GRUNDLAGEN • PRAXIS • ARCHITEKTUR • AGILE • INNOVATION

3 / 2018

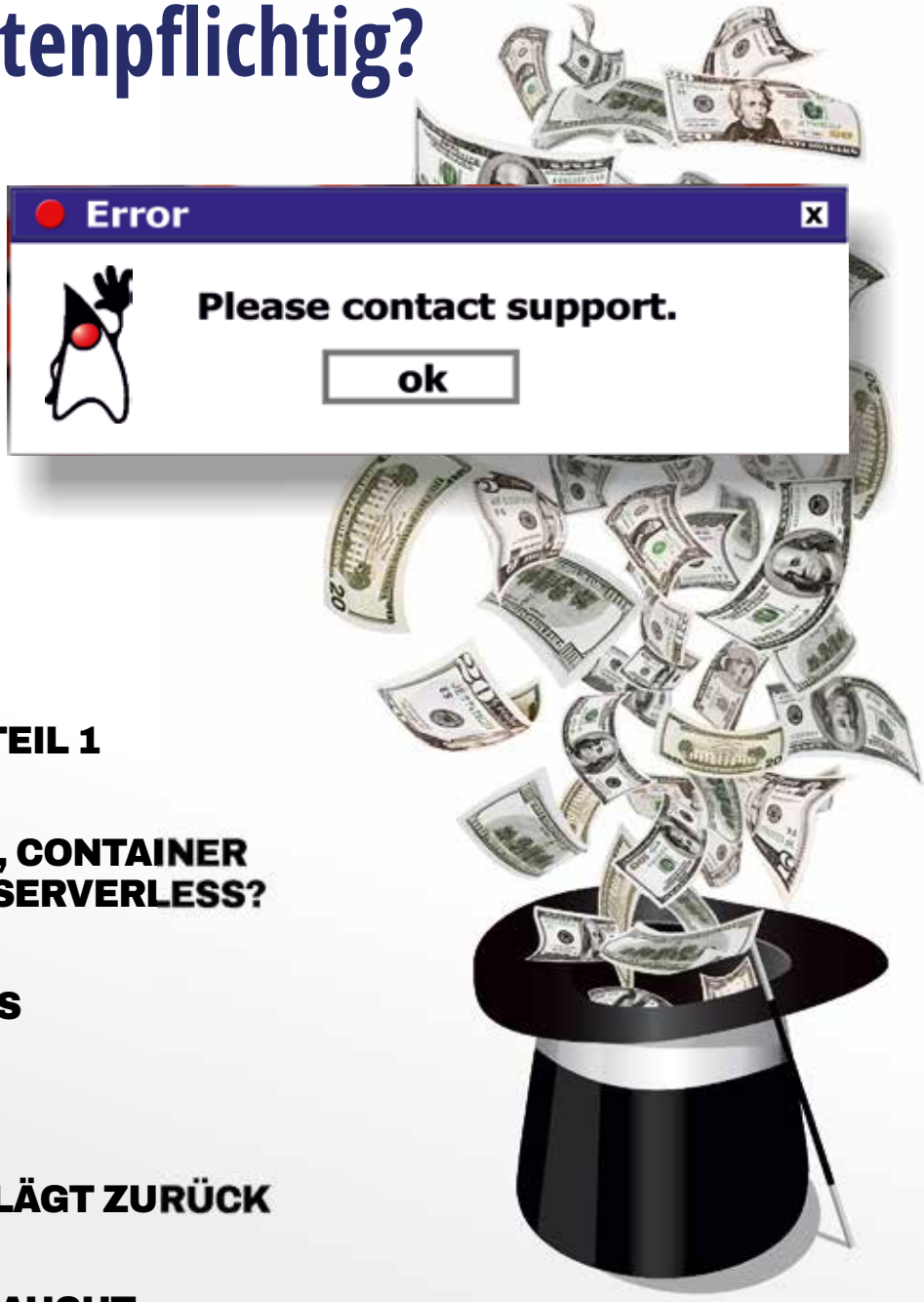
JAVAPRO

Magazin für professionelle Java Entwicklung in der Praxis

#JAVAPRO

Oracle ändert Lizenz- & Support Modell !

Java jetzt kostenpflichtig?



www.java-pro.de

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 2018

GOLD PARTNER

ORACLE eclipse XDEV Fast Lane Google Cloud

SILBER PARTNER

RAPID clipse jetstreamDB

BRONZE PARTNER

viadee IT-Unternehmensberatung consol 4DSoft GEBIT Solutions ETECTURE raytion Michael Page

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:
JAVAPRO
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:
Markus Kett (V.i.S.d.P.)

Redaktion:
info@java-pro.de

Gestaltung, Layout, Produktion:
Impuls Mediengruppe GmbH
Im Gewerbepark 29
92681 Erbendorf

Preis: kostenfrei

Illustrationen:
Pixabay (Public Domain)

Erscheinungsweise: Vier mal jährlich

Gründungsjahr: 2017

#JAVAPRO #Editorial

Die vergangenen Monate haben uns gravierende Veränderungen bei Java beschert. Früher mussten wir für gewöhnlich zwischen zwei bis vier Jahre auf die nächste Java-Version warten. Die Freigabe musste so gut wie immer um viele Monate verschoben werden. Spätestens mit Java 8 kamen Zweifel an einem Feature-getriebenem Release-Zyklus auf. Verzögern sich bestimmte Features, muss das gesamte Release verschoben werden und alle anderen, bereits fertigen Features können ebenfalls monatelang von der Community nicht genutzt werden. Deshalb sprach sich bereits damals Mark Reinhold, Chief Architect der Java Platform Group bei Oracle, für einen zeitbasierten Release-Zyklus aus.

In den letzten Monaten haben sich die Ereignisse nun regelrecht überschlagen. Für die Java Platform Standard Edition (Java SE) gibt es jetzt eine neue Roadmap mit stark verkürztem und vor allem regelmäßigem Release-Zyklus. Seit Java 9, das im September 2017 veröffentlicht wurde, erscheint nun alle sechs Monate eine neue Java Version, jeweils im März und September. Damit drückt Oracle bei Java mächtig aufs Gaspedal.

Gleichzeitig hat Oracle kräftig Ballast abgeworfen und bereits ebenfalls im Herbst 2017, für die Java-Community damals völlig überraschend, die Java Enterprise Edition abgestoßen und an die Eclipse Foundation übergeben. Seit Anfang 2018 wird Java EE als neues Eclipse-Projekt unter dem neuen Namen Jakarta EE von der Eclipse-Community weiterentwickelt.

Zur Freigabe von Java 11 im September hat der Java-Hersteller nun auch ein neues Lizenz- und Support-Modell für sein neues JDK vorgestellt, das für mächtig Gesprächsstoff in der Java-Community sorgt und auch viele Anwender verunsichert. Die wichtigste Frage, die uns alle bewegt ist: Kostet Java jetzt Geld? Um es vorweg zu nehmen, die Antwort lautet: Jein.

Wer das Oracle JDK im Produktivbetrieb einsetzen möchte, braucht in jedem Fall eine Lizenz – und diese kostet Geld! Aber gleichzeitig bietet Oracle auch eine kostenfreie Variante an: Das Oracle OpenJDK. Und es gibt noch weitere Alternativen dazu seitens der Community. Wolfgang Weigend beschreibt ab Seite 12 in seinem Artikel „Java 11 und die freie JDK Wahl" die verschiedenen Varianten sehr ausführlich und auch Falk Sippach geht in seinem Artikel „Neues in Java 11" ab Seite 14 auf dieses Thema ein. Der neue Java-Release-Zyklus ist in jedem Fall vorteilhaft und gut für die Community. Neue Java-Features stehen uns ab sofort sehr viel schneller zur Verfügung als früher. Auf der anderen Seite bedeutet die kurze Zeitspanne von nur sechs Monaten auch eine große Herausforderung für Anwender, die ihre Anwendung regelmäßig auf den neuesten Stand bringen wollen oder müssen. Viele Framework-, API- und Tool-Hersteller haben bereits reagiert und ihren eigenen Release-Zyklus entsprechend verkürzt. Bei der Eclipse IDE gibt es ab sofort sogar alle drei Monate ein neues Release.

Für Anwender, die nicht alle sechs Monate auf die neueste Java-Version umstellen können, gibt es von Oracle einen sogenannten Long-Time-Support.

Markus Kett
Chefredakteur
JAVAPRO


JCON

#JAVAPRO #JCON2018 #XDEVCON

JCON 2018 Rückblick

Die JCON ist bereits nach zwei Jahren eine der größten und beliebtesten Java-Community-Konferenzen in Deutschland. Insgesamt pilgerten dieses Jahr über 800 Teilnehmer nach Düsseldorf, knapp 40 Prozent mehr als im ersten Jahr. Im nächsten Jahr sollen es erstmals über 1.000 werden.

Ein besonderes Erlebnis ist die JCON durch ihre einzigartige Location, denn die JCON ist die einzige Java-Konferenz in Deutschland, die in einem modernen Multiplex-Kino stattfindet – in der UCI-Kinowelt im Düsseldorfer Medienhafen.
Auf den riesigen Kinoleinwänden wird Live-Coding zu einem einzigartigen Erlebnis und ist der entscheidende Mehrwert für Entwickler, denn selbst in den hintersten Reihen kann man immer noch alles perfekt erkennen und den Ausführungen der Speaker folgen ohne sich anstrengen zu müssen.

Über 70 großartige Speaker aus unserem JAVAPRO Autoren- und Speaker-Netzwerk waren auch heuer wieder der Garant dafür, dass das Vortragsprogramm erneut mit hochkarätigen Vorträgen gespickt war. An den drei Konferenztagen konnten die Teilnehmer insgesamt über 100 Vorträge besuchen.

Traditionell stehen auf der JCON die Themen Core-Java, Serverside- und Java-Enterprise, APIs und Frameworks im Fokus. In diesem Jahr ging es vor allem um die neuen Features in Java 9 und 10, um den neuen Release-Zyklus sowie um das neue Lizenz- und Support-Modell von Java, um die Weiterentwicklung von Java EE, das jetzt unter dem neuen Namen Jakarta EE unter dem Dach der Eclipse Foundation weiterentwickelt wird, sowie um funktionale und reaktive Programmierung.

Ergänzend zu den Java-Kernthemen gab es auch in diesem Jahr wieder drei hochkarätige Special-Days. Am Architecture-Day drehte sich auch heuer wieder alles um Microservices.
Am Cloud- und Serverless-Day dreht sich alles um Container, Docker, Kubernetes und um die Entwicklung von Serverless-Functions. David Delabassée von Oracle hat dazu in seiner Keynote das quelloffene Serverless-Framework fn-project vorgestellt, über das wir in unserer letzten JAVAPRO Ausgabe 2/2018 ausführlich berichtet haben.

Das Thema Agile ist mittlerweile schon ein Evergreen in der IT. Damit waren die Vorträge am Agile-Day wie gewohnt gut besucht.

Auch die XDEVCON war wieder erneut Teil der JCON. Die XDEVCON ist seit nunmehr einem Jahrzehnt die Top-Konferenz für Rapid-Cross-Platform-Development mit Java. Hier steht die Entwicklung von Web-Anwendungen und mobile Apps in der Praxis im Vordergrund. Alles dreht sich darum, wie man die Entwicklung von Business-Anwendungen mit Java vereinfachen und beschleunigen kann. XDEVCON-Vorträge sind erfahrungsgemäß immer gut gefüllt. Denn hier gibt es die meisten Live-Demos zu sehen. Im Fokus stehen hierbei die populäre Eclipse-Distribution RapidClipse, die neue Java In-Memory Database-Engine JetstreamDB sowie das UI-Framework Vaadin. Damit entstehen auf beeindruckende Weise oft ganze Anwendungsmodule während einer Session. Die XDEVCON richtet sich gezielt an Anwendungsentwickler, Projektleiter und IT-Entscheider, die businesskritische Geschäftsanwendungen mit Java entwickeln und sich dabei auf die schnelle Umsetzung neuer

Features konzentrieren möchten, ohne sich dafür mit Low-Level-Programmierung auseinandersetzen zu müssen. Dadurch zieht die XDEVCON auch viele Anwender aus dem 4GL-Umfeld an, die einen schnellen und einfachen Weg auf Java suchen, um wichtige Altanwendungen auf Java migrieren zu können.

Durch dieses breite Themenspektrum lässt die JCON keine Wünsche offen.

Neben dem Konferenzprogramm bietet die JCON auch eine 3-tägige Expo für Aussteller. Die diesjährigen sehr einfallsreichen Stände kamen bei den Teilnehmern richtig gut an, sodass die Ausstellung zwischen den Sessions und den längeren Pausen durchgehend stark frequentiert war. Unsere Aussteller waren demnach sehr zufrieden.

Wir bedanken uns bei allen Teilenehmern der JCON 2018 sowie bei unseren Partnern und Ausstellern, die die Durchführung der JCON 2018 möglich gemacht haben: Oracle, Fast-Lane-Gruppe und Google Cloud, Eclipse Foundation, XDEV Software Corp., RapidClipse, JetstreamDB, viadee, ConSol Consulting & Solutions Software GmbH, 4D Soft GmbH, GEBIT Solutions, ETECTURE, Raytion, MichaelPage, MANNING und dpunkt.verlag.

JCON 2019 - Call-for-Papers gestartet

Die JCON 2019 findet dieses Mal bereits vom 24. bis 26. September statt. Veranstaltungsort ist erneut die UCI-Kinowelt in Düsseldorf. Der Call-for-Papers wurde am 10. Dezember 2018 gestartet und läuft dieses Mal in 2 Phasen ab. Phase 1 läuft bis zum 28. Februar 2019. 60 Prozent aller Vorträge werden wir in dieser Phase ermitteln. Ziel ist es, den Sessionplan früher als in den ersten beiden Jahren veröffentlichen zu können. Die interessantesten Einreichungen werden als Featured-Sessions prominent hervorgehoben. Phase 2, über die wir die restlichen Vorträge auswählen werden, läuft bis 31. März 2019.

Haben Sie ein interessantes Thema, das Sie bei der JCON 2019 vorstellen möchten? Dann freuen wir uns über Ihre Einreichung. Mitmachen lohnt sich, denn für Ihre Einreichung erhalten Sie in jedem Fall einen 3-Tages-Pass und damit kostenfreien Zugang zu allen Sessions der JCON 2019. Als ausgewählter Speaker der JCON 2019 erhalten Sie neben einem 3-Tages-Pass weitere Tickets, mit denen Sie Ihre Kollegen, Geschäftspartner oder Freunde zu Ihrem Vortrag auf der JCON 2019 einladen können. Das Formular zum Call-for-Papers finden Sie auf der JCON-Webseite unter **`www.jcon.one`**.

#JAVAPRO #CoreJava #JDK11 #Java

Java 11 – die freie JDK-Wahl

Das neue Oracle JDK 11 wurde am 25. September 2018 als Long-Term-Support-Release (LTS) veröffentlicht[1]. Neben Bug-Fixes und Security-Patches stecken im JDK 11 Feature-Release, Sprach- und JVM-Änderungen, Erweiterungen, zugefügte und entfernte Klassen und Methoden, deprecated-markierte Features und JDK-Tool-Änderungen[2]. Anwender werden sich darüber freuen, dass sie ein JDK oder ein OpenJDK verschiedener Hersteller auswählen können.

Mit den neuen Java-Release-Zyklen bringt Oracle zwei JDK-Versionen pro Jahr heraus (Abb. 1). Nach dem JDK 8 LTS-Release in 2014, folgt jetzt das Oracle JDK 11 LTS. Danach soll im Dreijahresrhythmus als nächstes das Oracle JDK 17 LTS-Release das Licht der Welt erblicken.

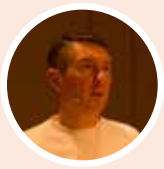
JDK Feature-Release und Critical-Patch-Update. (Abb. 1)

Die bestehende Oracle-Technology-Network-Lizenzvereinbarung (OTN-Lizenz) besagt, dass man das Oracle JDK 11 nur mit der Oracle Java SE Subscription[3] im Produktivbetrieb einsetzen darf. Die Nutzung vom Oracle JDK 11 ist für Entwicklung, Testing, Prototyping oder zu Demozwecken mit der OTN-Lizenz dabei ausdrücklich erlaubt. Beispielsweise benötigt man im unternehmensweiten Entwicklungsprozess mit Oracle JDK 11 und den Entwicklungsumgebungen Eclipse IDE, IntelliJ IDEA oder NetBeans IDE keine Oracle Java SE Subscription, weil die OTN-Lizenz die Entwicklung explizit erlaubt.

Anders ist es, wenn bespielsweise JIRA im Entwicklungsprozess mit dem Oracle JDK eingesetzt wird. In diesem Fall benötigt man die Oracle Java SE Subscription, weil es sich bei JIRA um eine Produktivumgebung handelt.

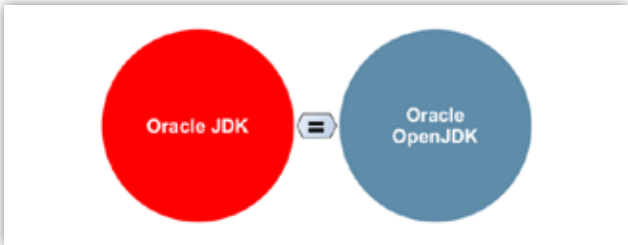
Mit Einführung der JDK-Version 11 wurden die technischen Unterschiede bei der Gleichstellung vom Oracle JDK 11 mit dem Oracle OpenJDK entfernt. Bei diesem Technologietransfer sind alle Bestandteile vom Oracle JDK als Open-Source-Implentierungen in das OpenJDK geflossen, sodass das Oracle JDK jederzeit problemlos durch das OpenJDK ausgetauscht werden kann (Abb. 2). Das Oracle JDK besitzt eine OTN-Lizenz und das OpenJDK wird über die GNU General Public License mit Classpath-Exception (GPL v2 & CPE) lizenziert.

Autor:



Wolfgang Weigend arbeitet als Senior Leitender Systemberater bei der Oracle Deutschland B.V. & Co. KG. Er beschäftigt sich mit Java-Technologie und Architektur für unternehmensweite Anwendungsentwicklung.

Im Gegensatz zum Oracle JDK, mit der OTN-Lizenz, kann das Oracle OpenJDK mit GPL v2 & CPE-Lizenz frei eingesetzt werden. Die Anwender können zwischen verschiedenen Hersteller-JDKs (Oracle, IBM, SAP, o.a.) auswählen oder sich für einen OpenJDK-Anbieter (Oracle, Azul, BellSoft, MicroDoc, Red Hat, SAP, o.a.) entscheiden. Zudem besteht die Möglichkeit, die vorgefertigten OpenJDK-Binärdateien von AdoptOpenJDK einzusetzen.



Oracle JDK und Oracle OpenJDK. (Abb. 2)

JDK 11 In & Out

Die im JDK 11 enthaltenen Merkmale[4] sind über die einzelnen JDK Enhancement Proposals (JEPs) in **(Abb. 3)** aufgeführt. Durch die kürzeren Java-Release-Zyklen sind weniger Features im JDK 11 enthalten als das bisher bei früheren Java-Major-Releases der Fall war. Kleinere Technologieportionen sollen bei der Anwendungsentwicklung schneller eingesetzt werden, um einen Innovationsstau zu vermeiden. Vorangegangene JEPs aus den früheren JDK-Versionen 9 und 10 sind mit dem JDK 11 vervollständigt und abgerundet worden. Hier ist zum Beispiel der JEP 323, Local-Variable Syntax for Lambda Parameters und die Standardisierung vom HTTP Client API (JEP 321) zu nennen. Der JDK Flight Recorder ist mit seinen Bestandteilen zum Datensammeln, durch die Module **`jdk.jfr.jmod`** und **`jfr.management.jfr.jmod`** im JDK 11 enthalten.

Das mit dem JDK Flight Recorder korrespondierende Visualisierungswerkzeug JDK Mission Control, befindet sich außerhalb vom JDK und muss eigenständig installiert werden[5].

Die JavaScript-Engine Nashorn (JEP 335) ist nicht mehr im JDK 11 enthalten. Eine Open-Source-Wartung von Nashorn kam nicht zustande. Mit GraalVM steht eine polyglotte Umgebung für interpretierte Sprachen (C/C++ LVVM, R, Ruby, JavaScript) und native Sprachunterstützung für Java, Scala und Kotlin bereit. Der experimentell eingestufte Graal JIT-Compiler benutzt das bereits mit dem JDK 9 eingeführte JVM Compiler Interface (JVMCI) über das Java-Kommando **`-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler`**.

Die UI-Technologie JavaFX wurde durch die Java-Modularisierung aus dem JDK 11 entfernt. JavaFX 11 wird seitdem über eigenständige Module zum JDK 11 hinzugefügt **(Abb. 4)**. Es kann auch über das JavaFX SDK 11 separat installiert werden. Der Download von JavaFX 11 wird von Gluon unter der GPL v2 & CPE-Lizenz bereitgestellt[6]. Der JavaFX-Support wird ebenfalls von Gluon angeboten. JavaFX 11 wird vom OpenJFX 11 abgeleitet und passt zum Oracle OpenJDK 11 oder zum Oracle JDK 11.

```
181: Nest-Based Access Control
309: Dynamic Class-File Constants
315: Improve Aarch64 Intrinsics
318: Epsilon: A No-Op Garbage Collector
320: Remove the Java EE and CORBA Modules
321: HTTP Client (Standard)
323: Local-Variable Syntax for Lambda Parameters
324: Key Agreement with Curve25519 and Curve448
327: Unicode 10
328: Flight Recorder
329: ChaCha20 and Poly1305 Cryptographic Algorithms
330: Launch Single-File Source-Code Programs
331: Low-Overhead Heap Profiling
332: Transport Layer Security (TLS) 1.3
333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)
335: Deprecate the Nashorn JavaScript Engine
336: Deprecate the Pack200 Tools and API
```

JDK 11 Feature-Liste mit JDK Enhancement Proposals (JEPs). (Abb. 3)

```
javafx.base.jmod
javafx.controls.jmod
javafx.fxml.jmod
javafx.graphics.jmod
javafx.media.jmod
javafx.swing.jmod
javafx.web.jmod
```

JavaFX 11 Module. (Abb. 4)

Fazit:

Mit einer überschaubaren Anzahl von technischen Änderungen im JDK 11 und der Vervollständigung von bereits in früheren JDK-Versionen angefangenen Arbeiten und Feature-Einführungen, sind keine großen Überraschungen im JDK 11 zu erwarten. Das breite Java-Ökosystem mit seiner Vielfalt ermöglicht Anwendern die freie Entscheidung über den Produkteinsatz des Oracle JDK 11 mit Oracle Java SE Subscription oder der alternativen Nutzung des Oracle OpenJDK sowie die Auswahl eines anderen Hersteller-JDK oder Hersteller-OpenJDK.

Quellen:

1 https://bit.ly/2Mrc9el
2 https://bit.ly/2DAuR1s
3 https://bit.ly/2PQKpFJ
4 https://openjdk.java.net/projects/jdk/11/
5 https://jdk.java.net/jmc/
6 https://openjfx.io/

#JAVAPRO #CoreJava #JDK11 #Java

Neues in Java 11

Oracle hat Anfang 2018 für Java einen völlig neuen Release-Zyklus eingeführt und dazu jetzt ein neues Lizenz- und Support-Modell bekannt gegeben. Im Rahmen dieses Artikels werfen wir einen Blick auf die Neuerungen, die neuen Features der neuesten Java Version 11 und auf die bereits im Frühjahr 2019 erwartete Java Version 12.

Autor:

Falk Sippach hat 20 Jahre Erfahrung mit Java und ist bei der Mannheimer Firma OIO Orientation in Objects GmbH als Trainer, Software-Entwickler/Architekt tätig. Er publiziert regelmäßig in Blogs, Fachartikeln und auf Konferenzen. In seiner Wahlheimat Darmstadt organisiert er mit anderen die örtliche Java User Group.

Twitter: @sippsack

Seit Frühjahr 2018 werden Java-Entwickler mit halbjährlichen Major-Releases beglückt. In diesem Zuge wurde im vergangenen September nun bereits Java 11[1] veröffentlicht. Darüber hinaus gab es auch politisch einige Änderungen. Denn mit Java 11 führt Oracle eine neue Lizenz- und Support-Strategie ein. Freie Updates für die letzte LTS-Version 8 (Long-Term-Support) wird es ab Januar 2019 nicht mehr geben. Zudem ist ab Java 11 das Oracle JDK nur noch in der Entwicklung kostenlos nutzbar, in Produktion muss ein Support-Vertrag[2] mit Oracle gegen entsprechende Gebühren abgeschlossen werden. Das Ziel dieser Lizenzgebühr

ist die Sicherstellung der wirtschaftlichen Weiterentwicklung des JDK durch das Oracle-Engineering-Team.

Wer weiterhin auf eine freie JDK-Version aufbauen möchte, kann das mittlerweile zum Oracle JDK binär kompatible OpenJDK einsetzen, welches als Open-Source unter der GNU General Public License v2 (with the Classpath Exception – GPL v2 + CPE) veröffentlicht ist. Allerdings wird Oracle das OpenJDK immer jeweils nur ein halbes Jahr mit freien Updates versorgen. Entweder aktualisiert man dann alle sechs Monate die eingesetzte Java-Version oder man nutzt alternative Distributionen. Das AdoptOpenJDK-Projekt[3] will beispielsweise die LTS-Versionen 8 und 11 noch bis zu vier Jahre mit kostenlosen Updates versorgen. Auch für Kunden von diversen Linux-Distributionen (z. B. Red Hat Enterprise Linux[4]) wird das OpenJDK 8 noch mindestens vier Jahre im Rahmen der Betriebssystem-Support-Verträge kostenlos mit Updates unterstützt. Für Anwender der Java-Plattform gibt es also zahlreiche Optionen zwischen kommerziellen Lösungen von Oracle, Azul, IBM[5] usw. und weiterhin freien Lösungen wie dem OpenJDK oder dem AdoptOpenJDK-Projekt. Es sind somit ausreichend Alternativen vorhanden, die darauf warten ausprobiert zu werden.

Neue Features in Java 11

Insgesamt wurden für das OpenJDK 11 etwa 2400 Tickets geschlossen[6]. Fast 80 Prozent davon hat Oracle bearbeitet. An den restlichen 20 Prozent waren aber viele andere Firmen wie SAP, Red Hat, Google, IBM usw. beteiligt. Ein Großteil der Neuerungen wurde im Rahmen der folgenden Java Enhancement Proposals (JEPs) umgesetzt:

- 181: Nest-Based Access-Control
- 309: Dynamic Class-File Constants
- 315: Improve Aarch64 Intrinsics
- 318: Epsilon: A No-Op Garbage-Collector
- 320: Remove the Java-EE and CORBA-Modules
- 321: http-Client (Standard)
- 323: Local-Variable-Syntax for Lambda-Parameters
- 324: Key-Agreement with Curve25519 and Curve448
- 327: Unicode 10
- 328: Flight-Recorder
- 329: ChaCha20 and Poly1305 Cryptographic-Algorithms
- 330: Launch Single-File Source-Code Programs
- 331: Low-Overhead Heap-Profiling
- 332: Transport-Layer-Security (TLS) 1.3
- 333: ZGC: A scalable Low-Latency Garbage-Collector
- 335: Deprecate the Nashorn JavaScript-Engine
- 336: Deprecate the Pack200-Tools and -API

Aus Entwicklersicht sind nur einige wenige Punkte wirklich relevant. So wurde im JEP 323 eine Erweiterung der in Java 10 eingeführten Local-Variable Type-Inference umgesetzt. Typinferenz ist das Schlussfolgern der Datentypen aus den restlichen Angaben des Quellcodes und den Typisierungsregeln heraus. Das spart Schreibarbeit und bläht den Quellcode nicht unnötig auf, wodurch sich wiederum die Lesbarkeit erhöht.

