-Anmeldung mit JavaFX bauen (Abb. 1). In Java könnte der Code möglicherweise wie in (Listing 1) aussehen:

(Listing 1)

```
GridPane gridPane = new GridPane();
gridPane.setVgap(10.0);
gridPane.setHgap(10.0);
```

```
gridPane.setPadding(new Insets(10.0,10.0,10.0,10.0));

Label label = new Label("Email");
GridPane.setConstraints(label,1,1);

TextField textField = new TextField();
GridPane.setConstraints(textField,2,1);

Button button = new Button("Subscribe");
GridPane.setConstraints(button,1,2);

gridPane.getChildren().addAll(label,textField,button)
```

Oberfläche der Newsletter-Anmeldung (Abb. 1)

Zuerst benötigen wir ein Layout, in unserem Fall eine GridPane, auf welcher wir ein Label, ein Textfeld und einen Button platzieren. Das Styling lässt sich mit JavaFX in CSS auslagern, aber wir starten erstmal klein.

An dieser Stelle kommt nun TornadoFX in Spiel, das unter anderem einige sehr brauchbare Erweiterungen mitbringt um JavaFX-Oberflächen zu konstruieren. Dabei macht es intensiv von einigen Kotlin-Features Gebrauch, die wir genauer durchleuchten wollen. Die selbe Oberfläche mit Tornado und Kotlin könnte in etwa so aussehen:

(Listing 2)

```
class NewsletterView : View() {
  override val root = gridpane {
      row {
          label("Email")
          textfield()
      }
      row {
          button("Subscribe")
      }
      vgap = 10.0
      hgap = 10.0
      padding = Insets(10.0, 10.0, 10.0, 10.0)
  }
}
```

Was hier wirklich ins Auge sticht ist, dass wir den Aufbau und die Struktur unserer Elemente auf den ersten Blick auch in der Struktur unseres Codes wiederfinden. Tornado bietet uns hier eine domänenspezifische Sprache (DSL) um unser UI zu konstruieren. Solche syntaktischen Muster dürften einigen Entwicklern bereits aus Groovy bekannt sein. Die Idee ist an sich nichts Neues. Der große Unterschied den Kotlin hier macht ist, dass diese DSL typsicher ist und anders als in Groovy der Compiler prüfen kann, ob zum Beispiel **`row { label() }`** syntaktisch korrekt ist. Wenn wir verstehen wollen wie das Ganze funktioniert, müssen wir zuerst wissen, was eine Extension-Function in Kotlin ist. Ein einfaches Beispiel finden wir in der Kotlin-Dokumentation:

(Listing 3)

```
To declare an extension function, we need to prefix its name with
a receiver type, i.e. the type being extended. The following
adds a swap function to MutableList<Int>: fun MutableList<Int>.
swap(index1: Int, index2: Int) {
```

Natürlich können wir rein technisch eine bestehende Klasse nicht wirklich erweitern. Was Kotlin uns hier bietet ist syntaktischer Zucker, denn eigentlich sind das nicht mehr als (meist) statische Hilfsmethoden, die (hier) eine MutableList als zusätzlichen Parameter erhalten. Wirklich angenehm daran ist, dass wir nun so tun können als wären wir in der Lage, auf einer MutableList die **`swap`** Methode zu benutzen. Tatsächlich ist es keine echte Methode von **`MutableList`**, denn dazu müsste das Classfile von **`MutableList`** nachträglich geändert werden. Was stattdessen entsteht, ist ein neues Classfile mit einer statischen Methode, die eine List als ersten Parameter hat. Mittels des Tools **`javap`** erhalten wir Zugriff auf den Code:

(Listing 4)

```
javap  MutableSwapKt.class
Compiled from "MutableSwap.kt"
public final class de.eiswind.tornado.MutableSwapKt {
 public static final void swap(
    java.util.List<java.lang.Integer>, int, int);
}
```

Es gibt also immer eine kleine Einschränkung: Wir können in der Extension-Function nur auf die öffentliche API der Klasse zugreifen, die wir erweitern. Doch wie wird daraus jetzt die strukturierte DSL mit der wir unser User-Interface so schön bauen konnten? Um das zu verstehen, werfen wir am besten mal einen Blick auf die erste Funktion, die wir aufrufen, nämlich **`root = gridpane { ...`** Diese Funktion ist folgendermaßen definiert:

(Listing 5)

```
fun EventTarget.gridpane(op: (GridPane.() -> Unit)? = null) =
opcr(this, GridPane(), op)
```

Der Typ **`EventTarget`** taucht in der Vererbungshierarchie unserer Basisklasse **`View`** auf. Damit ist **`gridpane`** eine Extension, die wir in unserer eigenen Klasse ohne weiteres benutzen können. Auf die **`opcr`** Funktion gehen wir im späteren Verlauf dieses Artikels noch genauer ein. Der eigentlich spannende Teil kommt jetzt erst noch. Unsere Extension hat einen einzigen Parameter, der wie folgt spezifiziert ist:

(Listing 6)

```
gridpane(op: (GridPane.() -> Unit)? = null)
```

Das bedeutet, dass **gridPane** als Parameter eine parameterlose Funktion **()->Unit** erwartet. Doch hier ist zusätzlich noch ein Extension-Typ angegeben:

(Listing 7)

```
GridPane.() -> Unit
```

Dies hat zur Folge, dass jede Funktion die wir hier übergeben, eine Extension-Function von **GridPane** wird. Wenn wir also folgendes schreiben:

(Listing 8)

```
gridpane {
     vgap = 10.0
}
Gridpane op:{}
node = GridPane()
op.invoke (node)
⇦„node.op“
```

Dann ist **{ vgap = 10.0 }** eine Extension-Function von **gridPane**. Die GridPane selbst wird beim Aufruf der opcr Funktion erzeugt und ist gleichzeitig die Instanz, auf der unsere neue Extension aufgerufen wird. **this** innerhalb der Lambda ist also eine GridPane:

(Listing 9)

```
fun <T : Node> opcr(parent: EventTarget, node: T,
                    op: (T.() -> Unit)? = null): T {
  parent.addChildIfPossible(node)
  op?.invoke(node)
  return node
}
```

Da **vgap** eine Property von **GridPane** ist, können wir hier typsicher einen Wert zuweisen. Auf der Classfile-Ebene entsteht eine Lambda, in der wir die GridPane als Parameter von **invoke** wiederfinden:

(Listing 10)

```
javap  'NewsletterView$root$1.class'
Compiled from "NewsletterKotlinApp.kt"
final class de.eiswind.tornado.NewsletterView$root$1 extends
kotlin.jvm.internal.Lambda implements kotlin.jvm.functions.Func-
tion1<javafx.scene.layout.GridPane, kotlin.Unit> {
 public static final de.eiswind.tornado.NewsletterView$root$1
INSTANCE;
 public java.lang.Object invoke(java.lang.Object);
 public final void invoke(javafx.scene.layout.GridPane);
 de.eiswind.tornado.NewsletterView$root$1();

 static {};
}
```

Wie können wir das Prinzip nun für unsere eigenen Zwecke nutzen? Nehmen wir an, wir wollen folgende Syntax erlauben:

(Listing 11)

```
cat {
  times = 2
  meow()
}
```

Dazu benötigen wir eine Hilfsklasse, die wie folgt aussehen könnte:

(Listing 12)

```
class Cat{

  var times : Int = 1

  fun meow(){
      for(i in 1..times){
          println("Meow")
      }
  }
}
```

Der Trick ist nun eine Funktion **cat** zu schreiben, die als Parameter eine Extension für die Klasse **Cat** erwartet. So haben wir in der Extension Zugriff auf alle Funktionen von **Cat**:

(Listing 13)

```
fun cat(op: Cat.() -> Unit)
```

In dieser Funktion konstruieren wir eine Instanz von unserer Hilfsklasse und rufen die übergebene Funktion op auf:

(Listing 14)

```
fun cat(op: Cat.() -> Unit) {
  val cat = Cat()
  op.invoke(cat)
}
```

Dadurch wird die Variable **times** auf **2** gesetzt und **meow** ausgeführt. Zugegeben, dies ist ein sehr einfaches Beispiel, aber es lässt sehr gut erkennen, mit welch einfachen Konstrukten eine solche domänenspezifische Erweiterung gebaut werden kann. Unser erstes JavaFX-Beispiel nutzt TornadoFX, eine Library, die sehr intensiv Gebrauch von diesem Muster macht um eine sehr umfangreiche DSL für JavaFX bereitzustellen. Es gibt inzwischen zahlreiche solcher typsicheren DSLs in Kotlin z.B. für Vaadin, SQL, Gradle, etc. Wie wir gesehen haben, ist es kein Hexenwerk, selbst eine DSL zu erstellen.

#JAVAPRO #Equifax #ApacheStruts

Wie sicher ist Open Source?

2017 wurde der Finanzdienstleister Equifax Opfer eines groß angelegten Hackerangriffs. Die Ursache soll eine Sicherheitslücke in einer Open-Source-Software gewesen sein, die Equifax einsetzte. Nach derart großen Pannen werden immer wieder Stimmen laut, Open Source wäre unsicher. Die Ursachen sind jedoch meist auf einen fahrlässigen Umgang mit Open-Source-Software und bereits bekannten Sicherheitslücken zurückzuführen. Wer selbst Open Source einsetzt, sollte prüfen, ob man nicht ebenfalls durch Nachlässigkeit gefährdet ist.

Am 8. März 2017 hat das US-amerikanische Ministerium für Heimatschutz, genauer das Computer-Emergency-Readiness-Team (US-CERT), eine Warnung herausgegeben, dass eine Sicherheitslücke in bestimmten Versionen von Apache Struts besteht und diese gepatched werden muss. Der Finanzdienstleister Equifax, eine der wichtigsten Kreditauskunfteien der USA, nutzte Struts in ihrem Online-Dispute-Portal[1], eine Website auf der Verbraucher Beanstandungen über ihre Kreditauskunft anfechten können. Tausende dieser Kreditauskunftsdatensätze gelangten so in die Hände von Hackern. Laut Aussagen von Ex-Equifax-CEO Richard Smith weiß man nun, dass die anfällige Version von Apache Struts bei Equifax weder gefunden noch gepatcht wurde. Über die Hintergründe ist nichts bekannt.

Welche Struts-Sicherheitslücke lag eigentlich vor?

Erste Spekulationen deuteten darauf hin, dass es die Schwachstelle CVE-2017-9805 war, die etwa zur gleichen Zeit bekannt wurde, wie die Nachricht, dass Equifax gehackt wurde. Schuld war jedoch CVE-2017-5638 , die im März letzten Jahres berichtet wurde. Zu diesem Zeitpunkt waren bereits Exploits verfügbar. Mehrere Websites berichteten bereits von aktiven Versuchen, sie zu scannen und diese Exploits zu nutzen.

Die Datendiebstähle sind laut Equifax von Mitte Mai bis Juli geschehen. Die OSSRA-Research (Open Source Security & Risk Analysis) von Black Duck legt nahe, dass Equifax vermutlich keine vollständige Sicht auf den von ihnen verwendeten Open-Source-Code hatte. Im OSSRA-Bericht haben wir festgestellt, dass Unternehmen doppelt so viel Open Source nutzten, wie sie selbst annehmen. Insgesamt 67 Prozent der analysierten Anwendungen mit Open-Source-Code hatten Schwachstellen in den verwendeten Komponenten; im Durchschnitt waren die in diesen Anwendungen identifizierten Schwachstellen seit mehr als vier Jahren öffentlich bekannt.

Das Sicherheitsteam von Equifax mag sich zwar dieser Sicherheitslücke bewusst gewesen sein, doch die Anwendungen, die gehackt wurden, flogen offenbar völlig unter ihrem Radar. Dies würde erklären, wie eine im März offenbarte Open-Source-Schwachstelle im Zeitraum von Mai bis Juli bei Equifax weiterhin von Hackern genutzt werden konnte.

Autor:

Als Vice President of Product Strategy bei Black Duck Software konzentriert sich Patrick Carey auf die Zusammenarbeit mit Technik- und Produktmanagement-Teams. Er entwickelt und verbessert mit ihnen Lösungen, die Unternehmen dabei unterstützen, schnell und sicher Software zu erstellen, wenn sie Open-Source-Code-Komponenten integrieren.
Die Software von Black Duck ermöglicht es Kunden sicherzustellen, dass der Open-Source-Code in ihren Anwendungen frei von bekannten Open-Source-Schwachstellen sowie mit Open-Source-Lizenzen konform ist - von der Code-Auswahl, über den Entwicklungsprozess, bis zur Auslieferung der Anwendung.

Der Patch für CVE-2017-5638 stammt also aus März – lange vor Mai und damit lange vor dem Equifax-Vorfall. Die meisten Fehlerbehebungen erfordern den Download und die Installation eines Patches sowie den Neustart des Servers. Im Gegensatz dazu erfordert die Behebung dieser speziellen Sicherheitslücke in der Regel, dass jede Web-Anwendung, die mit der anfälligen Version von Apache Struts entwickelt wurde, mit einer gepatchten Version neu kompiliert wird.

Warum sollte Equifax die Sicherheitslücke für weitere zwei Monate auf sich beruhen lassen? Möglich, dass das Team bei Equifax die ganze Zeit daran gearbeitet hat, diesen Patch zu installieren und in der Ausführung einfach nicht schnell genug war. Aber angesichts der Schwere der Lücke scheint es unwahrscheinlich, dass sie diese wissentlich so lange offen gelassen hätten. Die wahrscheinlichere Antwort scheint zu sein, dass niemand diese bestimmte Software auf dem Schirm hatte.

Wie man mit Sicherheitslücken umgehen sollte

Das Apache-Struts-Projektmanagementkomitee veröffentlichte eine Stellungnahme zum Equifax-Vorfall , der Vorschläge zum Sichern aller offenen oder geschlossenen Quellbibliotheken in Softwareprodukten und -diensten enthält:

1. Stellen Sie sicher, dass Sie wissen, welche Frameworks und Bibliotheken, in welchen Versionen, in Ihrer Software verwendet werden. Verfolgen Sie Sicherheitsankündigungen, die diese Produkte und Versionen betreffen.
2. Richten Sie Prozesse ein, um schnell ein Sicherheits-Update durchführen zu können, sobald die Unterstützung von Frameworks oder Bibliotheken aus Sicherheitsgründen aktualisiert werden muss. Am besten ist es dabei in Stunden oder ein paar Tagen zu denken - nicht in Wochen oder Monaten. Die meisten Verstöße, die wir bemerken, werden dadurch verursacht, dass bekannte Schwachstellen, die monatelang oder sogar jahrelang angreifbar sind, nicht gefixt werden.
3. Denken Sie daran: Jede komplexe Software enthält Fehler. Man sollte seine Sicherheitsrichtlinien nicht auf der Annahme aufbauen, dass die eingesetzten Softwareprodukte fehlerfrei sind.
4. Richten Sie Sicherheitsebenen ein. Es ist ein gutes Verfahren des Software-Engineerings, individuell gesicherte Schichten hinter einer öffentlichen Präsentationsschicht wie dem Apache-Struts-Framework zu haben. Eine Lücke in der Präsentationsschicht sollte niemals den Zugriff auf signifikante oder sogar alle Back-End-Informationsressourcen ermöglichen.
5. Überwachen und erkennen Sie ungewöhnliche Zugriffsmuster auf öffentliche Web-Ressourcen. Heutzutage gibt es viele Open-Source- und kommerzielle Produkte, um solche Muster zu erkennen und Warnungen zu geben. Das Monitoring von geschäftskritischen, webbasierten Diensten ist gute Praxis und wird empfohlen.

Diese Ratschläge stellen eine gute Grundlage dar, um solche Vorfälle wie bei Equifax zu vermeiden.

Mangel an Sichtbarkeit könnte Tür zum Exploit geöffnet haben

Ähnlich wie bei Heartbleed beleuchtet dieser Vorfall das Problem der Open-Source-Sicherheit und den Wettlauf zwischen den Anwendern und (potenziellen) Open-Source-Hackern, sobald eine Sicherheitslücke gemeldet wird. Da Open Source so weit verbreitet und leicht zugänglich ist, sind Exploits direkt auf YouTube und Hacker-Seiten verfügbar, nachdem eine Sicherheitslücke zum ersten Mal gemeldet wurde. Diese Exploits wirken wie ein Türöffner. Hacker nutzen sie, um in tausende von Websites und Anwendungen einzudringen. Zu diesem Zeitpunkt besteht größtes Risiko für einen Angriff und daraus folgenden Schaden.
Equifax verfügt - wie viele andere Unternehmen - über ein großes Anwendungsportfolio. Wie die OSSRA-Studie belegt, überwachen die meisten Unternehmen die von ihnen verwendeten Open-Source-Komponenten nicht ausreichend. Falls Equifax also keine Lösung wie Black Duck Hub implementiert hat, ist davon auszugehen, dass sie höchstwahrscheinlich auch kein vollständiges und verlässliches Inventar der Open-Source-Komponenten in ihren Anwendungen haben. Als die Sicherheitslücke im März aufgedeckt wurde, wusste Equifax wahrscheinlich nicht einmal, dass sie gefährdet waren. Selbst dann nicht, wenn ihr Sicherheitsteam über die Sicherheitslücke informiert war. Anders ausgedrückt: Sie waren im Blindflug.

Während die Exploits für CVE-2017-5638 bereits bekannt waren und fast unmittelbar nach der Offenlegung der Sicherheitslücke auch genutzt wurden, lief Equifax ahnungslos in die Falle. Bei Equifax stand die Tür vier Monate lang offen – in diesem gesamten Zeitraum scannten und sondierten Hacker aktiv nach verwundbaren Websites und Anwendungen.

Der Datenzugriff wurde bei Equifax bemerkt. Bei durchschnittlich 206 Tagen für die Identifizierung und Behebung von Sicherheitslücken (vgl.siehe Data Breach Report des Ponemon-Instituts für das Jahr 2017) zeigt dies, dass die Reaktionszeit von Equifax überdurchschnittlich schnell war.

Was können Unternehmen aus dem Equifax-Vorfall lernen?

Aufgrund möglicher Auswirkungen auf die Privatsphäre der User, anschließende Rechtsstreitigkeiten und sogar Folgen für die Politik, sollten sich Führungskräfte und Sicherheitsteams fragen, ob sie nicht ebenfalls von derartigen Sicherheitslücken gefährdet sind. Am Ende gibt es viele Lektionen, die gelernt werden müssen und Teams helfen können, diese Art von Sicherheitslücke zu vermeiden:

1. Einsicht und Kontrolle ist unabdingbar. Man kann sich nicht schützen, wenn man nicht weiß, was im Code enthalten ist.

Wenn man nicht über ein vollständiges Bild des von den Teams verwendeten Open-Source-Codes verfügt, setzt man seine Anwendungen großer Gefahr aus.

2. Das Schwachstellenmanagement von Open Source muss automatisiert und eng in Entwicklungs- und DevOps-Tools sowie Prozesse integriert werden. Der Code ist nur so sicher wie sein schwächster Teil. Nur wenn sichergestellt ist, dass der gesamte Code einschließlich der schwächsten Glieder gescannt wurde, bevor man mit dem Roll-Out der Anwendung beginnt, ist der Open-Source-Code in Ordnung.
3. Man sollte alles Mögliche tun, um das Zeitfenster zwischen der Meldung der Sicherheitslücke und dem Moment, in dem sie gepatcht oder behoben wird, zu minimieren. Jeden Tag werden mehr als zehn neue Open-Source-Sicherheitslücken öffentlich gemacht. Leider reicht die National Vulnerabilities Database (NVD) als frühzeitige Warnung nicht aus. Die Untersuchungen von Black Duck haben gezeigt, dass es durchschnittlich drei Wochen dauert, bis Schwachstellen in der NVD dokumentiert sind.

