Ausgabe 05/2023

Java aktuell

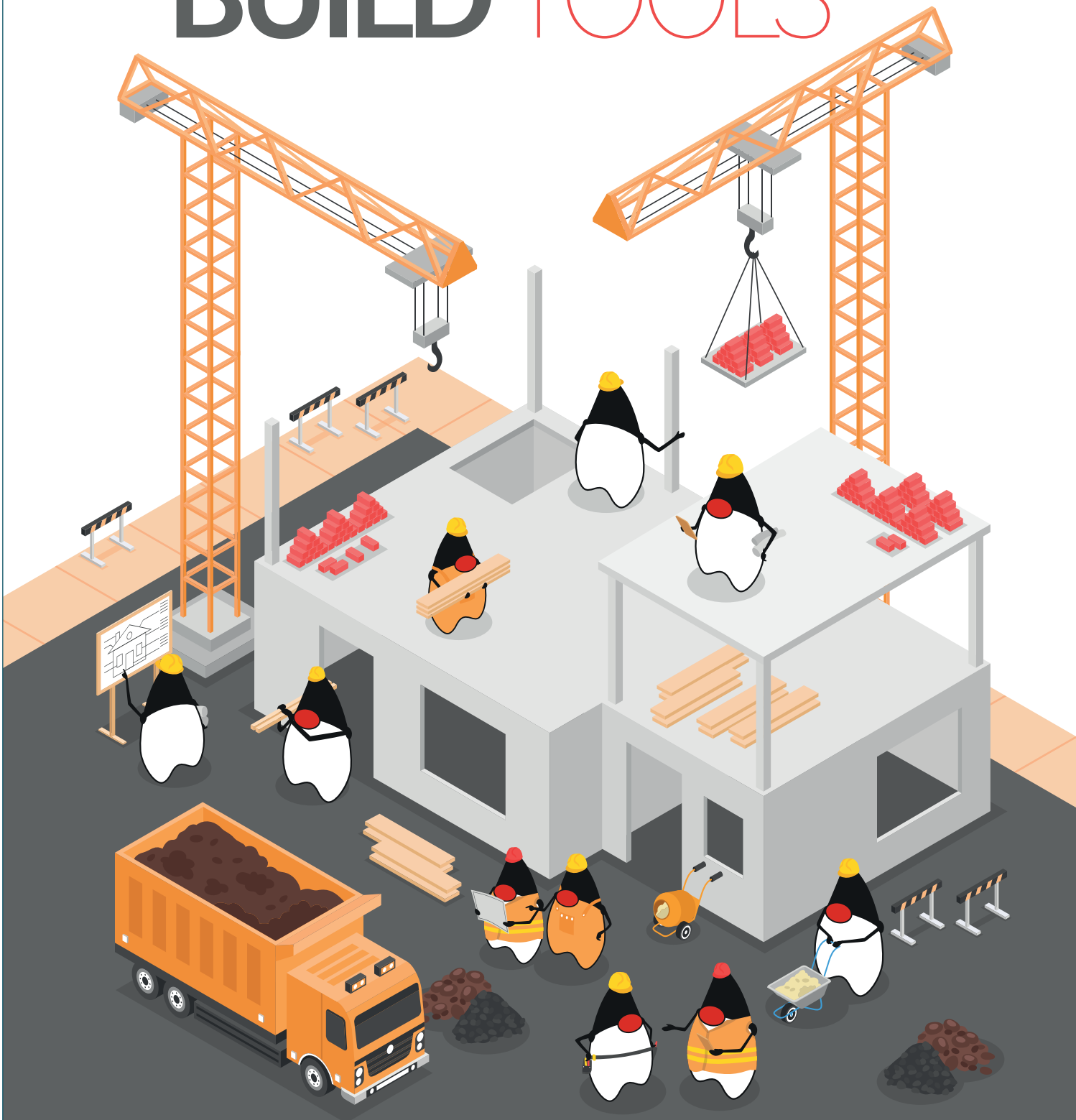
iJUG Verbund
www.ijug.eu

Build Tools
Maven Daemon, Maven Enforcer, Gradle

Kotlin
Für Backend-Applikationen

Performance
Performance-Analyse mit Lastkurven

BUILD TOOLS



Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 05

*Weinberge, Innovation und Vernetzung aller, deren Herz für die IT und Softwareentwicklung schlägt - Auf Sachsens größter IT-Community-Konferenz am 29. September 2023 in Radebeul bei Dresden.