Seit Java 10 können lokale Variablen mit dem Schlüsselwort **var** folgendermaßen deklariert werden:

```
// Funktioniert seit Java 10

var zahl = 5; // int
var string = "Hello World"; // String
var objekt = BigDecimal.ONE; // BigDecimal
```

Neu in Java 11 ist, dass man nun auch Lambda-Parameter mit **var** deklarieren kann. Das mag auf den ersten Blick nicht sonderlich sinnvoll erscheinen, da man den Typ von Lambda-Parametern sowieso weglassen und über die Typinferenz ermitteln lassen kann. Nützlich wird die Erweiterung aber für die Verwendung von Type-Annotations wie **@Nonnull** und **@Nullable**.

```
// Inference von Lambda Parametern
Consumer<String> printer = (var s) -> System.out.println(s); //
statt s -> System.out.println(s);

// aber keine Mischung von "var" und deklarierten Typen möglich
// BiConsumer<String, String> printer = (var s1, String s2)
-> System.out.println(s1 + " " + s2);

// Nützlich für Type Annotations
BiConsumer<String, String> printer = (@Nonnull var s1, @Nullable
var s2) ->
    System.out.println(s1 + (s2 == null ? "" : " " + s2));
```

Die nächste interessante Neuerung ist die Standardisierung der bisher noch experimentellen neuen HTTP Client API, die mit JDK 9 eingeführt und in JDK 10 aktualisiert wurde (JEP 110). Neben HTTP/1.1 werden nun auch HTTP/2, WebSockets, HTTP/2 Server-Push, synchrone und asynchrone Aufrufe und Reactive-Streams unterstützt. Garniert mit einem gut lesbaren Fluent-Interface wird die Verwendung von anderen HTTP-Clients (z. B. von Apache) dann in Zukunft voraussichtlich obsolet sein.

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
      .uri(URI.create("http://openjdk.java.net/"))
      .build();
client.sendAsync(request, asString())
      .thenApply(HttpResponse::body)
      .thenAccept(System.out::println)
      .join();
```

Durch den JEP 330 (Launch Single-File Source-Code-Programs) können jetzt Klassen gestartet werden, die noch nicht kompiliert wurden. Programme mit einer einzigen Datei sind heutzutage beim Schreiben kleiner Hilfsprogramme üblich und insbesondere

die Domäne von Skriptsprachen. Nun kann man sich auch in Java die unnötige Arbeit sparen und das verringert zugleich die Einstiegshürde für Java-Neulinge.

```
# java HelloWorld.java
// statt
# javac HelloWorld.java
# java -cp . hello.World
```

Auf unixoiden Betriebssystemen können Java-Dateien als Shebang-Files sogar direkt ausgeführt werden:

```
#!/path/to/java --source version

[...]
```

Der Aufruf erfolgt dann als ausführbare Datei ähnlich wie bei Shell-Skripten:

```
# ./HelloWorld.java
```

Weitere erwähnenswerte Änderungen sind die Unterstützung des Unicode-10-Standards und die Integration des bisher nur mit einer kommerziellen Lizenz verwendbaren Profiling-Tools Mission Control und Flight Recorder in das OpenJDK (sie wurden bisher nur mit dem Oracle JDK ausgeliefert). Das Ziel des Flight Recorders ist das möglichst effiziente Aufzeichnen von Anwendungsdaten, um bei Problemen die Java-Anwendung und die JVM analysieren zu können. Unsinnig scheint zunächst der JEP 318 (Epsilon: A No-Op Garbage-Collector) zu sein. Dabei handelt es sich um einen neuen Garbage-Collector, der aber gar keine Garbage-Collection durchführt (No-Operation). Interessant ist dieses Verhalten aber für Serverless-Functions für das Oracle Projekt fn. Da es sich hier um sehr kurz laufende Java-Anwendungen handelt, wäre ein zwischenzeitlicher Garbage-Collector-Lauf eher kontraproduktiv und sinnlos, wenn die Anwendung kurz darauf sowieso wieder beendet wird.

API-Änderungen

An der Java-Klassenbibliothek gab es natürlich auch unzählige kleine Änderungen. Besonders viel hat sich bei String und Character getan:

```
|  Welcome to JShell -- Version 11
|  For an introduction type: /help intro
// Unicode zu String
jshell> Character.toString(100)
$1 ==> "d"
jshell> Character.toString(66)
$2 ==> "B"
// Zeichen mit Faktor multiplizieren
jshell> "-".repeat(20)
$3 ==> "--------------------"

// Enthält ein Text keine Zeichen (höchstens Leerzeichen)?
jshell> String msg = "hello"
msg ==> "hello"
jshell> msg.isBlank()
$5 ==> false
jshell> String msg = "  "
msg ==> "  "
jshell> msg.isBlank()
$7 ==> true

// Abschneiden von führenden oder nachgelagerten Leerzeichen
jshell> " hello world ".strip()
$8 ==> "hello world"
jshell> "hello world    ".strip()
$9 ==> "hello world"
jshell> "hello world    ".stripTrailing()
$10 ==> "hello world"
jshell> "        hello world    ".stripLeading()
$11 ==> "hello world    "
jshell> "    ".strip()
$12 ==> ""

// Texte zeilenweise verarbeiten
jshell> String content =  "this is a multiline content\nMostly
obtained from some file\rwhich we will break into lines\r\nusing
the new api"
content ==> "this is a multiline content\nMostly obtained fro ...
ines\r\nusing the new api"
jshell> content.lines().forEach(System.out::println)
this is a multiline content
Mostly obtained from some file
which we will break into lines
using the new api
```

Was wurde aus Java entfernt?

Die Ankündigungen in Form von Deprecations in den Versionen 9 und 10 wurden nun in Java 11 in die Tat umgesetzt. So wurden im Rahmen des JEP 320 diverse Java-Enterprise-Packages aus Java SE entfernt, dazu zählen JAX-WS (XML basierte SOAP-Webservices inklusive den Tools wsgen und wsimport), JAXB (Java-XML-Binding inklusive den Tools schemagen und xjc), JAF (Java-Beans-Activation-Framework), Common-Annotations (@PostConstruct, @Resource, ...), CORBA und JTA (Java-Transaction-API).

Neu ist auch, dass das Oracle JDK kein JavaFX mehr enthalten wird, was mit dem OpenJDK übrigens noch nie ausgeliefert wurde. Stattdessen wird JavaFX über OpenJFX[7] als separater Download angeboten und kann wie jede andere Bibliothek in beliebigen Java-Anwendungen verwendet werden. Neben JavaFX wird auch der Support für Applets und Java Web-Start eingestellt. Die Open-Source-Community plant allerdings bereits Nachfolgeprojekte[8]. Wenn man im Moment noch Java Web-Start nutzen möchte, muss man zunächst beim Oracle JDK 8 bleiben und entweder ohne Sicherheits-Updates leben oder für den

kommerziellen Support ab 2019 Geld ausgeben.
Als Deprecated markiert wurde in Java 11 zudem die JavaScript-Engine Nashorn. Es ist davon auszugehen, dass sie in zukünftigen Java-Versionen verschwinden wird. Nashorn hat sich nie so richtig als serverseitige JavaScript-Implementierung gegenüber Node.js durchsetzen können. Und mit der GraalVM geht Oracle mittlerweile alternative Wege, um andere Programmiersprachen nativ auf der JVM auszuführen.

Übrigens wird es ab Java 11 die Java-Laufzeitumgebung (JRE) nur noch in der Server-Variante und nicht mehr für Desktops geben. Allerdings kann man für Desktop-Anwendungen mit dem Modulsystem und dem Werkzeug jlink mittlerweile selbst sehr einfach in der Größe angepasste Laufzeitumgebungen erstellen.

Fazit und Ausblick

Java 11 finalisiert angefangene Arbeiten aus den beiden vorangegangenen Versionen, damit es als LTS-Release für die nächsten drei Jahre gut gewappnet ist. Dazu wurden auch einige alte Zöpfe abgeschnitten und zum Beispiel JavaFX und diverse Java-EE-Packages aus dem JDK entfernt.

Mit Java 12 steht bereits das nächste Major-Release in den Startlöchern und wird voraussichtlich im März 2019 erscheinen[9]. Die Liste der Neuerungen wächst noch und enthielt Anfang November 2018 bereits die folgenden JEPs:

- JEP 325: Switch-Expressions
- JEP 326: Raw-String-Literals
- JEP 340: One AArch64 Port, Not Two
- JEP 341: Default CDS-Archives

Während es sich bei JEP 340 um Aufräumarbeiten an der ARM-Portierung des JDK und bei JEP 341 um Erweiterungen zur Verbesserung der Performance durch Class-Data-Sharing handelt, sind für Entwickler insbesondere die ersten beiden Punkte interessant. Beide Sprach-Features stammen aus sogenannten Inkubator-Projekten (Amber[10], Valhalla[11], Loom[12]), in denen kleinere, die Entwicklerproduktivität steigernde Sprach- und VM-Features ausgebrütet und dann als Java-Enhancement-Proposals (JEPs) akzeptiert werden.

Switch-Statements können demnach in Zukunft auch als Expression verwendet werden, die das Ergebnis des entsprechenden Case-Zweigs direkt zurückliefert.

```
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
```

Mit Raw-String-Literals wird Java endlich die Möglichkeit bekommen, mehrzeilige Zeichenketten zu definieren, die zusätzlich Escape-Sequenzen ignorieren. Damit lässt sich viel einfacher mit regulären Ausdrücken und Windows-Dateipfaden umgehen. Einzig das Ersetzen von Variablen (String-Interpolation) ist im Moment noch nicht geplant, ein Feature, welches alternative Sprachen wie Groovy, Ruby und JavaScript schon länger unterstützt.

```
Runtime.getRuntime().exec(`"C:\Program Files\foo" bar`);
// statt
Runtime.getRuntime().exec("\"C:\\Program Files\\foo\" bar");

String script1 = `function hello() {
  print('"Hello World"');
}

hello();`;

// statt
String script2 = "function hello() {\n" +
                 "   print(\'\"Hello World\"\');\n" +
                 "}\n" +
                 "\n" +
                 "hello();\n";
```

Das sieht schon sehr vielversprechend aus und wir dürfen gespannt sein, was es bis zum Feature-Freeze beim Start der Rampdown-Phase 1 Mitte Dezember noch in das JDK 12 schaffen wird.

Das nächste LTS-Release (Java 17) wird erst im Herbst 2021 herauskommen. Bis dahin sollte in Produktion entweder weiterhin auf Java 8 oder das aktuelle Java 11 gesetzt werden. Die Änderungen der nächsten Zwischen-Releases kann man natürlich leicht verfolgen, immerhin sind die halbjährlichen Versionen gut überschaubar. Die nächsten Jahre scheinen für Java-Entwickler spannend zu bleiben und man darf sich auf regelmäßige Verbesserungen an der Sprache freuen.

Quellen:

1 JDK 11 Projektseite: http://jdk.java.net/11/
2 Java SE Support-Verträge: https://bit.ly/2Brc3Ay
3 AdoptOpenJDK: https://adoptopenjdk.net/
4 RHEL OpenJDK: https://red.ht/2adYXY6
5 IBM SDK: https://ibm.co/2r286MB
6 OpenJDK Community Zusammenarbeit: https://bit.ly/2QcIwm5
7 OpenJFX: http://openjdk.java.net/projects/openjfx/
8 Opensource Java Webstart: https://dev.karakun.com/webstart/
9 OpenJDK 12 Projektseite: https://bit.ly/2E5OLT1
10 Project Amber: http://openjdk.java.net/projects/amber/
11 Project Valhalla: http://openjdk.java.net/projects/valhalla/
12 Project Loom: http://openjdk.java.net/projects/loom/

#JAVAPRO #Annotation

Deep-Dive into Annotations - Teil 1

Java-Annotationen sind ein mächtiges Sprachmerkmal, deren Interna vielen Entwicklern wahrscheinlich nicht sehr bekannt sind. In Teil 1 unserer dreiteiligen Serie wird aufgezeigt, was Annotationen eigentlich sind, und für welche Szenarien die fünf Standard-Annotationen gedacht sind.

Mit Version 5 der Java-Plattform, Standard-Edition (Java SE) im Jahre 2004 erhielten Annotationen nebst diversen anderen bedeutenden Sprachänderungen erstmals Einzug in die mächtige Programmiersprache. Obwohl deren Einführung also bereits 14 Jahre her ist, führen Annotationen gefühlt heute noch immer ein Schattendasein. In der Populärliteratur zur Java SE werden meist nur die gängigen Standardannotationen wie z. B. `@Override` oder `@SuppressWarnings ("unchecked")` angesprochen. Beschreibungen zu komplizierteren Annotationen wie z. B. `@SafeVarargs` findet man sogar in der einschlägigen Spezialliteratur nur selten.

Jeder Java-Programmierer wird mit Annotationen sicher schon seine Bekanntschaft gemacht haben. Sei es mehr oder weniger freiwillig, dass einen z. B. die Verwendung des soeben erwähnten `@Override` überzeugt hat bzw. ans Herz gelegt wurde, oder dass man mit einem `@SuppressWarnings("unchecked")` am richtigen Ort im Quelltext endlich seine Ruhe vor den lästigen Compiler-Warnungen hat, die auch nach langem Tüfteln mit Generics, Type-Casts und Arrays nicht auf dem regulären Weg verschwinden wollten.

Die Beschränkung auf die zwei typischen Standardannotationen verfehlt allerdings die Tatsache, dass es sich bei Annotationen um ein sehr flexibles und mächtiges Sprachmerkmal handelt, dessen Potenzial es noch auszuschöpfen gilt. In Teil 1 dieser Serie werden zuerst die bekannten wie auch selteneren Standardannotationen vorgestellt. In Teil 2 wird anschliessend

Autor:

Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Software-Entwicklung, -Schulung und -Beratung v. a. für MINT-Anwendungen und technische Implementierungsthemen in Java spezialisiert hat. Er ist seit 15 Jahren mit Java vertraut und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.

christian.heitzmann@simplexacode.ch
https://www.simplexacode.ch

aufgezeigt, wie sich eigene Annotationstypen erstellen lassen und in Teil 3, wie man diese Annotationen programmgesteuert auswerten kann. Denn gerade in beiden letzteren liegt die eigentliche Stärke der Annotationen, mit dem Java die Tür hin zur deklarativen Programmierung einen Spalt weit geöffnet hat.

Definition und Beispiele

Die wohl prägnanteste Definition für Annotationen findet sich gleich im ersten Satz von Abschnitt „9.7. Annotations" der Java Language Specification: „An annotation is a marker which associates information with a program construct, but has no effect at run time."

Bei Annotationen handelt es sich also um Metainformationen, oder salopp ausgedrückt, um „Informationen über Informationen", vergleichbar mit der ISBN-Nummer eines Buches. Die ISBN-Nummer ändert nichts am Inhalt des Buches (den eigentlichen Informationen), denn die Inhalte des Buches lassen sich genau gleich erschließen, unabhängig davon, ob das Buch nun eine ISBN-Nummer trägt oder nicht.

Annotationen tun selbst nichts, sie deklarieren nur Programmkonstrukte. Ein Buch mit einer ISBN-Nummer kann sich nicht selbst zum Haushalt eines Kunden liefern. Dazu braucht es immer noch jemand anderen, der das Buch entweder manuell oder automatisiert verpackt, transportiert und ausliefert. Gleich verhält es sich bei Annotationen. Um mit ihnen etwas anfangen zu können, braucht es einen Dritten, der den Java-Quelltext oder die Klassendatei inspiziert und bei Annotationen entsprechend reagiert, z. B.:

- Der Compiler, der die Korrektheit der Standardannotationen (z. B. **`@Override`**) im Zusammenspiel mit dem Quelltext überprüft und bei Bedarf eine Fehlermeldung generiert.
- Ein Code-Analyse-Tool, welches die technische Code-Qualität beurteilt, dabei aber bewusst platzierte Ausnahmebemerkungen (also Annotationen) des Entwicklers im Quelltext in der Analyse ignoriert.
- Ein Test-Framework, in denen die Testfälle annotiert werden, so dass sie als automatisierte Tests auffindbar, korrekt ausführbar und sinnvoll auswertbar sind.
- Ein Web-Server, der einen Java-Webservice, seine Methoden und deren Parameter anhand von Annotationen erkennt, automatisch konfiguriert und korrekt typisiert.
- Eine Java-XML-API, die entsprechend annotierte Java-Komponenten automatisch und korrekt zwischen Java und XML abbilden kann.
- Eigene Programme, die andere Klassen oder Objekte inspizieren, insbesondere mit Hilfe von Reflection oder sogenannten Annotationsprozessoren.

Die Grundidee von Annotationen war eigentlich auch in Java 5 nicht mehr neu. Bereits Javadoc, welches seit den Urtagen von Java existiert, ermöglichte dem Programmierer in seinen Kommentaren zu den Paket-, Klassen- und Methodenbeschreibungen gewisse Javadoc-Tags wie z. B. **`@author`**, **`@param`**, **`@return`** oder **`@throws`** zu verwenden, die dann vom Javadoc-Tool ausgewertet wurden und zu hervorragend strukturierten und indexierten HTML-Dokumentationen führten. Während die Javadoc-Kommentare und die daraus resultierenden HTML-Dokumentationen nur für Menschen als Zielpublikum gedacht sind, sind die Java-Annotationen vor allem für Maschinen gemacht. Einfach ausgedrückt, kann man Annotationen auch als Javadoc für Maschinen verstehen. Dass sowohl den Javadoc-Tags als auch den Annotationen ein **`@`** vorangestellt wird, ist also sicher kein Zufall.

Standard-Annotationen

Ein Blick in die Liste der All-Known-Implementing-Classes der Dokumentation des Interfaces **`java.lang.annotation.Annotation`** verrät, welche und wie viele vordefinierte Annotationen es in der Java SE gibt. Jede Annotation erweitert die Schnittstelle **`Annotation`**, allerdings lassen sich keine eigenen Annotationen erzeugen, indem man einfach dieses Interface erweitert (mehr dazu in Teil 2). Für Java 8 zählt man über 80 Annotationen. Ähnlich wie bei den Exceptions befinden sich dabei aber sehr viele in Spezialpaketen (beginnend mit **`javax`**) oder dienen nur ganz speziellen und eher untypischen Anwendungszwecken. Als Standardannotationen im engeren Sinn gibt es in der Java SE fünf im Paket **`java.lang`** (namentlich **`@Deprecated`**, **`@FunctionalInterface`**, **`@Override`**, **`@SafeVarargs`** und **`@SuppressWarnings`**) sowie sechs im Paket **`java.lang.annotation`**, wobei letztere lediglich Annotationen für Annotationen darstellen, sich also selbst annotieren. In Teil 2 werden die sogenannten Metaannotationen vorgestellt, nämlich die beiden wichtigen Annotationen **`@Retention`** und **`@Target`** sowie **`@Documented`**, **`@Inherited`**, **`@Native`** und **`@Repeatable`**. Im folgenden Code-Beispiel, das lediglich zur Erklärung konstruiert wurde und nicht für den Einsatz in der Praxis gedacht ist, werden alle fünf Standardannotationen verwendet.

(Listing 1)

```
import java.util.*;

public final class StandardAnnotationsDemoClass
    implements StandardAnnotationsDemoInterface {

  public StandardAnnotationsDemoClass() {}

  public static void main(String[] args) {
    StandardAnnotationsDemoClass demoClass
        = new StandardAnnotationsDemoClass();
    StandardAnnotationsDemoInterface.staticInterfaceMethod();
    demoClass.defaultInterfaceMethod();
    demoClass.regularInterfaceMethod();
    demoClass.safeVarargsMethod(Optional.of("Alpha"),
                                Optional.of("Beta"),
                                Optional.of("Gamma"));
    demoClass.outdatedMethod();
  }
```

```
  @Override
  public void regularInterfaceMethod() {

  /* Fails at compile time. Note the typo! */
//  public void regluarInterfaceMethod() {

    System.out.println("I'm a regular interface method.");
  }

  /* Emits compiler warnings if annotation removed. */
  @SafeVarargs
  public final void safeVarargsMethod(Optional<String>...
                                      arguments) {
    Object[] objectArray = arguments;
    objectArray[1] = Optional.of("Delta");

    /* Pollutes heap and makes loop below fail. */
//    objectArray[1] = Optional.of(new Integer(1));

    /* Emits compiler warning if annotation removed. */
    @SuppressWarnings("unchecked")
    Optional<String>[] stringOptionalArray
        = (Optional<String>[]) objectArray;

    for (Optional<String> currentStringOptional
         : stringOptionalArray) {
      String string = currentStringOptional.get();
      System.out.println(string);
    }
  }

  @Deprecated
  public void outdatedMethod() {
    System.out.println("I'm an outdated method.");
  }
}

@FunctionalInterface
interface StandardAnnotationsDemoInterface {

  public static void staticInterfaceMethod() {
    System.out.println("I'm a static interface method.");
  }

  public default void defaultInterfaceMethod() {
    System.out.println("I'm a default interface method.");
  }

  public void regularInterfaceMethod();

  /* Fails at compile time. */
//  public void anotherRegularInterfaceMethod();
}
```

Im Folgenden werden die einzelnen Standard-Annotationen in der Reihenfolge aufsteigender Komplexität kurz erläutert.

@Deprecated

Veraltete Methoden, die nicht mehr verwendet werden sollen, werden mit der Annotation **`@Deprecated`** versehen. Dies entspricht dem Javadoc-Tag **`@deprecated`** (klein geschrieben!), nur dass mit der Annotation diese Information nun auch dem Compiler zur Verfügung steht. In Eclipse wird im Code-Beispiel die **`outdatedMethod`** durchgestrichen dargestellt. Denkbar wäre auch eine Compiler-Warnung. Im Gegensatz zum Javadoc-Tag ist die Annotation immer verfügbar, also auch dann, wenn der Java-Quelltext oder die Dokumentation einer Bibliothek mit einer als deprecated-markierten Methode nicht zur Verfügung steht, oder wenn in der IDE die Verknüpfung zur Dokumentation nicht richtig eingerichtet ist.

@Override

Die **`StandardAnnotationsDemoClass`** implementiert das **`StandardAnnotationsDemoInterface`**, deren **`regularInterfaceMethod`** es zu überschreiben gilt. Wird eine Methode einer Oberklasse oder eines Interfaces überschrieben, so kann der Programmierer dies mit einer **`@Override`** Annotation versehen, was jedoch kein Muss ist. Dennoch ist es empfehlenswert, in der IDE den Compiler so einzustellen, dass bei fehlender Annotation eine Warnung ausgegeben wird.

Der Zweck ist folgender: Wird bei einer im guten Glauben überschreibenden Methode die **`@Override`** Annotation angebracht, dann MUSS die damit annotierte Methode auch wirklich überschreiben. Wenn die zu überschreibende Methode in der Oberklasse oder im Interface nicht (mehr) existiert, oder wenn die Methodensignaturen nicht identisch sind, was insbesondere bei Schreibfehlern oder falschen Parametertypen passieren kann, führt dies zu einem Compiler-Fehler. Das kann man ausprobieren, indem an die Methode **`regularInterfaceMethod`** durch **`regLUarInterfaceMethod`** ersetzt, die mit einem Schreibfehler behaftet ist. **`@Override`** stellt somit eine Absicherung dar, um in der Praxis schwierig auffindbare Fehler durch fehlerhaftes Überschreiben von Methoden zu vermeiden. Die Folgen eines kleinen Schreibfehlers können nämlich verheerend sein, insbesondere bei nicht abstrakten Oberklassen: Der Compiler wird sich nicht beschweren, die Laufzeitumgebung ebenso wenig, aber der Polymorphismus geht verloren. Das Programm wird stillschweigend die falsche Methode ausführen, nämlich diejenige der Oberklasse.

@FunctionalInterface

Die Annotation **`@FunctionalInterface`** kam mit Java 8 hinzu, was mit der elementaren Bedeutung der Functional-Interfaces in Zusammenhang mit der ebenfalls in Java 8 eingeführten funktionalen Programmierung und den Lambda-Ausdrücken zu erklären ist. Zur Erinnerung: Eine funktionale Schnittstelle ist ein Interface, das genau eine abstrakte Methode besitzt, mit Ausnahme der Methoden aus **`Object`**. Seit Java 8 lassen sich Interfaces ebenfalls mit Default-Implementierungen und statischen Implementierungen versehen. Diese Methoden sind ausdrücklich nicht abstrakt und können daher in beliebiger Menge auch in einer funktionalen Schnittstelle vorkommen. So ist z. B. **`java.util.Comparator<T>`** eine funktionale Schnittstelle, obwohl sie aus 18 Methoden besteht.

Die Annotation **`@FunctionalInterface`** markiert ein Interface als funktionale Schnittstelle. Ähnlich wie die Annotation **`@Override`** dient auch sie damit als Sicherheit, falls die Schnittstelle eines Tages durch Hinzufügen einer weiteren abstrakten Methode plötzlich nicht mehr funktional sein sollte. In der Folge würden nämlich sämtliche Lambda-Ausdrücke nicht mehr funktionieren. Würde man in aufgezeigtem Code-Beispiel der funktionalen Schnittstelle **`StandardAnnotationsDemoInterface`** die zusätzliche abstrakte Methode **`anotherRegularInterfaceMethod`** hinzufügen, so würde dies aufgrund der **`@FunctionalInterface`** Annotation sofort zu einem Compiler-Fehler führen.

@SuppressWarnings

In der Methode **`safeVarargsMethod`** des Code-Beispiels wird ein **`objectArray`** (also ein Array vom Typ **`Object[]`**) einem Array **`stringOptionalArray`** vom generischen Typ **`Optional<String>[]`** zugewiesen. Es ist klar, dass dieses „maskierte" **`Object`**-Array zur Laufzeit ein **`Optional`** Array sein muss, da es ansonsten sofort eine unmissverständliche **`ClassCastException`** geben würde. Weniger klar hingegen ist, welchen generischen Typ die **`Optionals`** im Array haben, denn Java verfügt bei der Verwendung von Generics im Zusammenspiel mit Arrays diesbezüglich über keine Laufzeitinformationen. Auch hat der Compiler keine Möglichkeit, die Typsicherheit vorab sicherzustellen. Er gibt daher eine Warnung aus: **`Type safety: Unchecked cast from Object[] to Optional<String>[]`**. Mit der Annotation **`@SuppressWarnings("unchecked")`** gibt der Programmierer dem Compiler ein Versprechen ab, sich von der Typsicherheit vergewissert zu haben, also dass es sich in diesem Array wirklich um **`Optional<String>`** und nicht etwa um **`Optional<Integer>`** handelt. Der Compiler kann daraufhin auf seine Warnung verzichten.

Es versteht sich von selbst, dass eine **`@SuppressWarnings`** Annotation nicht leichtfertig angewendet werden darf und nur den kleinstmöglichen Geltungsbereich umfassen soll. Man sollte also auf keinen Fall eine ganze Methode oder gar eine ganze Klasse damit annotieren! Das Zusammenspiel von Generics und Arrays in Java ist speziell und alles andere als trivial.[1]

@SafeVarargs

Seit Java 5 kann man Methoden eine variable Anzahl an Parametern übergeben. Deklariert werden solche Parameter mit Auslassungspunkten, z. B. **`safeVarargsMethod(Optional<String>... arguments)`** in vorliegendem Code-Beispiel. In Java bezeichnet man diese variable Anzahl an Methodenparametern oder -argumenten als Varargs. Intern werden Varargs ganz einfach durch ein Array entsprechenden Typs und entsprechender Größe realisiert. Jedes Argument des Varargs entspricht dann einem Element des Arrays, auf das die Methode Zugriff hat. Bei **`Optional<String>...`** führt diese Umwandlung zu einem generischen Array **`Optional<String>[]`**. Wie im vorangegangenen Punkt zu **`@SuppressWarnings`** bereits erwähnt, vertragen sich Generics und Arrays nicht sonderlich gut, was der Compiler direkt, wenn auch unverständlich, zur Sprache bringt. Beim Aufrufer meldet er: **`Type safety: A generic array of Optional<String> is created for a varargs parameter.`** Bei der Methodendeklaration lautet die Warnung: **`Type safety: Potential heap pollution via varargs parameter arguments`**. Diese Heap-Verschmutzung kann insbesondere dann auftreten, wenn das Array anschliessend ungeschickt beschrieben oder aus der Hand gegeben wird, also die Methode verlässt. Ein nicht vertrauenswürdiger Client (und davon ist grundsätzlich immer auszugehen) könnte dieses Array dann direkt bearbeiten.

Im Code-Beispiel ist eine solche Möglichkeit einer Heap-Pollution aufgezeigt: Das Argumenten-Array **`arguments`** vom Typ **`Optional<String>[]`** wird einem **`Object[]`** zugewiesen. In diesem **`Object`**-Array können die Elemente auch beschrieben werden. Im Code-Beispiel wird das Überschreiben des Elementes am Index **`1`** von **`Optional.of("Beta")`** auf **`Optional.of("Delta")`** funktionieren. Erstaunlicherweise wird aber auch die (noch auskommentierte) Zuweisung mit **`Optional.of(new Integer(1))`** funktionieren, ohne dass es an dieser Stelle zu einer Exception kommt. Erst weiter unten im Code, wenn in der Schleife das Element aus dem **`Optional`** extrahiert und auf einen **`String`** gecastet wird, kommt es zu einer **`ClassCastException`**. Auf diesen Umstand möchte der Compiler mit seiner Warnung hinweisen.

Hat sich der Programmierer davon vergewissert, dass eine derartige Heap-Pollution nicht auftreten kann, so kann er die Warnung mit einer **`@SafeVarargs`** Annotation deaktivieren. Diese Garantie besteht auf jeden Fall dann, wenn die Methode das Argumenten-Array weder beschreibt noch von außen Zugriff auf dieses Array gewährt – auch nicht aus Versehen! Für den normalen, klassischen Anwendungsfall von Varargs dürfte es eigentlich keine Schwierigkeiten geben.

Fazit:

Die fünf hier vorgestellten Standardannotationen dienen (noch) ausschließlich dem Compeiler. Außer einer gewissen Sicherheit und allenfalls etwas weniger händischer Dokumentationsarbeit ist der Vorteil von Annotationen noch nicht auf Anhieb offensichtlich. Aus diesem Grund werden in den folgenden beiden Teilen die Definition und Auswertung eigener Annotationen detailliert vorgestellt.

Quellen:

1 Mehr zum Thema: link.simplexacode.ch/rk57

#JAVAPRO #Architecture #IODA #FlowDesign

Alles im Fluss – Wider den Abhängigkeiten

Abhängigkeiten sind das Grundübel in Software-Systemen. Sie machen Software komplex und oft unwartbar. Auf unterster Ebene werden Abhängigkeiten durch Methodenaufrufe induziert, die sich dann auf abstrakteren Ebenen zu einem Abhängigkeitsgestrüpp hochschaukeln, das manchmal kaum noch zu entwirren ist. Flow-Design ist angetreten, die Abhängigkeiten einzudämmen und auf ein gesundes Maß zu beschränken.

Autor:

Denis Kuniß arbeitet bei Diebold Nixdorf in der Platform-Software. Er hat an der TU Berlin studiert und sich dort mit Compiler-Generierung beschäftigt. Seine Interessen gelten den formalen und domänenspezifischen Sprachen und dem Architekturentwurf. Er entwickelt in Java, Xtend, Scala und mit Xtext und Gradle.

blog.grammarcraft.de
https://www.xing.com/profile/
https://www.linkedin.com/in/denis-kuniss-3037929/
https://github.com/kuniss
kuniss@grammarcraft.de

Ohne Methoden und ohne Methodenaufrufe – das geht doch gar nicht! Wie sonst soll Logik modularisiert und zu größeren Logikkonstrukten zusammengebaut werden als durch Methoden und Methodenaufrufe. Wir sind so in der Denkwelt der Subroutinen, Methoden und Funktionen verhaftet, dass eine andere Lösung undenkbar erscheint. Aber ist dies wirklich die einzige Möglichkeit der Programmierung, alles in aufrufbare Subroutinen zu zerteilen?

Message-Passing

An dieser Stelle soll nicht dagegen argumentiert werden, sondern ein Beispiel gegeben werden, dass es auch anders geht. Die Unix-Shell-Programmierung unter Verwendung des Pipe-Operators ist eines der bekanntesten Programmiermodelle, die in ihrem zentralen Prinzip nicht auf Subroutinen-Aufrufen, sondern

auf Message-Passing beruht. Und man kann nicht behaupten, dass es nicht erfolgreich wäre. Nachfolgend ein Beispiel-Skript aus der RPM-Toolbox.

(Listing 1) – find-requires