Nach derartig großen Sicherheitspannen wie bei Equifax, werden immer wieder Stimmen laut, Open Source wäre grundsätzlich unsicher. Doch Open-Source-Software ist nicht mehr oder weniger unsicher als jede andere Software auch. Wenn man sie jedoch nicht konsequent überwacht und bei Sicherheits-Updates nicht auf dem Laufenden bleibt, setzt man sich selbst sowie die Systeme und Daten seiner Kunden einem Risiko aus.

Links:

1 Equifax Online Portal: https://goo.gl/1qWXBz
4 Apache Struts, Schwachstelle CVE-2017-980: https://goo.gl/rEs7cJ
5 Apache Struts, Schwachstelle CVE-2017-5638 https://goo.gl/nGfB1S
6 Open Source Security & Risk Analysis von Black Duck: https://goo.gl/tqEsN6
7 ArsTechnica-Artikel über die Verzögerung bei Equifax: https://goo.gl/xngAm2
8 Stellungnahme der Projektmanager https://goo.gl/qRzHyF
9 Black-Duck-Blog über Heartbleed: https://goo.gl/8YY1F7
10 Vgl. ArsTechnica-Artikel: https://goo.gl/RMPhoc
11 Data Breach Report 2017 https://goo.gl/fKkvZi
12 National Vulnerabilities Database https://nvd.nist.gov/

Equifax Online Portal: https://goo.gl/1qWXBz
Stellungnahme der Projektmanager https://goo.gl/qRzHyF
Apache Struts, Schwachstelle CVE-2017-980: https://goo.gl/rEs7cJ
Apache Struts, Schwachstelle CVE-2017-5638 https://goo.gl/nGfB1S
Open Source Security & Risk Analysis von Black Duck: https://goo.gl/tqEsN6
ArsTechnica-Artikel über die Verzögerung bei Equifax: https://goo.gl/xngAm2
Black-Duck-Blog über Heartbleed: https://goo.gl/8YY1F7
Vgl. ArsTechnica-Artikel: https://goo.gl/RMPhoc

#JAVAPRO #Scrum #Metriken

Metriken für agile Softwareentwicklung

Bei agiler Softwareentwicklung mit SCRUM stehen die Anpassung an veränderte Bedingungen bzw. die Anforderungen und die Optimierung des Entwicklungsprozesses im Mittelpunkt. Anpassungen und Optimierungen gehen dabei von den Entwicklern selbst aus. Um Entscheidungen besser beurteilen zu können, kommen i.d.R. Metriken zum Einsatz. Als Grundlage werden Daten zu Arbeitszeiten und Aufgaben benötigt. Die Metriken ermöglichen den Teammitgliedern die Metadaten ihrer eigenen Arbeit besser zu verstehen und seine Prozesse sinnvoll anzupassen.

Autor:

Richard Fichtner ist bei XDEV Software in verschiedenen Projekten eingebunden. Als zertifizierter Scrum Master und Product Owner unterstützt er interne und externe Kunden bei Ihrem Wandel hin zu einer agilen Entwicklungskultur.

Dabei profitiert er von seiner mehr als 15-jährigen Projektpraxis in wechselnden Rollen. Seine Interessenschwerpunkte sind Clean-Code, neue Technologien und alles was agil ist. Er studierte an der Wilhelm Büchner Hochschule berufsbegleitend Wirtschaftsinformatik und behandelte in seiner Masterarbeit das Thema „Metriken für agile Softwareentwicklung".

Motivation

Softwareentwicklung ist ein anspruchsvolles Aufgabenfeld auf vielschichtigen Ebenen. Neben technischen und fachlichen Aspekten sind auch die Gestaltungsmöglichkeiten bei dem Prozess der Softwareentwicklung vielfältig. In den vergangenen Jahren haben sich agile Vorgehensmodelle etabliert, damit die geradlinigen Wasserfälle aufgewirbelt und in Wildwasserbahnen verwandelt. So wird der bekannte Entwicklungsprozess und alle damit verbundenen Gewohnheiten in Frage gestellt. Entwicklungsteams erhalten neue Frei- und Spielräume, um ihre Arbeitsabläufe zu gestalten. Durch eine optimale Anpassung an die jeweilige Realität des konkreten Projektes lassen sich Wettbewerbsvorteile schaffen. Kurze Iterationen und enge Zusammenarbeit mit Kunden ermöglichen einen hohen Wirkungsgrad der knappen Ressource „Softwareentwickler".

Gleichzeitig bedeutet dies, dass Softwareentwicklungsteams in agilen Projekten stärker mit Entscheidungen rund um das eigene Vorgehensmodell konfrontiert werden. Die sprichwörtliche Qual der Wahl kommt als Preis der neuen Freiheiten. Welche Veränderungen sind sinnvoll? Muss überhaupt etwas geändert werden? Wo haben wir Verbesserungspotenzial? Woher wissen wir, ob Änderungen den gewünschten Effekt erzielen? Dies sind grundlegenden Fragen die auch meine Kollegen und mich in den letzten Jahren intensiv beschäftigt haben. Daher haben wir Metriken konzipiert, um die Metadaten zur Arbeit des Entwicklungsteams den Teammitgliedern in visualisierter Form zugänglich zu machen. Im Spannungsfeld zwischen Datenschutz und Transparenz soll es dem Team möglich sein, die eigene Arbeit besser zu erfassen und zu verstehen. Auf Basis der so gewonnenen Informationen ist das Team in der Lage, seine Prozesse sinnvoll anzupassen.

Welchen Zweck erfüllen Metriken?

Metriken können bei einer Vielzahl von Anwendungsfällen hilfreich sein. In der Regel werden Metriken in der Vorbereitung von Entscheidungen eingesetzt:

> „Metrics are measurements or properties
> that help in making decisions."[1]

Dabei lassen sich Metriken drei grundlegenden Aufgaben zuordnen:[2]

1. Metriken können helfen besser zu verstehen, was tatsächlich passiert. Durch die Untersuchung der aktuellen Situation lässt sich eine Richtlinie herstellen, die es ermöglicht, Ziele für zukünftiges Verhalten zu definieren. In diesem Sinn werden Veränderungen an Eigenschaften von untersuchten Prozessen bzw. Produkten im Zusammenhang mit Maßnahmen besser sichtbar und verständlich.
2. Metriken erlauben es zu kontrollieren was passiert. Durch das gewonnene Verständnis über Beziehungen zwischen Maßnahmen und deren Auswirkungen können gezielt Maßnahmen durchgeführt werden, die mit hoher Wahrscheinlichkeit zur Erreichung der angestrebten Ziele führen. In diesem Bereich spielen Metriken eine wichtige Rolle beim Risikomanagement.
3. Metriken können zur Verbesserung von Prozessen und Produkten ermutigen. Metriken können Menschen motivieren ihr Verhalten zu verändern. So kann mit Gamification[3] auf Basis von Metriken eine Motivationssteigerung für Arbeiten erreicht werden, die ansonsten als wenig herausfordernd oder zu komplex empfunden werden.[4] Im Gegenzug können Metriken auch dazu führen, dass Demotivation einsetzt. Aus Angst vor negativen Auswirkungen, als Reaktion auf schlechte Werte, werden die Basisdaten systematisch geschönt. Die Metrik wird somit unbrauchbar für den angestrebten Zweck.[5]

Herleitung von Metriken

Metriken sind oft auch ein iterativer Prozess. Dies zeigt beispielsweise die Geschichte der Messung der Temperatur. In **(Tabelle 1)** werden einige Meilensteine der Entwicklung der Temperaturmessung dargestellt.

Jahr	Messverfahren
2000 v. Chr.	Einordnung: " heißer als"
1600 n. Chr.	Das erste Thermometer misst "heißer als"
1720 n. Chr.	Fahrenheit-Skala
1742 n. Chr.	Celsius-Skala
1854 n. Chr.	Absolut Null, Kelvin-Skala

(Tabelle 1) Entwicklung der Temperaturmessung

Wir können anfangen die Welt zu verstehen, indem wir intuitiv Beziehungen beschreiben, ohne dass wir diese reproduzierbar und absolut messen können. Nachdem einige Erfahrungen gesammelt worden sind und ein Grundverständnis für den Gegenstand der Betrachtung entstanden ist, wird eine Möglichkeit benötigt, um eine genauere Messung durchzuführen. Dafür wiederum müssen meist spezielle Methoden oder Hilfsmittel herangezogen werden. Die anschließende Analyse der Ergebnisse führt oft zu neuen Erkenntnissen, höherer Messgenauigkeit und einem tieferen Verständnis. Metriken müssen also erarbeitet werden.

Im Zusammenhang mit SCRUM gibt es bereits einige Metriken und Visualisierungen, die weit verbreitet sind und in der Community diskutiert werden. Dazu zählen Metriken wie Velocity, die angibt, wie viel Storypoints ein Team in einem Sprint leistet und Visualisierung wie das Burn-Down-Chart. Über die gängigen Metriken lassen sich grundlegende Fragen zu agiler Softwareentwicklung beantworten. Wir brauchen für unsere Fragen jedoch andere Metriken.

Rahmenbedingungen und Annahmen

Für alle Metriken gelten folgende Rahmenbedingungen und Annahmen:

- Basis für die Metriken sollen Daten aus bestehenden Systemen sein.
- Die Metriken sollen für SCRUM-Teams unabhängig vom Einsatz verschiedener Software-Tools anwendbar sein.
- Als Betrachtungsrahmen soll ein Entwicklungsteam und dessen Aufgaben in einem bestimmten Zeitraum

herangezogen werden. Der Zeitraum ist standardmäßig eine Iteration des Projekts.
- Die Visualisierungen der Metriken sollen sich durch Standarddiagrammbibliotheken in Web-Anwendungen implementieren lassen, um eine einfache Integration in bestehende Web-Anwendungen zu ermöglichen.
- Es wird angenommen, dass bezüglich der Verarbeitung persönlicher Daten eine der im Bundesdatenschutzgesetz festgelegten Optionen erfüllt ist und die Verarbeitung daher zulässig ist.

Woher kommen die Daten?

Auf die manuelle Erhebung von Daten verzichtet man i.d.R. aus zwei Gründen. Zum einen ist die manuelle Erhebung von Daten mittels Formularen, egal ob analog oder digital, der Gefahr von absichtlicher und unabsichtlicher Verfälschung, Verweigerung und Verspätung ausgesetzt. Zum anderen wirken sich zusätzliche bürokratische Maßnahmen negativ auf die Nettoarbeitszeit des Entwicklungsteams aus. Daher soll ausschließlich auf bereits vorhandene bzw. bereits durch den Prozess oder die Verwendung von Tools entstehenden Daten zurückgegriffen werden.
Softwareentwicklungsteams generieren über den gesamten Softwarelebenszyklus hinweg eine Menge Daten in verschiedenen Systemen:

- Projekt-Tracking-System (PTS) Informationen zu Aufgaben und Aufwänden.
- Version- bzw. Source-Control-System (VCS) führt Buch über Änderungen an Dateien.
- Continuous-Integration (CI) Server prüfen stetig die Qualität der Software.

Durch den Einsatz der genannten Systeme existiert für die meisten Teams bereits eine breite Datenbasis als Grundlage für eine Vielzahl an Metriken, die mit Daten aus einem System hergeleitet werden können. Diese Metriken beschränken sich auf eine spezielle Sicht auf das Team. Interessant wäre eine ganzheitlichere Sicht auf das Team, um Zusammenhänge über Grenzen der Dateninseln hinaus zu verstehen. Dafür müssen die Daten aus den einzelnen Systemen zusammengeführt werden. Die Herausforderung dabei ist es die jeweiligen Datenmengen sinnvoll zu integrieren.

Es müssen Verbindungspunkte zwischen den Datenmengen gefunden oder geschaffen werden, um die Daten aus mehreren Quellen sinnvoll zu verbinden. Als Verbindungspunkte können Daten dienen, die in jeweils zwei zu verbindenden Datenmengen vorliegen.

Ein universelles Konzept, welches in vielen Datenmengen existiert ist die Zeit. Genauer ein Zeitpunkt, der einem Wert in einer Datenmenge zugeordnet ist. Über den Zeitpunkt können Datenmengen integriert werden. Anspruchsvoller wird die Integration von Datenpunkten, die sich nicht eindeutig zuordnen lassen. Unterschiedliche Datenstrukturen, welche dieselben Entitäten abbilden, sollten normalisiert werden. Ein prominentes Beispiel dafür sind Benutzer. In der Regel halten die Systeme eigene Schlüssel für ihre Benutzerdatensätze. Daher sind diese Schlüssel alleine in der Regel nicht ausreichend, um die Datensätze zusammenzuführen. Über ein eindeutiges Merkmal, wie die E-Mailadresse, können Benutzer auf eine einheitliche Basis konsolidiert werden. Kann kein durchgängig verwendetes Merkmal ausgemacht werden, kann auf eine externe Zuordnung der Datensätze ausgewichen werden. Diese Lösung ist nicht wartungsfrei (neue Benutzer müssen in der Zuordnungsregel eingetragen werden) und sollte daher nur als Workaround zum Einsatz kommen.

Metriken – Das Team-Time-Commitment (TTC)

Das Team verpflichtet sich bei SCRUM auf ein Sprint-Ziel. Wie genau diese Verpflichtung zu interpretieren ist, geht aus dem SCRUM-Guide nicht hervor. Es ist lediglich die Rede von Commitment. In der SCRUM-Community gibt es verschiedenen Interpretationen, die an dieser Stelle jedoch keine Rolle spielen.

Interessant ist in der Regel, ob das Sprint-Ziel erreicht werden konnte oder nicht. Das Sprint-Ziel zu erreichen ist umso einfacher, je:

- genauer die Schätzungen sind,
- geschützter der Sprint ist,
- genauer die Sprintkapazität bekannt ist.

Die Genauigkeit von Schätzungen ist ein breites Thema, das besser separat behandelt werden sollte. Team-Time-Commitment (TTC) adressiert die letzten beiden Punkte. Der Schutz des Sprints wird oft nur aus Sicht der Aufgaben im Sprint betrachtet, es gibt jedoch einen zweiten Angriffspunkt: die Mitglieder im Team[6]. Würden Teammitglieder dauerhaft entfernt, dann würde das Team explizit darauf reagieren und sich möglicherweise neu organisieren. Unmerklicher sind kleine Veränderungen, z.B. wenn ein Entwickler einige Stunden bei einem anderen Projekt aushelfen soll, weil er dort vor Jahren die Grundlagen geschaffen hat oder einfach nur Expertise zu einer Technologie besitzt, die im anfordernden Projekt fehlt. Genauso ist aber auch ein Teamzuwachs möglich. Beispielweise könnte für eine Story stundenweise ein Mitarbeiter aus dem Infrastrukturteam das Entwicklungsteam unterstützen. Hier grenzt sich Sprint-Schutz und Sprint-Kapazität ab. Waren die Änderungen am Team geplant, ist der Sprint geschützt und die Kapazität korrekt geplant. Bei der Planung der Sprint-Kapazität werden gut planbare Termine, wie Urlaub und Feiertage berücksichtig. Weniger gut planbare Termine, wie Elternzeitbeginn oder Krankheitstage, stellen eine Herausforderung dar.

Rückblickend können das geplante TTC mit dem tatsächlichen TTC verglichen werden. Abweichungen in beide Richtungen

sind zu untersuchen. Wurde signifikant mehr Zeit eingebracht als geplant, könnte dies auf verdeckte technische oder fachliche Probleme hindeuten. Die Ursache kann auf Aufgabenebene gesucht werden. Wurde signifikant weniger eingebracht als geplant, deutet dies auf Störungen des Teams hin.
Vorausschauend kann TTC verwendet werden, um eine bessere Grundlage für die Planung der Sprint-Kapazität zu erhalten.

Abgrenzung von Velocity

Die Metrik Velocity beschäftig sich mit der Leistung des Teams bezogen auf fertig gestellte Aufgaben im Sprint-Zeitraum.

$$Leistung = \frac{\Delta\, Arbeit}{\Delta\, Zeit} \rightarrow Velocity = \frac{Tasks\ Completed}{Sprint\ Timeframe}$$

TTC hingegen bezieht sich auf die von Team eingebrachte Zeit. Diese Größe kommt bei der Berechnung der Velocity nicht zum Tragen.

Variable	Beschreibung
$t_{m,b,p}$	Eingebrachte Arbeitszeit in Stunden eines Teammitglieds (m) im Betrachtungszeitraum (b) für das zu untersuchende Projekt (p)

(Tabelle 2)

Team-Time-Commitment := $\sum_m t_{m,b,p}$

Der Wertebereich der Metrik stellt sich wie folgt dar:

```
WTTC = [u;o]
```

```
u = 0
```

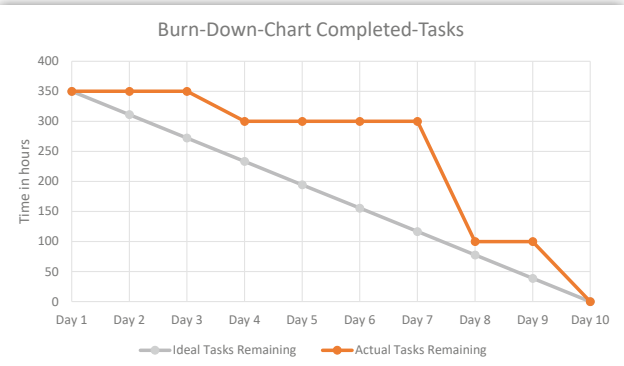
Das Team hat keine Zeit eingebracht.

```
o = |m| * |b|
```

Alle Teammitglieder haben ununterbrochen am Projekt gearbeitet. Dieser Wert wird mit steigender Dauer von **b** zunehmend unwahrscheinlich, da Menschen schlafen müssen. Die Metrik ergibt einen Zeitwert. Als Einheit wird Stunden vorgeschlagen.

Über die Kostenstellenzuordnung kann die investierte Zeit des Teams für den Betrachtungszeitraum ermittelt werden. Durch das Zusammenführen der Aufgaben aus dem PTS und den dazugehörigen Zeiten aus dem Zeiterfassungssystem lässt sich TTC berechnen.

TTC lässt sich in einer Abwandlung des Burn-Down-Charts darstellen, sodass das Team während des Sprints ein anschauliches Feedback bekommt. Das Burn-Down-Chart ist eine weitverbreitete Visualisierung für agile Softwareentwicklung. Herkömmlich wird der verbleibende Aufwand in einem Projekt in Relation zur verbleibenden Zeit dargestellt. Ein Beispiel für ein Burn-Down-Chart zeigt (Abb. 1).