Liebe Leserinnen und Leser,

diese Ausgabe widmet sich dem Thema "Build Tools". Doch zuerst bringen euch das Java-Tagebuch und die Eclipse Corner wieder auf den neuesten Stand rund um alle aktuellen Geschehnisse in der Java- und Eclipse-Community. Außerdem lässt Fin Labusch die diesjährige JavaLand 4 Kids Revue passieren und berichtet, welche Workshops die Mentorinnen und Mentoren für die Developer von morgen vorbereitet hatten.

Zum Einstieg in das Titelthema stellt uns Peter Palaga den Maven Daemon vor, ein Hintergrundprozess, der den eigenen Build beschleunigen soll. Ab Seite 22 geht Roland Weisleider näher auf das Maven-Enforcer-Plug-in ein und präsentiert, wie man mit diesem einerseits Anforderungen und Vorbedingungen für ein erfolgreiches Build definieren und andererseits die "Dependency-Hölle" bezwingen kann. Er zeigt uns, wie wir das Plug-in einrichten und nutzen. Dr. Jendrik Johannes zeigt in seinem Artikel Möglichkeiten auf, wie man das Java-Modulsystem mit Gradle und Gradle-Plug-ins für eine übersichtlichere und wartbarere Projektstruktur einsetzen kann. Wie mithilfe von jlink, jpackage oder GraalVM eigenständig lauffähige Java-Applikationen erstellt werden können, präsentiert Christian Heitzmann in seinem Artikel. Das Ziel dieser Applikationen ist, dass sie vom Endbenutzer ausgeführt werden können, ohne die Voraussetzung einer installierten JVM.

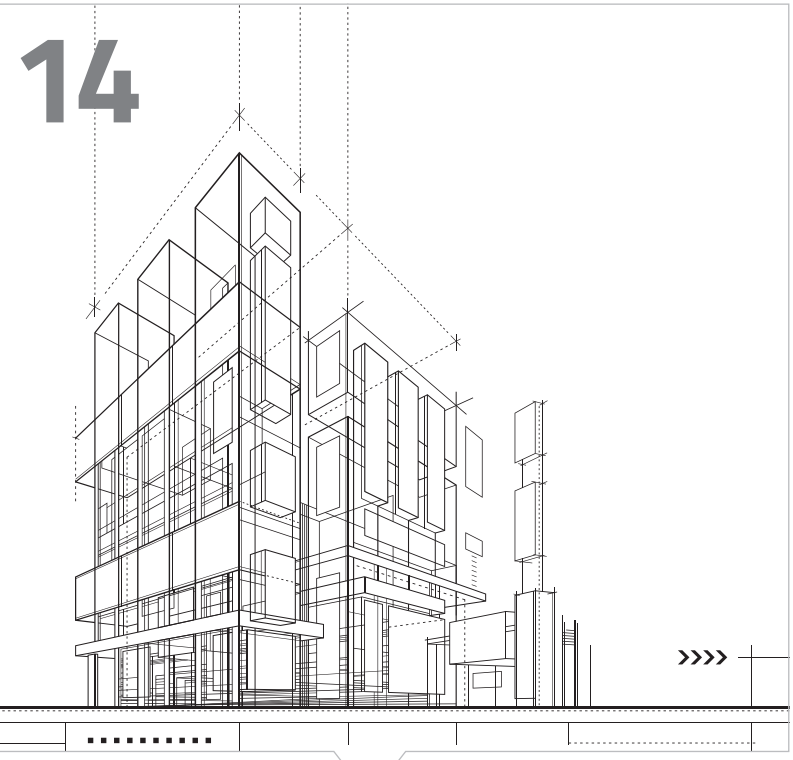
Bei dieser großen Anzahl existierender Build-Tools kann man schon einmal den Überblick verlieren. Dabei hilft das Open-Source-Projekt Gum! Es erkennt, auf welchem Werkzeug ein Projekt basiert und führt die entsprechenden Befehle aus. Einen Einblick in die Verwendung und die Vorteile von Gum gibt uns Marcus Fihlon ab Seite 42.

Kotlin wird hauptsächlich für Android-Anwendungen verwendet. Warum die JVM-Sprache jedoch auch für Backend-Applikationen bestens geeignet ist, zeigt uns Andreas Voigt in seinem Beitrag ab Seite 46. Zum Abschluss dieser Ausgabe steigen wir mit Dr. Stefan Koch in die Theorie der Performance-Analyse ein. Anhand der Erstellung und Interpretation von Lastkurven können Verbesserungen in der Performanz erreicht werden.

Wir wünschen euch viel Spaß beim Lesen!