```
#!/bin/sh
. ./determineLibraryDependencies.sh
echo $* | determineLibraryDependencies | sort -u -r | xargs -r -n
1 basename
```

(Listing 1) zeigt ein leicht angepasstes Shell-Skript des Kommandos **find-requires**. Das Skript bestimmt für eine Liste von ausführbaren Dateien die Bibliotheksabhängigkeiten (im Linux-System) durch Anwendung des Befehls **determineLibraryDependencies** auf jede einzelne Datei - lokal definiert im importierten Shell-Skript **determineLibraryDependenies.sh**.

Die Liste der Bibliotheksabhängigkeiten, alles .so-Dateien, wird anschließend sortiert, Duplikate werden eliminiert, absolute Pfade abgeschnitten.

Jeder der hier beschriebenen Schritte wird im Skript durch eine Funktionalität realisiert, die entweder als Kommando des Betriebssystems bereits existiert oder in einem externen Skript definiert ist. Betrachtet man die letzte Zeile des find-requires-Skripts eingehender, wird der eigentliche Zweck des Skriptes selbst klar: es ist nur dazu da, vorhandene Funktionalität zu integrieren und Funktionseinheiten miteinander zu verbinden, damit sie miteinander kommunizieren können. Die Daten fließen quasi durch die Funktionseinheiten hindurch und werden durch sie verändert. Es gibt im Skript selbst keinerlei logische Ausdrücke oder Kontrollstrukturen, nur integrative Logik – den Pipe-Operator und eine Skript-Import-Anweisung.

In (Listing 2) wird die einzige gegenüber dem Original neu programmierte Funktionseinheit **determineLibraryDependencies** näher betrachtet.

(Listing 2) – DetermineLibraryDependencies.sh

```
determineLibraryDependencies() {
  read input
  for f in $input; do
    ldd $f | awk '/=>/ { print $1 }'
  done
}
```

Auch wenn es wie eine Methode aussieht, ist es keine im Sinne einer prozeduralen Sprache! Die Eingaben werden vom Standard-Eingabestrom gelesen ohne Kenntnis, welche Funktionseinheit diesen Eingabestrom erzeugt. Aber noch wichtiger ist: Der Funktionseinheit ist auch unbekannt, wer den von ihr selbst erzeugten Ausgabestrom konsumieren wird. Diese Funktionseinheit ist tatsächlich unabhängig von ihrem Kontext.

Dies ist ein signifikanter Unterschied zu Subroutinen, die andere Subroutinen, welche im logischen Datenfluss später folgen, direkt aufrufen. Dadurch wird eine direkte Abhängigkeit zwischen der aufrufenden und der aufgerufenen Subroutine induziert, die bei späteren Anpassungen des Codes nicht unbeachtet bleiben darf. In Message-Passing basierten Design-Ansätzen wie der Unix-Kommando-Shell-Programmierung mit dem Pipe-Operator ist die Abhängigkeit von der aufgerufenen Subroutine in die integrierende Funktionseinheit, hier z.B. **find-requires**, verschoben. Aber dort gehört sie auch hin, denn integrierende Funktionseinheiten haben nach der IODA-Architektur[1] nur den einen Zweck zu integrieren. Sie sollen keine Logik direkt implementieren.

Prinzip der gegenseitigen Nichtbeachtung

In der Unix-Shell-Programmierung hat sich dieses Programmierprinzip fast automatisch ergeben. Es resultiert aus dem Umstand, dass alle vom Betriebssystem bereitgestellten Kommandozeilenprogramme eine eindeutige Schnittstelle haben – die zeilenbasierte Verarbeitung auf **stdin** und **stdout**. Die daraus folgende Methode des Komponierens vorhandener Funktionseinheiten zu neuen, abstrakteren Funktionseinheiten ist in Shell-basierten Programmsystemen weit verbreitet. Außerdem ist es auch für Externe einfach zu verstehen und zu erweitern. Die beteiligten Kommandos können sich gegenseitig nicht kennen, denn sie laufen in vom Betriebssystem voneinander abgeschotteten, separaten Prozessen. Bei der Implementierung der Kommandos konnte nicht auf die Funktionalität der anderen Kommandos zugegriffen werden, so wie wir es von der objektorientierten Programmierung her für Klassen und Methoden kennen.

Für prozedurale Programmiersprachen (dazu werden im weitesten Sinne alle, die auf Subroutinen basierenden, also auch objektorientierte und funktionale Programmiersprachen, gezählt) ist es möglich, nach diesem Prinzip Implementierungen vorzunehmen. Aber die ergeben sich nicht automatisch, sondern erfordern Disziplin, weshalb es Sinn macht, diesem Prinzip einen Namen zu geben. Ralf Westphal und Stefan Lieser, die den Begriff des Flow-Designs geprägt haben, nennen[2] es das „Prinzip der gegenseitigen Nichtbeachtung" (Principle of Mutual Oblivion – PoMO).

Das Prinzip drückt sich zuallererst darin aus, mit welcher Notation Flow-Design-basierte Systeme modelliert werden. Es ist eine sehr einfache Notation, kein Vergleich zur UML, die für viele überspezifiziert und zu detailliert ist. Im ersten Teil dieser Serie „Bessere Abstraktion mit IODA"[1] konnte man von der Notation schon einen ersten Eindruck bekommen. Im Flow-Design setzen sich Systeme aus Funktionseinheiten zusammen. Solch eine Funktionseinheit, wie als Beispiel in (Abb. 1) zu sehen, wird als Ellipse oder Rechteck dargestellt und mit der Funktionalität beschriftet, die es realisiert. Die Funktionseinheit hat eine oder mehrere Ein- und Ausgänge, welche die in die Funktionseinheit ein- und ausfließenden Daten repräsentieren sollen und Ports genannt

werden. Die Ein- und Ausgabe-Ports sind bei Bedarf mit in Klammern gesetzten Typen der Daten, die darüber fließen, annotiert. Dies kann auch eine Aufzählung oder Liste (mit Stern markiert) sein. Eine leere Klammer markiert ein Signal ohne spezifisches Datum und symbolisiert zum Beispiel den parameterlosen Aufruf eines Skripts oder Programms.

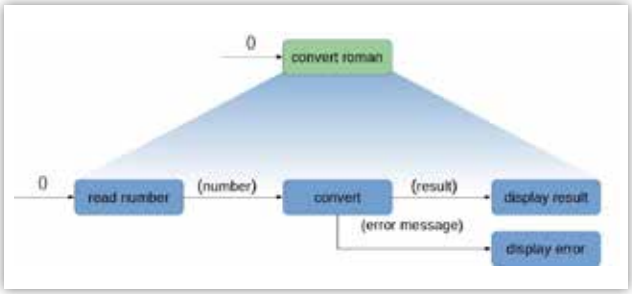
find-requires als Funktionseinheit. (Abb. 1)

Über die Ein- und Ausgabe-Ports werden die Funktionseinheiten im Kontext einer integrierenden Funktionseinheit verbunden. Sie ergeben damit eine Datenverarbeitung, die die Funktionalität der integrierenden Funktionseinheit repräsentiert, wie im Beispiel in (Abb. 2) zu sehen. Die Grundidee der Flow-Design-Notation folgt dem Box-Bullet-Line-Prinzip[3] und versucht, die Notation nicht zu überfrachten, um die Einstiegshürden für das Verstehen einer Design-Zeichnung möglichst gering zu halten. Im Idealfall sind die obigen Ausführungen ausreichend. Weitere Details zur Flow-Design-Notation sind in Stefan Liesers Cheatsheet[4] zu finden. Generell sollte man die Notation nicht zu formal begreifen. Hauptziel ist ein gemeinsames Verständnis des Designs, damit man darüber diskutieren kann. Wir werden noch sehen, dass das Flow-Design eines Systems sehr einfach aus dem Code ableitbar ist, sofern dessen Umsetzung vom PoMO-Prinzip geleitet wurde.

Flow-Design implementieren

Zurück zum Beispiel aus Teil 1 unserer Serie[1]. Dort wurde die Umwandlung von römischen Zahlen in arabische als Flow-Design entworfen, mit dem Ziel, eine IODA-Architektur zu erhalten. Dieses Ziel wird mit Flow-Design immer erreicht, denn jedes Flow-Design ergibt automatisch eine IODA-Architektur, wenn man den Code nach dem Prinzip der gegenseitigen Nichtbeachtung (PoMO) umsetzt.

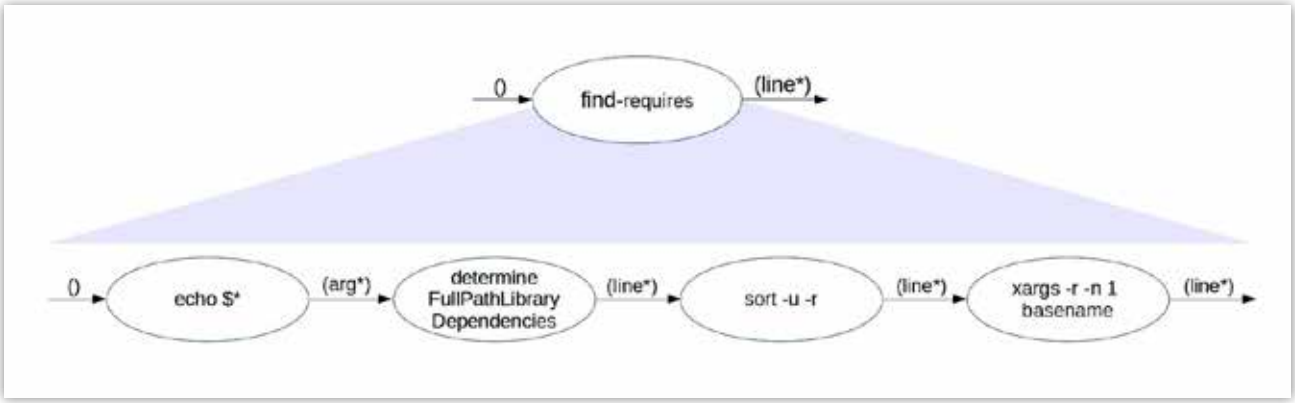
Aber wie soll eine Implementierung aussehen, die keine Methoden-Aufrufe verwendet? Nun, wir werden die Methodenaufrufe nicht ganz los, dafür müssten wir eine andere (wahrscheinlich nicht-existente) Programmiersprache als Java verwenden, die nicht auf der von-Neuman-Architektur heutiger Rechner basiert. Aber es gibt Konzepte wie das Observer-Pattern, die Actor-Programmierung oder die Verwendung von Continuation-Parametern, über die sich die Bestimmung welcher Methodenaufruf erfolgen soll, auf die Laufzeit verschieben lässt. Somit kann das Prinzip der gegenseitigen Nichtbeachtung zur Programmierzeit eingehalten werden.



Oberste Funktionseinheit convert roman. (Abb. 3)

Als erstes Beispiel zur Realisierung einer Java-Implementierung wird die oberste Funktionseinheit **`convert roman`** aus Teil 1 der Serie verwendet, wie in (Abb. 3) zu sehen ist. Es ist der Einstiegspunkt für das Programm.

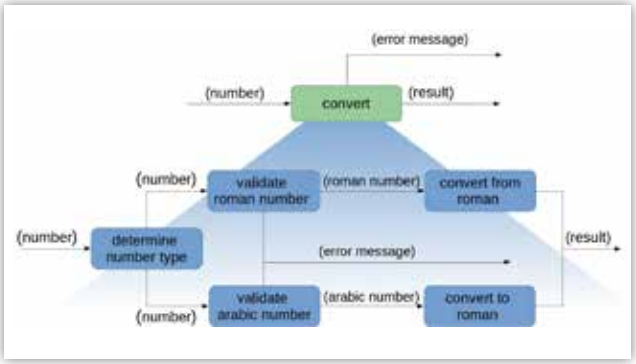
Der resultierende Code ist in (Listing 3) zu sehen. Man liest den Code einfach von oben nach unten und von links nach rechts. Einrückungen repräsentieren verschiedene Pfade im Datenfluss. Die beiden hinteren Argumente sind Lambda-Ausdrücke, wie sie seit Java 8 möglich sind. Sie repräsentieren die beiden Fortsetzungsalternativen im Kontrollfluss, nachdem die Logik in **`convert`** abgearbeitet ist. Zudem entsprechen sie den beiden Ausgängen aus dieser Funktionseinheit, wie im Diagramm zu sehen ist. Die erste Lambda-Funktion entspricht der Funktionseinheit **`display result`** aus (Abb. 3), die zweite der Funktionseinheit **`display error`**.



Verbundene Funktionseinheiten in find-requires. (Abb. 2)

(Listing 3) – convert roman

```
public void run() {
    String number = this.input.read_number_to_convert();
    this.body.convert (number,
        result -> this.output.display_result(result),
        errorMessage -> this.output.display_error(errorMessage));
}
```



Funktionseinheit convert. (Abb. 4)

Im zweiten Beispiel zeigt sich, wie die Funktionseinheit **`convert`** implementiert wird, die ja in **`convert roman`** integriert ist. Jedes Rechteck im Diagramm aus **(Abb. 4)** wird in einen Methodenaufruf übersetzt, mit den Eingängen als Parametern. Schwieriger wird es bei den Ausgängen: Wenn das Rechteck nur einen Ausgang hat, wird eine Funktion verwendet, die das Ergebnis der Berechnung oder Transformation als Return-Wert zurückliefert. Hat ein Rechteck jedoch mehr als einen Ausgang, dann erfolgt die Umsetzung der Ausgänge als Methodenparameter des Java-Typs **`Consumer`**[5] mit dem Datentyp als generischem Parameter. Auf diese Weise wird die Implementierung auch typsicher. **(Listing 4)** zeigt diese Implementierung der Methode **`convert`** mit zwei Consumer-Parametern. Diese Art von Parametern werden englisch auch Continuations genannt, weil der Kontrollfluss des Programms über sie weiterläuft.

(Listing 4) – Funktionseinheit convert in Java

```
public void convert(String number,
                    Consumer<String> onSuccess, Consumer<String>
                    onError)
{
    RomanConversions.determineNumberType (number,
        romanNumber -> RomanConversions.validateRomanNumber
        (romanNumber,
            () -> {
                    int result = FromRomanConversion.convert
                    (romanNumber);
                    onSuccess.accept(Integer.toString(result));
                },
            onError),
        arabicNumber -> RomanConversions.validateArabicNumber
        (arabicNumber,
            () -> {
                    String result =
ToRomanConversion.convert(arabicNumber);
                    onSuccess.accept(result);
                },
            onError)
    );
}
```

Anfänglich sind Continuations ungewohnt zu lesen. Als Entwickler ist man darauf trainiert, geschachtelte Aufrufe von rechts nach links zu lesen, obwohl es der natürlichen Leserichtung in unserem Kulturkreis widerspricht. Vielleicht ein zusätzlicher Grund, warum Quellcode oft schwer zu verstehen ist. Aber mit ein wenig Übung ist es erstaunlich, wie gut Quellcode dadurch lesbar wird.

Dies ist so ähnlich wie mit den ersten objektorientierten Implementierungen, die noch auf C basierten. Nachdem man sein Denken darauf umgestellt hatte, dass das erste Argument immer den Empfänger des Aufrufs darstellt, erschloss sich einem der Zweck des Codes viel einfacher.

Fazit:

Wie der objektorientierte Ansatz ist auch der Flow-Design-Ansatz so fundamental, dass sich die Umsetzung als eigenständiges Sprachkonstrukt geradezu aufdrängt. Wie wäre es, wenn man das Flow-Design aus **(Abb. 3)** auf folgende Weise in einer Sprache ausdrücken könnte:

(Listing 5)

```
read_number_to_convert → convert
convert.result → display_result
convert.error → display_error
```

Textuell kommt man wohl kaum näher an die ursprüngliche Intention der Zeichnung heran. Aber man muss nicht gleich eine neue Sprache entwerfen. Heutige moderne Multi-Paradigmen-Sprachen erlauben den Entwurf interner DSLs. In Java sind solche Notationen nicht realisierbar, da die Implementierung interner DSLs leider nicht unterstützt wird.

Quellen:

1 JAVAPRO Ausgabe 2/2018 – Bessere Abstraktion mit IODA: https://java-pro.de/bessere-abstraktion-mit-ioda
2 Ralf Westphal: Blogartikel unter http://blog.ralfw.de/2012/12/prinzip-der-gegenseitigen-nichtbeachtung.html
3 Ulf Schneider: „The Box-Bullet-Line (BBL)“, Blogartikel unter http://ulf.codes/bbl/
4 Stefan Lieser: „Flow-Design Cheatsheet“, PDF-Dokument unter https://bit.ly/2Rm1zru
5 Oracle: Java SE 8 API Documentation: https://bit.ly/2KKUNsK

Quellcode des Beispiels auf GitHub
https://bit.ly/2AyNuQo

#JAVAPRO #Serverless #Kubernetes #Container

Application-Server, Container oder lieber gleich Serverless?

Rückblickend auf Software-Projekte der letzten Jahre ist ein klarer Trend erkennbar. Die Snowflake-Server haben in modernen Architekturen ausgedient. Was aber ist die nächste Stufe der Evolution: Automatisierung mit Ansible, Chef und Co., Container à la Docker, Cluster-Lösungen wie Kubernetes oder komplett ohne Server als Serverless-Functions?

Autor:

In den letzten Jahren hat Thomas Pawlitzki in verschiedenen Rollen Erfahrungen zu Architekturen, DevOps, Development und Automatisierung sammeln können. Dabei hat er mit Tools wie Ansible, Docker und Kubernetes gearbeitet. Der aktuelle Technologie-Stack umfasst zusätzlich Serverless Functions.

Twitter: @thopaw

Alle Ansätze haben ihre Vor- und Nachteile und sollten je nach Zweck richtig eingesetzt werden. Wie jede Architekturentscheidung ist auch die Laufzeitumgebung (und der darunterliegende Technologie-Stack bis zur Hardware) eine Trade-Off-Entscheidung zwischen verschiedenen Qualitätszielen und sollte bewusst getroffen werden.

Früher war in den meisten Projekten der Graben zwischen Development und Operation noch sehr tief. Die Server wurden in allen Umgebungen ausschließlich von den Administratoren

des Betriebs betreut und jede Änderung an der Konfiguration oder Software musste mehr oder weniger aufwendig beauftragt werden. Das führte oftmals zu einem enormen Waste-in-Development, da man im Rahmen des Entwicklungsprozesses von (Team-) externen Arbeiten abhängig war. Für Operations bedeuten diese vielen kleinen manuellen Änderungen einen enormen Aufwand und erfordern ein großes Maß an Disziplin, alle Umgebungen immer konsistent zu halten. Eine Reproduzierbarkeit beim Aufsetzen einer neuen Umgebung konnte nicht sichergestellt werden.

Weiterhin war die Plattform für Java-Anwendungen in dieser Zeit oftmals ein "fetter" Application-Server. Dieser benötigte oft viele Ressourcen, die Applikationen waren oft von dem eingesetzten Application-Server des jeweiligen Herstellers abhängig[1] und die Administration dieser Server war meistens recht komplex.

In der letzter Zeit setzen sich die beiden Trends DevOps und Microservice[2] immer mehr durch und immer mehr Unternehmen beschäftigen sich damit. Bei der Software-Entwicklung wird heute oftmals auf autonome und interdisziplinäre Teams gesetzt, die die Skills und Kompetenzen haben, all ihre Aufgaben möglichst alleine zu bewältigen. Und auch bei den Applikationen geht der Trend weg von den in "liebevoller" Handarbeit administrierten Servern hin zu isolierten und self-contained Applikationen, die alles, was sie brauchen, selbst mitbringen. Gerade mit Containern hat man nun auch ein etabliertes Format und das entsprechende Tooling, um diese Anwendungen zu verpacken und zu transportieren. Für diese Container etablieren sich immer mehr Plattformen, mit denen man die einzelnen Container zu größeren Anwendungen orchestrieren, skalieren und betreiben kann. Eine andere Möglichkeit ist der Betrieb komplett ohne Server - also quasi Serverless. Hier nutzt man eine von einem Anbieter angebotene Laufzeitumgebung und gibt betriebliche Aspekte wie Skalierung, Verfügbarkeit oder Patch-Management an den Anbieter ab.

Im Folgenden werden Tools und Technologien beleuchtet, die alle ihre Stärken und Schwächen haben und je nach Zweck richtig eingesetzt werden sollten. Die Entscheidung für ein Betriebsmodell für eine Software-Lösung ist nämlich eine Architekturentscheidung. Und solche Entscheidungen sollten ein gutes Zusammenspiel aus Anforderung, Lösung und Kontext[3] darstellen, da eine nachträgliche Änderung bei solchen fundamentalen Entscheidungen oftmals teuer und aufwendig werden kann.

Um die einzelnen Lösungen etwas näher zu beleuchten, wird ein einfaches Hello-World-Beispiel mithilfe der verschiedenen Technologien in eine lauffähige Umgebung deployt. Für die ersten Beispiele wird eine sehr einfache Spring-Boot-Anwendung benutzt, die einfach eine HTTP-Route mit einem freundlichen **`Hallo`** beantwortet.

(Listing 1) - Einfache Spring-Boot-Anwendung

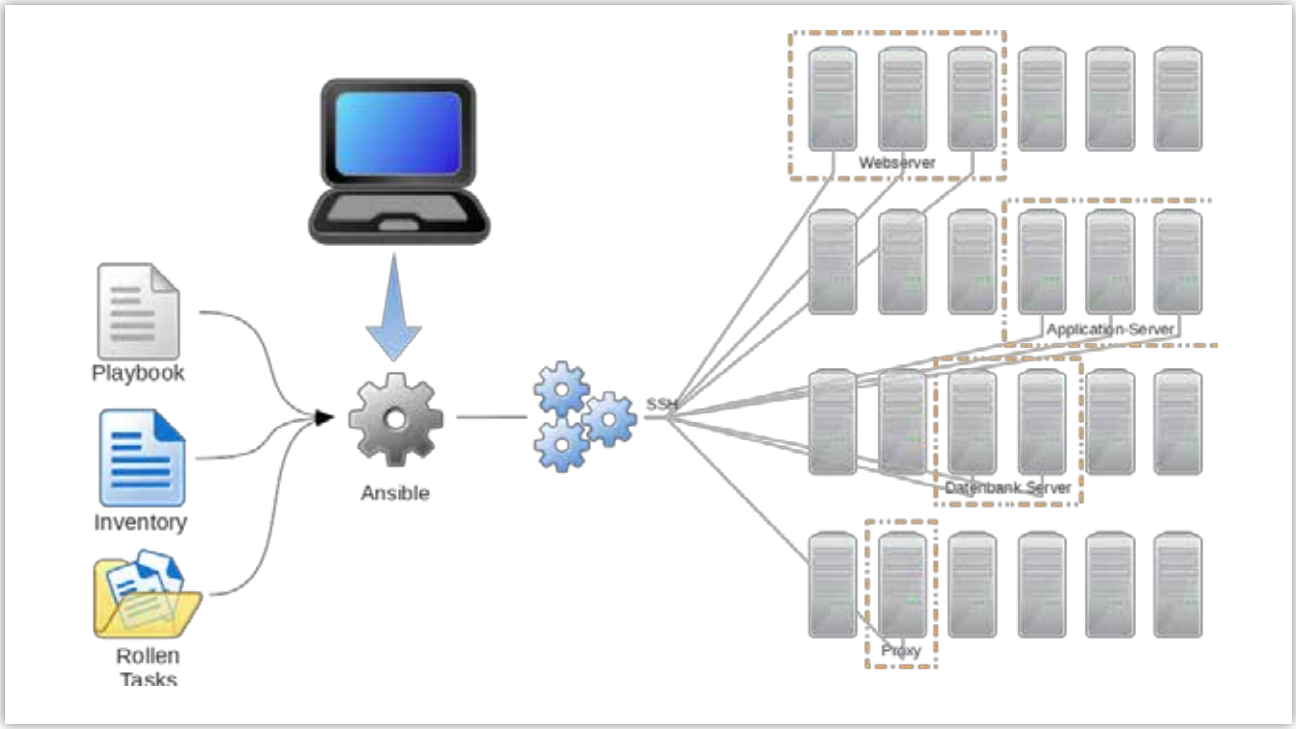
```
@SpringBootApplication
@Controller
public class App  {
   public static  void main(String...  args) {
      SpringApplication.run(App.class, args);
   }
   @GetMapping(value =  {  "/",  "/hello" })
   @ResponseBody
   public  String  getHello(@RequestParam(