Burn-Down Chart Completed Tasks (Abb.1)

Element	Beschreibung
x-Achse	Zeitlicher Verlauf
y-Achse	Verbleibender Aufwand im Betrachtungszeitraum. Der geschätzte Restaufwand zu jedem Zeitpunkt wird an dieser Achse gemessen.
Startpunkt	Am weitesten links liegender Punkt im Diagramm, der Beginn des Betrachtungszeitraums.
Endpunkt	Am weitesten rechts liegenderDer äußerste rechte Punkt im Diagramm, der das Ende des Betrachtungszeitraums markiert. Sein Abstand zum Startpunkt ergibt sich aus dem gesamten Aufwand im Betrachtungszeitraum geteilt durch die täglich leistbare Menge an Arbeit.
Ideal Tasks Remaining	Eine gerade Linie von Start- zu Endpunkt, die die idealisierte Annahme, der Aufwand im Projekt würde über die Zeit gleichmäßig geleistet werden, repräsentiert. Am Startpunkt zeigt die Ideallinie den geschätzten gesamten Aufwand im Betrachtungszeitraum. Am Endpunkt trifft sie die X-Achse, denn dann verbleibt kein Aufwand.
Actual Tasks Remaining	Diese Linie zeigt den tatsächlich verbleibenden Restaufwand. Zu Anfang und, bei Einhaltung des Endtermins auch am Ende, liegt diese Linie gleichauf mit der Ideallinie.

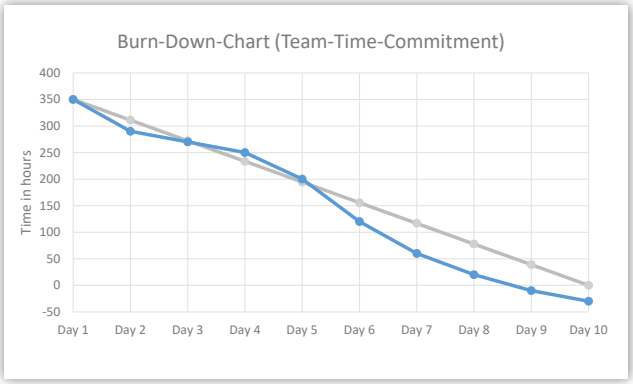
Erläuterungen zum Burn-Down-Chart – Completed-Tasks (Abb. 1) - (Tabelle 3)

Wenn Actual-Tasks-Remaining oberhalb des Ideal-Tasks-Remaining liegt, ist noch mehr zu erledigen als geplant war, sodass das Projekt hinter dem Zeitplan liegt. Ist das Gegenteil der Fall, dann ist weniger Aufwand notwendig als geplant war. Damit liegt das Projekt vor dem Zeitplan.

Im Gegensatz zu dem herkömmlichen Burn-Down-Chart mit der Darstellung der Completed-Tasks, wird beim Burn-Down-Chart mit der Abbildung von TTC die geleistete Zeit des Teams in Stunden beschrieben. Damit werden zwei wesentliche Effekte erzielt:

1. Es wird sichtbar, was bzw. wie viel Zeit das Team geleistet hat.
2. Es wird sofort sichtbar wo das Team steht. Die Darstellung von Completed-Tasks ist jedoch träge, da erst nach Abschluss eines Tasks eine Veränderung sichtbar wird.

(Abb. 2) zeigt ein Beispiel dafür:



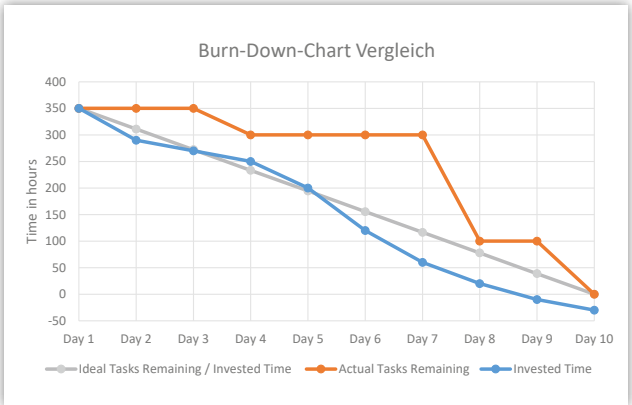
Burn-Down-Chart (Team-Time-Commitment) (Abb. 2)

Das Burn-Down-Chart für TTC ist analog zu dem oben beschrieben Burn-Down-Chart zu interpretieren, mit folgenden Ausnahmen:

Element	Beschreibung
Ideal Invested Time	Eine gerade Linie von Start- zu Endpunkt, die die idealisierte Annahme, die investierte Zeit im Projekt würde über die Zeit gleichmäßig geleistet werden, repräsentiert. Am Startpunkt zeigt die Ideallinie die geplante Gesamtzeit im Betrachtungszeitraum.
Actual Invested Time	Diese Linie zeigt die tatsächlich investierte Zeit.

Erläuterungen zum Burn-Down-Chart - Completed-Tasks in (Abb. 2) - (Tabelle 4)

Obwohl die Linie ideal genannt wird, muss es im konkreten Projekt nicht richtig sein, ihr im gesamten zeitlichen Verlauf möglichst genau zu folgen. Dennoch hilft die Ideallinie den Projektfortschritt einzuschätzen. Im Beispiel liegt Actual-Invested-Time zum Ende des Betrachtungszeitraums unter Ideal-Invested-Time. Das bedeutet, das Team hat mehr Zeit investiert als geplant.



Burn-Down-Chart Vergleich Completed-Tasks“ und Team-Time-Commitment (Abb. 3)

In **(Abb. 3)** werden die beiden Serien Completed-Tasks und Invested-Time in einem Burn-Down-Chart gezeichnet. Im Vergleich lässt sich gut erkennen, wie unterschiedlich die Werte sind. Completed-Tasks zeigt zwischen Tag vier und Tag sieben keine Veränderung und vermittelt den Eindruck, dass das Team keine merklichen Fortschritte macht. Invested-Time hingegen verändert sich täglich. Das Team scheint in diesem Beispiel am Tag fünf zu erkennen, dass es mehr Zeit einbringen muss, um das Sprint-Ziel zu erreichen.

Metriken – Das Sprint-Profile (SP)

Agile Softwareentwicklung möchte Kundenzufriedenheit durch die schnelle Lieferung von funktionierender Software erreichen. Im Idealfall wird möglichst viel Arbeitszeit in die Entwicklung von Features gesteckt, da diese den Wert der Software steigern. Arbeitsaufwand, der in die Behebung von Bugs oder die Bearbeitung von Hotfixes fließt, vermindert die Zeit, die für Features zur Verfügung steht. Hinzu kommt die Zeit, die das Team in Meetings verbringt. Hierzu zählen z.B. Daily-SCRUM, Sprint-Retrospektive und Sprint-Review.[7] Das Team soll die Möglichkeit erhalten, Sprints in Bezug auf den Aufgabentyp zu vergleichen und Schlussfolgerungen daraus zu ziehen.

Mit diesen Informationen kann das Team folgende Sprints besser planen, da Erfahrungswerte für die Zeitanteile je Aufgabentyp aus den letzten Sprints vorliegen.[8] Auch Überlegungen zum Thema Qualität können angestellt werden, zumal viele Bugfixes ein Indikator für mangelnde Qualität sein können.

Variable	Beschreibung
$t_{m,b,p,a}$	Eingebrachte Arbeitszeit in Stunden eines Teammitglieds (m) im Betrachtungszeitraum (b) für das zu untersuchende Projekt (p) für den Aufgabentyp (a)

(Tabelle 5)

Element	Beschreibung
Achsen	AchsenJe Aufgabentyp wird eine Achse eingeführt. Auf der Achse werden die Stunden pro Aufgabentyp von innen nach außen angetragen.

(Tabelle 5)

$$\texttt{Sprint Profile a} := \sum_m t_{m,b,p,a}$$

Der Wertebereich der Metrik stellt sich wie folgt dar:

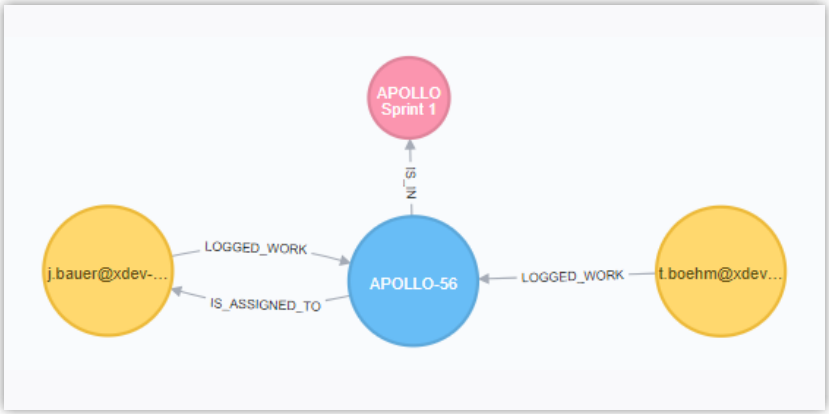
```
WSP = [u;o]
u = 0
```

Das Team hat keine Zeit für den Aufgabentyp eingebracht.

```
o = |m| * |b|
```

Alle Teammitglieder haben ununterbrochen am Aufgabentyp gearbeitet. Dieser Wert wird mit zunehmender Dauer von **b** zunehmend unwahrscheinlich, da Menschen schlafen müssen.
Die Metrik ergibt einen Zeitwert. Als Einheit wird Stunden vorgeschlagen.

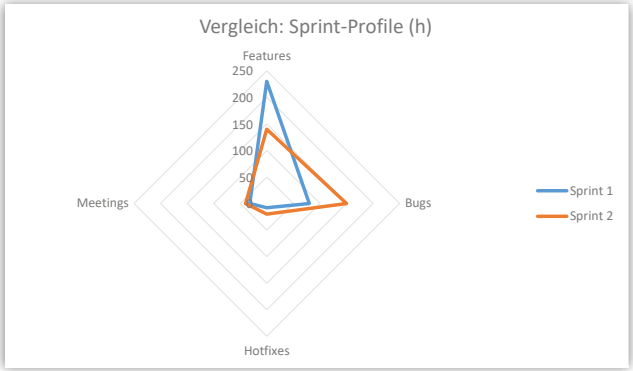
Über die Kostenstellzuordnung kann die investierte Zeit des Teams für einen Sprint ermittelt werden. Diese Zeiten müssen mit den Daten aus dem PTS angereichert werden, um eine Zuordnung zum Aufgabentyp zu erhalten.



Zuordnung von Zeiterfassung und Aufgabentyp (Abb. 4)

In (Abb. 4) ist beispielhaft die Aufgabe **APOLLO-56** (blauer Knoten) aus dem **Sprint 1** (rosa Knoten) dargestellt. Zugeordnet sind die Zeiteinträge zweier Mitarbeiter, die an dieser Aufgabe gearbeitet haben (gelbe Knoten). Der Aufgabentyp ist wiederum mit der Aufgabe verknüpft. Somit lassen sich alle notwendigen Daten für SP ermitteln.

Als Visualisierung wird ein Netzdiagramm vorgeschlagen, da sich damit mehrere Profile gut vergleichen lassen.



Vergleich Spint Profile (h) (Abb. 5)

(Abb. 5) zeigt, zwei Sprint- Profile im Vergleich. **Sprint 1** hat ein wünschenswertes Profil, es wurde viel Zeit in Features investiert und wenig Zeit in andere Aufgaben. Im Gegensatz dazu steht das Profil von **Sprint 2**, es wurde mehr Zeit in Bugs investiert als in Features.

Metriken – Team-Homogeneity (TH)

Das Team soll bezüglich seiner Arbeits- bzw. Aufgabenhomogenität untersucht werden. Dies lässt sich in zwei Dimensionen betrachten: Technisch (horizontal) und fachlich (vertikal).

Bei der technischen Dimension wird zwischen den horizontalen Elementen eines Produkts unterschieden, z.B.: Frontend, Backend, Tests, Dokumentation, Framework A, Framework B, etc.

Bei der fachlichen Dimension wird zwischen den vertikalen Elementen eines Produkts unterschieden, z.B.: Feature A, Feature B, Module C, Module D, etc.

Für beide Dimensionen gilt, um so gleichmäßiger die Aufgaben über das Team verteilt sind, um so homogener ist das Team. Wird jedes Element nur von einem Entwickler bearbeitet, ist das Team vollständig heterogen. Beide Extreme sind nicht erstrebenswert.

Vollständige Heterogenität bedeutet, dass spezifisches Wissen nur an einer Stelle im Team existiert (Single-Point-of-Failure). Ist ein Teammitglied nicht verfügbar, fehlt dem Team ersatzlos das Wissen um einen Teil des Produkts. Die Gründe für Engpässe in

der Verfügbarkeit von Teammitgliedern sind vielfältig: Urlaub, Elternzeit, Krankheit, Team- oder Arbeitgeberwechsel sind mögliche Faktoren. Um weiterarbeiten zu können, muss das Team das Wissen ohne den designierten Spezialisten erarbeiten oder warten, bis dieser wieder zur Verfügung steht. Somit bringt vollständige Heterogenität Risiken für die Planbarkeit neuer Features sowie für die Durchführung von Wartungsarbeiten.
Vollständige Homogenität bedeutet, dass jedes Teammitglied allumfassendes Wissen über die technischen bzw. fachlichen Gegebenheiten des Produkts besitzt (Total-Knowledge-Sharing). Ist ein Teammitglied nicht verfügbar, können dessen Aufgaben durch ein beliebiges anderes Teammitglied übernommen werden. Der hohe Grad der Redundanz bzw. die Kosten in Form von Zeit, um diese Redundanz herzustellen, stehen in Konkurrenz zum Ziel Business-Value in Form von Features zu liefern. Somit birgt vollständige Homogenität das Risiko, dass dem Team wenig Zeit bleibt, um neue Features zu entwickeln.

In der Regel liegt die Teamhomogenität zwischen den Extremen. Folgende Motivationen können bestehen, um die Teamhomogenität zu messen, zu beobachten und in eine Richtung zu entwickeln.

Bei SCRUM gibt das Product-Backlog bzw. das Sprint-Backlog die Priorität und damit die Reihenfolge der Aufgaben vor.[9] Im Idealfall kann ein Team im Sprint genau nach dieser Reihenfolge die Aufgaben bearbeiten. Je höher die Heterogenität im Team ist, desto höher ist die Wahrscheinlichkeit, dass die Aufgaben nicht in der vorgegebenen Reihenfolge erledigt werden können. Es entstehen Knowledge-Bottlenecks, die dazu führen, dass das Team Aufgaben überspringen oder warten muss. Somit wird das Team nicht dem Anspruch gerecht, die Features zu liefern, die am dringendsten benötigt werden.

Eine weitere Motivation ist der Bus-Factor oder Truck-Number, eine Kennzahl die Coplien[10] zur Abschätzung von Risiken in Software-Projekten vorschlägt. Der Wert gibt die Wahrscheinlichkeit des Scheiterns eines Projekts bei Ausfall eines Mitarbeiters an.:

„How many or few would have to be hit by a truck (or quit) before the project is incapacitated?“[11]

Die Ermittlung dieses Faktors wird in der Regel aus dem Kopf durchgeführt. Frei nach der Überlegung: „Wer müsste ausfallen, damit wir ein Problem haben?“. Dabei besteht das Risiko, dass man nicht ausreichend informiert ist und eine falsche Annahme trifft. Sicherer ist eine Herleitung des Faktors über die Teamhomogenität. Je nach Qualität der Datenbasis kann man so sehr genaue Aussagen erhalten.

Weiterhin kann Teamhomogenität helfen, Sub-Teams, also Teams im Team, zu identifizieren. Möglichweise arbeiten zwei Untergruppen im Team stark disjunkt an fachlichen oder technischen Themen. Mit dieser Information kann man entweder erkennen, dass es sich tatsächlich um zwei Teams handelt und die Teams trennen oder man arbeitet daran, dass aus zwei Teams eins wird.

Variable	Beschreibung
k	Komponente: Was als Komponente angesehen wird, kann für das jeweilige Projekt definiert werden. Die erste Instanz ist eine Unterscheidung in technische bzw. fachliche Ebene. Anschließend können projektspezifische Komponentengrenzen definiert werden. Eine fachliche Komponente in einer E-Commerce-Anwendung könnte z.B. der Warenkorb sein.
tk	Zeit, die für die Arbeit an der Komponente k vom Team aufgebracht wurde. Im Idealfall fließen hier die Zeiten aller Aufgaben und derer Prozessschritte ein, d.h. auch Spezifikation und Tests.
mk	Anzahl der Teammitglieder, die an der Komponente k gearbeitet haben.
mall	Anzahl der Teammitglieder (die dem Projektteam angehören)
hk	Teamhomogenität für die Komponente k

(Tabelle 7)

Erster Ansatz für die Berechnung der Teamhomogenität:

$$h1_k := \frac{m_k}{m_{all}}$$

Diese Berechnung ist nicht nach Arbeitszeit bzw. Aufwand gewichtet. Dies bedeutet, dass ein Teammitglied nur eine Code-Zeile ändern muss, um mit dem gleichen Gewicht in die Berechnung einzugehen, wie ein anderes Teammitglied, welches die Komponente seit Jahren betreut. Der Anteil der Teammitglieder wird im Folgenden mit der aufgewendeten Arbeitszeit gewichtet, um eine Abbildung näher an der Realität zu bekommen. Dabei steht die Annahme im Raum: Arbeitszeit an einer Komponente und Kenntnis der Komponente sind direkt proportional zueinander.

$$\mathrm{TH_k} := \prod_m \left(1 + \frac{t_{m,k}}{t_k}\right)$$

Diese Formel hat den Vorteil, dass die Arbeitszeit als Gewichtung einfließt und gleichzeitig die Werte beim Vergrößern des Teams stabil bleiben. Der Definitionsbereich und Wertebereich der Metrik stellen sich wie folgt dar:

$D_{TH}(m) = [2;\infty[$

WTH = [u;o]

u = 2

Das Team ist vollständig heterogen.

$$o = \left(1 + \frac{1}{m_{all}}\right)^{m_{all}}$$

Das Team ist vollständig homogen.

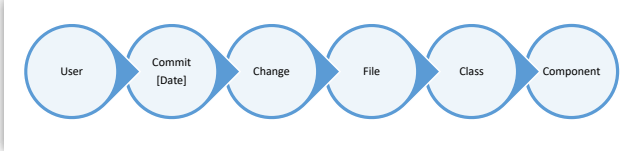
Zur Vereinfachung der Darstellung kann der Wertebereich wie folgt auf WTH = [0;1] korrigiert werden:

$$TH_{k,\,korrigiert} := \frac{TH_k - 2}{o - 2}$$

Die Metrik ergibt somit einen Wert zwischen 0 und 1. Es wird eine Darstellung in Prozent empfohlen.

Datenquellen – Das Version-Control-System (VCS)

Über das VCS lässt sich nachvollziehen, welches Teammitglied wann Änderungen an welchen Dateien vorgenommen hat. Dateien lassen sich auf Klassen abbilden. Klassen wiederum können Komponenten zugeordnet werden.[12] (Abb. 6) zeigt den Datenpfad von einem User zu einer Komponente.



Zusammenhang von Änderung der Codebasis und Komponenten (Abb. 6)

Diese Variante der Datengewinnung funktioniert gut in Projekten, deren Implementierung eine Zuordnung von Klassen zu Komponenten aus der Code-Basis zulassen. Beispiele hierfür sind Java oder C#, die über eine Paket- bzw. Namespace-Struktur eine Hierarchie erzeugen, über welche üblicherweise Komponenten abgebildet werden.