Lisa Damerow
Redaktionsleitung Java aktuell

Inhalt



Einführung in den Maven Daemon

Mit dem Maven-Enforcer-Plug-in die Dependency-Hölle bezwingen

Kotlin für Backend-Anwendungen im Enterprise-Umfeld

Verbesserte Leistung durch Performance-Analyse

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

30. März 2023

„Character"schwäche

Nicht nur manche Java-Anwendungen haben Probleme mit falschem Encoding, manche wurden vielleicht sogar von dem – allerdings sinnvollen – Wechsel auf UTF-8 als Default ab Java 18 überrascht. Auch das OpenJDK-Projekt selbst hat so seine Schwierigkeiten damit, wie ein vor einigen Wochen erstelltes Ticket demonstriert, in dem UTF-8 als Encoding für den gesamten Source Code gefordert wird. Die Code-Basis ist besteht zwar erwartbar größtenteils aus ASCII-Zeichen, aber eben nicht nur *[1]*.

5. April 2023

Jakarta EE – es gibt kein „Full" vor Platform oder Profile

Auch das Jakarta-Projekt muss sich hin und wieder in terminologischen Diskussionen üben, diesmal ausgelöst von der Wortwahl für die neue Starter UI. Zumindest scheint jetzt ein für alle Mal geklärt zu sein, dass es zwar ein „Web Profile" und ein „Core Profile" gibt, aber kein „Full Profile", geschweige denn eine „Full Platform", sondern nur die „Platform" für das Gesamtpaket. Ob es sich rumspricht? ;-)

Kotlin 1.8.20, WASM

Im „Incremental Release" Kotlin 1.8.20 versteckt sich ein neues „Multiplatform Target" *[2]*. Allerdings ist das Feature auch vorerst als experimentell deklariert, um Feedback für die Stabilisierung und weitere Entwicklung zu bekommen. Grundsätzlich kann Kotlin-Code aber damit nun in WebAssembly für die direkte binäre Ausführung in Browsern kompiliert werden.

Virtual Threads Performance – Loom gegen Kotlin Coroutines

Für diejenigen, die sich mit den Tiefen der „Virtual Threads Performance" oder mit dem Vergleich mit Kotlin-Koroutinen beschäftigen wollen, könnte diese Analyse (beziehungsweise der gesamte Thread) in der OpenJDK Mailing-Liste interessant sein *[3]*. Wer auf ein „The Winner is" hofft, wird aber enttäuscht. Man könnte es auf gemeine Weise so zusammenfassen: „Richtig" gewählt zeigen Benchmarks nur das, was sie zeigen sollen (analog zu: Traue nur der Statistik, die du selbst gefälscht hast). Allerdings ist in diesem Fall der Weg das Ziel, beziehungsweise die Diskussion.

7. April 2023

WebAssembly: Abwarten, oder Tee trinken?

Noch mal zurück zum Thema WebAssembly, das native Anwendungen im Browser möglich machen soll und inzwischen auch – zumindest experimentell – als „containerless" Lösung für Kubernetes Anwendung findet. Allerdings waren Java und andere JVM-Sprachen bislang nicht als Quell-Sprachen im Fokus. Neben dem oben erwähnten Kotlin-Feature gibt es aber noch ein paar andere Projekte, die das ändern wollen. Die TeaVM ist als „Ahead-of-Time"-Compiler von JVM-Sprachen (allem, was zunächst zu Java-Bytecode kompiliert werden kann) nach JavaScript bekannt, bietet seit einiger Zeit aber auch die Möglichkeit, WebAssembly als Ziel zu wählen.

Allerdings ist der Support „experimentell" und das TeaVM-Projekt in erster Linie ein persönliches Projekt des Haupt-Committers, sodass sich dieser Zustand sicher nicht schnell ändern wird. Mit JWebAssembly und CheerpJ (letzteres allerdings kommerziell, es sind wohl nur die Anfänge als Open Source auf GitHub verfügbar) liegen aber noch mindestens zwei weitere Eisen im Feuer.

10. April 2023

Jakarta EE Starter UI

Der Jakarta EE Starter ist jetzt offiziell als Web-UI online *[4]*. Die relativ schlichte Website macht genau das, was sie soll: Unter Berücksichtigung der Einschränkungen und Kompatibilitäten der SE- und EE-Versionen, Profile und Runtimes eine Auswahl ermöglichen und daraus ein Projekt-„Skelett" zum Download generieren. Gradle-Fans kommen bislang nicht auf ihre Kosten, Maven ist halt der Standard für Jakarta EE, aber eine Feature-Request gibt es schon.