      name   =  "name", required  =  false, defaultValue =
      "World")
      String  name) {
         return  "Hello " +  name   +  "!";
  }
}
```

Automatische Software-Provisionierung und -Konfiguration

Ansible[4] ist ein Konfigurations- und Management-Tool, mit dem man auf einer beliebigen Anzahl an Servern eine Reihe an definierten Kommandos und Konfigurationen ausrollen kann. Es ist etwas jünger (erstes Release in 2012) als andere Konfigurations-Tools, wie Puppet[5] oder Chef[6] und benötigt anders als seine Konkurrenz keine Agenten auf den zu konfigurierenden Systemen. Dadurch wird der Einsatz sehr vereinfacht, da im Grunde nur ein Ansible-Host benötigt wird, auf dem ein sogenanntes Playbook gestartet wird. Der Host verbindet sich per SSH mit den zu konfigurierenden Servern und führt die im Playbook enthaltenen Befehle aus. Die Server, bzw. die Gruppen von Servern, werden vorher aus einem Inventory selektiert. Die Selektion erfolgt dabei durch Patterns, die einzelne Server, Gruppen von Servern oder aber auch Schnittmengen oder disjunkte Mengen zwischen den Gruppen selektiert.

Ein Playbook ist eine einfache YAML-Datei, in der eine Reihe von Variablen definiert werden, die während der Ausführung des Playbooks genutzt werden können. Dabei können die Variablen je nach Gruppe oder Host aus dem Inventory anders belegt werden. So kann z.B. die Java-Version für Application-Server auf 9 und für Datenbank-Server auf 8 gesetzt werden. Neben den Group- und Host-Vars enthält das Playbook noch eine Liste von Tasks, die auf den selektierten Servern ausgeführt werden. Diese Tasks verwenden dabei üblicherweise die definierten Variablen und können so für jede Gruppe spezifische Einstellungen vornehmen. Ein weiteres Element der Playbooks sind Rollen, welche eine Menge an Variablen und Tasks umfassen und für die selektierten Server ausgeführt werden. Die Rollen bilden also eine Art wiederverwendbaren Building-Block und können auch unabhängig vom Playbook versioniert und referenziert werden.

(Abb. 1) zeigt vereinfacht den Ablauf einer Playbook-Ausführung. Aus einer Menge an Servern werden einzelne Server oder Gruppen aus dem Inventory selektiert. Es werden evtl. noch



Funktionsweise von Ansible. (Abb. 1)

Rollen oder Tasks in das Playbook inkludiert und dann per SSH auf den selektierten Rechnern zur Ausführung gebracht.

Aufbauend auf der Theorie wird nachfolgend ein einfaches Playbook für die sehr einfache Hello-World-Anwendung betrachtet. In diesem simplen Playbook wurden keine Rollen benutzt, aber man sieht, dass z.B. die Installation einer Applikation aus einem Maven-Repo oder das Einrichten einer SystemD-Unit als wiederverwendbare Rolle definiert werden kann. Allerdings sieht man auch, dass die einzelnen Tasks, die man definiert, oftmals von dem verwendeten Betriebssystem abhängig sind. In dem Beispiel sind die verwendeten Tasks speziell für den apt-Paket-Manager, der in einigen Linux-Distributionen verwendet wird.
Aus diesem Grund gibt es in Ansible auch die Möglichkeit Tasks bedingt auszuführen. Ein Task oder eine ganze Rolle kann z.B. vor der Ausführung auf eine Betriebssystemfamilie oder ein spezielles Betriebssystem prüfen und nur bei einem Match ausgeführt werden. Ein robustes Playbook oder eine robuste Rolle, welche auf vielen verschiedenen Systemen einwandfrei funktioniert, ist also eine Menge Arbeit und verlangt viele Tests.

(Listing 2) - Ansible-Playbook zum Ausrollen der Beispielanwendung

```
- name: installs Java Service on machine
  hosts: default
  become: yes

  vars:
    group_name: java
    user_name: java
    home_path: /opt/java-app
    java_pkg: default-jre
    mvn_repo:/home/thopaw/.m2/repository
    app_artifact: jcon-java
    app_group: thopaw
    app_version: 0.0.1-SNAPSHOT
    app_path:'{{ mvn_repo }}/{{ app_group }}/
    {{app_artifact }}/{{app_version }}/
    {{ app_artifact }}-{{ app_version }}.jar'
    ansible_python_interpreter: '/usr/bin/python3'

  tasks:
    - name: Update all packages to the latest version
      apt:
        upgrade: dist
        update_cache: yes
    - name: Installs JRE
      apt:
        name: '{{ java_pkg }}'
    - name: 'Ensure group {{ group_name }} exists'
      group:
        name: '{{ group_name }}' state: present
    - name: 'Adds user {{ user_name }} for application'
      user:
        name: '{{ user_name }}'
        group: '{{ group_name }}'
        create_home: true
        home: '{{ home_path }}'
        system: true
    - name: copy from maven repo to server
      copy:
        src: '{{ app_path }}'
        dest: '{{ home_path }}/app.jar'
        owner: '{{ user_name }}'
        group: '{{ group_name }}'
        mode: 0644
    - name: Ensure SystemD Unit is configured
      template:
        src: template/systemd.service.j2
        dest: '/etc/systemd/system/{{app_artifact}}.
```

```
            service'
            owner: root
            group: '{{ group_name }}'
            mode: 0644
    - name: Ensure Service  is started
      systemd:
            name: '{{ app_artifact }}'
            daemon_reload : true
            enabled: true
            state: started
```

An diesem einfachen Beispiel sieht man, dass man Ansible für die Automatisierung bei der Software-Verteilung und der Administration der Server einsetzen kann. Wenn man eine Analogie zu Programmiersprachen herstellt, ist man damit aber eher "low-level" unterwegs. Man muss sich auf diese Weise noch um alle Server-Aufgaben bei der Administration selber kümmern: Patch-Management, Updates, Backups, Housekeeping, etc. Viele Aufgaben können in zentral gewarteten Rollen ausgelagert werden, aber man muss diese immer noch warten, pflegen oder überprüfen. Allerdings hat man so auch alle Freiheiten und kann das System optimal für seinen Einsatz einrichten und nutzen.

Weiterhin wächst die Anzahl an built-in Ansible-Modulen[7], mit denen man auf höherer Ebene Aufgaben beschreiben und automatisieren kann. So gibt es z.B. schon Module, mit denen Kubernetes-Objekte verwaltet oder AWS Lambda Functions erstellt werden können.

Ansible ist sehr nützlich und mächtig und kann in unterschiedlichen Szenarien eingesetzt werden. So kann es z.B. für die Provisionierung von Kauf-Software benutzt werden, die sich nicht in Container packen lassen oder deren Lizenzmodell es nicht zulässt. Ebenfalls eignet sich Ansible sehr gut, um betriebssystemnahe Aufgaben zu automatisieren: Systeme patchen, eine Grundkonfiguration ausrollen, bestimmte Härtungsmaßnahmen vornehmen, etc. Mit seinen Erweiterungen kann Ansible auch ein Kandidat sein, um Container, Kubernetes-Objekte oder sogar Serverless-Functions zu verwalten. Allerdings existieren wahrscheinlich bessere Tools, so dass sich die Komplexität die Ansible mit sich bringt, vermeiden lässt. Wenn die komplette IT aber sonst mittels Ansible automatisiert ist, kann man den Skill in dem Unternehmen nutzen, um beispielsweise auch Container mittels Ansible zu orchestrieren oder auf Container-Plattformen zu deployen.

Container

Eine Technologie, die mittlerweile einen großen Einfluss auf die Software-Entwicklung und deren Architektur, die Paketierung und Verteilung hat, ist die Container-Technologie. Selbst eher konservative Unternehmen setzen immer mehr auf diese Technologie, weil sie eine Menge Vorteile bringt. Entwickler sehen einen der Hauptvorteile in Containern wie Docker oder Rocket, dass sie Entwickler davon befreit, die Software für ein bestimmtes Zielsystem zu erstellen. Ähnlich wie eine Java-Klasse auf jeder (versions-kompatiblen) JVM läuft, läuft ein Docker-Container auf allen Docker-Hosts. Der Entwickler kann einfach in einer für ihn bereits gewohnten Form alle Systemvoraussetzungen in einer Datei definieren, beschreiben und daraus ein portables Paket mit der Anwendung und all ihren Abhängigkeiten bauen, dieses lokal testen und in einem Repository bereitstellen. Weiterhin lassen sich eine Unmenge an Containern, die von anderen bereitgestellt werden, wie Bausteine benutzen oder für eigene Zwecke erweitern.

Führt man das Beispiel weiter fort und schreibt ein einfaches Docker-File für die Hello-World-Applikation, so könnte das wie folgt aussehen:

(Listing 3) - Docker-File für das Beispiel

```
FROM  java:alpine
COPY app.jar app.jar
CMD   java -jar  app.jar
```

Mit diesen drei Zeilen wird ein neues Image basierend auf dem Image **`java:alpine`** erzeugt und das Applikations-JAR wird in das Image kopiert. Die dritte Zeile definiert was passieren soll, wenn ein Container mit diesem Image gestartet wird. Dieses Beispiel zeigt, dass man bestehende Images wiederverwenden und erweitern kann, anstatt ein neues Image für seine Anwendung von Scratch aufbauen zu müssen. Dies bietet für die Entwicklung enorme Vorteile, da man Images als Basis für eine Laufzeitumgebung seiner eigenen Anwendung oder Images als fertige Services mit seinen eigenen Anwendungen nutzen kann, um diesen weitere Dienste wie z.B. Datenbanken, Proxies oder Queues anzubieten.

Allerdings schafft dies in einem Unternehmen auch eine zusätzliche Komplexität. Neben den verwendeten Bibliotheken, die in der Anwendung benötigt werden, müssen nun auch die verwendeten Images (und deren Ableitungskette bis hin zum Ursprung) überprüft und gemanagt werden. Folgende Fragen müssen nun beispielhaft beantwortet und für jedes Image nachgehalten werden:

- Sind in den verwendeten Images bekannte CVEs[8]?
- Werden durch die Images Lizenzen verwendet, die nicht passen?
- Sind die verwendeten Images aktuell?

Wenn man "nur" Docker-Container verwendet um seine Anwendung zu betreiben, muss weiterhin geklärt werden, wie die Images auf den einzelnen Servern verteilt, gestartet und überwacht werden. Dazu könnte man Technologien wie Ansible verwenden, aber damit müssten alle betrieblichen Themen rund um die Container-Instanzen selber gelöst werden.[9]

Allerdings würde man das Potenzial der Container-Technologie

nicht ohne großen Aufwand ausschöpfen können, sondern man würde das Container-Format vielmehr als Paketierungs- und Transportformat benutzen. Diese Technologie erlaubt aber noch andere Dinge, wie z.B. das sehr schnelle skalieren einer Applikation, indem neue Instanzen gestartet oder Instanzen gestoppt werden. Oder es können auch defekte Instanzen einer Anwendung einfach gestoppt und zerstört werden, wenn die Anwendungen diese Elastizität unterstützen. Da das Management dieser dynamisch schwankenden Anzahl an Container-Instanzen manuell nur sehr schwer möglich ist, haben sich hier auch Tools etabliert, um die Menge an Container zu managen und zu überwachen. Dabei hat sich Kubernetes als äußerst potenter Kandidat etabliert.

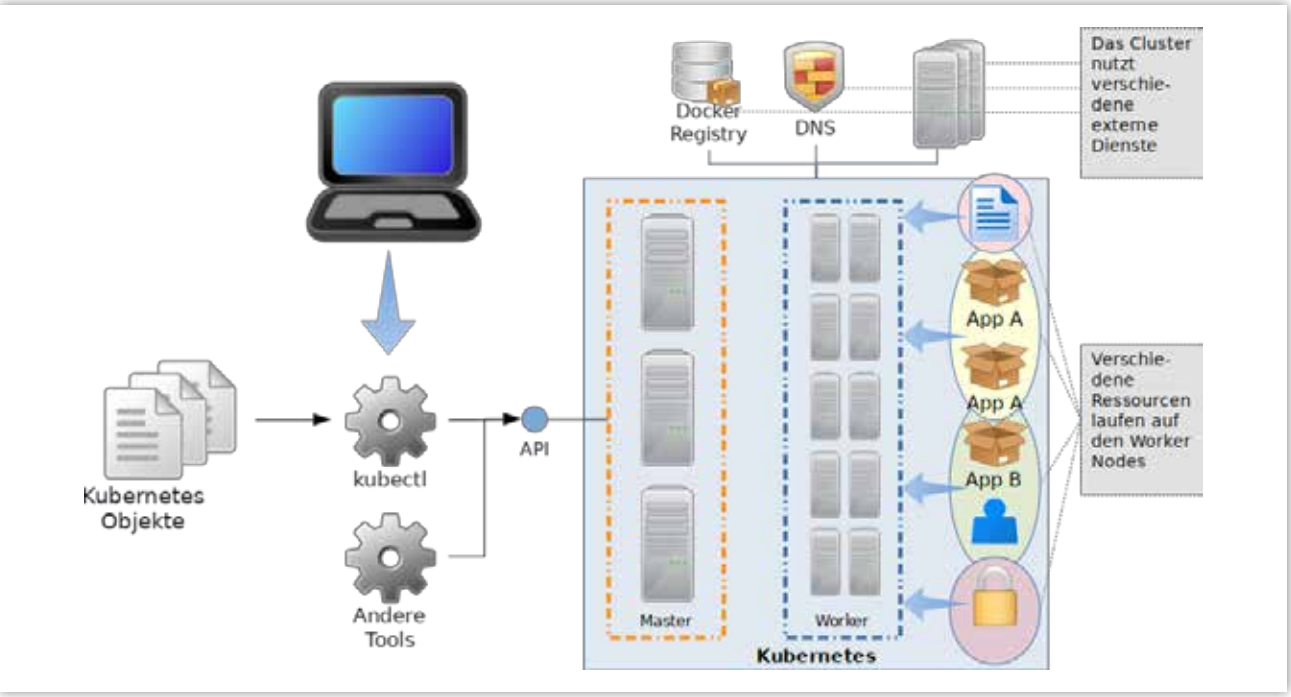
Kubernetes

Das initial von Google entwickelte Kubernetes[10] ist eine Container-Management- und Orchestrierung-Plattform, auf der Container zu Anwendungen geschnürt und betrieben werden können. Neben der reinen Plattform bietet Kubernetes auch noch eine Menge an Tools und Services an, mit denen der Lebenszyklus der Container automatisiert, gemanagt und überwacht werden kann.

Ein Kubernetes-Cluster selbst besteht aus Master- und Worker-Nodes. Die Pods - so heißt in Kubernetes die kleinste Einheit aus einem oder mehreren Containern - laufen auf den Worker-Nodes. Die Master-Nodes überwachen die Worker-Nodes und die dort laufenden Container und stellen die API bereit, mit der man mit dem Cluster kommunizieren kann. Sie starten und stoppen bei Bedarf Container oder andere Objekte, die vom Cluster gemanaged werden. Die API ermöglicht es neben Containern noch eine Reihe anderer Objekte auf dem Cluster zu betreiben.

Mit der API bietet Kubernetes eine sehr einfache und für Entwickler gewohnte Schnittstelle, um Objekte in dem Cluster anzulegen, zu löschen und zu ändern. Objekte werden dabei als Ressourcen bezeichnet. Typische Ressourcen sind dabei z.B. die schon angesprochenen Pods, aber auch Firewall-Regeln, Services, Konfigurationsparameter oder Berechtigungen. Um die unterschiedlichen Ressourcen voneinander unterscheiden oder auch selektieren zu können, besteht die Möglichkeit, diese mit Label oder Annotationen zu versehen. Ein Label oder eine Annotation besteht dabei aus einem Namen und einem Wert. Neben dem Wiederfinden der Ressourcen werden Labels und Annotationen auch zur Konfiguration verwendet. Denn Kubernetes erlaubt es auch, Ressourcen auf dem Cluster zu betreiben, die z.B. auf das Erzeugen einer neuen Ressource reagieren und diese konfigurieren - oder Dienste für diese anzubieten, wenn die neu erzeugte Ressource mit bestimmten Annotationen ausgestattet ist. Auf diese Weise kann ein Kubernetes-Cluster um zusätzliche Funktionen erweitert werden und so einen größeren Funktionsumfang anbieten.

Ein Beispiel hierzu ist external-DNS[12]. Diese Komponente hat für neu angelegte Ingress-Rules (eine Regel, die beschreibt wie und wo das Cluster Netzwerk-Anfragen erhalten soll) einen DNS-Record in einem zuvor konfigurierten DNS-Server angelegt. Auf diese Weise kann man Applikationen auf dem Cluster sehr einfach einen DNS-Namen zuordnen und auch wieder löschen, wenn die entsprechende Ingress-Ressource vom Cluster entfernt wird. Dieser Mechanismus lässt sich noch mit einer weiteren Komponente mit dem Namen kube-lego[13] kombinieren. Für speziell annotierte Ingress-Rules kann kube-lego dann für den angelegten DNS-Eintrag ein Let's-Encrypt[13]-X.509-Zertifikat anfragen und einrichten. Auf diese Weise können auf dem



Übersicht eines Kubernetes-Clusters (Abb. 2)

Cluster sehr komfortabel für neue Applikationen DNS-Einträge und auch Zertifikate eingerichtet und gemanaget werden, so dass die Anwendungen direkt per HTTPS erreichbar ist.

Damit die verschiedenen Ressourcen auch komfortabel und programmatisch (und damit automatisierbar) angelegt, gelöscht und verändert werden können, bietet Kubernetes neben der API auch noch verschiedene Tools an. Das wohl wichtigste Tool ist dabei kubectl, das Command-Line-Tool zur Cluster-Verwaltung. Dieses Tool wird auch in unserem Beispiel verwendet, um die Docker-Container auf dem Cluster zum Laufen zu bringen. Dazu müssen aber zunächst die Kubernetes-Objekte deklariert werden, die eine Anwendung ausmachen.

(Listing 4) - Definition der Kubernetes Objekte

```
apiVersion:  apps/v1
kind:  Deployment
metadata:
   name: hello-app-deployment
spec:
   replicas: 3
   selector:
         matchLabels:
         app: hello-pod
   template:
      metadata:
         labels:
            app: hello-pod
      spec:
         containers:
         - name: nginx
            image:  app
            imagePullPolicy:  Never #  Wird wegen minikube
            benötigt
            ports:
            - containerPort:  8080
---
kind: Service
apiVersion:  v1
metadata:
   name: hello-service
   labels:
      app: hello-app
spec:
   selector:
      app: hello-pod
   ports:
   - name: hello-http
      protocol: TCP
      port:  80
      targetPort: 8080
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
   name: hello-ingress
   annotations:
      nginx.ingress.kubernetes.io/rewrite-target: /
spec:
   rules:
   - http:
      paths:
      - path: /hello
         backend:
         serviceName: hello-service
         servicePort:  80
```

In der YAML-Datei in **(Listing 4)** werden drei Objekte (getrennt durch ---) definiert:

1. Das erste Objekt ist ein Deployment, welches definiert, mit wie vielen Instanzen ein Pod auf dem Cluster laufen soll. Weiter wird in der Sektion **`template`** definiert, welche Container zu dem Pod gehören und welche Ports genutzt werden sollen.
2. Das zweite Objekt definiert einen Service. Dieser selektiert eine Reihe von Pods (über ihr Label **app**) und definiert einen Port, über welchen diese Pods erreichbar sind. Ein Service ist langlebiger als ein Pod und sollte für die Cluster-interne und -externe Kommunikation genutzt werden. Ein Pod kann nämlich z.B. von den Master-Nodes plötzlich gestoppt und auf einen anderen Worker geschoben werden, wenn der Master entschieden hat, dass die Ressourcen des Workers anders genutzt werden sollen.
3. Das dritte Objekt ist eine Ingress-Regel. Sie wird von einem Pod mit dem Namen **`ingress controller`**[14] genutzt, der auf dem Cluster läuft und quasi die Tür zu Applikationen im Cluster darstellt. Er erzeugt aus allen Ingress-Regeln ein Routing für Netzwerkanfragen an das Cluster. Dabei verteilt die Ingress-Regel den Traffic an den Pfad **`/hello`** auf den Service, der unter Punkt 2 beschrieben wurde.

Ein **`kubectl apply -f app.yml`** erzeugt nun die Objekte, die in der Datei **`app.yml`** auf dem Cluster definiert wurden.

Wie man schon an diesem kleinen Beispiel sieht, bietet Kubernetes eine große Flexibilität bei der Gestaltung einer ganzen Plattform, die für die Container-Orchestrierung genutzt werden kann. Zusammen mit den entsprechenden Tools, wie kubctl oder auch Helm[15], kann man die Prozesse auch recht einfach automatisieren und z.B. in Continuous-Delivery-Pipelines einbauen.

Allerdings ist der Betrieb eines Kubernetes-Clusters auch eine Herausforderung und benötigt spezielle Kenntnisse, die im Unternehmen vorhanden sein oder aufgebaut werden müssen. Auch kann schon alleine die Geschwindigkeit, mit der Kubernetes weiterentwickelt wird, problematisch sein, da in der Vergangenheit ungefähr alle zwei Monate eine neue Kubernetes-Version veröffentlicht wurde. Das Update des Clusters selber funktioniert i.d.R. problemlos, die Erweiterungen, die auf dem Cluster betrieben wurden, finden die neue Version nicht immer gut. Es kann vorkommen, dass bestimmte Plugins mit einer neueren Kubernetes-Version einfach nicht mehr funktionieren.
Eine andere Möglichkeit die Vorteile von Kubernetes zu nutzen, ohne den Aufwand eines Cluster-Betriebs selbst zu stemmen, ist die Nutzung eines Managed-Kubernetes-Clusters. Aufgrund des Erfolges von Kubernetes gibt es immer mehr Angebote von

Kubernetes-Cluster, die von dem Anbieter gemanaged werden. Wenn man Kubernetes nur nutzen und es nicht betreiben möchte, dann lohnt sich hier eine Recherche.

Serverless-Functions

Ein anderer interessanter Ansatz, mit dem Software entwickelt und betrieben werden kann, stellen Serverless-Functions dar. Die großen Cloud-Anbieter bieten für diese Art des Anwendungsbetriebs eine Lösung an. Bei AWS heißen diese Funktionen Lambda[16], bei Google heißen sie Cloud Functions[17] und bei Microsoft Azure einfach nur Functions[18].

Bei Serverless-Functions abstrahiert man sich noch weiter von Betriebsthemen und selektiert nur noch eine Laufzeitumgebung des Cloud-Anbieters, in der der Code ausgeführt wird. Den einzelnen Funktionen kann man zusätzlich rudimentär Ressourcen wie Speicher, Berechtigung oder ein Timeout mitgeben, ansonsten überlässt man alle Betriebsthemen dem Cloud-Anbieter.

Den wirklichen Vorteil bei dieser Art des Application-Betriebs konnte der Autor bei der Arbeit in einem kleinen Team an einem MVP/Prototyp bemerken. Es war keinerlei Infrastruktur vorhanden, auf der man aufsetzen konnte. Es stellte sich also die Frage, wie die Software betrieben werden soll und dabei gleichzeitig noch genügend Geschwindigkeit bei der Entwicklung sichergestellt werden kann. Da lag die Entscheidung für Serverless-Functions nahe, weil hier nur eine minimale Konfiguration notwendig ist und diese von Tools wie Serverless[19] auch noch sehr komfortabel ist.

Allerdings muss man bei der Nutzung von Serverless-Functions auch einiges beachten. Das erste, was auffällt ist, dass man Software für eine bestimmte Cloud-Plattform schreibt. Hier kann zwar durch entsprechende Pattern bei der Implementierung eine Minderung geschaffen werden, aber an manchen Stellen muss einfach Glue-Code geschrieben werden, der speziell auf die API der Provider passen muss. Ein weiterer Punkt, der erwähnt werden sollte, ist die Gefahr des Vendor-Lock-Ins. Die oftmals nahtlose Integration und Kombination weiterer Services mit den Serverless-Functions erhöht die Gefahr, dass die entwickelte Lösung nicht ohne große Aufwände zu einem anderen Cloud-Provider portiert werden kann. Ob das ein Problem ist, muss man abwägen. In dem genannten Fall war durch die sehr regulierte Branche Versicherungen, in der sich das Projekt bewegt, und die Anforderung der BaFin[20] eine Portierung zu einem anderen Cloud-Anbieter sowie eine Exit-Strategie zwingend notwendig.

Ein weiterer Nachteil, der bei der Verwendung von Serverless-Functions schmerzhaft werden kann, ist die fehlende Freiheit, die ein Betrieb von z.B. Containern ermöglicht. Es stehen nur die Laufzeitumgebungen zur Verfügung, die der Cloud-Provider anbietet. Und es stehen auch nur die Versionen der Laufzeitumgebungen zur Verfügung, die angeboten werden. Ob man nun also Java 8, 9 oder sogar noch 7 verwendet, ist keine freie Entscheidung mehr. Auch die Nutzung von schon fertigen Software-Bausteinen, wie z.B. Containern, in denen eine SQL, NoSQL, eine Queue oder Kafka läuft, ist nicht möglich. Hier ist man auf SaaS-Lösungen des Cloud-Providers oder eines anderen Anbieters angewiesen.

Als Beispiel haben wir mittels Serverless ein Projekt generiert und minimal angepasst. Mit folgendem Befehl kann man sich das Projekt erzeugen lassen:

```
serverless create -t aws-java-gradle
```

(Listing 5) zeigt den **RequestHandler**, der als Lambda ausgeführt wird. Als Input wird eine Map in die Methode gegeben und als Rückgabe wird ein Response-Objekt zurückgegeben, welches mit in das Projekt generiert wurde.

(Listing 5) - Java RequestHandler für das Serverless-Beispiel

```
public class Handler  implements RequestHandler<Map<String,
Object>,
ApiGatewayResponse> {
   private static final Logger LOG =  Logger.getLogger(Handler.
   class);
   @Override
   public ApiGatewayResponse handleRequest(Map<String,  Object>
   input,  Context context)  {
      LOG.info("received: " +  input);
      @SuppressWarnings("unchecked")
      Optional<Map<String,String>>  query =
         Optional.ofNullable(
            (Map<String,String>)input.get("queryString
            Parameters")
          );
      String name   =  query
         .orElse(Collections.emptyMap())
         .getOrDefault("name",  "World");
      Response responseBody  =  new  Response("Hello " +  name
      +  "!", input);
      return  ApiGatewayResponse.builder()
            .setStatusCode(200)
            .setObjectBody(responseBody)
            .build();
     }
}
```

Die Datei **serverless.yml**, die ebenfalls in das Projekt generiert wurde, muss an die eigenen Bedürfnisse angepasst werden. Nach den Anpassungen sieht die Konfiguration wie in (Listing 6) zu sehen ist, aus. Man sieht in der Konfiguration, dass als Cloud-Anbieter AWS und eine Java 8 Umgebung ausgewählt wurde. Interessant ist noch die Deklaration der Funktionen und das Mapping der Events zu den Funktionen. In diesem Fall sollen alle Requests an den Pfad **hello** von dem RequestHandler beantwortet werden. Als Event lassen sich noch sehr viele andere Typen[21] auswählen, wie z.B. SNS, SQS, S3 oder Schedule. Dies sind meistens andere AWS-Dienste, die sich so nahtlos mit den

Serverless-Functions verbinden. Allerdings zahlt man damit den Preis, dass die Lösung von dem jeweiligen Anbieter abhängig und eine Portierung zu einer anderen Plattform schwieriger ist.

(Listing 6) - Konfiguration des Serverless-Frameworks

```
service: jcon-java-serverless
provider:
   name: aws  runtime:
   java8 region:  eu-
   central-1
package:
   artifact: build/distributions/hello.zip
functions:
   hello:
      handler:  com.serverless.Handler
      events:
      - http:
      path: hello
      method: get
```

Eine Automatisierung von Konfigurationen oder Deployments ist durch Tools wie Serverless meist schon problemlos möglich. In dem genannten Fall war der Betrieb und die Weiterentwicklung des MVPs selbst in einem sehr kleinen Team möglich, da die komplette Umgebung gemanaged wurde. Eine Fokussierung auf die Implementierung von Features war problemlos möglich, weil das Team sich auf die Plattform und den Anbieter geeinigt und die damit verbundenen Einschränkungen in Kauf genommen hat.

Fazit

Keine der aufgeführten Technologien ist als klarer Gewinner zu sehen, sondern nur besser oder weniger gut geeignet für die Umsetzung einer Architektur und die Entwicklung eines bestimmten Systems. Ansible oder andere Tools zur Software-Provisionierung sind bestens geeignet, um administrative Prozesse zu automatisieren. Mit zunehmender Reife entstehen auch immer mehr Module und Rollen, die man in seinen Playbooks verwenden kann, um so auf eine höhere Abstraktionsebene zu kommen. Mittlerweile gibt es schon Ansible-Module, die sich um AWS-Ressourcen kümmern oder Kubernetes-Objekte managen. Auch eignet sich Ansible immer noch dafür, eine große Menge an Servern zu administrieren, deren Anwendungen sich nicht so einfach auf eine Container-Plattform portieren lassen. Gründe hierfür könnten z.B. Lagecy-Systeme, Lizenz-Restriktionen oder fehlende Kompatibilität bei eingekaufter Software sein.

Docker alleine ist schon eine hervorragende Lösung, um

1. fertige Software-Bausteine inklusiver aller Abhängigkeiten zu nutzen ohne eine komplette Umgebung dafür aufzubauen,
2. seine eigenen Bausteine mitsamt der benötigten Umgebung als Paket zur Verfügung zu stellen.

Kombiniert mit Technologien wie Kubernetes zur Container-Orchestrierung schafft man es, eine automatisierbare und sehr flexible Plattform anzubieten, auf der eine Vielzahl an unterschiedlichen Lösungen von unterschiedlichen Teams betrieben werden können. Allerdings muss man sich auch bewusst sein, dass schon Container alleine eine weitere Komplexitätsebene bedeuten. Die genutzten Images müssen zu den Policies des Unternehmens passen und separat auf Compliance und Sicherheit geprüft werden. Wenn man die Container dann auf einer Plattform wie Kubernetes betreibt, bringt diese ebenfalls eine eigene Komplexität mit, die beherrscht oder als managed-Lösung eingekauft werden muss.

Software komplett Serverless zu betreiben, ermöglicht eine sehr starke Fokussierung auf die Funktionalität und das Design der Applikation, solange man mit den angebotenen Laufzeitumgebungen auskommt und weitere Anforderungen an die Umgebung der Software über andere Services oder andere Anbieter bedient. Auch sollte man darauf achten, dass die gebaute Software nicht zu stark von dem genutzten Cloud-Anbieter abhängig wird. Hierzu sollte man ein sauberes Design und eine klare Kapselung des Glue-Codes anstreben, der zwischen der eigentlichen Application-Logik und der API des Cloud-Anbieters eingezogen werden muss.

Quellen:

1 Wenn man nicht sehr sauber gearbeitet hat, waren die Anwendungen oft von den speziellen Application Servern abhängig. Selbst, wenn man im Code wirklich sauber entwickelt hat, so musste man trotzdem noch häufig vendor-spezifische Deployment-Deskriptoren schreiben.
2 Mit Microservices ist hier die Abkehr von Deployment-Monolithen hin zu kleineren, unabhängig deploybaren Services gemeint. Die Rede ist eher vom Architekturstil, eine Software aus unabhängigen Prozessen mit sprachneutralen APIs zu bauen.
3 Als Kontext wird hier die Umgebung aufgefasst, in der die Anwendung eingebettet wird. Dieser ist nicht nur technologisch zu sehen, sondern betrifft auch das Unternehmen (Mitarbeiter, Skills, Prozesse, bestehende Infrastruktur, etc.).
4 https://www.ansible.com/
5 http://www.puppetlabs.com/company/
6 http://www.chef.io/
7 https://bit.ly/2FOaCjJ
8 Sicherheitslücken, Common Vulnerabilities and Exposures
9 Dies wären z.B. Themen wie Monitoring, Skalierung und Deployment.
10 https://kubernetes.io/
11 https://bit.ly/2KJ5RXA
12 Dieser wird nicht weiterentwickelt und wurde durch cert-manager ersetzt.
13 Eine freie und automatisierte Certificate Authority: https://letsencrypt.org/
14 https://bit.ly/2SlpB5X
15 Paket-Manager für Kubernetes: https://helm.sh/
16 https://aws.amazon.com/lambda/
17 https://cloud.google.com/functions/
18 https://azure.microsoft.com/services/functions/
19 https://serverless.com
20 Bundesanstalt für Finanzdienstleistungsaufsicht: https://www.bafin.de
21 https://bit.ly/2tOvIVq

Fast Lane

Softwareentwicklung beschleunigen

Fast Lane ist weltweiter Spezialist für Technologie- und Business-Training und Beratung im Highend-Bereich. Wir bieten Ihnen die passenden Trainings, damit Sie bei der Entwicklung Ihrer Unternehmensanwendungen optimal von Container-Techniken und Microservices, agilen Methoden und DevOps-Konzepten profitieren.

Auswahl unserer Trainingsangebote für Entwickler

Ob als Training im Klassenraum, Online-Kurs, E-Learning oder Blended Learning, informieren Sie sich jetzt über die geeignete Lösung für Ihre Aus- und Weiterbildung!

AWS Development
- AWS Technical Essentials
- Developing on AWS
- DevOps Engineering on AWS

Google Cloud Development
- Google Cloud Fundamentals: Core Infrastructure
- Developing Applications with Google Cloud Platform

Microsoft Development
- Microsoft AI Cognitive Services
- Microsoft Azure IoT Services
- Azure Developer, Core & Advanced Solutions
- Developing Windows Azure & Web Services
- Programming in HTML5 with JavaScript & CSS3
- Programming in C#
- Developing ASP.NET MVC 4 Web Applications
- DevOps Practices

Oracle Java SE
- Java SE 8: Fundamentals
- Java SE 8: New Features
- Java SE 8: Programming

Linux Development
- Linux, Open Source Development & GIT
- Developing Applications for Linux
- Linux Kernel Internals & Development
- Developing Linux Drivers
- Linux Kernel Debugging & Security

Pivotal
- Core Spring
- Spring Boot Developer
- Spring Cloud Services
- Cloud Foundry Developer

Container und agile Softwareentwicklung
- Docker Fundamentals
- Advanced Docker Design & Troubleshooting
- Kubernetes Administration
- Scrum Product Owner, Scrum Master, Scrum Developer
- Agiles Anforderungsmanagement
- Agiles Projektmanagement
- Testen in der agilen Softwareentwicklung

Weitere Informationen und Kursbuchung: www.flane.de

aws partner network Training Partner

Partner Google Cloud

Gold Microsoft Partner Microsoft

ORACLE Approved Education Center

THE LINUX FOUNDATION AUTHORIZED TRAINING PARTNER

Pivotal AUTHORIZED TRAINING PARTNER

#JAVAPRO #Microservices #Container

Software-Modernisierung mit Microservices und Container

In nahezu allen Unternehmen hat sich die IT zum zentralen Faktor für den Geschäftserfolg entwickelt. Damit sie nicht zum Hemmschuh wird, muss die IT eine hohe Flexibilität unterstützen und damit die zügige Umsetzung neuer Geschäftsanforderungen ermöglichen. Der Einsatz von Microservices und Container-Technologie spielt dabei eine entscheidende Rolle.

Wie bei anderen IT-Trends auch, folgt die Entwicklung bei den Microservices einer typischen Hype-Kurve: von der Euphorie über die Ernüchterung zum Pragmatismus. Allzu begeisterte und manchmal auch unbedarfte Herangehensweisen haben in der Praxis teils skurrile Blüten getrieben. So kam es vor, dass der Anspruch „Micro" sehr strikt interpretiert wurde und ein regelrechter Wettbewerb um die kleinste funktionale Einheit, die in einem Service isoliert werden kann, ausbrach. In anderen Fällen waren siebenköpfige Scrum-Teams plötzlich für 50 Microservice-Projekte verantwortlich und waren somit personell gar nicht in der Lage, einen Kernvorteil dieser Architektur auszunutzen – nämlich, dass jedes Projekt unabhängig von allen anderen weiterentwickelt werden kann.

Pragmatismus statt der buchstabengetreuen Umsetzung

Die Lehren aus diesen Entwicklungen: Statt den Code der reinen Lehre zufolge und ohne Berücksichtigung des Kontextes zu

Autor:

Oliver Weise ist Development-Teamleiter bei Consol Software in München.

zergliedern, sollte „Right-Sizing" das Motto sein. Wenn einem Software-Architekten die Segregation von Funktionalitäten in mehrere Services vermeintlich dazu zwingt, etablierte Lösungen wie eine Datenbankplattform, in Frage zu stellen, dann ist das sehr wahrscheinlich keine gute Idee. Auch dann, wenn das Endergebnis laut Domain-Driven-Design korrekt aussehen würde. Wichtiger als die Reinheit der Domäne eines Services sind am Ende eher praktische Eigenschaften wie schneller Start/Shutdown, Statuslosigkeit und der problemlose Betrieb auf Container-Plattformen, also klassische Tugenden der 12-Faktor-App[1].

In verteilten Systemen neue Konzepte zur Datenkonsistenz und -aggregation – etwa Event-Sourcing und CQRS (Command-Query-Responsibility-Segregation) – zusammen mit Microservices zu verwenden, klingt vielversprechend, ist aber auch sehr ambitioniert. Dennoch wurden diese in einigen Fällen zu leichtfertig eingesetzt. Am Ende entstand ein hochkomplexes System, das erhebliche administrative Aufmerksamkeit und Knowhow erfordert, für das sich dann aber niemand wirklich verantwortlich fühlte. Entsprechende Pläne und die Auswirkungen, die sich aus dem Einsatz dieser Systeme ergeben, sollten früh kommuniziert und gegebenenfalls auf den Einsatz verzichtet werden.

Unter Architekturaspekten empfiehlt es sich, eine evolutionäre statt einer revolutionären Transition existierender Applikationslandschaften zu betreiben: Ein funktionierender Anwendungmonolith wird nicht in einem Schritt dekonstruiert, sondern es werden nacheinander einzelne, leicht isolierbare, funktionale Domänen ausgegliedert und in separate Microservice-Projekte ausgelagert. Das gibt allen Beteiligten die Möglichkeit, sich sowohl technisch als auch kulturell an das neue Modell zu gewöhnen und ein klareres Bild von den Herausforderungen zu entwickeln, bevor die großen Aufgaben angepackt werden.

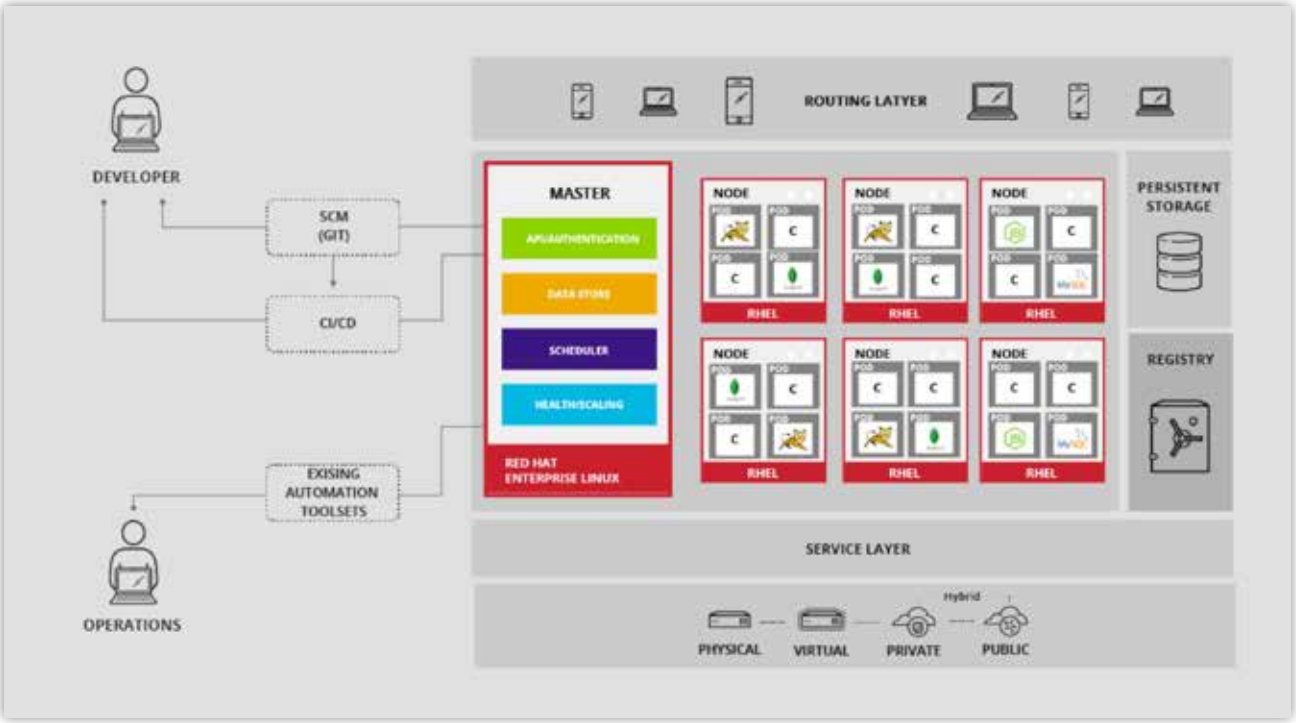
Microservices und Programmiersprachen

Aktuell verbreitet sich Googles neue Programmiersprache Go sehr rasch im Microservice-Umfeld. Entwickler schätzen an Go die Gradlinigkeit bei der Umsetzung gängiger, nicht allzu komplexer Use-Cases – beispielsweise Web-Services mit Datenbank-Backend – und die hohe Performance. Im Zusammenspiel mit dem hochoptimierten Kommunikations-Framework gRPC ist Go eine prädestinierte Lösung für sehr stark frequentierte Web-Services, bei denen massive Skalierbarkeit erforderlich ist.

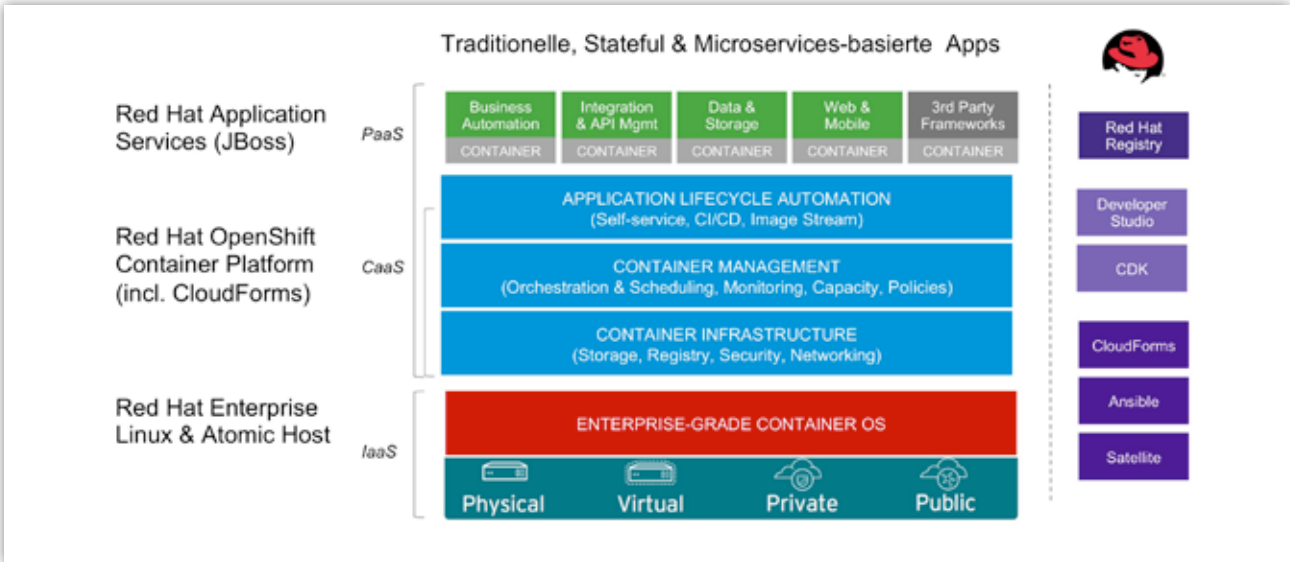
Aber auch Java- und andere JVM-basierte Sprachen spielen eine wichtige Rolle; unter den Frameworks sind Spring Boot und Java EE die Spitzenreiter. Funktionale Plattformen wie AWS Lambda oder Google Cloud Functions sind im Kommen und stoßen auf großes Interesse. Sie müssen aber noch zeigen, ob sie für den Mainstream der Anwendungsfälle das Werkzeug der Wahl sind oder ob sie eine Nischenlösung bleiben.

Container-Plattformen für den Betrieb von Microservices

Eine Reihe von Entwickler-Teams haben mit dem Aufkommen von Docker vor ein paar Jahren bereits viel Energie und Knowhow in den Aufbau eigener Plattformen für den containerisierten Betrieb



Für den Betrieb und die Bereitstellung von Microservices bieten sich vor allem Container-Lösungen an, wie sie etwa mit Red Hat OpenShift Container Platform zur Verfügung stehen (Quelle: Red Hat). (Abb. 1)



Container dienen der Kapselung und Isolierung von Applikationen mit allen erforderlichen Systemkomponenten. Damit stellen sie eine optimale Möglichkeit für die einfache und vor allem schnelle Bereitstellung von Anwendungen dar (Quelle: Red Hat). (Abb. 2)

und das Testing investiert. Nun aber etabliert sich eine neue Generation von integrierten Lösungen für das Container-Clustering wie Google Kubernetes, Red Hat OpenShift Container Platform (Abb. 1) oder Docker Datacenter, die gerade hinsichtlich einfachen Betriebs und funktionalen Umfangs punkten.

Bei den Pionieren ist die Bereitschaft aktuell natürlich gering, die eigenen Lösungen nun wieder zu ersetzen. Nach dem betriebenen Aufwand sind die Prozesse vielleicht gerade erst wirklich etabliert und man identifiziert sich natürlich auch mit der eigenen Herangehensweise. Daher könnte sich diese Early-Adoption jetzt als Hemmschuh herausstellen, da sich die neuen Plattformen, den Eigenbau-Lösungen in Hinsicht Flexibilität und Zukunftssicherheit als überlegen herausstellen könnten. Wer also bislang noch nicht auf den Container-Zug (Abb. 2) aufgesprungen ist, könnte nun bei der Umstellung seiner Infrastruktur so etwas wie die „Gnade der späten Geburt" erleben und direkt auf eine der gereiften Lösungen der neuen Generation setzen.

Sind Container-Plattformen sicher?

Ein Aspekt, der Betreibern von Container-Plattformen teilweise Kopfzerbrechen bereitet, ist die Kontrolle der deployten Systeme hinsichtlich ihrer Sicherheit – insbesondere wegen der Geschwindigkeit, mit der neue, sehr heterogene Apps aus dem Boden schießen können. Man fürchtet einen allzu ausufernden Wildwuchs von Systemen mit einer unübersehbaren Anzahl aktiver Komponenten in diversen Versionen. Tatsächlich muss sich hier aber erst noch zeigen, wie sensibel das Thema tatsächlich ist, oder ob der partielle Kontrollverlust beziehungsweise die Verlagerung der Verantwortung zum Projekt-Team nicht notwendige Übel sind. Red Hat bietet hier mit der auf einheitlichen Basis-Images aufbauenden **`s2i-Strategie`**[2] eine potentielle Lösung.

Public-Cloud oder lieber selber hosten?

Darüber hinaus existiert eine gewisse Unsicherheit, ab welcher Unternehmensgröße es sich lohnt, eine Container-Plattform On-Premise selbst zu betreiben oder ob es nicht oft besser wäre, eines der Angebote von großen Cloud-Providern, wie Amazon AWS, Microsoft Azure oder Google Cloud, zu nutzen. Einigen Firmen erscheinen letztere Lösungen zunächst einmal als teuer, dafür sparen sie sich einen Großteil des administrativen Aufwandes und als Bonus können sie gefühlt grenzenlos skalieren, auch wenn dies die wenigsten Anwender wohl jemals benötigen.

Gerade in Deutschland spielt aber auch der Datenschutz eine zentrale Rolle, der bei US-amerikanischen Cloud-Anbietern aufgrund herrschender Gesetzgebung und gemessen an den hiesigen Anforderungen zumindest mit einem Fragezeichen zu versehen ist. Potentielle Alternativen: Microsofts Azure-Deutschland, das von T-Systems als deutschem Datentreuhänder kontrolliert wird (jedoch seit Herbst diesen Jahres nur noch von Bestandskunden genutzt werden kann), oder dem auf Red Hat OpenShift Container-Platform-basierten AppAgile-Angebot, ebenfalls von T-Systems, das deutschem Recht unterliegt.

Quellen:

```
1 https://12factor.net/de/
2 https://red.ht/2AAvZiG
```

#JAVAPRO #DevOps

Skills, Tools und das richtige Mindset für DevOps

DevOps gilt als zukunftsträchtiger Prozessverbesserungsansatz zwischen Entwicklung und Betrieb von Software, der einen großen Vorteil für Entwickler, Betreiber wie auch Auftraggeber einer Software darstellt. Anknüpfend an den Ansatz der agilen Software-Entwicklung findet eine Transformation hin zum „digitalen Fließband" statt – der Code wird unter Zuhilfenahme automatisierter Prozesse von der Idee bis zum Endprodukt durchgereicht.

Die Arbeitsschritte der Spezialisten sind miteinander verknüpft. Durch diese Methode herrscht eine hohe Produktivität, schnelleres Time-to-Market und somit ein besserer Durchlauf der initialen Software-Idee zum anwendbaren Produkt. Im Folgenden wird neben Tools und Zielen für das Thema DevOps auch das entsprechende Mindset vorgestellt, das für eine gelungene Umsetzung nötig ist.

Autor:

Dr. Florian Zierer arbeitet seit 2016 bei der PENTASYS AG. Seine Schwerpunkte liegen in den Bereichen Continuous Integration und Automatisierung sowie in der Programmierung mit Java im Bereich Webapp-Entwicklung und Scala im Bereich Spark/Hadoop. Er interessiert sich besonders für die Qualitätssicherung in agilen Projekten und die Transition von klassischen Systemlandschaften hin zu einer DevOps-Kultur.

Welche Geschäftsziele sich durch DevOps erreichen lassen

Oft spricht man beim Thema DevOps nur über die Tools und Umsetzungslösungen – doch sollte immer im Mittelpunkt stehen, wie sich die Geschäftsziele eines Software-Produkts besser (oder überhaupt) erreichen lassen. Denn DevOps spricht auch die

Fachseite an, die das Software-Produkt in Auftrag gibt. Nachfolgend werden kurz drei Beispiele für Geschäftsziele genannt, die durch den DevOps-Gedanken erreicht werden können.

Ein mögliches Geschäftsziel ist die Robustheit der Software, die durch ein starkes Bewusstsein der Entwickler für den Verlauf der Wertschöpfungskette und ihren Einflussfaktoren ganz von selbst verstärkt wird. Das Wie und Wo des Software-Betriebs (Operations) sollte genauso im Fokus stehen wie die Architektur (Development). Es ist daher nicht abwegig, diese Form von gleichwertiger Anerkennung aus dem großen O von DevOps herauszulesen.

In Bezug auf die Wertschöpfung lässt sich der zeitliche Einfluss mit dem Stichpunkt Time-to-Market aufgreifen. Dessen Verkürzung ist ein weiteres Geschäftsziel. Klar strukturierte Abläufe und ein Bewusstsein über diese, sorgen beim „Fließband" der agilen Software-Entwicklung für ein schnelles Umsetzen der fachlichen Anforderung. Eingespielte Teams sind dank DevOps zudem in der Lage, konstante Leistung zu erbringen und bei der Planung neuartiger Anforderungen zuverlässige Abschätzungen zu geben. Der ständige Blick über den Tellerrand in den Betrieb ermöglicht es dem Entwickler-Team, ein breites Wissen aufzubauen. Ein gutes Beispiel hierfür ist die folgende Situation: Fehlende Kenntnisse über langsame Backend-Systeme führen dazu, dass Leistungsengpässe erst beim Marktgang einer neuen Software-Anforderung erkannt werden. Im schlimmsten Fall ist das Produkt für den Marktgang untauglich. Dann bedarf es schnelle Nachbesserungen, die jedoch folgende Probleme aufwerfen:

- Einerseits erlangen Entwickler erst dann genauere Kenntnisse über diese Systeme, wenn es zu erheblichen Ausfällen kommt: Dann fehlt die Zeit, sich genauere Gedanken über den Betrieb und das dortige Ökosysteme zu machen. Mit dem Ziel ein Pflaster für das Problem zu finden, wird intensiv, aber nur ad hoc, nach dem Ursprung gesucht. Was weiterhin ausbleibt, ist ein tiefer Einblick in den Bereich Operations.
- Anderseits suchen die Verantwortlichen in dem Pflaster nur eine punktuelle Lösung für die Leistungsengpässe, aber sie führen keine komplette Überarbeitung der Software-Architektur durch. Denn Zeit und Ressourcen fehlen nach einem geplatzten Live-Gang. Die Fachseite drängt nämlich auf eine rasche Platzierung des Produkts auf dem Markt oder im Geschäftsablauf und gibt damit den Takt an. Die eigentlich notwendige Überarbeitung der Architektur fällt damit aus.

Es braucht den täglichen Kontakt mit der Denkweise der Systeme, um ein Gespür für sie zu entwickeln und dieses Wissen bei Architekturentscheidungen einfließen zu lassen.

An dieser Stelle soll erwähnt werden, dass modern aufgesetzte Software-Betriebe mit cloud- und container-basierten Plattformen (Kubernetes, OpenShift etc.) zwar bereits einen sehr transparenten DevOps-Ansatz haben und viele Schwachstellen eliminieren. Sie verhindern aber nicht die Probleme, die durch unbekannte schwache Glieder in der Betriebskette entstehen. Sind geeignete Monitoring-Werkzeuge verfügbar, um diese Glieder zu finden, müssen die Entwickler diese Werkzeuge auch bedienen und interpretieren lernen.

Ein drittes Geschäftsziel, das sich durch DevOps erreichen lässt, ist Nachhaltigkeit. Dieses schließt an die obigen zwei Geschäftsziele Robustheit und Time-to-Market an. Durch besseres Wissen über die Wertschöpfung und an ihr beteiligten Parteien, entsteht ein qualitativ hochwertig aufgebautes Software-Produkt. Der DevOps-Ansatz erleichtert dadurch einem zukünftigen, neuen Entwickler-Team das Verständnis und die Wiederaufnahme von Arbeiten an diesem Code. Die oben erwähnten Pflaster werden dann seltener benötigt.

Mindset und Wissensaustausch sind die Erfolgszutaten für DevOps

Natürlich ist nicht gesagt, dass die genannten Probleme durch den DevOps-Gedanken nie wieder auftreten – sie werden aber deutlich seltener, sofern ein eingespieltes Team mit entsprechend hoher DevOps-Erfahrung beteiligt ist. DevOps ist gewissermaßen eine Anleitung dafür, wie ein Team einen solchen Grad an Abstimmung erreichen kann. Besonders das Mindset ist essentiell. Denn es sind die Team-Mitglieder, die vom Wissensaustausch getrieben sind, die den DevOps-Gedanken voranbringen. Das heißt, sie sind nicht nur bestrebt ständig Neues zu lernen und ihr Arbeitsumfeld bewusst wahrzunehmen, sondern teilen dieses Wissen auch mit ihren Mitmenschen. Der DevOps-Erfolg zeigt sich darin, dass ein Team sich gerne stetig verbessert und über sein Arbeiten reflektiert.

Zwei Hürden bei der Realisierung von DevOps: Management und Operations

Es ist viel zu beachten, um den richtigen Katalysator finden zu können, der den DevOps-Prozess ins Rollen bringt. So setzt ein täglicher Kontakt mit der Denkweise der Systeme eine Öffnung und Kooperationsbereitschaft bei zwei organisatorischen Einheiten einer Firma voraus, damit überhaupt der erste Schritt von Dev nach Ops passieren kann: Einerseits muss sich Ops in Richtung Dev bewegen und andererseits muss das Management diese Bestrebungen unterstützen.

Die Bereitschaft der Abteilung Operations, dass sie Ressourcen und eine Bühne zum kooperativen Austausch bietet, ist grundlegend, damit die Entwicklungsabteilung einen Überblick zum Thema Ops bekommt. Beim täglichen Geschäft ist es wichtig, dass Entwickler auf direktem Weg den Kontakt zu Personen im Betrieb finden können. Zudem muss es einfach und klar sein, sich in der Server-Landschaft per Login oder Frontend-Tools zu bewegen. Neben der Schaffung einer übersichtlichen

technischen Basis muss der Betrieb auch bereit sein, die Rolle eines Lotsen zu übernehmen und damit einem breiten Publikum für Support-Anfragen bereitstehen. Dem entgegenkommend müssen Entwickler fähig sein, Einblicke in Schnittstellen und Konfigurationen ihres Software-Produkts zu geben. All dies kann nur dann funktionieren, wenn das Management den Wandel der Verantwortungen sowie das Ineinandergreifen der zwei klassisch getrennten Abteilungen Entwicklung und Betrieb bevollmächtigt.

Diese Werkzeuge erleichtern die Arbeit

Eine technologische Entwicklung, die DevOps erheblich erleichtert, ist Virtualisierung. Der von Kief Morris geprägte Begriff „Infrastructure-as-Code" steht für die neuen Möglichkeiten, die sich Entwicklern im Cloud-Zeitalter bieten. Dank einer virtualisierten Infrastruktur muss kein direkter Zugriff mehr auf die Hardware erfolgen. Ganze Server können per Programmierzeile ausgeführt, dupliziert oder gelöscht werden. Zudem möglich wird eine zentrale Verwaltung des Quellcodes mit Versionierung sowie automatisiertes Testen der Konfiguration von virtuellen Maschinen und den entsprechenden Deployments. Besonders die Automatisierung hilft den Zeitaufwand bei Entwicklern und Betreibern zu senken.

Für DevOps sind vor allem die folgenden Tools bei der Verknüpfung von Entwicklung und Betrieb hilfreich:

- Virtualisierungslösungen wie VirtualBox sind das A und O gelungener DevOps. Sie helfen den Entwicklern dabei, ihren Code auf lokalen virtuellen Maschinen zu testen. Damit lässt sich bereits in der Entwicklung testen, wie die Server-Umgebung auf die Software reagiert.
- Automatisierungslösungen wie Puppet, Vagrant oder Ansible unterstützen Entwickler und Betreiber dabei, sich auf die wesentlichen Aspekte ihrer Aufgaben zu konzentrieren, indem sie routinemäßige Prozesse selbst übernehmen. Dies lässt sich fürs Untersuchen, Testen und Ausführen sowohl auf Software- als auch auf Infrastrukturseite anwenden.
- Container-Plattformen wie Docker helfen bei der Bereitstellung von Containern, d.h. lauffähiger virtueller Zusammenfassungen einzelner Anwendungen – mitsamt allen Bibliotheken, Hilfsprogrammen und sonstigen Daten, jedoch ohne ein Betriebssystem. Container können im Umfeld eines Host-Betriebssystems gestartet werden und sind daher ressourcenschonender als eine komplett virtualisierte Anwendungsumgebung. In Verbindung mit der eingebauten Standardisierung in Skalierung, Service-Discovery oder Load-Balancing (z.B. bei Docker-Swarm) ist das Arbeiten mit Containern für DevOps-Entwickler mittlerweile essentiell.

Welche Skills DevOps brauchen

Die von David Guest eingeführte „T-Shape" gibt einen Anhaltspunkt: Die vertikale Säule des Buchstabens „T" symbolisiert die Spezial-Fähigkeiten des Mitarbeiters in einem eingegrenzten Fachgebiet; der horizontale Balken symbolisiert die Fähigkeit zum transdisziplinären Arbeiten. Für DevOps ist eine enge Einbindung der Anforderungen der Betreiber- bzw. Infrastrukturseite notwendig. Deshalb ist für die Mitarbeiter vor allem die Horizontale wichtig. Generalisten mit Offenheit für einen neuen Ansatz und den entsprechenden Technologien sind gefragt. Der Umgang mit diesen lässt sich vergleichbar leicht erlernen, da manche von ihnen ohnehin einen überschaubaren Funktionsumfang besitzen oder sich auch in einer Basisversion einsetzen lassen. Ein ebenso wichtiger Skill ist die Offenheit für einen Wissensaustausch und die Fähigkeit, den Mitmenschen im Team eine neue Technologie näherzubringen und ihren Fokus auf die relevanten Stellen zu legen.

Ähnlich wie das Megathema Agilität hat sich der DevOps-Ansatz als Schlagwort etabliert, hinter dem sich eine Sammlung von Anforderungen und Business-Problemen verbirgt. Bei der agilen Software-Entwicklung haben sich im Laufe der Zeit feststehende Methoden, Werkzeuge und Strukturen entwickelt und etabliert. Ein ähnlicher Prozess lässt sich bei DevOps feststellen. Dennoch sollte DevOps nicht mit einem Stellenprofil verwechselt werden. Auch wenn es mittlerweile Experten gibt, die sich auf DevOps spezialisiert haben, ist das Thema in einem Querschnitt angesiedelt und erfordert, ganz analog zur Agilität, ein Mindset, einen Kulturwandel. Gerade deshalb geht es nicht um einzelne Kompetenzen, Fähigkeiten oder gar Tools – DevOps ist eine radikal neue Art zu arbeiten.

Eine neue Stufe für die IT

Sowohl bei Entwicklern als auch IT-Verantwortlichen ruft der Begriff DevOps immer noch einige Fragezeichen hervor. Es führt aber auf lange Sicht nichts an dem Thema vorbei, denn der DevOps-Ansatz wird aller Voraussicht nach in den nächsten Jahren zum Industriestandard werden und ist in manchen Sparten bereits gängige Praxis. Damit DevOps zum Standard in Unternehmen werden kann, müssen Entwickler umlernen und sich stärker mit Fragen des Betriebs der Software beschäftigen. Dabei helfen die in DevOps etablierten Rollen, sowie Tools und spezifische Skills.

Insbesondere gilt es, all diese Zutaten so zu vermengen, dass die beteiligten Parteien Appetit darauf bekommen, die Realisierung des DevOps-Gedankens mitzugestalten. Die eingangs erwähnte Transformation hin zum „digitalen Fließband" erinnert an die Einführung des Fließbands in den Automobilfabriken Henry Fords. Sie hat nicht weniger als eine Revolution in der industriellen Produktion ausgelöst. Eine ähnliche Revolution zeichnet sich derzeit in der IT-Welt ab.

#JAVAPRO #Container #DevOps #Agile

Neun Best-Practices für Container

IT-Prioritäten verschieben sich: Waren es früher Konsolidierung und Kostenreduktion, stehen heute Agilität und Geschwindigkeit ganz oben auf der Agenda. Neue IT-Modelle, Entwicklungs- und Betriebsprozesse wie Container, DevOps und Microservices gewinnen daher zunehmend an Bedeutung.

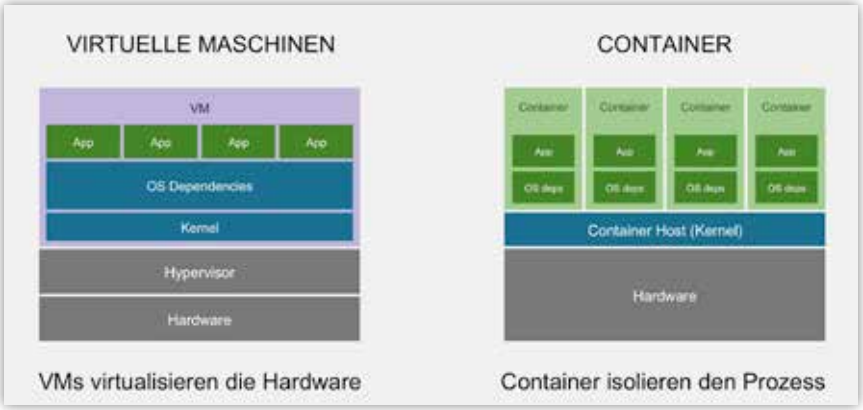
Die Vorteile von Containern liegen auf der Hand: Erstens definieren sie ein einheitliches Format für Applikationen, die deshalb erheblich schneller produktiv eingesetzt werden können (Time-to-Market), zweitens kapseln Container Anwendungen von der darunterliegenden Hardware und machen sie deshalb wesentlich portabler. Die Gleichartigkeit von Container-basierten Anwendungen beim Starten, Stoppen und dem Auslesen von Metriken und Logs bildet die Basis effizienter Betriebsprozesse. Außerdem benötigen Container keinen Hypervisor, sondern nutzen Betriebssystemfunktionen, sodass sie sehr ressourcenschonend und schneller im Vergleich zu virtuellen Maschinen arbeiten (Abb. 1). Trotz all dieser Vorteile fällt es Unternehmen nach wie vor schwer, Container-Technologien einzuführen und sie in die vorhandene IT-Systemlandschaft zu integrieren.

Einer der Gründe dafür: Die Container-Technologie erfordert einen grundlegenden Wandel, denn das Silodenken zwischen der Software-Entwicklung, der Storage- und Netzwerk-Administration und dem Server-Betrieb muss überwunden werden. Wenn neue Software-Releases nach dem DevOps-Modell schneller und häufiger bereitgestellt werden sollen, müssen bei der Integration von Container-Technologien in die Unternehmens-IT vorhandene Prozesse angepasst und neue etabliert werden. Neun Best-Practices unterstützen Unternehmen bei der Einführung der Container-Technologie.

Autor:

Sebastian Faulhaber ist Senior Solution Architect, Technical Teamlead Middleware D/A bei Red Hat.

Sebastian Faulhaber hat über acht Jahre als technischer Berater mit den Schwerpunkten Business Integration und Business Process Management (BPM) bei IBM gearbeitet. 2013 kam er als Solution Architect zu Red Hat. In dieser Funktion setzt Faulhaber seine umfassende Praxiserfahrung ein, um Kunden bei der erfolgreichen Implementierung der Open-Source-Technologien von Red Hat zu unterstützen.



VMs und Container - Linux-Container teilen sich einen Container-Host und Kernel und verwenden gleichzeitig Kernel-Technologien wie cgroups, SELinux und Kernel-Namensräume, sodass ein Container weder auf die anderen Container, die auf demselben Host laufen, noch auf den Host selbst zugreifen oder diese beeinflussen kann (Quelle: Red Hat). (Abb. 1)

1. Transparenz herstellen und Wertschöpfung definieren

Ein wichtiger Startpunkt besteht darin, dass alle Beteiligten aus der Software-Entwicklung und dem IT-Betrieb ein gemeinsames Verständnis schaffen, was es heißt, Software in den IT-Betrieb zu überführen. Entscheidend ist dabei der Unterschied zwischen Lead-Time und Processing-Time. Die Lead-Time beginnt mit der Spezifikation einer Anforderung und endet, wenn der Auftrag erledigt ist. Die Processing-Time läuft erst dann, wenn die eigentliche Arbeit anfängt; Liegezeiten zählen nicht dazu. Je kürzer die Durchlaufzeit, desto besser.

Dazu gehört eine Klärung folgender Fragen: Welcher Schritt im Prozess dauert wie lange? Wie viel Zeit vergeht, bis eine bestimmte Funktion in den produktiven Betrieb eingeführt wird? In der traditionellen Software-Entwicklung kann das einige Wochen oder gar Monate dauern. Relevant ist die Zeit, bis die Software produktiv läuft und einen Nutzen erzielt. In vielen Fällen ist bislang der Prozess der Software-Entwicklung eine Black-Box. Durch eine Visualisierung wird der Prozess transparent: Welche Schritte gibt es? Welches Team ist wofür zuständig? Wo gibt es Liegezeiten? Eines der Werkzeuge dafür ist Value-Stream-Mapping (VSM), bei der die Wertschöpfungskette dargestellt wird.

2. Grundprinzipien effizienter Prozesse berücksichtigen

Value-Stream-Mapping ist eine Methode, die ihren Weg aus der Fertigung in die Software-Entwicklung gefunden hat und kommt nun auch immer häufiger bei der Software-Entwicklung zum Einsatz. Die entsprechenden Tools zeichnen den Material- und Datenfluss vom Lieferanten bis zu den Kunden nach. Damit sind auf einen Blick Ursachen für Verzögerungen im Workflow erkennbar, aber auch Engpässe, die sich auf den Produktionsprozess auswirken. Entstanden ist VSM in engem Zusammenhang mit dem Toyota-Production-System, bei dem es unter anderem darum geht, wie Prozesse im Allgemeinen optimiert werden können.

Eine der Aktivitäten besteht darin, Arbeitsschritte sichtbar zu machen. In der agilen Software-Entwicklung gibt es dazu Techniken wie das Kanban-Board, mit dem sich die Arbeitseinheiten visualisieren lassen. Das gibt es beispielsweise auch in digitaler Form mit der Projektmanagement-Software Trello. Eine wichtige Kenngröße ist Work-in-Progress und die Ermittlung, wie viele und welche Arbeitsschritte parallel stattfinden. Idealerweise sollten so wenig Schritte wie möglich parallel durchgeführt werden. Ein weiterer Aspekt ist, die Größe der Software-Pakete zu begrenzen. Je größer die Pakete, desto größer auch das Risiko, dass sich Fehler einschleichen, die erst im produktiven Einsatz bemerkt werden – und nur aufwendig zu beheben sind. Werden kleinere Pakete in die Produktion gebracht, sinkt das Fehlerrisiko.

3. Geeignetes Pilotprojekt auswählen

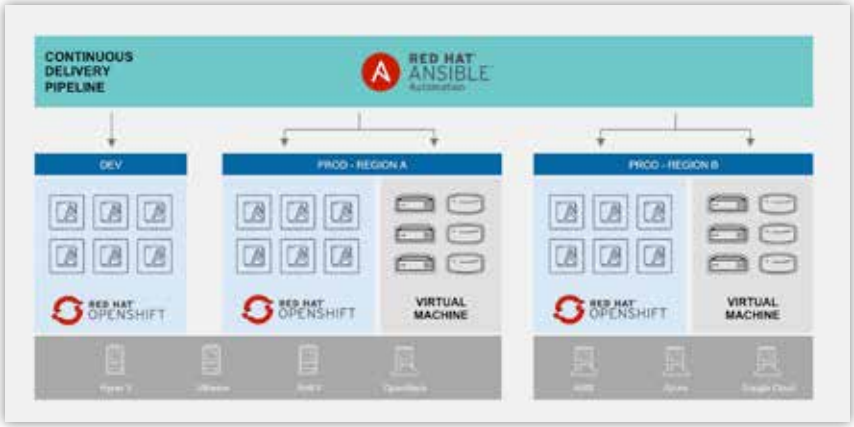
Bei einem Greenfield-Projekt gestaltet sich aus organisatorischer Sicht die Einführung von Container-Technologien einfacher als bei einem Brownfield-Projekt, bei dem manchmal mit allen Mitteln versucht wird, vorhandene Applikationen in die Container-Welt zu migrieren. Vor allem aber sind bei neuen Projekten die Komplexität und die Abhängigkeiten geringer und damit auch die Risiken des Scheiterns. Zudem hat es sich in der Praxis bewährt, dass in der Anfangsphase der Container-Einführung eher Entwickler beteiligt sein sollten, die als innovativ gelten und neuen Technologien gegenüber aufgeschlossen sind.

Von der technischen Seite gesehen, eignen sich für ein Pilotprojekt häufig Web-basierte Applikationen. Diese sind in der Regel einfacher strukturiert und lassen sich in Programmiersprachen wie Java, PHP oder Perl erstellen. Zudem sollte die Anwendung nicht zu viele Komponenten haben und außerdem sollte diese möglichst wenig zustandsbehaftet sein (Stateless vs. Stateful).
Ideale Voraussetzungen liegen vor, wenn die IT-Abteilung mit Microservices experimentiert. Microservices und Container passen perfekt zusammen. Aber auch für Brownfield-Anwendungen werden Container immer interessanter: Selbst die Anbieter von Standard-Software orientieren sich immer stärker in Richtung Container.

4. DevOps-Kultur etablieren

Grundsätzlich gilt: Es existiert keine universell gültige DevOps-Definition. Allerdings gibt es eine Reihe von wichtigen Rahmenbedingungen. Dazu gehören die offene Kommunikation zwischen

den Team-Mitgliedern, ein gutes Vertrauensverhältnis und die offene Aussprache über Fehler. In der DevOps-Kultur gelten Fehler als Möglichkeiten zur Verbesserung. Es wird versucht, Fehler mit Swarming-Technologien zu beheben. Wenn etwa ein Software-Release in der Produktion nicht funktioniert, wird die Weiterentwicklung gestoppt bis das Problem durch das gesamte Team, bestehend aus Entwicklung und Betrieb, gelöst wird. Vor allem aber soll aus dem Vorfall gelernt werden, um in künftigen Situationen schneller reagieren zu können. Dazu gehört auch, schnell und regelmäßig Feedback zwischen allen Beteiligten auszutauschen.



Continuous-Delivery-Pipelinie - Durch eine Kombination von Red Hat Ansible Automation und Red Hat OpenShift Container Platform können Unternehmen die Bereitstellung und Verteilung hybrider Anwendungen automatisieren, und diese auf OpenShift sowie auf anderen On-Premises- und Cloud-Plattformen implementieren und betreiben (Quelle: Red Hat). (Abb. 2)

Dieser Lösungsweg stammt ebenfalls aus dem Toyota-Production-System, bei dem es eine sogenannte Andon-Cord gibt. Kommt es im Produktionsprozess zu Problemen, zieht ein Mitarbeiter diese Reißleine. Das Grundprinzip dahinter: Es wird sofort auf Fehler reagiert. Jeder Mitarbeiter hat in diesem Prozess die gesamte Wertschöpfung und nicht nur seine „kleine Insel" im Blick. Das unterscheidet sich stark davon, wie heute noch in siloartigen Organisationsformen in der IT gearbeitet wird. Hier wird bestenfalls das eigene Silo, nicht aber das große Ganze optimiert. Gerade die Einführung der Container-Technologie bietet die Möglichkeit, das Silodenken zu überwinden. Denn bei der Nutzung von Containern müssen Mitarbeiter aus der Entwicklung, dem Storage- und dem Networking-Bereich zusammenarbeiten.

5. Schrittweise Automatisierung einführen

Dadurch, dass Container, bedingt durch die zugrundeliegende Technologie und das einheitliche Format, Anwendungen gleichartig aussehen lassen, bieten sie die ideale Grundlage für die Automatisierung. Es muss beispielsweise nur einmal ein Prozess implementiert werden, um Daten zu verschieben oder Artefakte zu deployen. Vor allem aber lässt sich die Automatisierung auch schrittweise einführen.

Die IT-Abteilung kann beispielsweise mit Continuous-Integration (CI) – also mit dem täglichen Erstellen von lauffähigen Software-Paketen – anfangen. Der nächste Schritt ist der Aufbau von Continuous-Deployment (CD)-Pipelines, um Container von einer Betriebsumgebung in die nächste zu bringen **(Abb. 2)**. Dazu kommt schließlich das automatisierte Testen. Hier wird mit Unit-Tests gearbeitet, um die Qualität, Wiederholbarkeit und Verlässlichkeit beim Testen zu optimieren. CI und CD funktionieren mit Containern deutlich einfacher als mit traditionellen Mitteln; hier nutzt jede Umgebung ihre eigenen Verfahren. Container ermöglichen es, CI und CD für sehr viele Applikationen auf eine einheitliche und einfache Art und Weise zu betreiben.

6. Grundlegende Sicherheitsmaßnahmen berücksichtigen

Als die Container-Technologie vor einigen Jahren aufkam, hat man sich anfangs noch nicht so stark wie heute mit dem Thema Sicherheit beschäftigt. Ein Applikations-Container verschnürt bekanntlich den Programm-Code zusammen mit dessen Laufzeitabhängigkeiten in einem Paket. Es ist daher unabdingbar, dass die zugehörigen Images und das Container-Betriebssystem regelmäßig mit den neuesten Sicherheits-Updates aktualisiert werden.

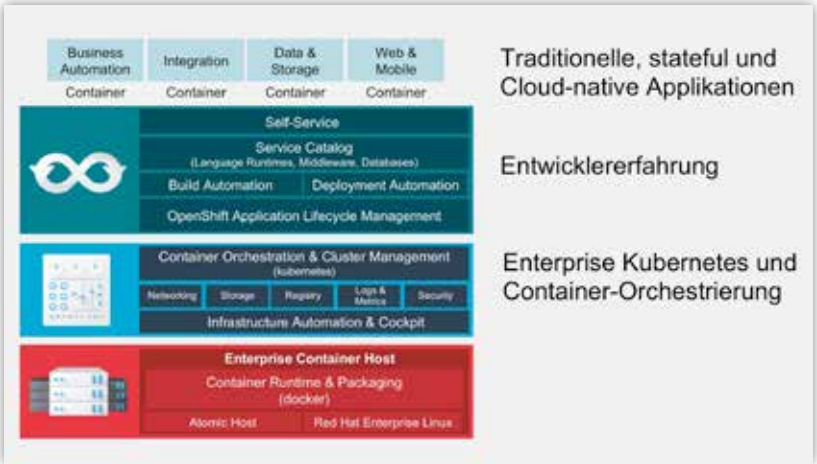
Darüber hinaus muss der Kernel Funktionen bieten, die einen sachgemäßen Umfang von Isolation und Begrenzung bieten. Diese Anforderungen erfüllen etwa SELinux, als integraler Bestandteil von Red Hat Enterprise-Linux, Seccomp oder Namespaces. SELinux isoliert Container voneinander und erlaubt nur den Zugriff auf notwendige Ressourcen. Zudem befasst sich seit Neustem auch das BSI (Bundesamt für Sicherheit in der Informationstechnik) mit der Container-Sicherheit und hat dazu einen Grundschutzbaustein entworfen.

7. Auf Standards setzen

Innerhalb weniger Jahre haben sich Container-Technologien von einem Nischenstatus zu einem zentralen Faktor für die Applikations-Bereitstellung weiterentwickelt. Das Herzstück dabei sind Linux-Container auf Basis des OCI-Formatd (Open Container Initiative), das von vielen führenden IT-Unternehmen, darunter Amazon, Google, Hewlett Packard Enterprise, IBM, Microsoft und Red Hat unterstützt wird. Das OCI-Container-Format ist aus dem Docker-Format hervorgegangen und gilt in der Zwischenzeit als Industriestandard unter Schirmherrschaft der Linux-Foundation. Nicht nur Software-Hersteller, sondern auch Unternehmen, die mit Linux-Containern Anwendungen entwickeln und betreiben, beteiligen sich aktiv in der Linux-Foundation.

OCI besteht erstens aus einem Format für die Container-Images selbst – ein Pendant zu den Docker-Images des Unternehmens Docker. Dazu kommt zweitens die sogenannte Runtime-Specification, die das Laufzeit-Management von Containern definiert.

Mitte 2017 publizierte die OCI die Version 1.0 der Standards für die Runtime-Spezifikation und die Image-Format-Spezifikation für Linux-Container. Mit diesen Standards als Basis können Container-Anbieter, aber auch Unternehmen, Lösungen entwickeln, die erweiterte Funktionen enthalten und dennoch miteinander kompatibel sind.



Pakettierung einer Applikation als Container mit Red Hat OpenShift (Quelle: Red Hat). (Abb. 3)

8. Eine Orchestrierungsplattform einsetzen

Die Standardisierung ist darüber hinaus auch ein Trend bei der Orchestrierungsplattform. Anfangs hatte nahezu jeder Anbieter seine eigene Container-Orchestrierungsplattform. In der Zwischenzeit hat sich Kubernetes als Industriestandard durchgesetzt. Kubernetes ermöglicht es, Container auf Clustern physischer, virtueller Maschinen oder auch in der Cloud bereitzustellen und auszuführen.

Linux und eine Container-Runtime bilden Schlüsseltechnologien für Kubernetes. Linux betreibt die Container und ist für das Ressourcen-Management und die Sicherheit zuständig. Die Container-Runtime verwaltet die Instanziierung und die Ressourcenzuweisung auf dem Host. Red Hat OpenShift Container-Platform beispielsweise kombiniert Enterprise-Grade-Kubernetes mit leistungsstarken Funktionen für die Erstellung, die Implementierung und das Management von Applikations-Containern (Abb. 3).

9. Mit anderen Anwendern vernetzen

Um mit Applikations-Containern erfolgreich zu sein, ist es grundsätzlich wichtig, im Unternehmen eine DevOps-Kultur aufzubauen. Das haben viele Anwender erkannt und beteiligen sich daher zusätzlich an den unterschiedlichsten Meetups und Anwender-Communities oder gründeten bei Bedarf selbst welche.

Ziel dabei ist es, Erfahrungen in der eigenen Organisation, aber auch unternehmensübergreifend auszutauschen. Möglichkeiten dafür bieten darüber hinaus vielfältige Open-Source-Communities rund um Container-Technologien, die sich durch ein offenes Modell und die enge Zusammenarbeit von Software-Herstellern, Anwendern und freiberuflichen Open-Source-Entwicklern auszeichnen. Wie geht man bestimmte Herausforderungen an? Welche konkreten Erfahrungen haben andere Entwickler gemacht? Welche Lösungswege haben sie bei bestimmten Problemen gefunden? Wie lassen sich Innovationen vorantreiben? Die Kubernetes-Community befasst sich beispielsweise mit Cluster-Stability, Extensibility, Service-Automation, Security-Improvements und Workload-Diversity. Umfangreiche Informationen zu Kubernetes und zu Experten-Treffen finden sich unter anderem in der deutschsprachigen OpenShift-Anwender-Community.