Datenquellen – Das Project-Tracking-System (PTS)

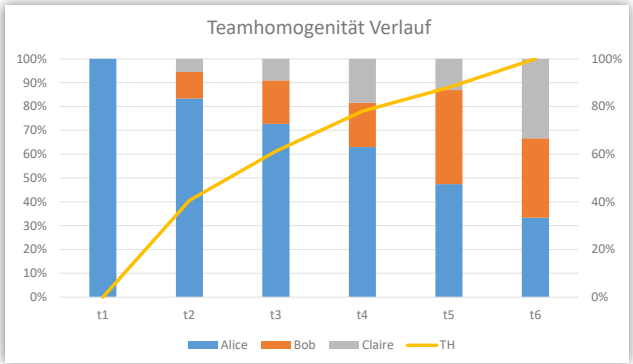
Wenn eine Herleitung des Komponentenbezugs aus dem VCS nicht oder nicht zufriedenstellend möglich ist, können diese Informationen ggf. auch alternativ oder ergänzend aus dem PTS gewonnen werden. Viele PTS sehen in der Standardkonfiguration Datenfeld „Komponente" vor. Dies ist die offensichtlichste Anlaufstelle. Über Metadaten wie Labels oder Tags kann auch ein Bezug zu Komponenten hergestellt werden. Manche Teams arbeiten auch mit Hashtags in Kommentaren. Durch die Analyse der Hashtags in Kommentaren lässt sich ebenfalls ein Komponentenbezug herstellen.

Die Qualität der Daten, die aus dem PTS gewonnen werden, ist dabei stark abhängig von der Disziplin des Teams, Labels und (Hash-) Tags überhaupt und zusätzlich korrekt zu setzen. Die Daten aus dem PTS können mit den Daten aus dem VCS kombiniert werden, um bessere Ergebnisse zu erzielen und ggf. Inkonsistenzen zu erkennen.

Als Visualisierung für **TH** wird ein kombiniertes Diagramm bestehend aus einem gestapelten Säulendiagramm (100 Prozent) und einem Liniendiagramm vorgeschlagen. Über das Liniendiagramm wird der Verlauf von **TH** dargestellt.

Element	Beschreibung
x-Achse	Zeitlicher Verlauf
y-Achse (links): Säulen	Arbeitsanteile der Teammitglieder in Prozent, die zum jeweiligen Zeitpunkt (gestapelt) als Grundlage für die Berechnung von TH stehen
Y-Achse (rechts): Linie	Teamhomogenität in Prozent

(Tabelle 8)



Teamhomogenität Verlauf (Abb. 7)

In (Abb. 7) wird beispielhaft die TH für ein Team bestehend aus Alice, Bob und Claire gezeigt. Über den Verlauf der Messpunkte wird die Veränderung der **TH** des Teams von vollständiger Heterogenität zur vollständigen Homogenität sichtbar. Im Beispiel wird die betrachtete Komponente zum Zeitpunkt der ersten Messung (**t1**) von Alice alleine betreut. Im Verlauf der Zeit beteiligen sich Bob und Claire zunehmend an den Arbeiten an der betrachteten Komponente. Zum Zeitpunkt der letzten Messung (**t6**) haben alle Teammitglieder schließlich gleiche Zeitanteile. Das Team ist damit für die betrachtete Komponente vollständig homogen. Verlässt ein Mitarbeiter das Team, sollten dessen Arbeitsanteile

nicht mehr in die Berechnung einfließen, da dessen Knowhow dem Team nicht mehr zur Verfügung steht. Zugänge zum Team, die nicht an der betrachteten Komponente arbeiten, haben durch die Art der Berechnung keinen Einfluss auf das Ergebnis.

Mit der Idee, durch Introspektion an sich selbst zu arbeiten und neue Erkenntnisse zu erlangen, setzen sich bereits um 400 v. Chr. griechische Philosophen wie Sokrates und Platon auseinander. „Gnothi seauton!" (griechisch: Erkenne dich selbst!)

Als Grundlage dafür wurde die Erkenntnis um das eigene Unwissen vorausgesetzt. Erst anschließend könne man sich daran machen, sich selbst zu verstehen. Die richtige Metrik - zur richtigen Zeit - mit einer sinnvollen Visualisierung kann ein guter Einstiegspunkt für nachhaltige Veränderungen für ein Entwicklungsteam sein. Die konzipierten Metriken können Softwareentwicklungsteams helfen, die Metaebene ihrer täglichen Arbeit besser zu verstehen.

Die persönlichen Meinungen zu verschiedenen Metriken gehen weit auseinander, es zeigt sich jedoch, dass alleine schon das Gespräch über Metriken meist sehr fruchtbar ist. Gespräche mit Kunden und Kollegen zum Thema Metriken zeigen immer wieder, dass ein großes Interesse an diesem Thema besteht. Daher kann ich nur ermutigen, über Metriken zu sprechen und getreu dem agilen Motto „Inspect and adapt" zu erforschen, was daraus erwächst.

Links:

1 Christopher Davies: "Agile Metrics in Action: How to Measure and Improve Team Performance."
2 Norman Fenton, James Bieman: „Software Metrics: A Rigorous and Practical Approach."
3 Gamifiction: https://de.wikipedia.org/wiki/Gamification
4 Brian Burke: „Gamify How Gamification Motivates People to Do Extraordinary Things"
5 David Nicolette: „Software Development Metrics"
6 Sven Röpstorff, Robert Wiechmann: „Scrum in der Praxis: Erfahrungen, Problemfelder und Erfolgsfaktoren"
7 Stefan Roock, Wolf Henning: „Scrum – verstehen und erfolgreich einsetzen"
8 Mike Cohn: „Agile Estimating and Planning"
9 Jeff Sutherland, Ken Schwaber: http://www.scrumguides.org/scrum-guide.html
10 Coplien James, Neil Harrison: „Organizational Patterns of Agile Software Development"
11 Laurie Williams, Robert Kessler: Pair Programming Illuminated
12 Dirk Mahler: "Shadows Of The Past: Analysis Of Git Repositories -jQAssistant" – https://goo.gl/hVM2Ni
13 Rafael Ferber, Platon: „Apologie des Sokrates"

eclipse FOUNDATION MEMBER

RAPID clipse™

THE VISUAL ECLIPSE

Eclipse Distribution

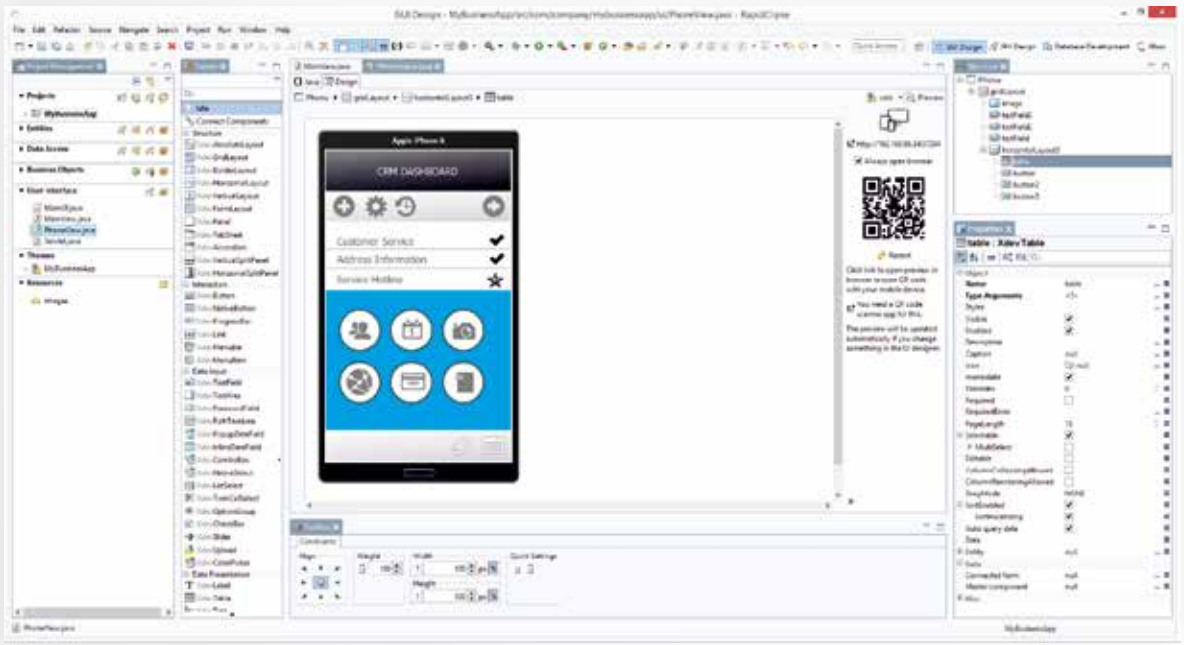
Alle wichtigen Eclipse-Plugins und viele Zusatz-Tools sind bereits vorinstalliert, vorkonfiguriert, auf einander abgestimmt. Keine aufwändigen Eclipse-Konfigurationen mehr. Einfach installieren

Stark verbesserte Hibernate Tools

- Endlich fehlerfreier Hibernate Datenbank-Import, fehlerfreies Datentyp-Mapping für alle Datenbanken und fehlerfrei Entity-Klassen Generierung
- JPA-SQL: Queris in SQL Syntax schreiben und typsicheren JPA Criteriea Querycode erhalten
- Vollautomatisierters Hibernate Entity-Lifecycle-Management

Cross-Platform-Development

Jedes RapidClipse Projekt kann als Web-Anwendung, hybride Mobile App oder klassische Java Desktop-Applikation deployt werden.



RapidClipse macht Eclipse zur visuellen Java IDE.

for Windows & Linux

DOWNLOAD FREE !
www.rapidclipse.com

#JAVAPRO #Kotlin #IoT #MQTT

IOT-Security mit MQTT

MQTT ist das de facto Standardprotokoll für das Internet-of-Things. Die Eclipse-Paho-MQTT-Client-Bibliothek und das Plugin-System des MQTT-Brokers HiveMQ ermöglichen die Integrationen mit Java. Dieser Artikel bietet eine kurze Einführung in MQTT und zeigt, wie MQTT sicher verwendet werden kann.

Das Internet-of-Things (IoT) ist heutzutage in aller Munde. Selbst kleinste Geräte verfügen dank moderner Technik über Möglichkeiten zur Datenübertragung. Diese Geräte werden im Allgemeinen als Smart-Devices bezeichnet. Waschmaschinen, Toaster oder elektrische Rollläden – kaum ein technisches Gerät ist nicht dazu in der Lage sich mit dem Internet der Dinge zu verbinden. Beispielsweise werden gesammelte Daten zur Weiterverarbeitung an ein Backend-System geschickt oder Geräte reagieren auf einen Befehl, welcher über eine Handy-App ausgelöst wurde.

Autor:

Florian Raschbichler hilft IoT-Interessierten bei Problemlösungen rund um die Themen MQTT und HiveMQ. Als MQTT-Experte schreibt er Artikel und hält regelmäßig Workshops rund um das Protokoll. Er ist Supportverantwortlicher bei der dc-square GmbH, die hochskalierbare IoT-Lösungen wie den MQTT-Broker HiveMQ entwickelt und steht daher in ständigem Kundenkontakt und kennt die Herausforderungen aus erster Hand.

Homepage: www.dc-square.de
LinkedIn: https://www.linkedin.com/in/florian-raschbichler-46310413b
Twitter: fraschbi

Dieser Artikel stellt MQTT vor, das sich inzwischen als de facto Standard Protokoll für IoT-Anwendungen etabliert hat und betrachtet anhand der Beispiele Verschlüsselung, Authentifizierung und Autorisierung die Umsetzung von Sicherheitsmechanismen mit MQTT. Außerdem werden beispielhaft Wege aufgezeigt, diese Mechanismen mit Java zu implementieren.

MQTT – Geschichte und eine kurze Einleitung

Das MQTT-Protokoll wurde im Jahr 1999 von Andy Stanford-Clark (IBM) und Arlen Nipper (Arcom, jetzt Cirrus Link) entwickelt[1], um die Übertragung von Daten mit möglichst geringen Anforderungen für Batterie- und Bandbreitenbedarf für die Verbindung von Öl-Pipelines über eine Satellitenverbindung zu realisieren. Folgende Merkmale, die das zukünftige Protokoll erfüllen sollte, wurden spezifiziert:

- Einfache Implementierung,
- mehrere Übertragungsgarantien für die Datenübertragung,
- leichtgewichtig und Effizient beim Verbrauch von Bandbreite,
- Datenagnostik,
- Informationen zu Sessions sollen serverseitig gespeichert werden, damit diese nicht beim Wiederaufbau einer Verbindung neu gesendet werden müssen.

Heutzutage stellen diese Merkmale noch immer den Kern von MQTT dar. Der Fokus des Protokolls hat sich inzwischen allerdings von proprietären eingebetteten Systemen hin zu offenen Anwendungsfällen im Internet der Dinge gewandelt.

Nachdem MQTT mehrere Jahre für unterschiedliche Anwendungsfälle intern von IBM genutzt wurde, veröffentlichte IBM das Protokoll im Jahre 2010 in Version 3.1 frei von Lizenzgebühren. Seither ist es jedem möglich das Protokoll kostenlos zu implementieren und zu nutzen. Ein weiterer wichtiger Meilenstein, der es MQTT ermöglichte eine so wichtige Rolle für den Anwendungsbereich Internet der Dinge einzunehmen, war die Gründung des Paho-Projekts[2] unter dem Schirm der Eclipse Foundation.

Unter Paho wurden unterschiedlichste MQTT-Client-Implementierungen veröffentlicht, welche den Ausbau eines Ökosystems für das Protokoll ermöglicht und beschleunigt haben. Im Jahr 2013 wurde bekannt gegeben, dass MQTT unter OASIS[3] standardisiert werden sollte. Ein gutes Jahr später, am 29. Oktober 2014, wurde MQTT-Version 3.1.1 offiziell als OASIS-Standard zugelassen. Der Übergang von 3.1 zu 3.1.1 symbolisiert, dass es sich nur um einen Minor-Change handelt und nur kleine Veränderungen am Protokoll vorgenommen wurden. Das Hauptziel war es, MQTT schnellstmöglich zu standardisieren und von diesem Stand aus stetig zu verbessern.

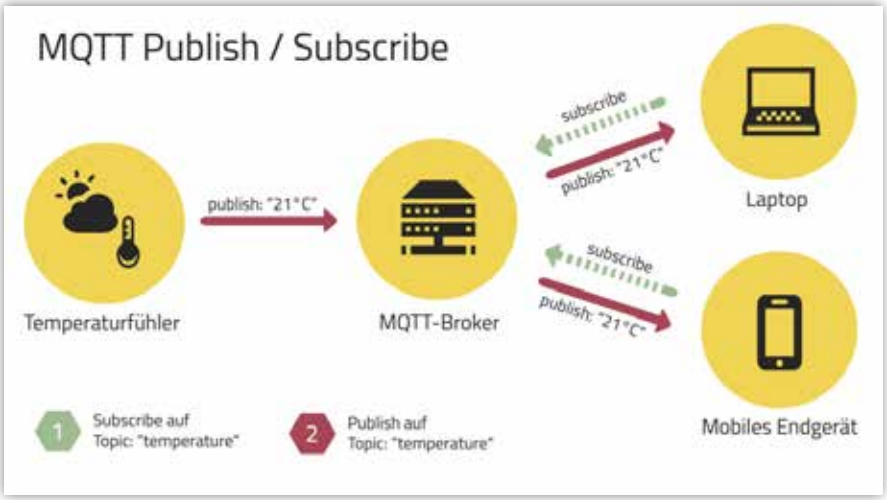
Grundprinzip von MQTT als Protokoll ist eine Publish-/Subscribe-Architektur. Anders als beispielsweise bei HTTP, mit seiner Request-/Response-Architektur, wird keine Ende-zu-Ende-Verbindung zwischen Sender und Empfänger aufgebaut. Sender und Empfänger von Nachrichten (Client) verbinden sich gleichermaßen mit einem zentralen Server (Broker). Das Senden (Publish) und Empfangen (Subscribe) von Nachrichten erfolgt über sogenannte Topics. Topics sind Strings, die vergleichbar mit URLs aufgebaut sind und den Betreff von Nachrichten darstellen. Beispielhaft könnte ein Temperaturfühler seine gemessenen Werte auf dem Topic `temperature` senden (publishen). Wichtig ist hierbei, dass es sich bei `temperature` nicht um die Adresse des Temperaturfühlers, sondern um einen Kommunikationskanal für alle Parteien handelt, die an der Temperatur interessiert sind **(Abb. 1)**.

Dieses Prinzip von Publish und Subscribe ermöglicht eine Push-gesteuerte Kommunikation. Das heißt, interessierte Parteien können vom Broker aktiv über neue Nachrichten informiert werden und müssen diese nicht ständig nachfragen, wie es bei Request-/Response-basierten Protokollen der Fall ist. Die Kombination aus Leichtgewichtigkeit und den daraus resultierenden niedrigen Anforderungen an die Bandbreite, Übertragungsgarantien, welche die Zustellung von Nachrichten auch unter nicht optimalen Voraussetzungen ermöglichen, eine Fülle an Bibliotheken für beinahe alle erdenklichen Systeme und Programmiersprachen sowie die Möglichkeit von Push-Notifikationen, haben unter anderem dazu geführt, dass sich MQTT bis dato als de facto Standardprotokoll im IoT etabliert hat.
Interessierte finden beispielsweise in der Blog-Serie MQTT-Essentials[4] detaillierte Informationen zu sämtlichen Features von MQTT.

Typische Anwendungsbereiche von MQTT umfassen die Vernetzung von Geräten beispielweise für sogenannte Smart-Home-Anwendungen. Auch beim Versenden von Sensordaten aus Regionen, welche schlecht vernetzt oder zugänglich sind, glänzt das Protokoll. Außerdem eignet sich die pushbasierte Kommunikation hervorragend für Chat-Anwendungen, weshalb beispielsweise der Facebook-Messenger[6] in Teilen auf MQTT setzt.

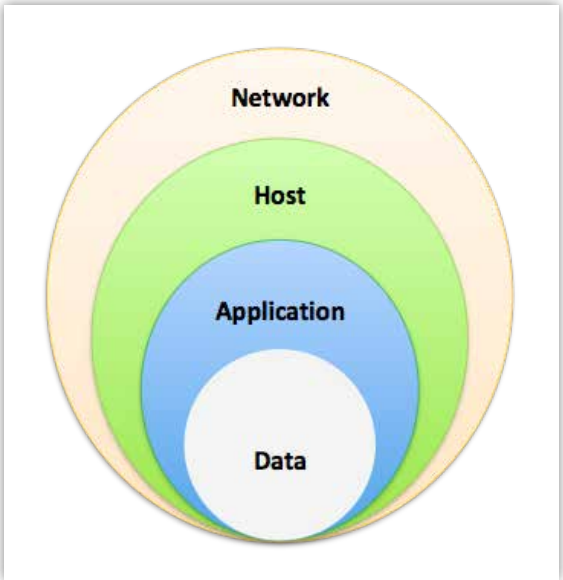
IT-Security-Prinzipien

Sicherheit in der IT ist ein wichtiges Thema. Grundprinzipien wie das CIA-Triad-Principle gelten im Internet-of-Things natürlich ebenso wie in der restlichen IT. **(Abb. 2)** verdeutlicht, visualisiert durch die verschiedenen Schichten einer Zwiebel, dass Sicherheitsmechanismen auf unterschiedlichen Ebenen angebracht werden müssen, um höchsten Sicherheitsansprüchen gerecht zu werden.