26. April 2023

Quarkus 3.0

Jetzt ist Quarkus 3.0 offiziell freigegeben, abgestimmt auf MicroProfile 6.0 und Jakarta EE Core Profile 10. Die Feature-Details gibt es hier: *[5]*.

Amazon, der weltgrößte JVM-Händler

Wenn die Zahlen von New Relics „2023 State of the Java Ecosystem Report" die Realität halbwegs abbilden, ist Oracle nicht mehr der führende „JVM Vendor". Oracles Anteil ist, nach den von Kunden des Monitoring-Spezialisten gesammelten Zahlen, von 75 % (2020) auf 34 % (2022) bereits dramatisch gesunken und ging jetzt noch einmal runter auf 28 %. Im Gegenzug stieg der Anteil von Amazon Coretto auf 31 % – vielleicht in etwa relational zur Größe der AWS-Cloud? Auf den weiteren Plätzen folgen ebenfalls mit Zuwächsen das Eclipse-Adoptium-Projekt (12 %) und Red Hat (10,5 %), während Azul Anteile verloren hat (auf knapp 6 %).

Eine weitere Statistik klingt dramatisch: 430 % Zuwachs für Java 17 in Produktion. Das ist trotzdem erst ein Anteil von 9 % aller Java-Versionen in Produktion. Doch das wird sich sicherlich weiter schnell verändern, da Frameworks wie Spring in den neuesten Versionen Java 17 voraussetzen. Ganz klar vorne liegt jetzt Java 11 (56 %), während Java 8 aber immer noch auf 33 % kommt. Das inzwischen nicht mal mehr mit Sicherheits-Fixes versorgte Java 7 (dessen OpenJDK-Projekt auch kürzlich nach vergeblicher Projektleiter-Suche aufgelöst wurde) ist „nur" noch mit 0,28 % vertreten – bei denen sich trotzdem die Fragen aufdrängen: In Produktion? Im Ernst?

Der ganze Report ist hier zu sehen *[6]*.

2. Mai 2023

AWS: Lambdas mit Java 17?

Passend zum Anstieg der Java-17-Nutzung unterstützen jetzt auch AWS-Lambdas die noch aktuelle LTS-Version. Früher waren Java und Lambdas nicht die besten Freunde, weil es ja „serverless" beim Hochfahren ganz schnell gehen sollte. Als Lösungsoption kamen dann Projekte wie die GraalVM mit nativer Kompilierung für den Schnellstart ins Spiel. Andere Lösungsansätze kommen von den Cloud-Anbietern selbst als Bestandteil ihrer Clouds dazu, insbesondere „Vorwärm"-Features wie SnapStart von AWS, das jetzt auch entsprechend für Java 17 genutzt werden kann *(siehe Java-Tagebuch 2/23)*.

8. Mai 2023

„Contributors Guide to the Jakarta EE 11 Galaxy"

Ein bisschen Werbung für einen guten Zweck: Reza Rahman tourt mit einem „Contributors Guide to the Jakarta EE 11 Galaxy" durch die Konferenzlandschaft und YouTube. Wer also wissen möchte, wie leicht es ist, bei Jakarta EE mitzumischen – ein kleines bisschen freie Zeit natürlich vorausgesetzt – kann sich das hier anschauen: *[7]*.

30. Mai 2023

MicroProfile und Jakarta – Klärung des Beziehungsstatus

Die zukünftige Beziehung von MicroProfile zu Jakarta EE – schon in der letzten Tagebuch-Ausgabe ein größeres Thema – wird weiterverhandelt. Zu den damals übrig gebliebenen Optionen ist eine weitere hinzugekommen. Hier noch mal die verbliebenen in Kurzform:

- Option 1: Ein MP-Release definiert eine minimale Jakarta-EE-Version (auf Plattform-Ebene).
- Option 3: Neue MP-Releases basieren immer auf den zuvor freigegebenen Jakarta-Releases.
- Und nun neu Option 5, als eine Art Kompromiss, die auch vom iJUG favorisiert wurde: MicroProfile stellt komplett auf Semantic Versioning um und übernimmt Jakarta-Releases entsprechend, das heißt, ein „Breaking Change" in Jakarta führt auch zu einer neuen Major-Version von MP.