Fazit:

Die generelle Empfehlung für den Einstieg in die Container-Technologie lautet klein anzufangen, beispielsweise mit einer neuen stateless Web-Applikation, die nicht allzu komplex sein sollte. Anschließend können IT-Teams iterativ vorgehen, um dann sukzessive die weiteren Möglichkeiten zu testen und zu erschließen. In dieser Phase stehen dann Themen wie CI und CD, automatisiertes Testen und der Einsatz einer Orchestrierungsplattform auf der Tagesordnung.

Da die Welt aber nur zu einem geringen Teil aus Greenfield-Anwendungen besteht, sollten Unternehmen sich auch frühzeitig Gedanken machen, was mit den Brownfield-Anwendungen geschehen soll. Wie lassen sich diese in die Container-Welt überführen? Die einfachste Form ist „Lift and Shift". Dabei wird die Applikation selbst jedoch nicht modernisiert, sie wird lediglich neu paketiert und in die Cloud migriert und dort betrieben. Beim „Re-Platforming" – der nächsten Stufe – analysieren Entwickler, welche Teile einer Applikation am meisten von Containern profitieren und bearbeiten diese dann gezielt und setzen sie beispielsweise in Microservices um oder ergänzen sie um einen API-Layer. Am aufwendigsten ist ein „Re-Factoring", bei dem eine Applikation eine völlig neue Architektur erhält und zum Beispiel komplett mit Microservices umgesetzt wird. Für diese letzte Stufe sollte jedoch ein passender Business-Case vorhanden sein.

Quellen:

1 https://bit.ly/2BH9YQS
2 https://www.openshift-anwender.de/.

#JAVAPRO #Security #DevOps #OpenSource

Sicherer Code durch Dev(Sec)Op

Traditionell ist der Software-Entwicklungsprozess in zwei verschiedene Phasen und zugehörige Teams unterteilt: Entwickler schreiben Code und geben diesen an das Operations-Team weiter. Die IT-Sicherheit sorgt dafür, dass bei Freigabe der Software keine Sicherheitsprobleme bestehen. In den letzten Jahren hat sich aber die Art und Weise der Software-Entwicklung grundlegend verändert. Heute muss Sicherheit in alle Phasen des Software-Development-Prozesses integriert werden.

Die Entwicklung von Anwendungen mit Open-Source-Komponenten bietet viele Vorteile. Dazu gehören die Möglichkeiten, Fachkenntnisse sowie Expertise größerer Entwickler-Teams wirksam einzusetzen, erstklassige Implementierungen einzubinden und natürlich die Kosten zu senken. Um dies auf eine sichere Art und Weise zu tun, benötigen Unternehmen jedoch Transparenz darüber, welche Open-Source-Komponenten sie verwenden, woher diese stammen und welches Sicherheitsrisiko mit ihnen verbunden ist. In die National-Vulnerability-Database wurden allein im Jahr 2017 täglich mehr als 30 neue Schwachstellen aufgenommen. Um diese Schwachstellen finden zu können, ist ein vollständiger Einblick in die verwendete Open-Source notwendig.

Meist wird Open-Source ohne Unterstützung eines Herstellers verwendet. Der Anbieter forciert aber weder Fixes noch benachrichtigt er bei Sicherheitsproblemen. Dies bedeutet, dass Unternehmen die genutzten Open-Source-Komponenten selbst auf Updates und Patches hin überwachen müssen. Allerdings gibt es oft nur wenig interne Dokumentation darüber, welche Version einer bestimmten Open-Source-Komponente von welchem bestimmten Anbieter übernommen wurde. Hinzu kommt das Problem des Software-Verfalls: Ältere, zuvor als sicher geltende Open-Source-Komponenten weisen neue Sicherheitsprobleme auf. Und schon wird die Aufgabe des Sicherheits-Teams noch ein bisschen schwieriger.

Autor:

Tim Mackey ist in technischen Communities gut vernetzt, und erfährt so aus erster Hand wie Black Duck by Synopsys den Kunden am besten unterstützen kann. Er kennt die aktuellsten Applikationssicherheitsprobleme und gibt diese an die Entwicklerteams weiter. Seine Spezialgebiete sind Open-Source-Sicherheit, Rechenzentrumssicherheit, Container, Virtualisierung und Cloud-Technologien.
Tim Mackey hat weltweit bereits zu verschiedenen Themen und auf bekannten Veranstaltungen wie OSCON, CloudOpen, Interop, CA World, Cloud Connect und der CloudStack Collaboration Conference gesprochen.

DevOps-Grundsätze

DevOps ist in erster Linie ein Konzept. Es kombiniert kulturelle Philosophien, technische Praktiken und Tools, um Entwicklungs- und IT-Betriebs-Teams bei der Zusammenarbeit zu unterstützen, mit dem Ziel, Software schneller und zuverlässiger zu erstellen, zu testen und schließlich freizugeben. DevOps wird langfristig verfolgt und schließlich Teil der Unternehmenskultur. Letztendlich besteht das Ziel von DevOps darin, Organisationen in die Lage zu versetzen, ihre Kunden effektiver zu bedienen und ihre Wettbewerbsfähigkeit zu erhöhen. Einige Unternehmen führen dazu Entwicklung und Betrieb in einem DevOps-Team zusammen, welches für den gesamten Anwendungslebenszyklus verantwortlich ist.

Die Three-Ways[1] sind Prinzipien, von denen alle DevOps-Muster abgeleitet sind. Sie beschreiben jene Philosophien, die die Prozesse, Verfahren und Praktiken von DevOps einrahmen. Jeder dieser Three-Ways trägt zum Gesamtkonzept DevOps bei und beinhaltet festgelegte Schritte, um Teams dabei zu helfen, DevOps in ihrer Organisation zu erreichen. Sie sollen an dieser Stelle nur kurz zusammengefasst werden:

- Flow: Systemdenken. Die Leistungsfähigkeit des Gesamtsystems ist entscheidend.
- Feedback-Loops: Feedback-Schleifen verstärken in allen Phasen. Gemeinsame Ziele für Dev und Ops definieren.
- Experimentation and Learning: Fortgesetztes Experimentieren und Lernen. Risiken zulassen, damit aus Erfolgen und Fehlern gelernt werden kann.

Der letzte Schritt zum Erreichen von DevOps ist die Integration von Sicherheits-, Änderungs-Management- und Compliance-Kontrollen in die täglichen Aufgaben von Entwicklungs- und Betriebs-Teams. Dadurch wird Sicherheit zu einem Teil jeder Rolle im Software-Entwicklungszyklus, anstatt diese Verantwortung auf das Sicherheits-Team zu beschränken. Der Aufbau von automatischen Sicherheitskontrollen in die DevOps-Prozesse und -Tools, die bereits von Entwicklungs- und Betriebs-Teams verwendet werden, kann viel dazu beitragen, den schnellen Feedback-Flow zu erzeugen, der für eine erfolgreiche DevOps-Implementierung erforderlich ist.

Und was ist jetzt DevSecOps?

Sichere DevOps oder DevSecOps beschreibt die Integration von Sicherheit in alle Phasen des Software-Development-Lifecycle (SDLC). Diese Praxis erfordert allerdings kulturelle und praktische Veränderungen, zum Beispiel:

- Integration von Sicherheit in das Defect-Tracking: Sicherheitsprobleme sollten mit den gleichen Tools verfolgt werden, die bereits von den Entwicklungs- und Betriebs-Teams genutzt werden. Das gebräuchlichste Tool hierzu ist sicherlich Jira, eine speziell für agile Teams entwickelte Planungs- und Verfolgungs-Software. Nach jedem gelösten Sicherheitsproblem sollte es zudem eine Art Manöverkritik geben, die verhindert, dass der gleiche Fehler vom Team wiederholt wird.
- Integration von Sicherheitskontrollen in gemeinsame Code-Repositories und Services: Alle Teams sollten ein Quellcode-Repository mit sicherheitszertifizierten Bibliotheken teilen. Dieses Repository kann zusätzlich zu Toolchains, der Deployment-Pipeline und Standards auch solche Pakete und Builds enthalten, die für die Entwicklung freigegeben wurden (z.B. sichere Versionen von OpenSSL mit korrekten Konfigurationen). Üblicherweise nutzen Teams Verwaltungs-Tools wie Git, Bitbucket und Mercurial sowie binäre Repositories wie Artefactory und Nexus.
- Integration von Sicherheit in die Deployment-Pipeline: Sicherheitstests sollten so weit wie möglich automatisiert ablaufen, so dass sie neben anderen automatisierten Tests in der Pipeline ausgeführt werden können. Die Sicherheitstests sollten bei jedem Commit durch Entwicklung oder Betrieb, selbst in frühen Phasen, durchgeführt werden. Ziel sollten kurze Feedback-Loops sein, so dass alle Teams über mögliche Sicherheitsprobleme bei Code-Commits informiert sind. Nur so können Sicherheitsprobleme schnell erkannt und als Teil der täglichen Arbeit behoben werden. Wartet man damit bis zum Ende des SDLC, sind Fixes in der Regel komplex, zeitraubend und damit teuer. Am einfachsten erreicht man diesen Schritt mit Continuous-Integration-Tools wie Jenkins, Travis und TeamCity sowie Continuous-Delivery-Tools wie Jenkins, XebiaLabs und GoCD.
- Gewährleistung der Sicherheit der Anwendung: Zur Qualitätssicherung sollten statische und dynamische Analysen (SAST, DAST), Software-Zusammensetzungsanalysen (SCA), Container-Scans und interaktive Anwendungssicherheitstests (IAST) durchgeführt werden. Viele dieser Verfahren können Teil einer CI/CD-Pipeline sein.

Anders wie DevOps und CI/CD dürfte Software-Composition-Analysis (SCA) ein Begriff sein, der denjenigen vielleicht nicht so vertraut ist, die sich direkt mit der Software-Zusammensetzung befassen – also Entwicklern und Technikern. SCA-Tools analysieren speziell Quell-Code, Module, Frameworks und Bibliotheken, um Open-Source-Komponenten zu identifizieren und zu inventarisieren sowie bekannte Sicherheitslücken oder Lizenzprobleme auszumachen, bevor die Anwendung für die Produktion freigegeben wird. Hinsichtlich der operativen Seite kann SCA schnell Probleme identifizieren, die in Implementierungsumgebungen wie Container-Images vorhanden sind; eine Situation, die außerhalb des traditionellen technischen Sicherheitsmodells liegt.

Der Weg zur DevOps-kompatiblen SCA-Lösung

Eine DevOps-kompatible SCA-Lösung lässt sich in Teams sowie in weitere Tools integrieren, um den Prozess der Identifizierung,

Kommunikation von Open-Source-Schwachstellen und Lizenzrisiken als Teil des Entwicklungs- und Bereitstellungs-Workflows zu automatisieren.

Eine vollständige SCA-Lösung muss folgende Funktionen umfassen:

- Inventar der verwendeten Open-Source-Software: Ein vollständiges und genaues Inventar des Open-Source-Codes, der in Anwendungen verwendet wird, ist von wesentlicher Bedeutung.
- Übersicht der bekannten Open-Source-Schwachstellen: DevOps-Teams müssen auf öffentliche Quellen zurückgreifen, um festzustellen welche der von ihnen verwendeten Open-Source-Komponenten angreifbar sind.
- Identifikation von Lizenz- und Qualitätsrisiken: Die Nichteinhaltung von Open-Source-Lizenzen kann für Unternehmen ein erhebliches Risiko an Rechtsstreitigkeiten und eine Gefährdung des geistigen Eigentums bedeuten.
- Richtlinien für Open-Source durchsetzen: Die Automatisierung der Software-Entwicklung nimmt zu, dementsprechend sollte es auch die Verwaltung der Open-Source-Richtlinien.
- Warnung zu neuen Sicherheitsbedrohungen: Angesichts der neuen Open-Source-Schwachstellen die jedes Jahr entdeckt werden, müssen Unternehmen kontinuierlich nach neuen Bedrohungen suchen, solange ihre Anwendungen im Einsatz sind.

Fazit

Ein Team, das mit einer solchen SCA-Lösung arbeitet, wird sich schnell von dem gängigen Gedanken verabschieden, dass Sicherheit die Entwicklungsarbeit lediglich behindert. Stattdessen wird Sicherheit schnell als ein wertvoller Teil des CI/CD-Prozesses wahrgenommen, in dem Sicherheitslücken und Lizenzprobleme in Open-Source gefunden und proaktiv behoben werden.

Wenn sowohl benutzerdefinierter als auch Open-Source-Code in der Entwicklung und der Bereitstellung kontinuierlich überprüft werden, haben Unternehmen eine sehr gute Grundlage für die Sicherung ihrer Anwendungen. Die DevOps-Teams sollten eine automatisierte Software-Composition-Analysis für Open-Source- und Drittanbieter-Bibliotheken verwenden, die in automatisierte Builds und als Teil des Code-Check-In integriert sind.

Quellen:

1 https://itrevolution.com/book/the-phoenix-project/

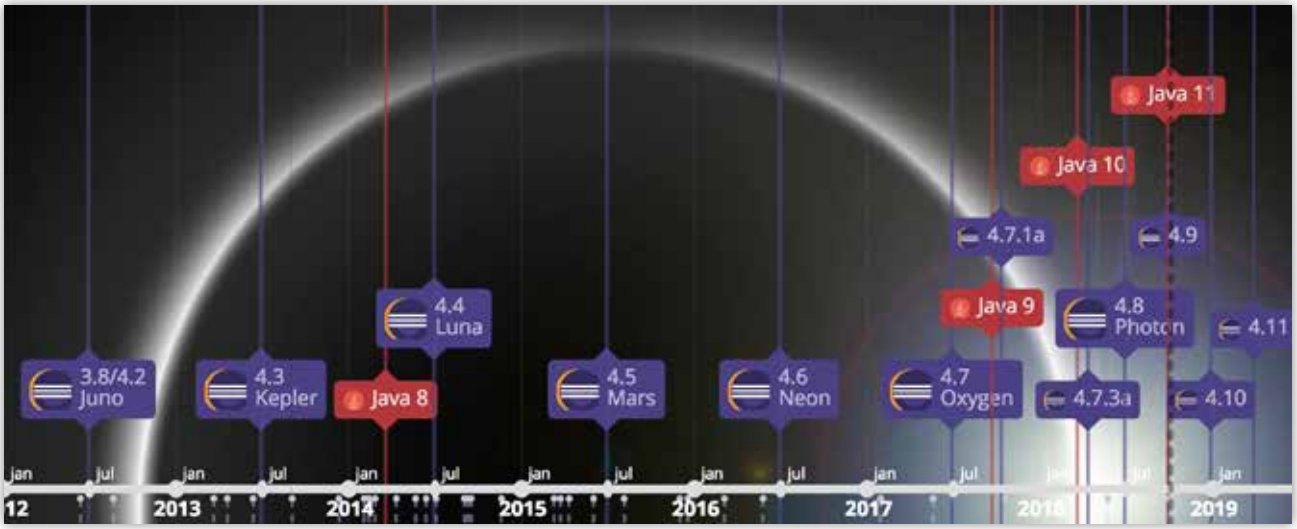
#JAVAPRO #Eclipse #IDE #Tools

IDE-Wars - Die Foundation schlägt zurück

Die Eclipse IDE ist nun bald 20 Jahre alt, für Software- und insbesondere Entwicklungswerkzeuge ein beinahe biblisches Alter. Während es zu damaligen Zeiten nur wenig Alternativen gab und Eclipse den Markt klar dominierte, hat sich das mittlerweile stark verändert. Immer mehr IDEs kämpfen um die Gunst der Entwickler und mittlerweile hat IntelliJ IDEA der Eclipse IDE den Rang abgelaufen. Andere leichtgewichtige IDEs wie VS-Code oder gar Web-basierte IDEs gewinnen immer mehr Nutzer. Eclipse, gerne als alter Dinosaurier verschrien und totgeredet. Nutzer, die Eclipse benutzen, werden bedauert oder verspottet. Steht es denn wirklich so schlecht um die Eclipse IDE? Bei weitem nicht!

Autor:

Karsten Thoms arbeitet seit über 15 Jahren als IT-Consultant und ist Experte in den Themen Domänenspezifische Sprachen (DSLs), Code-Generatoren, modellgetriebene Entwicklung und Eclipse. Er ist seit vielen Jahren aktiver Eclipse-Committer und gehört zum Kern-Team des Eclipse Xtext Projekts sowie der Eclipse IDE. Seine Kunden berät und unterstützt er bei dem Entwurf, der Entwicklung und Integration von DSLs und ist leidenschaftlicher Open Source Verfechter. Seine Erfahrungen teilt er gerne als Speaker auf internationalen Konferenzen, in Fachartikeln und in seinen Blogs.



Java & Eclipse Release-Timeline.[1] (Abb. 1).

Ein Blick zurück

Im Jahr 2012 wurde der Umstieg der Basisplattform von Version 3.x auf 4.x vollzogen. Die damalige Eclipse-Version Juno war tatsächlich eine mittlere Katastrophe. Die IDE war deutlich träger und voller neuer Bugs. Das hat viele Nutzer sehr nachhaltig verärgert. Zum gleichen Zeitpunkt begann der unaufhaltsame Siegeszug von IntelliJ IDEA und ihren Derivaten.

Seitdem sind aber bereits sechs Jahre vergangen. Sieben Major-Releases und viele Minor-Releases liegen seitdem dazwischen, in der die Entwicklung nie nachgelassen hat. In der vergangenen Zeit konnte das Team und insbesondere das Eclipse-Platform-Team, das sich für das Basis-Framework verantwortlich zeigt, vergrößert werden. Auch wenn natürlich immer wieder Committer sich anderen Aktivitäten zuwenden und aus dem Projekt ausscheiden oder passiv werden, gelingt es dennoch, immer neue engagierte Entwickler zu finden, die das Projekt weiter vorantreiben. Allein das Eclipse-Platform-Project konnte für die Entwicklung des Photon-Release 117 individuelle Contributoren zählen.

Neuer Release-Zyklus

Mit Eclipse-Photon ist im Juni, pünktlich wie immer, das letzte benannte Simultaneous-Release erschienen. Fortan wird Eclipse vier Mal im Jahr ein sogenanntes Rolling-Release liefern. Das erste, Eclipse 2018-09, wurde im September fertig gestellt und die Version Eclipse 2018-12 wird im Dezember erscheinen. Damit passt sich Eclipse den heutzutage üblichen kürzeren Release-Zyklen an. Dieses Release ist allerdings nicht nur ein Produkt, sondern es nehmen dazu rund 80 weitere Eclipse-Projekte an dem Release-Train teil, die bei Photon etwa 70 Mio. Lines-of-Code ausmachten. Auch auf diese Projekte kommt nun die Herausforderung zu, häufiger mit Releases am Rolling-Release teilzunehmen. Gerade für das erste Rolling-Release 2018-09 hat das einige Projekte vor Herausforderungen gestellt, so dass auch bis kurz vor Schluss sogenannte Respins erfolgten, die letzte notwendige Updates von Teilprojekten erforderten und integrierten.

Notwendig wurde der kürzere Release-Zyklus u.a. durch die nun aktualisierte Java-Roadmap. Nachdem zwischen Java 1.7, 1.8 und 9 jeweils etwa 3 Jahre lagen, gibt es nun alle 6 Monate ein neues Java-Release. Die entsprechenden Entwicklungswerkzeuge, bei Eclipse ist es das JDT (Java Development Tool), müssen die neuen Spezifikationen natürlich adaptieren und integrieren. Das ist eine ziemliche Mammutaufgabe, doch das JDT-Team leistet hier hervorragende Arbeit und liefert die Eclipse-Unterstützung umgehend. Im Eclipse-Oxygen-Stream passten die Release-Termine für Java 9 und 10 nicht ganz zu den Release-Terminen der Service-Releases, so dass mit Oxygen.1a und Oxygen.3a diese umgehend nachgeliefert wurden. Mit dreimonatigen Releases sind solche Zwischen-Releases nicht mehr notwendig - bis zum nächsten Release ist es nie lang. Die JDT-Unterstützung von Java 11 etwa wurde ein Tag nach dem Java-Release geliefert und kann auf Eclipse 2018-09 installiert werden. In Eclipse 2018-12 wird sie dann direkt integriert sein.

Robustheit durch Eclipse Automated Error Reporting

Mit Eclipse 4.5 wurde das Eclipse Automated Error Reporting eingeführt. Nutzer konnten damit über den Automated Error Reporting Client (AERI genannt) automatisch Berichte an Eclipse schicken, wann immer ein interner Fehler auftrat. Natürlich ist der Nutzer dazu nicht gezwungen, doch genug haben dieses Feature genutzt. Bereits nach einem Jahr wurden so 70.000 Berichte pro Woche[2] gesammelt, Tendenz steigend. Was für den Nutzer mit einem Klick erledigt ist, resultiert für die Entwickler in wertvollen Informationen.

AERI Problemreport. (Abb. 2)

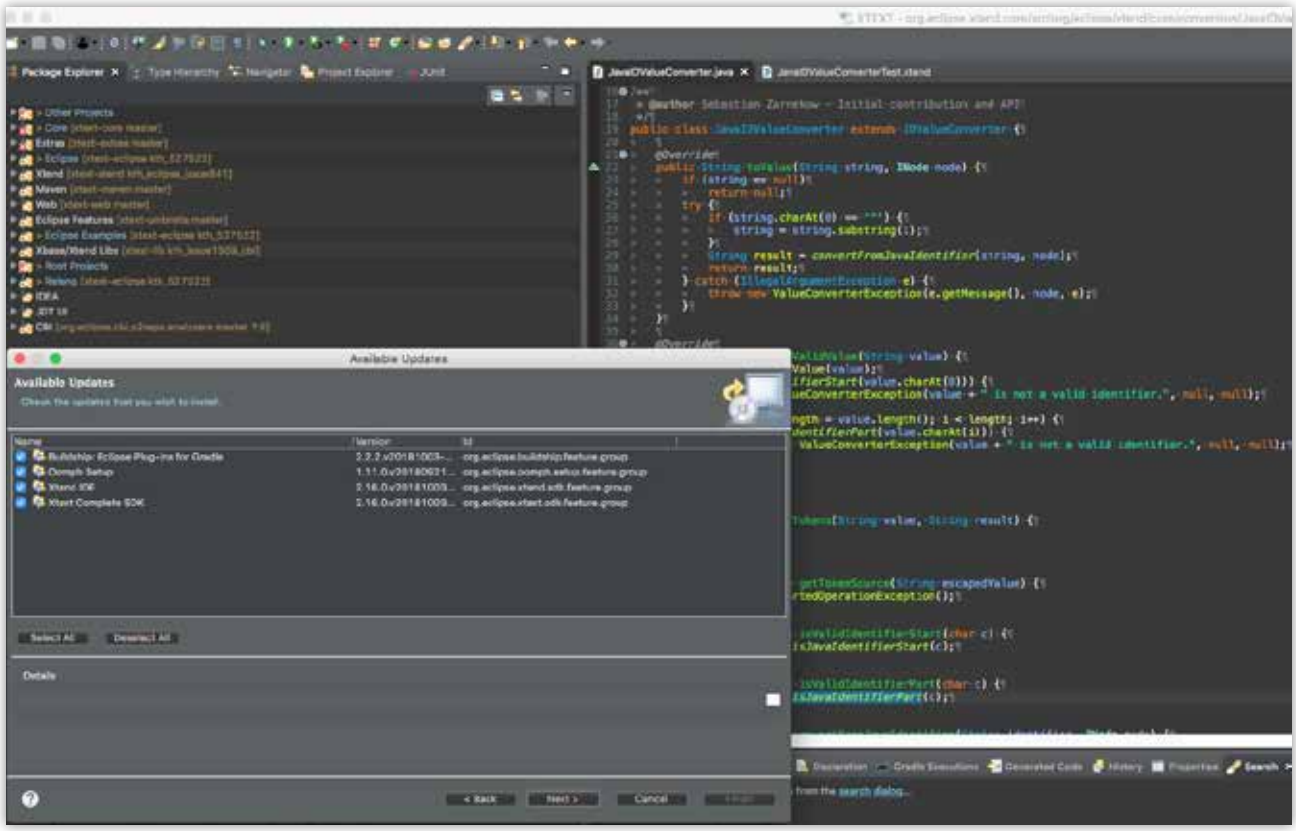
Über diese Fehlerberichte kann festgestellt werden, welche Probleme bei vielen Nutzern häufig auftreten und damit mit höherer Priorität bearbeitet werden. Vieles sind auch "Low-hanging-fruits": Etwa NullPointerException treten häufig auf, wenn Rückgabewerte von Methodenaufrufen nicht richtig vor der Weiterverarbeitung geprüft werden. Solche Fehler können relativ zügig identifiziert und gelöst werden. Dadurch, dass viele dieser kleinen Fehler nun über Jahre abgearbeitet wurden, konnte Eclipse an sich deutlich robuster gemacht werden.

Fokus auf Performance

Für Entwickler gibt es nichts Schlimmeres als unnötiges Warten. Dennoch bleibt es nicht aus, dass komplexe Operationen in der Entwicklungsumgebung auch mal Zeit kosten. Gerade hier lohnt es sich auch mal genauer zu untersuchen, ob diese Zeiten nicht verbessert werden können.

Daran haben die Eclipse-Entwickler in der Vergangenheit immer wieder gearbeitet und vor allem für das Photon-Release einige Prozesse in der IDE durch gezieltes Profiling genauer unter die Lupe genommen. Während andere IDEs durch zunehmende Komplexität doch spürbar träger werden, konnte Eclipse an vielen Stellen schneller werden. Vieles mag nicht direkt an einzelnen Aktionen spürbar sein, wie etwa optimiertes Event-Dispatching im Search-View-Trees UI-Framework, doch insgesamt fühlt sich Eclipse reaktiver an als zuvor. An anderen Stellen konnten Operationen, die spürbar viel Zeit verbrauchten, deutlich optimiert werden. So etwa die Expansion des SWT bei großen Treffermengen oder der Importprozess von Projekten aus Git-Repositories.

Performance ist aber nicht nur auf Geschwindigkeit beschränkt - auch die Optimierung der Speichernutzung wirkt sich unmittelbar auf die Geschwindigkeit aus. Weniger Garbage bedeutet



Eclipse IDE mit Dark-Theme. (Abb. 3)

weniger Garbage-Collection. Darauf wird in Zeiten moderner Garbage-Collectoren bei der Entwicklung häufig (und oft auch zu Recht) nicht gedacht. Doch eine IDE ist von Natur aus ein anderes Stück Software als etwa ein Microservice. In der IDE werden häufig sehr viele kurzlebige Objekte erzeugt, deren Beseitigung nicht viel, aber doch unnötig Zeit verbrauchen. Gerade bei häufig genutzten Operationen wirkt sich die Vermeidung von Garbage positiv auf die Geschwindigkeit aus.

Ebenso wurde am Speicherbedarf intensiv gearbeitet. Eine "blanke" Eclipse Java-IDE benötigt nach Start gerade mal 40 MB Heap minimal und kommt mit 256 MB gut aus. Das ist heutzutage fast gar nichts. Zugegeben kein realistischer Anwendungsfall. Aber nimmt man mal als Extrembeispiel den Workspace, mit dem der Autor an der Eclipse-Plattform und einigen darauf aufsetzenden Projekten simultan arbeitet. Dieser enthält 1.600 Projekte aus 40 Git-Repositories (einiges sind kleine Testprojekte) mit etwa 60.000 Java-Klassen. Diese Eclipse-Instanz gibt sich gar bei einem Full-Build aller Projekte mit einem guten GB-Heap zufrieden. Über Jahre hoch optimierte Builder sei Dank. Zugegeben auch kein normales Szenario, doch gerade in solchen Umgebungen fällt beim Profiling auf, wo man auch noch sinnvoll am kleinen Detail optimieren kann. In kleineren Workspaces würden viele Dinge einfach nicht im Profiler auffallen.

Die dunkle Seite

Heutzutage gelten IDEs mit dunklen Themes als hipp. Mit Einführung der Eclipse-4-Plattform ist ein Theming der IDE möglich und der klassische helle Stil durch ein Dark-Theme austauschbar **(Abb. 3)**. Allerdings war es bislang noch nicht ganz gelungen, dass Theming durchgängig rund zu bekommen. Mit Eclipse Photon hat man noch einmal kräftig daran gearbeitet und nun sieht das Dark-Theme sauber aus. Buttons haben durchgängig dunklen Hintergrund, Icons wurden auch für dunklen Hintergrund optimiert und vieles mehr. Wem das Dark-Theme noch nicht ganz gefällt sollte mal einen Blick auf das DevStyle-Plugin[3] werfen, das über den Eclipse-Marketplace installiert werden kann.

Features, Features, Features

Jedes Eclipse-Release kommt mit einer Hülle neuer Features, die sich hier gar nicht alle aufzählen lassen. Manchmal sind es nur kleine Änderungen, die einen großen Nutzen bringen. So in etwa, dass die Debug-Perspektive überarbeitet wurde und per Default mehr Platz für den zu debuggenden Code lässt. Oder dass im Debugger das Ergebnis des letzten Aufrufs automatisch im Variablen-View angezeigt werden.

Durch die Fülle an Entwicklern mit unterschiedlichem Fokus erhalten mit jedem Release viele nützliche Erweiterungen und Verbesserungen Einzug. Um da halbwegs am Ball zu bleiben sind die umfangreichen New- and Noteworthy Artikel[4, 5] zu empfehlen.

Krieg der IDEs?

Auch wenn es der Titel des Artikels ein wenig provoziert, einen echten Krieg der IDEs gibt es nicht. Es ist Geschmackssache und abhängig von Unternehmen und Projekten, welche IDE vorteilhaft ist oder halt einfach vorgegeben wird. Aussterben wird die Eclipse IDE jedenfalls noch viele Jahre nicht. Durch die aktive Entwicklung wird sie ständig an die neuen Bedürfnisse der Anwender angepasst. Nicht ändern wird sich, dass es eine Desktop-Anwendung ist und der Bedarf nach anderweitigen Entwicklungswerkzeugen durchaus vorhanden und berechtigt ist. Alle diese Werkzeuge werden ihren Anteil am Markt haben. Davon wird Eclipse nicht den größten Teil haben, aber weiter einen signifikanten.

Was man durchaus auch berücksichtigen sollte ist, dass hinter Eclipse nicht ein großes Unternehmen steckt, sondern vor allem viele kleine sowie enthusiastische Entwickler, die ihren Anteil dazu beitragen, damit dieses komplexe Open-Source-Projekt auch weiterhin erfolgreich vorangetrieben wird und stets kostenlos genutzt werden kann. Es ist damit nicht "die Eclipse-Foundation", welche die Entwicklung vorantreibt, sondern das Interesse vieler Beteiligter.

Fazit

Die Eclipse IDE ist aus dem Markt der Entwicklungswerkzeuge nicht wegzudenken. Die Entwicklung ist sehr aktiv und wird durch einen neuen dreimonatigen Release-Zyklus noch einmal beschleunigt. Die neuesten Java-Standards werden jeweils unmittelbar unterstützt und gleichzeitig wird an vielen hilfreichen Features gearbeitet. Mit dem Eclipse Photon-Release konnte noch einmal ein deutlicher Sprung nach vorn in den Bereichen Performance, Speichernutzung, Robustheit und Usability gemacht werden. An diesen Bereichen wird auch in Zukunft besonders intensiv gearbeitet. Die Eclipse IDE bleibt ein wichtiges und nützliches Werkzeug für die Entwicklung von Projekten aller Art.

Quellen:

1 Release Timeline Java, Eclipse, Xtext:
https://time.graphics/embed?v=1&id=99959
2 One Year of Automated Error Reporting https://bit.ly/2Q5HUj8
3 DevStyle Plugin: https://bit.ly/2So6XKZ
4 Eclipse 4.8 New & Noteworthy:
https://www.eclipse.org/eclipse/news/4.8/
5 Eclipse 4.9 New & Noteworthy:
https://www.eclipse.org/eclipse/news/4.9/

#JAVAPRO #SCM #Tools

Versionsverwaltung mit Expressions

Im Umgang mit Source-Control-Management-Systemen (SCM) wie Git oder Subversion haben sich im Lauf der Zeit vielerlei Praktiken bewährt. Neben unzähligen Beiträgen über Workflows zum Branchen und Mergen ist auch das Formulieren verständlicher Beschreibungen in den Commit-Messages ein wichtiges Thema.

Ab und zu kommt es vor, dass in kollaborativen Teams vereinzelte Code-Änderungen zurückgenommen werden müssen. So vielfältig die Gründe für ein Rollback auch sein mögen, das Identifizieren betroffener Code-Fragmente kann eine beachtliche Herausforderung sein. Die Möglichkeit jeder Änderung des Code-Repository eine Beschreibung anzufügen, erleichtert die Navigation zwischen den Änderungen. Sind die hinterlegten Kommentare der Entwickler dann so aussagekräftig wie „Layout-Anpassungen" oder „Tests hinzugefügt", hilft dies wenig weiter. Diese Ansicht vertreten auch diverse Blog-Beiträge [3, 4, 5] und sehen das Formulieren klarer Commit-Messages als wichtiges Instrument, um die interne Kommunikation zwischen einzelnen Team-Mitgliedern deutlich zu verbessern.

Autor:

Marco Schulz studierte an der HS Merseburg Diplominformatik. Sein persönlicher Schwerpunkt liegt in Software-Architekturen, der Automatisierung des Software-Entwicklungsprozess und dem Software-konfigurationsmanagement. Seit über fünfzehn Jahren entwickelt er für namhafte Unternehmen auf unterschiedlichen Plattformen umfangreiche Web-Applikationen. Derzeit arbeitet er als freier Consultant und ist Autor verschiedener Fachartikel.

E-Mail: marco.schulz@outlook.com

Das man als Entwickler nach vollbrachter Arbeit nicht immer den optimalen Ausdruck findet, seine Aktivitäten deutlich zu formulieren, kann einem hohen Termindruck geschuldet sein. Ein hilfreiches Instrument, ein aussagekräftiges Resümee der eigenen Arbeit zu ziehen, ist die nachfolgend vorgeschlagene Struktur und ein darauf operierendes aussagekräftiges Vokabular inklusive einer festgelegten Notation. Die vorgestellte Lösung lässt sich sehr leicht in die eigenen Prozesse einfügen und kann ohne großen Aufwand erweitert bzw. angepasst werden.

Mit den standardisierten Expressions besteht auch die Möglichkeit die vorhandenen Commit-Messages automatisiert zu Parsen, um die Projektevolution gegebenenfalls grafisch darzustellen. Sämtliche Einzelheiten der hier vorgestellten Methode sind in einem Cheat-Sheet auf einer Seite übersichtlich zusammengefasst und können so leicht im Team verbreitet werden. Das diesem Text zugrundeliegende Paper ist auf Research-Gate[1] in englischer Sprache frei verfügbar.

Eine Commit-Message besteht aus einer verpflichtenden (mandatory) ersten Zeile, die sich aus der Funktions-ID, einem Label und der Spezifikation zusammensetzt. Die zweite und dritte Zeile ist optional. Die Task-ID, die Issue-Management-Systeme wie Jira vergeben, wird in der zweiten Zeile notiert. Grund dafür ist, dass nicht jedes Projekt an ein Issue-Management-System gekoppelt ist. Viel wichtiger ist auch die Tatsache, dass Funktionen meist auf mehrere Tasks verteilt werden. Eine Suche nach der Funktions-ID fördert alle Teile einer Funktionalität zu tage,

auch wenn dies unterschiedlichen Task-IDs zugeordnet ist. Die ausführliche Beschreibung in Zeile drei ist ebenfalls optional und rückt auf Zeile zwei vor, falls keine Task-ID notiert wird. Das gesamte Vokabular zu dem nachfolgenden Beispiel ist im Cheat-Sheet notiert und soll an dieser Stelle nicht wiederholt werden.

(Listing 1)