Publish-/Subscribe-Architektur von MQTT. (Abb.1)

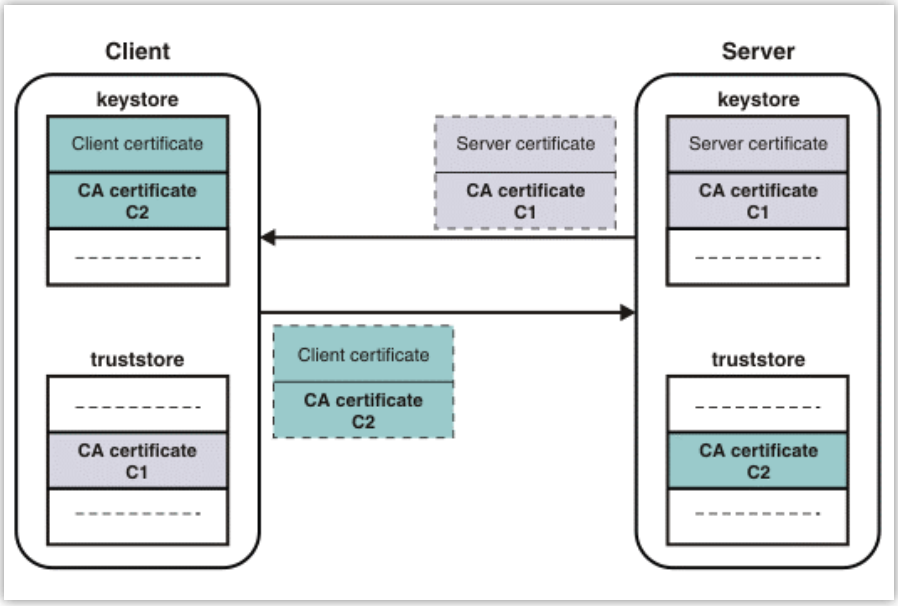
Dieser Artikel wird sich damit beschäftigen, wie Verschlüsselung, Authentifizierung und Autorisierung auf der Ebene der MQTT-Anwendung realisiert und unter Verwendung von Java auf Client- (Paho Java)[7] oder Brokerseite (HiveMQ-Plugin-System[8]) implementiert werden können. Die Beachtung von Sicherheitsstandards für weitere Teile des Gesamtsystems, durch beispielsweise standesgemäße Absicherung des Netzwerkes mit Firewalls oder auch des Betriebssystems der Host-Maschinen via SSH-Zugänge[9], werden vorausgesetzt.



Verschiedene Schichten der Sicherheit. (Abb. 2)

Verschlüsselung mit SSL/TLS

Auf der Anwendungsschicht des in (Abb. 2) aufgezeigten Zwiebelmodells eignet sich Transport-Layer-Security (TLS) und sein Vorgänger Secure-Sockets-Layer (SSL). Es handelt sich dabei um kryptografische Protokolle, die eine sichere Kommunikation in einem Computernetzwerk ermöglichen[10]. MQTT baut auf **`TCP/IP`** in der Transportschicht auf. Insofern werden keine Informationen unverschlüsselt versendet. SSL/TLS sorgt dafür, dass die Verbindung sicher wird, indem ein verschlüsselter Kommunikationskanal zwischen den Teilnehmern aufgebaut wird. Die Verwendung von SSL/TLS verhindert u.a. sogenannte Man-in-the-middle-Attacken[11], bei denen sich eine Dritte Partei, mit der Intention unverschlüsselte Daten auszulesen, als Kommunikationspartner ausgibt.



Key- und TrustStores (Abb.3)

Key- und TrustStores

Java verwendet Key- bzw. TrustStores (JKS)[12] um SSL/TLS-Zertifikate aufzubewahren. Im KeyStore befindet sich das eigene Zertifikat, welches für den SSL-Handshake angeboten wird, während im TrustStore die Zertifikate abgelegt sind, die als vertrauenswürdig eingestuft werden (vgl. (Abb. 3)).
Wird ein Zertifikat vom Kommunikationspartner verwendet, welches nicht von einer vertrauenswürdigen CA[13] ausgestellt wurde, dem aber vertraut werden soll, muss der Public-Key dieses Zertifikates in den TrustStore gelegt werden. (Abb. 3) visualisiert die Aufbewahrungsorte der Server- bzw. Clientzertifikate bei einer sogenannten Mutual-TLS-Verbindung. Also einer Verbindung, bei der sowohl Server- als auch Client-Zertifikate verwendet werden.

Um Zertifikate entsprechend der in (Abb. 3) abgebildeten Struktur aufzubewahren, gibt es unterschiedliche Tools, mit denen sich Key- und TrustStores erstellen bzw. editieren lassen. Beispielsweise das Kommandozeilentool Keytool[14] oder grafische Programme wie der Keystore Explorer[15].

Implementierung mit Paho

Das folgende Beispiel basiert auf der Annahme, dass auf Seiten des MQTT-Broker ein Serverzertifikat mit dem public key aus **`broker.jks`** hinterlegt wurde und dem Zertifikat in **`client.jks`** vertraut wird.

Es zeigt den Aufbau einer Mutual-TLS-Verbindung. Für die MqttConnectOptions wird eine SocketFactory gesetzt, in welche der JavaKeyStore **`client.jks`** als KeyStore (Hier liegt das Client-Zertifikat, das vom Client für den TLS-Handshake angeboten wird) und der JavaKeyStore **`broker.jks`** als TrustStore (Hier ist der public key des Server-Zertifikats hinterlegt, wodurch der Client weiß, dass er diesem Zertifikat vertrauen kann) geladen werden.

Listing 1

```
public static void main
(String... args) {
    try {
        String clientId =
        "sslTestWithCert";
        MqttClient client = new
        MqttClient("ssl://
        localhost:8883", clientId,
        new MemoryPersistence());

        MqttConnectOptions
        mqttConnectOptions =
        new MqttConnectOptions();

        try {
            mqttConnectOptions.
            setSocketFactory(get
```

```
        TruststoreFactory());
        } catch (Exception e) {
            log.error("Error while setting up TLS", e);
        }

        client.connect(mqttConnectOptions);
        client.publish("test", "test".getBytes(), 1, true);
        client.disconnect();

    } catch (MqttException e) {
        log.error("MQTT Exception:",e);
    }
}

public static SocketFactory getTruststoreFactory() throws
Exception {

    //Create key store

    KeyStore keyStore = KeyStore.getInstance("JKS");
    InputStream inKey = new FileInputStream("client.jks");
    keyStore.load(inKey, "your-client-key-store-password".
    toCharArray());

    KeyManagerFactory kmf = KeyManagerFactory
            .getInstance(KeyManagerFactory.
             getDefaultAlgorithm());
    kmf.init(keyStore,"your-client-key-password".toCharArray());

    //Create trust store

    KeyStore trustStore = KeyStore.getInstance("JKS");
    InputStream in = new FileInputStream("broker.jks");
    trustStore.load(in, "your-client-trust-store-password".
    toCharArray());

    TrustManagerFactory tmf = TrustManagerFactory
            .getInstance(TrustManagerFactory.getDefault
            Algorithm());
    tmf.init(trustStore);

    // Build SSL context

    SSLContext sslCtx = SSLContext.getInstance("TLSv1.2");
    sslCtx.init(kmf.getKeyManagers(), tmf.getTrustManagers(),
    null);
    return sslCtx.getSocketFactory();
}
```

Authentifizierung mit MQTT

Während die Verschlüsselung auf der Transportschicht garantiert, dass die ausgetauschten Informationen nur von den Kommunikationspartnern gelesen werden können, stellt Authentifizierung die Identität dieser Kommunikationspartner auf Applikationsebene sicher. Nicht nur in der Informationstechnologie sondern auch im echten Leben kennt man dieses Prinzip. Beim Check-In am Flughafen dient ein Reisepass als Authentifizierungsmittel. Am Computer weisen wir mehrmals täglich unsere Identität mit Hilfe einer Username-Passwort-Kombination nach. Das MQTT-Protokoll beinhaltet Felder für Username und Password, um die Identität des Clients am Broker überprüfen zu können. (Listing 2) zeigt wie die Verbindung eines Paho-Java-Clients, der Username und Passwort gesetzt hat, realisiert werden kann.

Listing 2

```
public void ConnectMQTTClientWithCredentials {

      MqttClient client = new MqttClient(
      "tcp://broker.mqttdashboard.com:1883", //URI
      MqttClient.generateClientId(), //ClientId
      new MemoryPersistence()); //Persistence

      MqttConnectOptions options = new MqttConnectOptions();
      options.setUserName("Fu");
      options.setPassword("Bar".toCharArray());
      client.connect(options);

}
```

Diese vom Client bei der Verbindung mitübertragenen Anmeldedaten können nun vom Broker ausgelesen werden. Beispielsweise kommt der MQTT-Broker HiveMQ[16] mit einem frei zugänglichen, auf Java basierten Plugin-System, mit dem der Nutzer individuell zusätzliche Geschäftslogik implementieren kann. Sogenannte Callbacks werden zu bestimmten Zeitpunkten der MQTT-Kommunikation oder durch bestimmte Ereignisse ausgelöst. Für die Authentifizierung existiert der sogenannte **`OnAuthentication Callback`**. Dieser wird ausgelöst, sobald ein Client einen Verbindungsversuch startet und eignet sich optimal dafür die Identität des Clients zu überprüfen.

(Listing 3) zeigt ein Beispiel bei dem der Broker überprüft, ob die Username-Passwort-Kombination des Clients **`Fu`** bzw. **`Bar`** entspricht. Nur in diesem Fall ist der Rückgabewert **`true`** und der Broker erlaubt die Verbindung. In allen anderen Fällen wird **`false`** zurückgegeben und der Broker beendet die Verbindung. Eine solch hart codierte Herangehensweise dürfte allerdings wenig praxistauglich sein. Java bietet Möglichkeiten beispielsweise eine Datei oder Datenbank als Quelle für die Identitätsprüfung zu nutzen.

Listing 3

```
public class UserAuthentication implements OnAuthenticationCall-
back {
...
    @Override
public Boolean checkCredentials(ClientCredentialsData clientData)
throws AuthenticationException {

      if (!clientData.getUsername().isPresent()) {
            return false; // kein Username
        }
      if (!clientData.getPassword().isPresent()) {
            return false; // kein Passwort
        }
      else {
if (clientData.getUsername().get().equals("Fu") && clientData.
getPassword().get().equals("Bar")) {
                return true; // richtige Kombination
            }
            return false; // falsche Kombination
        }
    }
...
}
```

Autorisierung mit MQTT

Authentifizierung und Verschlüsselung stellen sicher, dass Kommunikationspartner auch sind wer sie vorgeben zu sein und dass die Kommunikation nicht von Dritten abgehört werden kann. Autorisierung ist ein weiteres Sicherheitskonzept, das dafür sorgt, dass unterschiedliche Identitäten auch unterschiedliche Berechtigungen haben. Ähnlich wie beim Authentifizierungsbeispiel und dem Reisepass lässt sich auch die Autorisierung auf das echte Leben übertragen. Es zeigt sich, dass diese zusätzlich zur Authentifizierung sinnvoll sein kann. Zugang zu einem gesicherten Bereich wird beispielsweise erst gewährt, nachdem einerseits die Identität nachgewiesen wurde und andererseits sichergestellt wurde, dass diese Identität auch die nötige Berechtigung besitzt diesen Bereich zu betreten.

Für viele MQTT-Anwendungsfälle ist es ratsam, unterschiedlichen Teilnehmern auch unterschiedliche Zugriffe auf bestimmte Ressourcen zu gewähren. Unterschieden werden kann der Zugriff auf drei Ebenen:

- Topic,
- Art des Zugriffs (Publish und/oder Subscribe),
- Quality of Service Level (0,1,2).

Es wird zwischen zwei grundsätzlichen Prinzipien der Zugriffskontrolle unterschieden: Zwischen einem Blacklist- und einem Whitelist-Ansatz. Beim Whitelist-Ansatz ist alles verboten was nicht explizit erlaubt ist (auf der Whitelist steht). Beim Blacklist-Ansatz herrscht das umgekehrte Prinzip und alle Zugriffe, die nicht auf der Blacklist stehen, sind erlaubt. **(Listing 4)** zeigt einen Whitelist-Ansatz mit hart kodierten Werten für die Autorisierungsprüfung am MQTT-Broker-HiveMQ. Verwendet wird hierzu der **`OnAuthorizationCallback`** der Seitens des MQTT-Brokers jedes Mal ausgeführt wird, wenn ein Client auf einem Topic ein Subscribe oder Publish ausführen möchte.

Listing 4

```
public class WhitelistAuthorisation implements OnAuthorizati-
onCallback {

    @Override
    public List<MqttTopicPermission> getPermissionsForClient
    (ClientData clientData) {

        final List<MqttTopicPermission> permissions =
    new ArrayList<>();
    permissions.add(new MqttTopicPermission("client/" +
    clientData.getClientId(), TYPE.ALLOW, QOS.ONE, ACTIVITY.
    SUBSCRIBE)); // Subscribe mit QoS=1 auf topic "client/
    clientID" erlaubt.
    permissions.add(new MqttTopicPermission("client/#", TYPE.
    ALLOW, QOS.ALL, ACTIVITY.PUBLISH)); // Publish mit allen
    Quality of Service Leveln auf allen Topcis die mit "client/"
    beginnen erlaubt.

        return permissions;
    }

    @Override
    public AuthorizationBehaviour getDefaultBehaviour() {
        return AuthorizationBehaviour.DENY; // Whitelist =
    Autorisierung wird per Default abgelehnt.
    }
...
}
```

`Hinweis:` Da der **`OnAuthorizationCallback`** in der Praxis sehr häufig aufgerufen wird ist es sehr empfehlenswert, dessen Ergebnisse zu cachen und so für eine erhöhte Performance am Broker zu sorgen.

Fazit:

Dieser Artikel behandelt nicht das gesamte Spektrum an Sicherheitsmechanismen. Um bestmögliche Sicherheitsstandards zu gewährleisten ist es unumgänglich nicht nur die Anwendungsebene, in der sich MQTT wiederfindet, abzusichern. Die Beispiele Verschlüsselung, Authentifizierung und Autorisierung zeigen, dass es möglich ist gängige Sicherheitsmechanismen mit dem de facto Standardprotokoll für IoT-Anwendungen zu integrieren. Die Eclipse-Paho-MQTT-Client-Bibliothek und das Plugin-System des MQTT-Brokers HiveMQ ermöglichen diese Integrationen mit Java. MQTT ermöglicht die Implementierung weiterer Sicherheitsmechanismen. Beispielsweise ist es möglich die Payload der gesendeten Nachrichten gesondert zu verschlüsseln oder das Autorisierungsframework „OAuth2" zu verwenden . Abschließend kann resümiert werden, dass eine sichere Verwendung des MQTT-Protokolls und Java für Anwendungen im Internet der Dinge möglich ist.

Links:

1 IBM Podcast: https://goo.gl/dLpnxP
2 Eclipse paho: https://www.eclipse.org/paho/
3 OASIS: https://www.oasis-open.org/
4 MQTT-Blog: https://www.hivemq.com/mqtt-essentials/
5 IBM – "Why is facebook using MQTT on mobile?":
https://goo.gl/9Q4HCh
6 WhatIs.com – Das CIA-Triad-Principle: https://goo.gl/4RtpkY
7 https://www.eclipse.org/paho/clients/java/
8 HIVEMQ: https://www.hivemq.com/docs/plugins/latest/
9 https://de.wikipedia.org/wiki/Secure_Shell
10 https://de.wikipedia.org/wiki/Transport_Layer_Security
11 https://en.wikipedia.org/wiki/Man-in-the-middle_attack
12 https://www.digitalocean.com/community/tutorials/
java-keytool-essentials-working-with-java-keystores
13 https://en.wikipedia.org/wiki/Certificate_authority
14 https://docs.oracle.com/javase/6/docs/technotes/tools/
solaris/keytool.html
15 http://keystore-explorer.org/

https://www.hivemq.com/
https://oauth.net/2/
https://www.hivemq.com/blog/mqtt-security-fundamentals/

#JAVAPRO #AWS #Alexa

Alexa + Java = Fun

Mit AWS Lambda und einer Prise Java macht es richtig Spaß, Amazons Alexa neue Tricks beizubringen. Das ist kein Hexenwerk, sondern mit einfachem Java- und etwas AWS-Know-how leicht zu bewerkstelligen.

Alexa? Wer ist das denn?

Als Alexa bei uns eingezogen ist, war meine Frau nicht gerade erfreut. Neben Ihr sollte es keine Andere im Haus geben. Und das, obwohl es sich dabei eigentlich nur um einen schwarzen Zylinder handelt. Um etwas präziser zu sein: Alexa ist lediglich der Sprachassistent, der irgendwo in den Weiten des Internets existiert. Das Gerät heißt Amazon Echo. Es handelt sich um eine Lautsprecherbox und sieben Mikrofone, die auf das Netzwerk zugreifen können. Diese sieben Mikrofone lauschen beständig in den Raum hinein, ob jemand das Schlüsselwort **`Alexa`** sagt. Falls dem so ist, wird die aufgezeichnete Nachricht an Alexa geschickt, die dann hoffentlich weiterhelfen kann.

Autor:

Christian Grobmeier arbeitet selbst, ständig und gerne. Am liebsten mag er irgendwas mit Spring oder Solr. Er ist auch kein JavaScript-Kostverächter und treibt sich gerne mit Angular oder React rum. Hin und wieder greift er sogar zur Ruby und PHP Keule. Und wenn alles nichts mehr hilft, schreibt er halt ein Buch.

https://grobmeier.solutions
https://twitter.com/grobmeier
https://www.linkedin.com/in/grobmeier
https://github.com/grobmeier
cg@grobmeier.de

Datenschützer graust es hier bereits: Amazon hört mit, wenn auch nur kurz. Ständig. Immerhin muss man ja irgendwie das Schlüsselwort identifizieren. Man kann das Mikrofon ausschalten, doch wir wissen ja, dass die NSA auch ausgeschaltete Handys abhören kann. Das Grauen ist perfekt, wenn man sich Amazon Show ins Haus holt. Dort bekommt Alexa auch noch Bildschirm und Kamera. Offiziell um Modetipps zu geben. Aber was sonst noch so übertragen und wie es genutzt wird, kann man natürlich nie genau sagen.

Glücklicherweise sind wir zu Hause nicht paranoid. Trotzdem steht das Echo-Gerät nicht in besonders privaten Räumen wie dem Kinder- oder Schlafzimmer. Vertrauliche Gespräche im Büro darf Alexa auch nicht mithören.

Wer sich trotz aller Kritik mit Alexa anfreunden kann, der kann sie fragen: "Alexa, wie wird das Wetter heute?" Oder: "Wie viel Uhr ist es?" Das sind die langweiligen Dinge. Interessanter wird es, wenn die verschiedenen Anbieter wie die Deutsche Bahn Auskunft zum Zugfahrplan geben. Das was Alexa kann, nennt man Skills, also Fähigkeiten.