Die Option 4 – MP geht auch organisatorisch komplett in Jakarta auf – ist nicht völlig vom Tisch, wird aber erst mal als losgelöste Diskussion für die Zukunft gesehen. In den vergangenen Tagen fand dazu eine Abstimmung in der MP-Working-Group statt. Dabei gab es ein Kopf-an-Kopf-Rennen – jede Option hatte drei Befürworter – bis als Letzte sowohl Oracle als auch die Atlanta JUG für Option 1 stimmten und die Abstimmung damit entschieden. Mal sehen, was in der Praxis daraus wird – die Option der „Minimum Jakarta-Version" hat auch ihre Tücken. Hier nochmal für alle Detailversessenen der Link zum Dokument mit der Diskussion aller Optionen *[8]*.

8. Juni 2023

JDK 21 ist in „Rampdown Phase 1"

Die nächste Long-Term-Support-Version, Java SE, JDK 21, biegt in die Zielgerade ein – mit jeder Menge Features. Einige davon sind schon seit mehreren Versionen als Previews dabei und erwartungsgemäß jetzt produktionsreif (oder zumindest als solche deklariert): *Virtual Threads, Record Patterns, Pattern Matching for Switch*. Das *Foreign Feature & Memory API* allerdings geht in die dritte Preview-Runde und das sich weiterhin in der Inkubator-Phase befindende *Vector-API* geht dort sogar in die sechste Runde. Fünf neue Previews sind auch dazugekommen: Drei kleinere Syntax-Verbesserungen aus dem „Project Amber", nämlich *String Templates* (String Interpolation statt Concatenation, aber auf sichere Weise), *Unnamed Patterns and Variables* (Verwendung des Unterstrichs „_" für nicht genutzte Variablen) und *Unnamed Classes and Instance Main Methods* (simple Java-Programme mit weniger Code schreiben). Dazu zwei aus „Project Loom" für die (massiv-)parallele Entwicklung, die den „Aufstieg" aus dem Inkubator geschafft haben: *Scoped Values* („eine bessere Alternative zu ThreadLocals zum Teilen von unveränderlichen Daten") und das API für *Structured Concurrency* (zur gemeinsamen Behandlung verteilter, aber verwandter Tasks).

Aber es gibt noch weitere „produktionsreife" Features: Etwa der *Generational ZGC*, (eine Erweiterung des Z Garbage Collectors) und die *Sequenced Collections*. *Prepare to Disallow the Dynamic Loading of Agents* ist ein neues Sicherheits-Feature. Hier sollen zunächst nur Warnungen erfolgen, wenn Agents dynamisch in die JVM geladen werden – mit dem Ziel, dies in einer späteren Version per Default zu unterbinden. Zur Beruhigung: Die Start-Optionen `-agentlib` und `-javaagent` sind hiervon explizit ausgenommen.

Referenzen:

[1] *https://bugs.openjdk.org/browse/JDK-8301971*
[2] *https://kotlinlang.org/docs/whatsnew1820.html#new-kotlin-wasm-target*
[3] *https://mail.openjdk.org/pipermail/jdk-dev/2023-April/007571.html*
[4] *https://start.jakarta.ee*
[5] *https://quarkus.io/blog/quarkus-3-0-final-released*
[6] *https://newrelic.com/resources/report/2023-state-of-the-java-ecosystem*
[7] *https://www.youtube.com/watch?v=roHMl8V01tw*
[8] *https://docs.google.com/document/d/1FFYi8IcTdyBBXu-gw_ZCH-6mqPvdrL9QPohxPCN_eRJQ*

Andreas Badelt
stellv. Leiter der DOAG Cloud Native Community
andreas.badelt@doag.org

Andreas Badelt ist seit 2001 ehrenamtlich im DOAG e.V. aktiv und hat dort inzwischen seine Heimat in der Cloud Native Community gefunden, wobei ihn das Java-Ökosystem bis heute fasziniert. Beruflich hat er von Ende des vorigen Jahrtausends an als Entwickler und später auch Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet. Seit 2016 ist er als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Markus' ECLIPSE™-Corner

Nach dem Spiel ist bekanntlich vor dem Spiel, und so arbeitet die Jakarta EE Working Group seit dem Erscheinen von Jakarta EE 10 (wir erinnern uns: die Release mit Java-11-Unterbau und neuer Core Edition) an der Nachfolgerin mit der Nummer 11. Der geneigte Leser wird sich fragen, was denn als Nächstes kommen wird: Ob es die Hinwendung zu Microservices, Cloud-nativen Anwendungen oder einem anderen Hype ist und so wagen wir einen Blick in die Glaskugel.