```
[CM-0500] #CHANGE 'function:pom'
<QS-0815>
{Change version number of the dependency JUnit from 4 to 5.0.2}
```

Projektübergreifende Aufgaben wie das Anpassen der Build-Logik, Erzeugen eines Releases oder das Initiieren eines Repositories können in allgemeingültigen Funktions-IDs zusammengefasst werden. Entsprechende Beispiele sind im Cheat-Sheet angeführt.

Möchte man nun den Projektfortschritt ermitteln, ist es sinnvoll aussagekräftige Meilensteine miteinander zu vergleichen. Solche Punkte (POI) stellen üblicherweise Releases dar. So können bei Berücksichtigung des Standard-Release-Prozesses und des Semantic-Versionings[2] Metriken über die Anzahl der Bug-Fixes pro Release erstellt werden. Aber auch klassische Erhebungen wie Lines-of-Code zwischen zwei Minor-Releases können interessante Erkenntnisse fördern.

Quellen:

1 Paper: https://bit.ly/2Sr3USn
2 Semantic Versioning: https://semver.org
3 https://bit.ly/2BJ4dCu
4 https://chris.beams.io/posts/git-commit/
5 https://bit.ly/2rfh8FS

#JAVAPRO #Buchrezension

Java by Comparison

Become a Java Craftsman in 70 Examples

Simon Harrer, Jörg Lenhard und Linus Dietz zeigen in Ihrem Buch Java by Comparison anhand von 70 Beispielen wie man aus einem „unschönen" Code mit Tipps und Tricks einen sauberen Code hinbekommt. Aufgebaut ist diese Lektüre so, dass auf jeder Doppelseite ein Beispiel erklärt wird. Auf der linken Buchseite findet man jeweils einen programmierten Code vor, der nicht so gut gelungen ist und welche Probleme sich damit ergeben. Die rechte Seite zeigt, wie die gute und ordentlich Lösung aussehen kann. Zusätzlich wird ausführlich erläutert, warum diese Lösung gut ist und welche Tipps und Tricks man anwenden kann. Die kompakte Darstellung auf Doppelseiten erleichtert den Vergleich und das Verständnis. Jedes Beispiel wird mit einem leicht zu merkenden Satz beschrieben, so dass sich diese Empfehlungen auch in der eigenen Programmierung einfach umsetzen lassen. Ganz nebenbei wird auch die Syntax von Java 8 verständlich gemacht. Auch für erfahrene Programmierer ist das Buch eine Leseempfehlung, weil vergessenes wieder in Erinnerung gerufen wird und man auch neue Dinge dazu lernt. Es verleitet einen noch während dem Lesen dazu, seinen eigenen Code zu überprüfen. Die Lektüre schließt mit einem Kapitel mit hervorragenden Praxishinweisen und einer Übersicht existierender Code-Analyse-Tools.

Dieses Buch zeigt, wie man seine Java-Skills verbessern und auf ein neues Niveau heben kann.

Buchvorstellung:

Titel: Java by Comparison
Autoren: Simon Harrer, Jörg Lenhard, Linus Dietz
Verlag: O'Reilly UK Ltd., 2018
ISBN: 9781680502879
Umfang: 174 Seiten

#JAVAPRO #Agile #Transition

Agile Transition braucht kulturellen Wandel

Schnelllebige Märkte, wirtschaftliche Unwägbarkeiten und eine fortschreitende Digitalisierung stellen Unternehmen vor große Herausforderungen. Für das wirtschaftliche Überleben ist ein hohes Maß an Anpassungsfähigkeit gefragt. Unternehmen müssen agiler werden. Die Einführung agiler Methoden wird jedoch häufig nur aus technischer Sicht betrachtet. Dabei benötigt eine Organisation auf dem Weg zu mehr Resilienz und Antifragilität vor allem einen kulturellen Wandel.

Viele Unternehmen tun sich schwer damit, auf Veränderungen von außen wie auch auf Fehler flexibel zu reagieren. Besonders unbekannte, nicht kalkulierbare Stressfaktoren setzen ihnen zu. Erstrebenswert wäre daher ein resilientes System, das durch ständige Anpassung und Neuausrichtung auf Veränderungen jeglicher Art reagiert und so einen stabileren Status erreicht als zuvor. Einen solchen Zustand von Antifragilität erreichen Organisationen allerdings nicht mit ihren herkömmlichen Prozessen – sie müssen agiler werden. Für die meisten Unternehmen bedeutet das nicht weniger als einen Paradigmenwechsel in der Managementkultur: Weg von Top-down-Hierarchien hin zu einem agilen Unternehmen, in dem ergebnisoffen und streng an den Marktbedürfnissen orientiert, gearbeitet wird. Doch was können agile Methoden in Unternehmen bewirken?

Der Markt führt

Besonders vorteilhaft lassen sich agile Methoden zum Beispiel bei der Entwicklung neuer Produkte anwenden. Da hier die benötigte Zeit für die verschiedenen Teilaufgaben im Gesamtprozess allenfalls grob geschätzt werden kann, ist eine empirische Prozesskontrolle erfolgversprechender als die herkömmliche definierte. Das Ziel ist es, ein vom Kunden akzeptiertes Ergebnis in möglichst kurzer Zeit zu erreichen. Dazu werden Zeiträume von zwei bis vier Wochen, sogenannte Sprints, festgelegt. Welche Teilaufgaben in diesen Zeiträumen abzuarbeiten sind, entscheidet das Team eigenständig. Die Erfahrungen daraus fließen jeweils in den nächsten Sprint ein, so dass nicht nur das Produkt für den Kunden, sondern auch der Prozess stetig verbessert wird.

Dass sich das Team selbst seine Teilaufgaben setzt und Lösungen findet, wäre mit Vorgaben „von oben" nach dem Push-Prinzip nicht möglich. Nur als eine sich selbst organisierende Einheit entwickelt das Team die nötige Kompetenz, mit der Anforderungen des Kunden in Aufgaben „übersetzt" und effektiv verteilt werden können. Den daraus entstandenen kollektiven Erfahrungsschatz könnte ein einzelner übergeordneter Manager nicht haben. Für diesen Prozess braucht das Team allerdings ausreichend Freiheit und Unterstützung, um sich selbst in dieser Form organisieren zu können.

Im Mittelpunkt des Prozesses steht also konsequent der Kundennutzen als Voraussetzung für die Entwicklung innovativer Produkte und zum Erhalt bzw. zur Steigerung der Wettbewerbsfähigkeit. Mit anderen Worten: Der Markt führt. Agile Methoden zielen darauf ab, dem Kunden schnell und in bester Qualität genau das zu liefern, was er braucht. Alles, was nicht wertschaffend ist, entfällt, das betrifft unnötige Tätigkeiten ebenso wie überflüssige Variationen. Damit der Prozess nicht ins Stocken gerät, ist es wichtig, Überlastungen zu vermeiden – und damit verbunden Fehler und Nacharbeiten.

„Faktor Mensch" steht im Mittelpunkt bei der Einführung agiler Methoden

So einfach und transparent agile Vorgehensweisen auch sind, so schwierig gestaltet sich oftmals ihre Einführung. Besonders, wenn sie auf Unternehmen, Teams und Strukturen angewendet werden sollen, die jahrelang gegenteilig gedacht und gearbeitet haben. Da wurden Aufgaben vorgegeben, Vorgehensweisen genauestens definiert und viel Zeit und Energie in Statusmeetings investiert. Neue Ideen wurden durch langwierige Genehmigungsverfahren nicht selten im Keim erstickt. Und bei all diesen Prozessen kam ein Faktor immer zu kurz: der Mensch. Im Gegensatz dazu setzen agile Methoden auf den „Faktor Mensch", was eine große Umstellung für alle Beteiligten bedeutet.

Daher ist es wichtig, die Veränderung in kleinen Schritten vorzunehmen – am besten zunächst bei Prozessen, die nicht scheitern können, und den Teammitgliedern dabei das Experimentieren zu erlauben. Diese Vorgehensweise hilft, Widerstände abzubauen und hat zugleich den Vorteil, dass Wissen angehäuft wird, das für weitere Projekte genutzt werden kann. Der Fokus bei Entwicklungen liegt rein auf dem Kundennutzen. Der einfachste Weg dorthin sollte gefunden und alles Überflüssige auf dem Weg dorthin weggelassen werden. Dies hilft, die Time-to-Market zu verkürzen. Darüber hinaus verkürzt die Dezentralisierung von Kontrolle und Entscheidungshoheit Feedback-Schleifen, fördert die Zusammenarbeit und steigert Reaktionsfähigkeit und Flexibilität des Unternehmens.

Statt ein Gesamtprodukt als Ziel zu setzen, sollten einzelne, kleinere Features von mehreren Teams entwickelt werden. Die Teams arbeiten dabei ohne starre Deadline mit hoher Eigenverantwortlichkeit. Kleine Entwicklungseinheiten ermöglichen ein schnelles Feedback und eine prompte Reaktion auf eventuelle Fehler, was Entwicklungszeit und Kosten senkt, während Qualität und Teammotivation steigen. Um einen langfristigen Erfolg mit den

Autor:

Marion Eickmann ist Mitgründerin und Mitglied der Geschäftsführung von agile42. Seit über 15 Jahren ist sie im Bereich Software-Entwicklung und Projektmanagement tätig. Aufgrund dieser Erfahrung ist es Marion Eickmann und ihrem internationalen Team möglich, seit 2007 erfolgreich agile Projekte lokal und global umzusetzen.

www.agile42.de
E-Mail: marion.eickmann@agile42.com
Twitter:@agile42
https://de.linkedin.com/in/marioneickmann/de
https://www.xing.com/profile/Marion_Eickmann

agilen Methoden sicherzustellen, sollte eine Unternehmenskultur eingeführt werden, die diese Vorgehensweise unterstützt.

Agile Führung in der modernen Arbeitswelt

In der modernen Arbeitswelt nimmt die Komplexität stetig zu. Immer mehr Aufgaben verteilen sich auf immer weniger Schultern, daher sind in Unternehmen zunehmend Experten gefragt. Durch diese bestens ausgebildeten Mitarbeiter mit Spezialwissen steigt die Kompetenz innerhalb der Teams, die gemeinsam an einer Vision arbeiten. Eigenverantwortung und Kreativität der Teammitglieder sorgen für deutlich bessere und schnellere Ergebnisse als bei der alten Methode. Hierarchiedenken wäre hier also eher kontraproduktiv. Vielmehr sollte jeder dieser Spezialisten als eine Art „Manager" für seinen eigenen Bereich gesehen werden. Doch welche Rolle nimmt die Führungskraft in einer agilen Organisation ein, wenn immer mehr Entscheidungen in den Teams getroffen werden? Wird überhaupt noch eine Führung gebraucht? Gebraucht wird sie, allerdings nicht mehr in der Rolle des alleinigen Entscheiders, der Dinge von oben anordnet. Stattdessen werden nach dem Prinzip des „Shared Leaderships" Entscheidungen gemeinsam getroffen. Dabei fällt dem Vorgesetzten die Rolle eines Moderators und Mentors zu, der sein hoch kompetentes Expertenteam koordiniert und ihm beratend zur Seite steht.

Strukturelle Eingriffe als Basis für agile Methoden

Grundvoraussetzung für die Einführung einer agilen Organisationsstruktur ist bei allen Beteiligten die Einsicht und der Wille zur Veränderung – die allerdings mit größeren Eingriffen in die Organisation einhergeht. Es gilt, kürzere Entscheidungswege und straffere Prozesse zu schaffen. Auf Basis einer Prozessanalyse werden zunächst potentielle Aufwandstreiber identifiziert. Eine Analyse der Tätigkeitsstrukturen gibt Aufschluss über die Effektivität einzelner Arbeitsschritte innerhalb eines Prozesses und hilft, den wahren Ressourcenbedarf für dessen Abwicklung zu ermitteln. Mit diesen Erkenntnissen können die einzelnen Teams auf die neue, agile Arbeitsweise ausgerichtet werden.

Am Ende setzen interdisziplinäre und selbstorganisierte Mitarbeiter im Team Prozesse effizient um, moderierend und beratend unterstützt durch ihre Vorgesetzten und deren Führungskompetenz. Damit Entscheidungen nicht mehr über mehrere Hierarchieebenen hinweg getroffen werden müssen, werden die Verantwortlichkeiten sukzessive vom Vorgesetzten zu den Mitarbeitern verlagert. So sind die Ergebnisse der Gruppenarbeit schneller sichtbar und wirksam.

Loslassen seitens des Managements ermöglicht die Neuorganisation

Dieser Veränderungsprozess stößt nicht selten auf Widerstände bei den Führungskräften, die im agilen Umfeld für sich keine Aufgabe mehr sehen, wenn die Entscheidungshoheit zunehmend auf die Teams übergeht. Für sie ist es wichtig, hier umzudenken, ihre bisherige Rolle im operativen Geschäft loszulassen und der Fähigkeit ihres Teams zu vertrauen, sich selbst zu organisieren. Stattdessen sollten sich Manager auf die strategische Weiterentwicklung des eigenen Bereichs konzentrieren und so das Unternehmen voranbringen. Ein regelmäßiger Austausch mit dem Team hält sie über alle aktuellen Prozesse auf dem Laufenden, so dass sie bei Bedarf jederzeit aktiv mit einbezogen werden können.

Für einen nachhaltigen Erfolg des agilen Prinzips im Unternehmen besteht der erste und entscheidende Schritt darin, noch vor dem Erlernen bestimmter Methoden die dahinter liegenden Prinzipien und Überzeugungen zu verstehen. Wie das Team muss auch der Vorgesetzte begreifen und akzeptieren, welche Veränderungen agile Methoden mit sich bringen. Diese reichen von den Grundprinzipien des „Agile-Leadership" über die veränderte Funktionsweise des Bonussystems in agilen Umgebungen bis hin zur Frage, wie das gesamte Unternehmen agil wird. Aufgabe der Führungskräfte ist es, die Verantwortung für den agilen Wandel zu übernehmen und ihn zu koordinieren.

Unterstützende Tools für die agile Transition

Die Veränderungen bei der Umstellung auf agile Methoden sind enorm und weitreichend. Bei allen Beteiligten wandeln sich Zuständigkeiten und Rollenverständnis gewaltig. Agil sein bedeutet daher nicht nur, neue Vorgehensweisen zu erlernen, sondern sie im Kontext zu sehen, Hintergründe zu verstehen – einschließlich des Verständnis, warum die neue Arbeitsweise zielführend ist. Nur wenn die agilen Prinzipien von allen verstanden werden, kann sich die Team- und Unternehmenskultur ändern. Es empfiehlt sich deshalb, bei der Umsetzung einer agilen Unternehmensführung unterstützend auf Monitoring-Tools zu setzen, die die Transition transparent machen, den Prozess messen, Business-Ziele definieren lassen und dabei zugleich den menschlichen Faktor als wichtigste Erfolgskomponente für eine erfolgreiche Realisation im Fokus haben.

Denn eins muss allen Beteiligten vor Beginn einer agilen Transition klar sein: Sie braucht Zeit und erfordert Geduld, denn jedes Projekt ist einzigartig und daher nicht übertragbar auf andere. Daher muss jedes Unternehmen seinen eigenen agilen Weg finden. Das mag mühsam erscheinen, doch es lohnt sich allemal, wenn es darum geht, ein Unternehmen fit für alle zukünftigen Herausforderungen zu machen.

13 Billion
Devices Run Java

ATMs, Smartcards, POS Terminals, Blu-ray Players,
Set Top Boxes, Multifunction Printers, PCs, Servers,
Routers, Switches, Parking Meters, Smart Meters,
Lottery Systems, Airplane Systems, IoT Gateways,
Programmable Logic Controllers, Optical Sensors,
Wireless M2M Modules, Access Control Systems,
Medical Devices, Building Controls, Automobiles…

Java™ | #1 Development Platform

ORACLE®

oracle.com/java
or call 1.800.ORACLE.1

JAVAPRO
POWER - SEMINAR

JETZT NEU

Java 11

JAVAPRO POWER SEMINAR

- **JAVA 11 - ALLES ÜBER DAS NEUE JAVA SUPPORT- & LIZENZ-MODELL**
- **WAS KOSTET EINE JAVA SE LIZENZ ?**
- **WELCHE ALTERNATIVEN GIBT ES ? - ÜBERBLICK ÜBER DEN JDK-JUNGLE.**
- **JAVA WEB-START & JAVAFX SIND NICHT MEHR BESTANDTEIL VON JAVA** - WIE SIE JETZT SCHNELL & GÜNSTIG AUF JAVA WEB-APPLIKATIONEN ODER HTML-OFFLINE-CLIENTS UMSTELLEN KÖNNEN

TEILNAHMEGEBÜHR:

199,- €

zzgl. 19% ges.MwSt.!

TERMINE & ANMELDUNG:

www.java-pro.de/Seminare/Java11

ODER INDIVIDUELLEN TERMIN VEREINBAREN !

www.java-pro.de