Von einem erfolgreichen, werbefinanzierten oder bezahlten Skill habe ich bisher noch nie gehört. Was nicht ist, kann ja noch kommen. Amazon bietet solche Einstellmöglichkeiten in seiner Entwicklerkonsole bereits an. Aber auch kostenfrei ist es natürlich ziemlich cool, sein Produkt oder Unternehmen via Voice-Support zu unterstützen. Man stelle sich vor: "Alexa, frage Dr. Mustermann ob er morgen noch Termine frei hat" Oder: "Alexa, frage das Café Chapeau ob es morgen geöffnet hat." Und so weiter. Sprachsteuerung kann nützlich werden, nicht nur im Auto.

Mit Java ist es sehr einfach, ebenfalls einen Skill zu entwickeln. Obwohl JavaScript in AWS Lambda offenbar Einwohner Nummer Eins ist.

„Hallo Alexa" - Begriffe

Alexa ist der Sprachassistent. Genauso wie Siri Apples Sprachassistent ist. Also nur eine Menge an Software, die Sprache versteht und antwortet. Das Gerät, dass im Folgenden verwendet wird, ist Amazon Echo. Es gibt auch Varianten wie Amazon Echo Dot (etwas kleiner, mit schlechterer Box) oder Amazon Show (mit Bildschirm). Mittlerweile gibt es weitere alexaunterstützte Geräte. Wie zum Beispiel Sonos One. Es handelt sich dabei um eine sprachgesteuerte HiFi-Box.

Den folgenden Code nennt man Skill. Ein Skill ist eine Fähigkeit. Irgendeine Funktionalität, die das Gerät erfüllen kann. Es gibt Standard-Skills, also Fähigkeiten die Alexa von vorneherein mitbringt. Üblicherweise meint man mit dem Wort Skill aber eine Eigenentwicklung, also einen Custom-Skill. Dieser Skill kann eine oder mehrere Intents, also Absichten beinhalten. So könnten Sie beispielsweise sagen: "Alexa, frage Crypto nach den letzten Preisen." Und Sie würden derzeitige Preise der gängigen Cryptowährungen erhalten. Dann wäre **`GetCurrencies`** also möglicherweise Ihr erstes Intent.

Sie könnten zusätzlich fragen: "Alexa, frage Crypto ob es den Euro noch gibt." Und Sie hätten möglicherweise ein zweites Intent namens **`CheckEuro`**. Der Begriff Intent ist ebenfalls in der Android-Entwicklung bekannt, wo er recht ähnlich verwendet wird.

Ablauf eines Alexa Aufrufs

Prinzipiell ist der Ablauf eines Alexa-Requests relativ einfach. Das Gerät lauscht wie gesagt immer ein paar Sekunden mit. Sobald es das Keyword **`Alexa`** hört, wird die Aufnahme zur Auswertung an den Alexa-Service gesendet. Das ist die Domain von Amazon. Ihre Anfrage wir dort auch gespeichert, analysiert und offenbar zur Verbesserung von Alexa verwendet.

Der Alexa-Service interpretiert die Anfrage und leitet dann optimalerweise den Text an Lambda weiter. Zumindest meistens. Denn obwohl Lambda normalerweise die beste Wahl ist und auch von Amazon empfohlen wird, könnte man theoretisch auch seinen eigenen Server dahinterklemmen und damit ohne Lambda arbeiten.

Amazon Web Services (AWS) bietet Lambda an, womit man serverlose Funktionen im Netz verfügbar machen kann. Manch einer verbindet den Begriff Microservice damit. Die Funktion wird komplett von Amazon gewartet und ist daher extrem stabil. Es ist also möglich, auch ohne einen AWS-Zugang einen Alexa-Skill zu erstellen. Da es aber nicht die empfohlene Vorgehensweise ist, hält sich diese Einführung ebenfalls an AWS.

Alexa-Flow – von Gerät zu Ihrem Code (Abb. 1)

Speechlet und Lambda-Code

Um Lambda verwenden zu können, benötigt man einen AWS-Zugang. Für Einsteiger bietet AWS ein kostenfreies Kontingent an. Um schnell einsteigen zu können, könnte man beispielsweise mit Maven ein Standardprojekt generieren.

mvn archetype:generate

Wer die Defaults verwendet, kann sich damit ein sehr einfaches Java-Projekt generieren, dem man nur noch die Abhängigkeit zum Alexa-Skills-Kit konfigurieren muss, wie in **(Listing 1)** ersichtlich ist.

(Listing 1)

```
<dependencies>
   <dependency>
       <groupId>com.amazon.alexa</groupId>
       <artifactId>alexa-skills-kit</artifactId>
       <version>1.3.0</version>
   </dependency>
</dependencies>
```

Mit dieser Abhängigkeit hat man alle notwendigen Klassen zur Verfügung um den Skill zu bauen.

Im Prinzip geht es nun um zwei Dinge: zunächst muss die eigentliche Alexa-Funktionalität gebaut werden. Und zweitens, diese Alexa-Funktionalität muss via Lambda bereitgestellt werden. Der interessantere Code ist in **(Listing 2)** zu sehen, worin es um die eigentliche Abhandlung der Voice-Anfrage geht.

(Listing 2)

```
public class CrypoSpeechlet implements Speechlet {

   public void onSessionStarted(
       SessionStartedRequest sessionStartedRequest,
       Session session) throws SpeechletException {
...
   }

   public SpeechletResponse onLaunch(
       LaunchRequest launchRequest,
       Session session) throws SpeechletException {
       ...
   }

   public SpeechletResponse onIntent(
       IntentRequest intentRequest,
       Session session) throws SpeechletException {
       ...
   }

   public void onSessionEnded(
      SessionEndedRequest sessionEndedRequest,
      Session session) throws SpeechletException {
      ...
   }
```

Hier ist zu sehen, wie wir die Methoden des **Speechlet** Interfaces implementieren. Das **Speechlet** wird von den JavaDocs als Web-Service beschrieben. Dieser Service wird üblicherweise noch gewrapped, z.B. durch das **SpeechletServlet** oder den **SpeechletRequestStreamHandler**. Ersteres ermöglicht dem **Speechlet** in Java-EE-Containern zu existieren, zweiteres bindet das **Speechlet** an AWS Lambda an.

Die Methoden sind nahezu selbsterklärend. Es wird ganz klar deutlich, dass wir uns nicht darum kümmern müssen, die Bytes aus den MP3s zu ziehen und auf komplexe Art und Weise auszuwerten. Das alles macht Amazon für uns. Und wenn es das getan hat, ruft Amazon unser **Speechlet** auf.

Die **onLaunch**-Methode wird aufgerufen, wenn der Skill gestartet wird ohne einen besonderen Befehl abzugeben. Ein Beispiel wäre: "Alexa, öffne Crypto." Dadurch würde Alexa eben den Skill in Bereitschaftsmodus versetzen. Würde der Crypto-Skill nur eine Sache machen, dann könnte hier der wichtige Teil der Anwendungslogik implementiert werden. Wenn der Skill allerdings mehr Informationen braucht, dann sollte man hier mit einer Antwort aufwarten und weitere Anweisungen geben.

onLaunch öffnet immer auch eine neue Session die im Aufruf von **onSessionStarted** mündet. Ganz generell wir **onSessionStarted** bei jeder Interaktion mit dem Gerät aufgerufen, da jede Interaktion als neue Sitzung gilt. Ähnlich dazu wird **onSessionEnded** verwendet. Dieser Callback wird aufgerufen, wenn der Benutzer nicht weiter mit dem Gerät kommuniziert oder den Skill beendet hat. Besonders interessant ist dann **onIntent**. Hier lebt der Großteil der Anwendungslogik, bzw. von hier aus werden die verschiedenen Funktionalitäten aufgerufen.

(Listing 3)

```
public SpeechletResponse onIntent(
       IntentRequest intentRequest,
       Session session) throws SpeechletException {

      String name = intentRequest.getIntent().getName();
   String speechText = "You called " + name;

   // Cards - für Amazon Show
   Card card = new SimpleCard();
   card.setTitle("Hello Crypto");
   card.setContent(speechText);

   // Sprachausgabe
   PlainTextOutputSpeech speech = new PlainTextOutputSpeech();
   speech.setText(speechText);

   return SpeechletResponse.newTellResponse(speech, card);

}
```

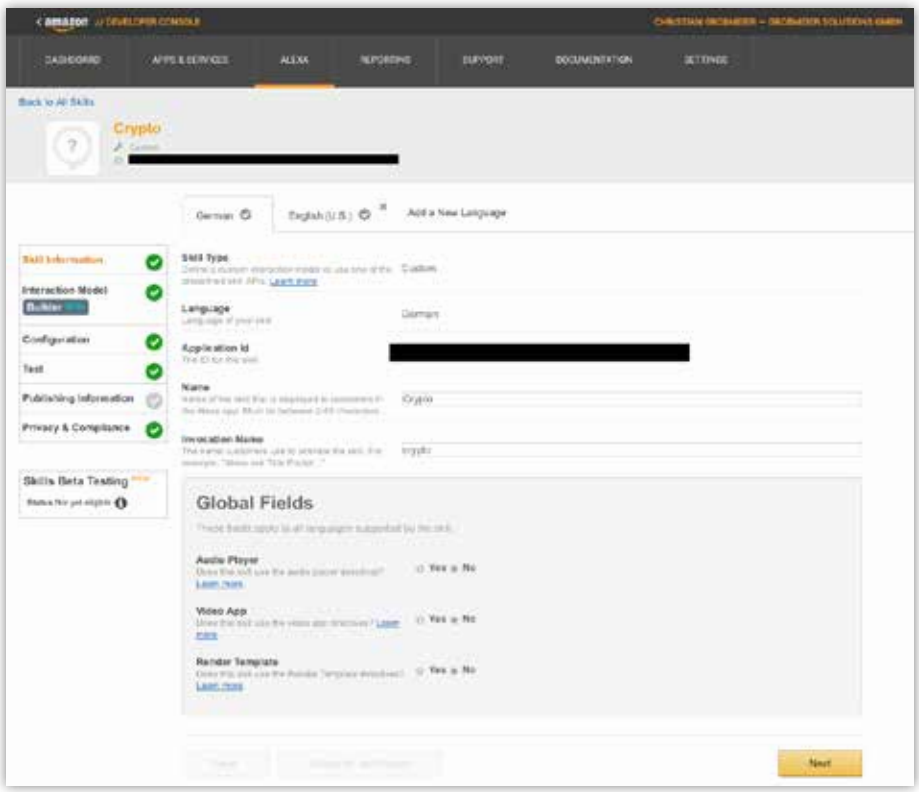
Eine beispielhafte Implementierung von **onIntent** findet man in **(Listing 3)**. Mittels dem **IntentRequest** Objekt können wir auf die Details des Requests zurückgreifen. In diesem Beispiel greifen wir lediglich auf den Namen zu. Es könnte sich hierbei um folgendes handeln: "Alexa, frage Crypto nach den Kursen." **Crypto** wäre der Skill, und **Kurse** ist der Name des Intents.

Weiter könnte man sogenannte Slots definieren. Das sind Platzhalter, die dem Request hinzugefügt werden. Zum Beispiel: "Alexa, frage Crypto nach den Kursen für Bitcoin." In diesem Fall wäre das Intent nach wie vor **Kurse**, aber der Begriff **Bitcoin** könnte auch mit **Ether** oder **Litecoin** ausgewechselt werden. Hier handelt es sich um einen Slot. Auch auf diese Slots könnte man im **Speechlet** mittels des **IntentRequest** zugreifen.

In dieser Implementierung ist lediglich der Name des Intents notwendig um daraus einen **String speechText** zu erzeugen. Der wird für zwei Fälle verwendet: Zum ersten, erzeugen wir eine

SimpleCard. Falls das Gerät einen Bildschirm besitzt, wird eine visuelle Ausgabe ebenfalls angezeigt. Das ist derzeit mit Amazon Show möglich. Im Anschluss wird die Sprachausgabe mit dem gleichen Text erzeugt. Obwohl es aufgrund der Neuartigkeit der Geräte vermutlich weniger Best-Practices gibt, ist es vermutlich sinnvoll für visuelle und Sprachausgabe den gleichen Text zu verwenden - ähnlich den Untertiteln im Fernsehen.

Zu guter Letzt wird die Antwort mit **`SpeechletResponse.newTellResponse(speech, card);`** erzeugt. Die Tell-Response ist dafür gedacht eine Antwort zu geben und ermöglicht keinen komplexen Workflow. Dem gegenüber steht die Ask-Response, welche die Möglichkeit bereitstellt Rückfragen an den Benutzer zu richten.



Skill erstellen (Abb. 2)

So weit so gut: Derzeit verfügt der Code bereits über eine Art Hello- World-Logik, doch es fehlt noch die Anbindung an Lambda. (Listing 4) zeigt den notwendigen Code.

(Listing 4)

```
public class CryptoSpeechletRequestStreamHandler
   extends SpeechletRequestStreamHandler {
   private static final Set<String> appIds =
   new HashSet<String>();

   static {
       appIds.add("<Ihre App ID>");
   }

   public CryptoSpeechletRequestStreamHandler() {
       super(new CrypoSpeechlet(), appIds);
   }
}
```

Im Prinzip muss lediglich von **`SpeechletRequestStreamHandler`** abgeleitet werden, der wiederum das erstellte **`Speechlet`** wrapped. Hinzu kommen eine oder mehrere App-IDs, die wir mit Java-Code nicht erzeugen können. Denn an dieser Stelle verlassen wir nun die Java-Welt und müssen mittels Amazon-Developer-Console und AWS-Management-Console die beiden Welten verdrahten und konfigurieren.

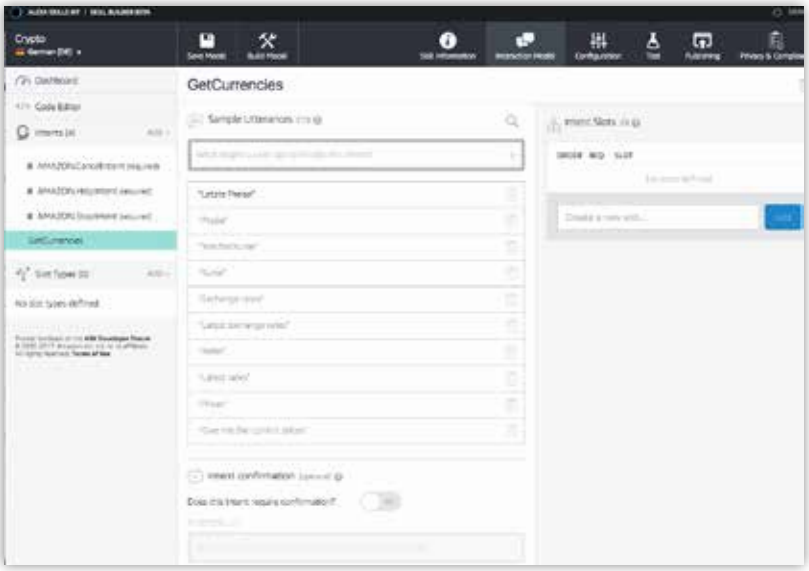
Amazon-Developer-Services konfigurieren

Um diesen Dienst nutzen zu können, muss man sich als Entwickler registrieren. Das ist glücklicherweise recht problemlos und auch kostenfrei. Wenn man sich zum Reiter **`Alexa`** und **`Create new Alexa App`** durch gehangelt hat, folgen zahlreiche Einstellungsmöglichkeiten die letztlich nur ein Ziel haben: irgendwie eine App-ID bekommen, die dem Code in (Listing 4) hinzugefügt werden kann. Zunächst gibt man den Namen an und auch die Sprachen, in denen man seinen Skill zur Verfügung stellen möchte.

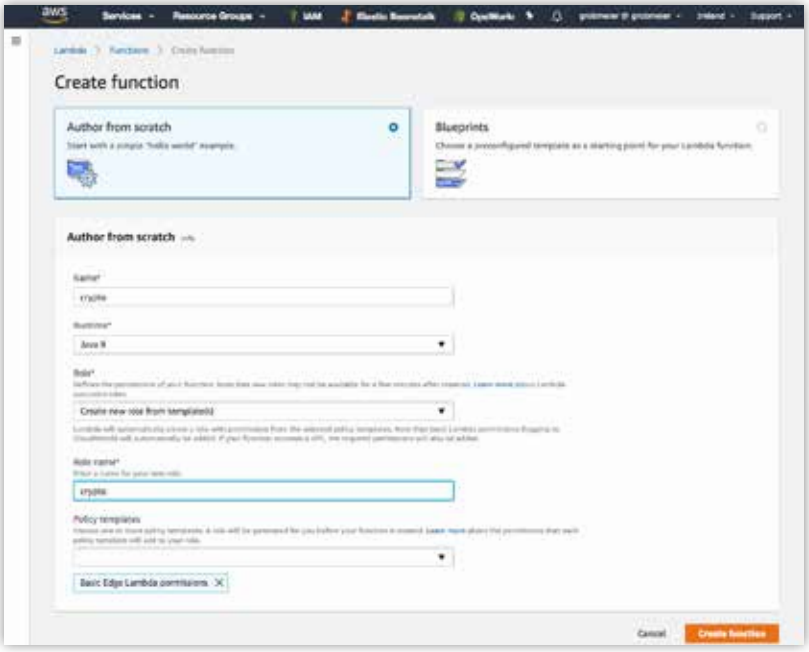
Wer weiterhin auf der bequemen Oberfläche arbeiten möchte, der hat seit neuestem die Möglichkeit mit dem Interaction-Model-Builder seine Anwendung angenehm zu konfigurieren. Vorher war dies nur mittels JSON möglich, was natürlich nicht wirklich schön war. Allerdings existiert ja bekanntermaßen nur Code, wenn er auch in Git committed wurde. Insofern sollte man bei professionellen Entwicklungen zumindest das generierte JSON kopieren und in die Versionskontrolle kopieren.

Wie man in (Abb. 3) sehen kann, besteht in dieser Oberfläche die Möglichkeit mehrere Intents anzulegen. Dazu konfiguriert man beispielsweise die Keywords, die das Intent auslösen sollen. Ich habe in meinem Fall einfach eine Menge von Wörtern definiert, die mir behilflich sein könnten. Dazu können auch die Slots, also die Platzhalter angelegt werden.

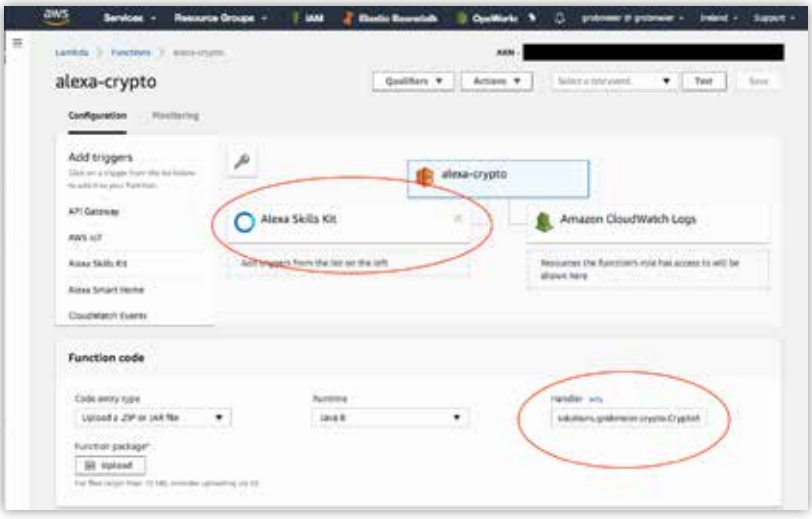
Im Punkt **`Configuration`** muss das ARN (eine Art AWS-ID einer AWS-Resource) der Lambda-Funktion angegeben werden - was wir derzeit noch nicht haben. Es handelt sich um ein Henne-Ei-Problem. Sie müssen sich also darauf vorbereiten, irgendwann die Lambda zu erstellen und dann die ARN hierhin zu kopieren.