So richtig entschieden ist das Ganze noch nicht wirklich, auch wenn hier und da unterschiedliche Leute aus der Jakarta-Blase bereits mit ihrer Ansicht zu Wort gemeldet haben. Im März dieses Jahres beispielsweise Steve Millidge, seines Zeichens Chef des Payara-Teams, der auf seinem Blog vier Kernpunkte als gesetzt betrachtet hat:

- Mehr Leute ins Boot holen – Ok, das kann nicht schaden, ist ja mein bekanntes Credo. Ist doch aber eher eine Marketingaufgabe und weniger das Ziel einer technischen Spezifikation.
- Mehr Einheitlichkeit – Ja, es kann nicht schaden, wenn Dinge überall gleich gehen. Man fragt sich aber natürlich, wie das ohne Verlust der Rückwärtskompatibilität funktionieren soll. Dazu schweigt man sich aber eher erstmal aus. Bei Jakarta REST beispielsweise wollen wir die CDI-Integration seit Jahren für exakt den genannten Benefit und sind aus exakt dem genannten Problem noch keinen Schritt weiter. Und mal ehrlich, bis Q4/24 wird das auch nichts mehr.
- Neue Spezifikation – Aha, nun wird es interessant. Genannt wird Jakarta Config, da würde ich mich tatsächlich drauf freuen, nachdem es mit der Übernahme von Microprofile Config nicht geklappt hat. Und ein API zur technologieneutralen Separation der Persistenzschicht (namens Jakarta Data *[1]*) ist sicherlich auch kein Fehler, aber mal ehrlich, sie ist auch nicht so der große Brüller. (Der messbare Vorteil eines allgemeinen API explizit für diesen Zweck gegenüber Selbstbau erschließt sich mir derzeit noch nicht, warten wir mal auf die Version 1 dieser Spezifikation. Vielleicht gehen mir ja dann die Augen auf, wie sinnlos mein Programmiererleben ohne diese bahnbrechende Neuerung bislang war.)
- Java 21 – Ok, die Amerikaner würden sagen: „Now we're talking!" Ja, darauf freue ich mich tatsächlich schon ziemlich, endlich Records und Record Patterns, Pattern Matching for Switch und viele kleine Neuerungen verwenden zu dürfen sowie die Performancesteigerungen durch virtuelle Threads und optimiertes NIO-Subsystem zu genießen. (Wenn ihr nicht wisst, wovon ich rede, schaut gerne auf meinem YouTube-Kanal vorbei.)

Ja… und sonst? Das war's? Hm, so richtig vom Hocker gerissen bin ich jetzt noch nicht. Auch der offizielle Jakarta EE 11 Release Plan *[2]* enthält kaum Belastbares, außer klare Hinweise auf seine Vorläufigkeit und ein paar Tippfehler. Nun, vielleicht ist es einfach noch zu früh, um zu wissen, was man bis Ende des Jahres abliefern will – es ist ja schließlich erst Juni. Da ist doch noch massig Zeit und die beteiligten Unternehmen hauen mit Sicherheit dieses Mal kräftig rein… oder?

Referenzen:

[1] Spezifikation „Jakarta Data" *https://jakarta.ee/specifications/data/*

[2] Jakarta EE 11 Release Plan *https://github.com/jakartaee/jakartaee-platform/blob/gh-pages/jakartaee11/JakartaEE11ReleasePlan.md*

Markus Karg

markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

Bits, Kids und Karusselle: JavaLand4Kids 2023

Fin Labusch

*Am Vortag der diesjährigen JavaLand-Konferenz im Phantasialand fand nach langer Corona-Pause wieder eine JavaLand4Kids statt. Die Veranstaltungsreihe richtet sich an Kinder und Jugendliche im Alter zwischen 8 und 16 Jahren und wird von engagierten IT-Expert*innen durchgeführt. Ziel ist es, den jungen Teilnehmer*innen spielerisch und praxisnah einen Einblick in die Welt der Informatik zu geben. Das Angebot der ungefähr dreistündigen Workshops bietet für Einsteiger*innen, aber auch für erfahrene Jung-Entwickler*innen, spannende Themen.*

In diesem Artikel stelle ich die sechs durchgeführten Workshops kurz vor. Zusätzlich erfährst du, wie eine Konferenz für Kinder abläuft, wie du dich einbringen kannst und welche Vorteile damit verbunden sind. Zum Abschluss werfen wir noch einen kurzen Blick auf die JavaLand4Kidadults, die erstmals von mir bei der diesjährigen JavaLand angeboten wurde. Denn auch große Kinder spielen immer noch gerne mit Lego, Calliope mini oder den quirligen Ozobots.