Skill-Builder (Abb. 3)



AWS Lambda - Funktion erstellen (Abb. 4)



AWS Lambda - Trigger erstellen (Abb. 5)

AWS Lambda konfigurieren

In der AWS-Management-Console kann man AWS Lambda bequem über eine Weboberfläche konfigurieren. Leider gibt es keine Anleitung für das Erstellen von Lambda-Funktionen mit Java. Aber das kann man alles eigenständig lösen, wie in (Abb. 2) ersichtlich. Wichtig ist hierbei lediglich nur die korrekte Runtime und die Rolle. Im Prinzip sollten für die meisten Anwendungen grundlegende Rechte (Basic-Permissions) reichen.

Im Anschluss kann der Trigger, also der Auslöser für diese Funktion, definiert werden. Das geschieht bequem über ein Dropdown-Menü. Außerdem wird ein Handler benötigt. Das ist die Klasse, die wir in unserem Paket von **`SpeechletRequestStreamHandler`** abgeleitet haben.

In meinem Fall ist das also **`solutions.grobmeier.crypto.CryptoSpeechletRequestStreamHandler`**. Dann kann die JAR-Datei bereits hochgeladen werden. Die JAR erstellt man übrigens am einfachsten mit folgendem Maven-Kommando:

(Listing 5)

```
mvn assembly:assembly -DdescriptorId=jar-with
-dependencies package
```

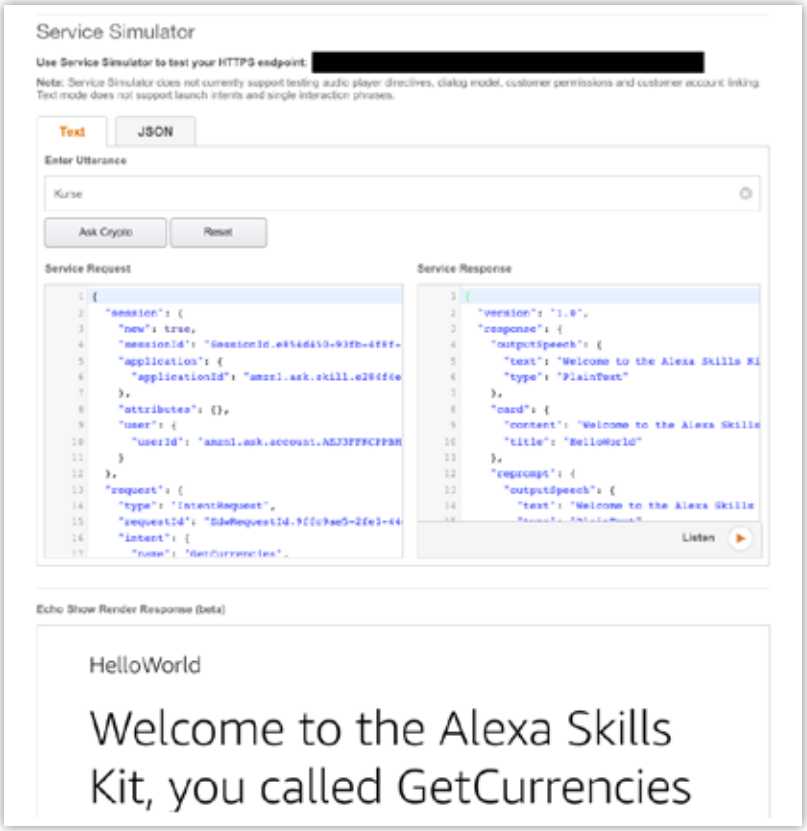
Hiermit wird ein Assembly erstellt das auch alle Abhängigkeiten für unser Paket beinhaltet.

Sobald die Lambda erstellt ist, vergessen Sie bitte nicht die ARN der Lambda, ersichtlich rechts oben in der Konsole, in die Amazon-Developer-Console zu kopieren. Damit sollte dann alles bereit sein für unseren ersten Test.

Test

Zurück in der Amazon-Developer-Console steht im Reiter **`Test`** der Service-Simulator bereit. Hier kann man sein Keyword eingeben und es so abschicken, als wäre es von Alexa erkannt worden.

Die Antwort erfolgt im JSON-Format und sogar die Card wird gerendert, als wäre eine Show vorhanden. Im Test ist aufgefallen, dass man die Lambda-Funktion nicht mit zu wenig



Alexa Testing (Abb. 6)

Zeit ausstatten sollte. In den Basic-Settings sollte man eher mit zehn Sekunden rechnen, bis die Lambda durchgelaufen ist. Es ist auch möglich, mit seinem eigenen Gerät zu testen.

Fazit

Allen Privatsphärenbedenken zum Trotz: Alexa macht einfach Spaß. Die Möglichkeit mit unserem Computer zu sprechen ist nicht erst seit Star Trek populär. Um Alexa-Geräte zu entwickeln, benötigt man allerdings auch etwas Geduld: Nicht nur Java und die SDKs der Sprachsteuerung wollen gemeistert werden. Auch AWS Lambda benötigt etwas Zeit und natürlich muss man sich auch mit der Amazon-Developer-Console etwas auskennen. Bis man dann vom Code zu einem fertigen, signierten Paket im Skill-Store kommt, kann es einige Zeit dauern. Die zugrundeliegenden Ideen sind zwar verständlich, aber neu. Trotzdem: An einem Wochenende sollte man auch ohne jegliche AWS-Vorkenntnisse zu einem halbwegs anständigen Skill kommen.

GEBIT Solutions

ZUKUNFT GESTALTEN

KREATIVE KÖPFE GESUCHT

LUST AUF NEUE HERAUSFORDERUNGEN?

Omnichannel, Digital Store, Mobile POS, Big Data, Internet of Things, Robotik, Augmented Reality, Location Based Services. Der Handel setzt Trends und erwartet passende Lösungen.

Genau hierfür realisieren wir maßgeschneiderten und innovativen IT-Lösungen für unsere Kunden wie z.B. ALDI SÜD, dm-drogerie markt, ESPRIT, Kaufland und OBI.

Komm zu GEBIT Solutions und gestalte mit uns die Zukunft des Handels als (w/m):
Java Softwareentwickler (Senior, Professional, Berufseinsteiger), Software-Architekt, Projektleiter, Java Webentwickler, Business Analyst, DevOps Engineer...

Berlin • Düsseldorf • Stuttgart **www.gebit.de/karriere**

GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

JAVA PRO

Magazin für professionelle Java Entwicklung in der Praxis

www.java-pro.de

Ausgabe 2 - 15. Juni 2018

Ausgabe 3 - 15. September 2018

Ausgabe 4 - 15. Dezember 2018

Das kostenlose Magazin für Java NEU!

Kostenlos anfordern unter: www.java-pro.de

... oder im Web unter www.java-pro.de

www.java-pro.de

www.java-pro.de

#JAVAPRO #Robotik #Telepräsenzrobotik

Telepräsenzrobotik mit Java

Projekt Avatar ist die prototypische Implementierung eines Telepräsenzrobotiksystems auf Basis eines NAO-Roboters. Ziel des Projektes ist es - ähnlich wie in James Camerons Film Avatar - in die Rolle eines Roboters hineinzuschlüpfen und eine außerkörperliche Erfahrung zu erleben.

Im Film Avatar – Aufbruch nach Pandora, schlüpft Captain Jack Sully durch eine Art Bewusstseinsübertragung in die Rolle eines außerirdischen Eingeborenen, um eine Mission zu erfüllen. Mit dieser speziellen Technologie kann er die Welt aus der Sicht dieses „Na'vi" erleben, fühlen und erkunden.

Das Ziel von Projekt Avatar ist es, ein System zu entwickeln, mit welchem man eine ähnliche außerkörperliche Erfahrung (Out-of-body-experience) erleben kann. Dies soll durch den Einsatz von handelsüblicher Hardware erreicht werden. Mit einer solchen Telepräsenz-Applikation soll von einem weit entfernten Ort ausgehend ein Roboter berührungslos ferngesteuert werden. 3D-Kameras tracken das Skelett des gesamten menschlichen Körpers. Die Software wandelt diese Positionsdaten in Bewegungsabläufe um. Ein vom Bediener entferntes Robotiksystem führt daraufhin Bewegungen auf dieselbe Art aus. Zum Einsatz kommt hier ein humanoider Roboter der Firma SoftBank Robotics namens NAO. Dieser ist mit zusätzlicher Sensorik ausgestattet, um weitere Daten aus der Umwelt zu sammeln. Der Roboter sendet die so gewonnenen Informationen an den Leitstand zurück. Dort werden die Eindrücke aus der entfernten Umwelt in geeigneter Weise dargestellt.

Autor:

Martin Förtsch und Thomas Endres sind IT-Consultants der TNG Technology Consulting GmbH aus Unterföhring bei München. Beide haben Informatik studiert und beschäftigen sich hauptberuflich mit agiler Softwareentwicklung. Sie halten auf zahlreichen IT-Konferenzen Vorträge über AR, VR und IoT und erhielten u.a. dafür den JavaOne Rockstar Award.
Firmen-Website: https://tngtech.com
Blog-URL: http://parrotsonjava.com/

Martin Förtsch
E-Mail: martin.foertsch@tngtech.com
Twitter: https://twitter.com/MartinFoertsch
LinkedIn: https://www.linkedin.com/in/martinfoertsch/
Xing: https://www.xing.com/profile/Martin_Foertsch2

Thomas Endres
E-Mail: thomas.endres@tngtech.com
Twitter: https://twitter.com/originalone1984
LinkedIn: https://www.linkedin.com/in/thomasendres/
Xing: https://www.xing.com/profile/Thomas_Endres5

Definition der Telepräsenzrobotik

Für die Definition eines Telepräsenzrobotiksystems sind zwei Begriffe von zentraler Bedeutung. Telerobotik beschreibt das Steuern von mechanischen Systemen aus der Ferne. Telepräsenz dagegen bedeutet, dass der Benutzer die Möglichkeit bekommt sich regelrecht in einen Teleoperator hineinzuversetzen. Das System vermittelt so dem Menschen im Leitstand den Eindruck, dass er sich an dem Ort befindet, an dem der Teleoperator stationiert ist.

Menschlichen Barrieren im Überblick (Abb. 1)

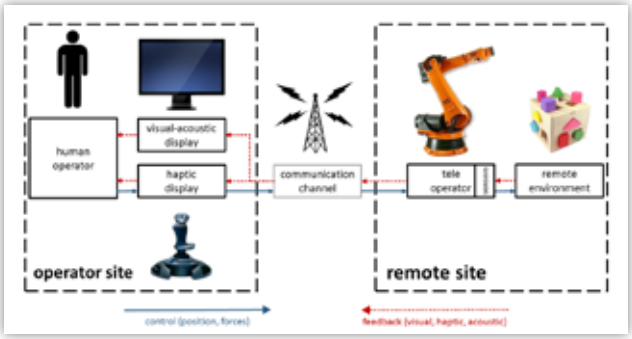
Ziele der Telepräsenzrobotik

Das erklärte Ziel der Telepräsenzrobotik ist es, menschliche Barrieren zu überwinden. Darunter zählen u.a. Distanz, Gefahr, Skalierung und Materie.[1] In (Abb. 1) werden vier menschliche Barrieren dargestellt. Die wichtigste Barriere, die es zu überwinden gilt, ist die der Kommunikation über weite Distanzen. Schon zu Frühzeiten haben insbesondere die Ureinwohner in Nordamerika Rauchzeichen zur Fernkommunikation benutzt. Diese Form der optischen Telegrafie kam z.B. dann zur Anwendung, wenn sich Stämme gegenseitig vor herannahenden Feinden warnen wollten. Heutzutage überwindet Satellitentechnik extrem weite Distanzen, um z.B. unbemannte Flugdrohnen an fernen Orten fernzusteuern.

Ein Telepräsenzrobotiksystem kann aber auch Orte mit für Menschen lebensfeindlichen Bedingungen aus sicherer Distanz untersuchen. Das Erkunden eines havarierten Kraftwerks ist dafür ein typischer Anwendungsfall, der vor kurzem im Kraftwerk Fukushima erfolgreich durchgeführt wurde.

Menschen können mit ihren Hand- und Fingerfertigkeiten oft nur bis zu einer bestimmten Größenordnung präzise arbeiten. Ein Chirurg ist typischerweise in der Lage, mit einer Präzision im Millimeterbereich zu operieren. Ein Robotiksystem kann die Bewegungen des Chirurgen so skalieren, dass eine Operation mit einer Genauigkeit im Mikrometerbereich möglich wird. Beispielsweise kann das Entfernen von Tumoren präziser erfolgen und die Gefahr, ungewollt Krebszellen zu verteilen, wird minimiert.

Orte, die aufgrund ihrer physikalischen Eigenschaften für den Menschen unerreichbar sind, stellen eine weitere Barriere dar. Ein Tiefseeroboter, welcher in mehreren Kilometer Meerestiefe Expeditionen durchführt, ist hier ein klassisches Anwendungsbeispiel.



Überblick über ein Telerobotiksystem (Abb. 2)

(Abb. 2) zeigt die schematische Darstellung eines multimodalen Telepräsenzrobotiksystems. Dieses System wird durch zwei unterschiedliche Orte charakterisiert. Auf der einen Seite befindet sich die Leitstelle (engl. „Operator Site"), in der ein menschlicher Operator Steuersignale erzeugt. Dies geschieht z.B. durch die Benutzung von Joystick oder Tastatur. Die Positions- und Steuerdaten werden über einen Kommunikationskanal an die Gegenstelle (engl. „Remote Site") gesendet. Der Teleoperator führt die Steuersignale entsprechend aus und kann dadurch die Umwelt der Gegenstelle manipulieren.[2]

Der Teleoperator verfügt aber auch über Sensorik und kann Veränderungen der entfernten Umwelt erkennen. Er sendet solche Daten über den Kommunikationskanal zurück an den Bediener. Die Daten aus dem Rückkanal werden dem Bedienenden auf eine geeignete Weise dargestellt. Monitore oder Virtual-Reality-Headsets können z.B. Videodaten von RGB-Kameras anzeigen, Lautsprecher oder Kopfhörer akustische Aufnahmen wiedergeben.

Die hier beschriebene Teleoperation bezeichnet wie bereits erwähnt das bloße Fernsteuern von Teleoperatoren. Diese werden dabei von einem menschlichen Leitstand aus gesteuert, um dort ausgelöste Aktionen an den sich in der Ferne befindlichen Teleoperator zu senden und auszuführen. Telepräsenzrobotik kombiniert die beiden Gebiete Teleoperation und Telepräsenz. Der Mensch im Leitstand hat neben der Möglichkeit einen Teleoperator fernzusteuern das Gefühl, sich an einem weit entfernten Ort zu befinden.

Multimodale Telepräsenz

Bei einem multimodalen Telepräsenzrobotiksystem ist die verzögerungsfreie Übertragung der Daten sehr wichtig. Der Roboter muss für das räumliche Sehen mit einem Stereo-Kameramodul ausgestattet werden. Diese Videodaten sollen beim Bediener in der Leitstelle idealerweise in einer Datenbrille oder einem Datenhelm angezeigt werden. Die Verzögerung bei der Übertragung über den Kommunikationskanal gilt es dabei so niedrig wie nötig zu halten, um die sog. Motion-Sickness (auch Kinetose genannt) zu verhindern. Zusätzlich sollen druckempfindliche Sensoren genutzt werden, welche Berührungen am Roboter an

den Operator übertragen. Wird der Kopf des Roboters berührt, soll der Bediener in der Leitstelle durch die Verwendung eines Vibrationsmotors ebenfalls das Gefühl vermittelt bekommen, dass er am Kopf berührt wird.

Fernsteuerung

Um dem Bediener in der Leistelle ein Maximum an Freiheit bei der Kontrolle des Teleoperators zu bieten, wird von herkömmlichen Steuerungen über Keyboard oder Joystick abgesehen. Das Steuern von 25 Freiheitsgraden ist über diese gewöhnlichen Eingabegeräte sehr schwierig.

Wesentlich intuitiver ist es, von einem 3D-Kamerasensor abgefilmt zu werden, um die daraus gewonnenen Skelett- und dazugehörigen Bewegungsdaten an den humanoiden Roboter zu senden. Bewegungen des Operators werden vom Roboter auf der Gegenstelle nachgeahmt.

Idee und Aufbau von „Projekt Avatar" (Abb. 3)

Nach der Identifizierung der wichtigsten Anforderungen von Projekt Avatar wurde als nächstes die passende Hardware gesucht. Wie in (Abb. 3) zu sehen wird neben einem humanoiden Roboter eine Datenbrille und ein 3D-Kamerasensor benötigt.

Die Hardwareauswahl

Projekt Avatar baut auf einem der am weitest verbreiteten humanoiden Roboter der Firma SoftBank Robotics (ehemals Aldebaran[3]) auf. Das menschliche Skelett dient als Vorlage für die Anatomie des NAO. Der Roboter verfügt über 25 Freiheitsgrade.

NAO verfügt zusätzlich über Hände, mit denen er Objekte greifen kann. Er ist mit einer integrierten Batterie ausgestattet, so dass er kabellos betrieben werden kann. Die zwei integrierten RGB-Kameras sind vertikal ausgerichtet und eignen sich deshalb nicht, um ein stereoskopisches Bild zu erfassen, welches dem menschlichen Sichtfeld ähnelt. Der Datenaustausch zum NAO geschieht wahlweise über WLAN oder Ethernet. Darüber hinaus verfügt er u.a. über taktile Sensoren, Mikrofone und Sonar.

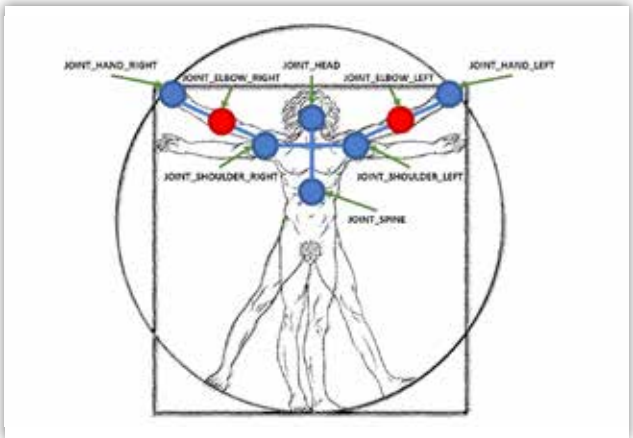
Um das Sichtfeld des Roboters dem Menschen in der Leitstelle zu vermitteln benötigt dieser eine Datenbrille, welche das räumliche Bild ins eigene Sichtfeld projiziert. Hier fällt die Wahl auf eine Oculus Rift DK2.