Willkommen im Phantasialand

Montagmorgen, kurz vor neun Uhr: Neugierige Kinderaugen (okay, die Jugendlichen versuchen es mit Coolness zu vertuschen) wurden von zirka 20 nicht weniger aufgeregten Mentorinnen und Mentoren begrüßt und auf insgesamt sechs Workshops verteilt.

Für die zirka 20 Schüler*innen der Hans-Christian-Andersen-Grundschule aus Brühl wurden die Workshops

- Musik programmieren mit SonicPI
- Lego WeDo

und für die Schüler*innen des Max-Ernst-Gymnasiums aus Brühl

- Einführung in die Programmierung mit Python
- CandyCrush Clone
- Programmieren mit MicroBits
- Graphen und Neo4J

angeboten. Während bei den Workshops für die Grundschüler*innen

viel Wert auf einen spielerischen Zugang gelegt wird, bot das Angebot für die Älteren sowohl für Einsteiger*innen als auch für Profis eine gute Auswahl.

In dem von Oliver Hook organisierten Workshop „SonicPi" erlernten die teilnehmenden Grundschüler*innen, wie sich Musik programmieren lässt. Der kreative Zugang über Töne, Noten und Samples erleichtert dabei den Einstieg in Programmierkonzepte wie Anweisungen („Spiel diesen Ton", „Leg ein Hall-Effekt auf dieses Sample"), Schleifen („Wiederhole den Refrain") und Bedingungen. In einer Dauerschleife gibt das kostenlose Programm SonicPi *[2]* die Musik wieder, sodass Änderungen am Code sofort zu hören sind. Das Beispielprogramm aus *Listing 1* zeigt eine Endlosschleife, in der ein Synth-Sample (`use_synth`) und eine von vier Noten gespielt wird (`tick`). Mit jedem Schleifendurchlauf wird die nächste Note gespielt und danach wieder von vorne begonnen (`ring`). Besonders gut gefallen hat den Schüler*innen, dass sie ihre eigenen Instrumente digitalisieren und in die Songs integrieren konnten.

```
live_loop :bassline do
  use_synth :tb303
  notes = [:C2, :C2, :Eb2, :Bb2].ring.tick
  play notes, release: 0.25, cutoff: rrand(60, 130)
  sleep 0.25
end
```

Listing 1: Eine Bassline in SonicPi

Deutlich hörbaren Spaß hatten die Kinder im zweiten Workshop, bei dem aus Legosteinen kleine Roboter und andere motorisierte Modelle gebaut und mit einer einfachen grafischen Programmiersprache in Bewegung versetzt wurden. Am Beispiel eines Rennwagens wurde so lange am Programm und Modell getüftelt, bis die Teststrecke am Ende in nur in einem Drittel der Zeit des ersten Entwurfs abgefahren wurde. Dies wurde nicht nur durch die Optimierung des Programms, sondern auch durch technische Anpassungen am Modell (größere Räder, bessere Übersetzung) erreicht. Bei diesem Workshop zeigte sich erneut, dass in jungen Jahren das Interesse an informatischen Themen bei beiden Geschlechtern vorhanden ist, aber erste Tendenzen für eine Maskulinisierung des Themas zu erkennen sind. Da gab es zum Beispiel Ken (Name geändert), der schon bei der Begrüßung fragte, ob er nicht auch an einem Workshop der Großen teilnehmen könne. Er arbeitete mit Ada (Name geändert) zusammen und riss gleich das Tablet an sich. Ada ließ sich jedoch nicht die Butter vom Brot nehmen und wusste sich zur Wehr zu setzen. Am Ende waren die beiden eines der harmonischsten Teams in der Gruppe.

Tobse Fritz hat sich bei der Vorbereitung des Workshops „Candy Crush Clone" viel Mühe gemacht und so konnten die Teilnehmer*innen hier schon nach kurzer Zeit erste Bonbons abräumen. Die jugendlichen Spieleentwickler*innen nutzten IntelliJ als Entwicklungsumgebung, die gegenüber dem in der Schule eingesetzten BlueJ besser angenommen wurde. Das Spiel wurde mit vorgefertigten Tests testgetrieben mit Kotlin als Programmiersprache entwickelt. Mit jedem grünen Test bekam das Spiel neue Funktionen, die sofort ausprobiert werden konnten. Das fertige Spiel wurde mit eigenen Grafiken und Sounds aufgewertet und anschließend in der Gruppe präsentiert. Zusätzlich wurde das Spiel auf der Source-Code-Hosting-Plattform GitHub als interaktive Web-Applikation veröffentlicht und steht für Interessierte zur Verfügung *[3]*.