Head-Mounted-Display und Tracking Sensor der Oculus Rift DK2 (Abb. 4)

Damit der komplette menschliche Körper getrackt werden kann, wird eine spezielle 3D-Kamera benötigt. Microsoft Kinect oder Intel RealSense sind klassische Hardwareprodukte für derartige Spezialkameras. Die Kinect ist auf das Ganzkörpertracking spezialisiert und kann bis zu 35 Gelenkpunkte (sog. Joints) verfolgen. Intel hat mit der Version 2016 R2 des RealSense SDKs ebenfalls Körpertracking für bis zu sechs Punkte im Oberkörperbereich implementiert. Eines der RealSense-Kameramodelle, welches diese Funktion nutzen kann, ist die R200.

Intel RealSense R200 3D-Kamera (Abb. 5)

In der folgenden Abbildung werden die von der RealSense nachverfolgbaren Joints dargestellt. Die blauen Kreise zeigen die Punkte, welche von der 3D-Kamera direkt abgefragt werden können. Die mit roten Kreisen markierten Gelenke werden zum aktuellen Zeitpunkt nicht von der Kamera erfasst.



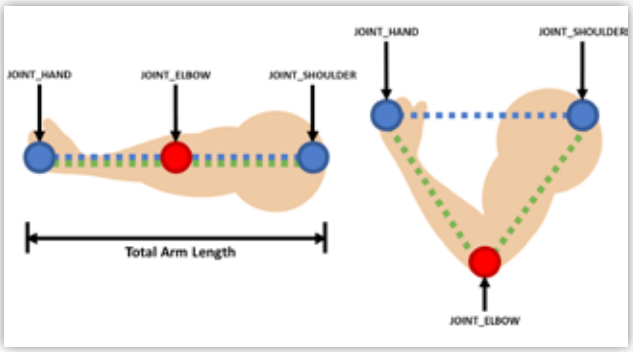
Körpertrackingpunkte der Intel RealSense R200 3D-Kamera (Abb. 6)

Der Java-Quelltext in (Listing 1) zeigt die Abfrage von Joints mit der Intel RealSense-API. Da die Java-Schnittstelle nur aus einem sehr dünnen Wrapper um die C++-Klassen besteht, ähnelt der gegen die API implementierte Code eher den C++-Konventionen als denen der Programmiersprache Java.

(Listing 1)

```
// personTracking is an instance of PXCMPersonTrackingData
int numberOfPersons = personTracking.QueryNumberOfPeople();

for (int personIndex = 0; personIndex < numberOfPersons; perso-
nIndex++) {
    // Retrieve the PersonTracking instance
    PXCMPersonTrackingData.Person personData =
personTracking.QueryPersonData(
    PXCMFaceTrackingData.AccessOrder.ACCESS_ORDER_BY_ID, perso-
nIndex);
PXCMPersonTrackingData.PersonJoint personJoints =
    personData.QuerySkeletonJoints();

    // work on the tracking data
    int numberOfJoints = personJoints.QueryNumJoints();
    PXCMPersonTrackingData.JointPoint[] joints = new
        PXCMPersonTrackingData[numberOfJoints];
    personJoints.QueryJoints(joints);

    // do some awesome things
}
```

Von der 3D-Kamera nicht erfasste Joints werden in der Implementierung von Projekt Avatar näherungsweise bestimmt. Basierend auf der bekannten gesamten Armlänge und der aktuell bestimmten Armlänge können die fehlenden Joints berechnet werden.



Trackingpunkte eines Armes für die Intel RealSense R200 3D-Kamera (Abb. 7)

Die Distanz zwischen **JOINT_HAND** und **JOINT_SHOULDER** wird durch die vom SDK erkannten Punkte berechnet. Darüber hinaus ist die gesamte Armlänge bekannt, welche durch die grüne Linie dargestellt ist. Wird der Arm gebeugt, kann die Position von **JOINT_ELBOW** näherungsweise bestimmt werden. Der berechnete Punkt für **JOINT_ELBOW** ist nicht eindeutig. Es wird daher immer der tiefste mögliche Punkt für den **JOINT_ELBOW** als korrekt angenommen. Andere Punkte wirken aufgrund der menschlichen Anatomie unnatürlich und werden auch vom Menschen eher selten eingenommen.

Die Beine des NAO-Roboters sind per Gesten nur eingeschränkt steuerbar. Hintergrund ist, dass bei einer freien Beinsteuerung der NAO zu schnell das Gleichgewicht verlieren würde. Stattdessen wird nach der Erkennung einer Laufgeste (vgl. auf der Stelle trippeln) ein Laufmakro auf dem NAO ausgeführt. Dieser voreingestellte Bewegungsablauf lässt den NAO sicher in eine bestimmte Richtung laufen, ohne die Balance zu verlieren.

Programmierung des NAO

Den NAO-Roboter zu programmieren ist sehr einfach. Über das SDK werden Schnittstellen für Java, JavaScript, Python und C++ angeboten. Die Unterstützung von .NET wurde dagegen eingestellt. Für alle von NAO unterstützten Programmiersprachen wird eine Bibliothek namens **NAOqi** angeboten, die von der Softbank-Website heruntergeladen werden kann. Die folgenden Beispiele sollen für ein besseres Verständnis sorgen, wie einfach der NAO-Roboter programmiert werden kann.

Setup des NAO

Der Import diverser Klassen aus der NAOqi-Bibliothek ist notwendig, um den NAO programmatisch anzusteuern. Jede einzelne Klasse stellt jeweils eine bestimmte Funktionalität des NAOs zur Verfügung. Es folgt ein Auszug einiger Klassen inklusive der dazugehörigen Funktionalität:

- **ALAudioPlayer** zur Wiedergabe von **.wav**, **.ogg** und **.mp3** Dateien über die integrierten Lautsprecher des Roboters.
- **ALTextToSpeech** zur Wiedergabe von Texteingaben in Form einer akustischen Sprachausgabe über die integrierten Lautsprecher. Es existieren Pakete für viele verschiedene Sprachen.
- **ALRobotPosture** zur Verwendung von vordefinierten und automatischen Bewegungsabläufen zwischen unterschiedlichen vorgegebenen Posen.
- **ALMotion** zur exakten Ansteuerung einzelner Gelenke.

(Listing 2)

```
public static void main(String[] args) throws Exception {
    Application application = new Application(args,
        String.format("tcp://%s:%s", IP_ADRESS_OF_NAO,
        PORT_NUMBER_OF_NAO));

    application.start();

    speechProxy = new ALTextToSpeech(application.session());
    audioProxy = new ALAudioPlayer(application.session());
    postureProxy = new ALRobotPosture(application.session());
    motionProxy = new ALMotion(application.session());
}
```

In (Listing 2) wird eine Konfiguration für den NAO-Roboter gezeigt. Die Instanziierung des Objektes der Klasse **Application** mit einer IP-Adresse und einer Portnummer erlaubt die Initialisierung der einzelnen NAOqi-Module. Die Standardportnummer ist 9559. Über diesen Socket findet später die gesamte Kommunikation bilateral zwischen dem Java-Programm und dem NAO statt.

Hello-World mit NAO

Nach der Konfiguration ist die programmatische Ansteuerung des NAO nun möglich. Die ersten Schritte mit einem Hello-World-Programm sind erstaunlich einfach.

(Listing 3)

```
speechProxy.say("Hello World!");
```

In (Listing 3) wird auf das Objekt der Klasse **ALTextToSpeech** mit dem Namen **speechProxy** zugegriffen. Die Methode **say** bringt den NAO zum Sprechen. Der Roboter wird den Satz **Hello World!** aussprechen.

NAO ist ein Poser

NAO kann verschiedene sog. Posen einnehmen. Zu derartigen Posen gehören u.a. das ganz normale auf den Beinen stehen (**Stand**), auf dem Bauch oder Rücken liegen (**LyingBelly** oder **LyingBack**) oder in die Hocke gehen (**Crouch**). Zwischen all den verschiedenen Posen kann der NAO selbstständig hin und her wechseln. Sollte NAO z.B. einmal das Gleichgewicht verlieren und auf dem Bauch landen, führt das Aufrufen der Pose **Stand** zu einem automatisierten Bewegungsablauf, der ihn dazu bringt wieder die aufrechte Köperhaltung einzunehmen.

(Listing 4)

```
postureProxy.goToPosture("Stand", 1);
```

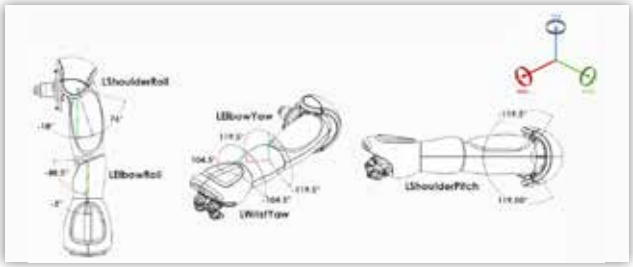
In (Listing 4) wird der NAO durch die Verwendung der Methode **goToPosture** aus der Klasse **ALRobotPosture** in eine beliebige Pose gehen. Als erstes Methodenargument wird die Zielpose (hier **Stand**) und im zweiten die Bewegungsgeschwindigkeit definiert. Dabei ist **0.0** die langsamste und **1.0** die schnellste Geschwindigkeit.

I like to move it, move it

Mit dem nächsten Listing erklären wir die grundsätzliche Herangehensweise, um einzelne Gelenke des NAO-Roboters anzusteuern.

Zielpose, die der NAO einnehmen soll (Abb. 8)

Im Folgenden soll der NAO-Roboter die in (Abb. 8) dargestellte Pose einnehmen. Da für diese Stellung der Joints keine vordefinierte NAO-Pose durch die **ALRobotPosture** Klasse zur Verfügung gestellt wird, muss die Stellung der Gelenke vollständig selbst implementiert werden.



Freiheitsgrade des linken Armes des NAO v5 aus der Entwicklerdokumentation (Abb. 9)

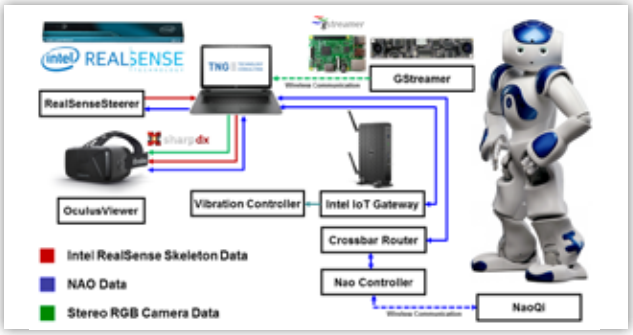
Um das zu erreichen müssen wir mehrere Armgelenke des rechten und des linken Armes in eine zuvor festgelegte Position versetzen. Die Joints, welche angesteuert werden, sind **ShoulderRoll**, **ShoulderPitch** und **ElbowRoll.** Da sich sowohl der linke als auch der rechte Arm bewegen müssen, werden in Summe sechs Joints bewegt. Der in (Abb. 9) dargestellte Auszug aus der Entwicklerdokumentation zeigt die verschiedenen Joints und damit verbundenen Freiheitsgrade des NAO-Roboters.

(Listing 5)

```
public void testMotion() throws InterruptedException, CallError {
    List<String> leftJoints = new ArrayList<>(Arrays.asList(
        "LShoulderRoll", "LElbowRoll", "LShoulderPitch"));
    List<String> rightJoints = new ArrayList<>(Arrays.asList(
        "RShoulderRoll", "RElbowRoll", "RShoulderPitch"));

    List<Float> leftAngles = new ArrayList<>(Arrays.asList(2.0f,
-1.5f,
        0.6f));
    List<Float> rightAngles = new ArrayList<>(Arrays.asList(-
2.0f, 1.5f,
        0.6f));

    motionProxy.setAngles(leftJoints, leftAngles, 0.5f);
    motionProxy.setAngles(rightJoints, rightAngles, 0.5f);
}
```

In (Listing 5) ist der Code zur Ausführung dieser Bewegung zu sehen. Die Methode **setAngles()** erwartet drei Parameter. Der erste Parameter ist eine Liste der Namen der anzusteuernden Joints. Der zweite Parameter ist ebenfalls eine Liste, in welcher die zu den Gelenken korrespondierenden Werte der Positionen in derselben Reihenfolge enthalten sind. Die für das jeweilige Gelenk möglichen Werte können über die NAO-Dokumentation nachgeschlagen werden. Der dritte Parameter definiert die Geschwindigkeit, mit welchem der angesteuerte Motor des entsprechenden Gelenks bewegt wird. Wird der Parameter mit **1.0** definiert, führt der NAO die Bewegung mit maximaler Geschwindigkeit aus. Der Parameterwert **0.0** würde dagegen eine sehr langsame Bewegung zur Folge haben.

Software-Architektur



Die Software-Architektur in der Übersicht (Abb. 10)

Die Soft- und Hardware-Architektur von Projekt Avatar ist, wie in **(Abb. 10)** zu sehen, relativ komplex. Es müssen Daten aus unterschiedlichsten Quellen gesammelt, verarbeitet und weitergeleitet werden, damit das Gesamtprojekt funktioniert.

Eine 3D-Kamera erfasst Bewegungsdaten, welche auf dem Laptop in NAOqi-kompatible Bewegungskommandos übersetzt werden. Diese Bewegungsanweisungen werden an einen über die Software Node-RED umgesetzten IoT-Router gesendet und an den NAO-Controller weitergegeben. Der Controller wiederum sendet diese Daten drahtlos an die NAOqi Schnittstelle. NAO führt diese Bewegungen daraufhin aus.

Die Erkennung von Joints am menschlichen Skelett und das Übersetzen in korrespondierende NAO-Bewegungen benötigt kaum zusätzlichen Aufwand über die Verwendung der entsprechenden SDKs hinaus. Die Verarbeitung der Feedbackdaten vom NAO zurück zum Leitstand ist dagegen wesentlich schwieriger, da insbesondere die Videodaten eine hohe Netzwerklast verursachen.

Da die im NAO integrierten RGB-Kameras vertikal ausgerichtet sind, eignen sie sich nicht zur Erstellung eines stereoskopischen Bildes für die Darstellung in der VR-Brille. Theoretisch ist es zwar möglich die Überlappungen der beiden Kameras auszunutzen, um ein Stereobild zu berechnen. Jedoch ist die Überlappung bei einer gleichzeitig relativ niedrigen Kameraauflösung von 1280x960 Pixel (~1,2 Megapixel) beider Kameras nur minimal. Aus diesem Grunde wurde stattdessen eine horizontal ausgerichtete Stereokamera in ein 3D-gedrucktes Gehäuse integriert, welche dem NAO als eine Art Brille am Kopf dient. Diese Stereokamera verfügt ebenfalls nur über eine Auflösung von einem Megapixel. Allerdings wird dank der horizontalen Ausrichtung der Kameras ein stereoskopisches Bild der Umwelt in guter Qualität aufgenommen. Die Stereokamera erzeugt eine große Menge an Videodaten, welche auf der NAO-Seite von einem Raspberry Pi verarbeitet werden. Der Minicomputer ist als Rucksack am NAO angebracht.

Die Komprimierung der Videosignale geschieht durch das MJPG-Verfahren. Es wird einfach ein Bild nach dem anderen JPG-kodiert übertragen. Diese Komprimierung hat den Vorteil, dass sie keinerlei zusätzliche Verzögerung verursacht, wie es z.B. bei H.264 der Fall wäre. Darüber hinaus ist diese Herangehensweise für den Prozessor ressourcenschonender. Auf der anderen Seite führt dies aber auch dazu, dass die Bandbreite des Netzwerkes schnell ausgeschöpft ist. Ist das Netzwerk überlastet, werden nicht mehr alle Frames angezeigt. Der Mensch im Leitstand erlebt dieses Verhalten als Ruckeln.

Die Lösung dieses Problems stellt für Projekt Avatar ein spezielles Router-Board dar. Dieses baut zwei separate WLAN-Netzwerke auf. Ein Netzwerk wird benutzt, um die Steuerdaten in beide Richtungen auszutauschen. Das andere Netzwerk wird exklusiv dafür verwendet, um die Videodaten zu übertragen. Somit wird die Bandbreite erhöht und keine Videodaten gehen mehr verloren. Zur Bildübertragung zwischen der entfernten Umwelt und dem Leitstand verwenden wir die Open Source Software GStreamer.

Der Transportkoffer enthält die für Projekt Avatar notwendige Hardware (Abb. 11)

Fazit

Bereits mit Hilfe von handelsüblicher Hardware kann ein multimodales Telepräsenzrobotiksystem realisiert werden. Den NAO mit Gestensteuerung zu kontrollieren ist durch die Verwendung von 3D-Kamerasensoren wie z.B. der Microsoft Kinect oder Intel RealSense, sowie der NAOqi-Schnittstelle sehr einfach. Eine große Herausforderung ist es, die Videosignale der Stereokamera über das Drahtlosnetzwerk verzögerungsarm und mit ausreichender Qualität zu übertragen. Mit Hilfe eines Intel IoT-Gateways und eines speziellen Routerboards, welches durch mehrere WiFi-Netzwerkkarten gleich mehrere WLAN-Netzwerke aufbaut, können schließlich die Videosignale wie auch sämtliche Steuer- und Sensorsignale in beide Richtungen nahezu latenzfrei übertragen werden, um eine außerkörperliche Erfahrung zu realisieren.

Links:

1 Paul M. Frank – “Advances in Control: Highlights of ECC’99”
2 Bruno Siciliano und Oussama Khatib:
“Springer Handbook of Robotics”
3 NAO Documentation – Aldebaran 2.4.3 NAOqi documentation.
https://goo.gl/33TD48

JOBS BEI JAVAPRO

ACCOUNT MANAGER m/w

Ihre Aufgaben:

- Vermarktung unserer Medienprodukte (u.a. Konferenzen, Werbekampagnen)
- Gewinnung und Betreuung von Neu- und Bestandskunden sowie Kooperationspartner
- Marktbeobachtung und Marktentwicklung
- Schnittstelle zwischen JAVAPRO (Redaktion, Event-Team, Marketing, Produktion) und unseren Kunden und Partnern

Voraussetzungen:

- Berufserfahrung im Bereich Vertrieb
- Erfahrung im Bereich Media, Verlag, Agentur, IT vorteilhaft
- Vertriebsorientierte und unternehmerische Denkweise
- Selbständige Arbeitsweise
- Hohe Kommunikationsfähigkeit
- Sicheres Auftreten
- Fließende Deutschkenntnisse, Englischkenntnisse vorteilhaft
- Reisebereitschaft (innerhalb Deutschlands)

Was wir Ihnen bieten:

- Mitarbeit bei einem hoch motiviertem Media-Startup, das anders denkt und funktioniert als konservative Verlage, mit gewaltigem Entwicklungspotential und entsprechenden persönlichen Chancen
- Home-Office Tätigkeit
- Selbständiges Arbeiten und flexible Arbeitszeiten
- Entscheidungskompetenzen
- Spielraum für die Umsetzung neuer Ideen
- Intensive Unterstützung und Betreuung

Wir freuen uns auf Ihre Bewerbung als E-Mail an:

jobs@java-pro.de