Wie man mit Variablen, Listen, Schleifen und Kontrollstrukturen richtig umgeht, konnten Einsteiger*innen im Kurs von Kevin Wit-

Abbildung 1: Präsentation des Baus und der Programmierung eines Schleusentors mit LEGO Education WeDo 2.0 (Foto: Fin Labusch)

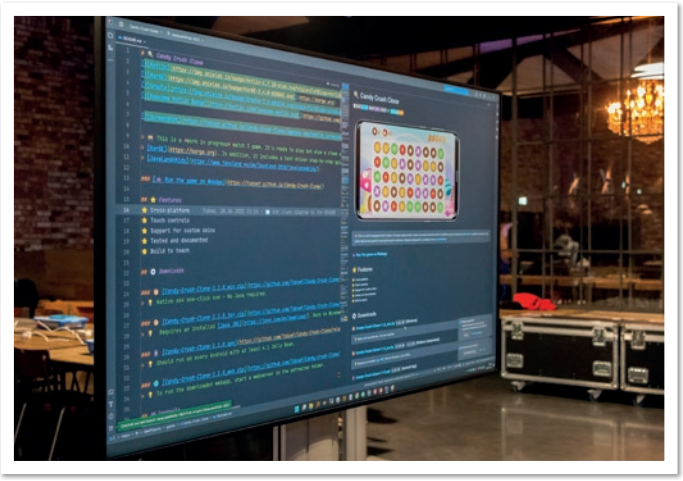


Abbildung 2: Candy Crush Clone (Foto: Thomas Ruhroth)

tek „Einführung in die Programmierung mit Python lernen". Mit der webbasierten Entwicklungsumgebung Replit *[4]* haben die Schüler*innen zunächst ein interaktives Quizspiel programmiert und anschließend mithilfe der Turtle Graphics Library von Python *[5]* ein paar Bilder auf den Monitor gezaubert. Besonders letzteres wurde von den Schüler*innen mit Begeisterung aufgenommen und auch hier zeigt sich, dass unmittelbares visuelles Feedback die Workshop-Teilnehmer*innen motiviert.

Dieses macht sich auch der „Hardware-nahe" Workshop „Programmieren mit micro:bits", organisiert von Christian Rucinksi, zunutze. Nach einer kurzen Einführung erschienen erste Smileys auf dem einfachen 5x5-Punkte-LED-Display und am Ende waren die Teilnehmer*innen in der Lage, ein Spiel zu programmieren, das per Funk auch Nachrichten zwischen den micro:bits austauschen konnte. Das Spiel wurde im Pair-Programming entwickelt, bei dem der/die jeweils andere Partner*in neu hinzugefügte Funktionalität geprüft und am Gerät selbst getestet hat.

Für bereits erfahrene Programmierer*innen bot der Workshop von Iryna Feuerstein einen spannenden Einblick in Java und die grafische Datenbank Neo4J. Nach einer Einführung in die Graphentheorie und die Abfragesprache Cypher nutzten die Schüler*innen Daten aus OpenStreetMap über ihren Heimatort Brühl, um verschiedene Fragestellungen zu beantworten, etwa, wo sich das nächste Café befindet oder wie lang die längste Tram-Strecke ist. Erwähnenswert ist das 50:50-Verhältnis zwischen Mädchen und Jungen, das sonst nur bei Workshops für die Jüngeren anzutreffen ist. Die persönliche Erfahrung des Autors zeigt, dass bis zu einem Alter von ungefähr zehn Jahren das Interesse an informatischen Themen gleich ist und danach bei den Mädchen deutlich abzunehmen scheint.

Wie spannend und motivierend alle Workshops waren, konnte man unter anderem daran erkennen, dass die eigentlich geplante gemeinsame Präsentation der Ergebnisse kurzfristig gestrichen wurde – die Schüler*innen wollten unbedingt ihre Projekte fertig stellen.

Nach einem leckeren Mittagessen ging es nach einer letzten Workshop-Einheit, der Verteilung der Urkunden und einem Abschlussfoto *(siehe Abbildung 4)* in den Vergnügungspark. Die Schüler*innen beendeten einen spannenden Tag mit einer Fahrt im Kettenkarussell und einer virtuellen Mäusejagd.

Warum du bei der nächsten JavaLand4Kids mitmachen solltest

Du bekommst einen offiziellen JavaLand-Speaker-Hoodie! Ja, das stimmt, aber das sollte nicht wirklich der Grund sein. Wenn du aber, wie wir, die Mentoren und Mentorinnen der JavaLand4Kids, das gleiche Interesse daran hast, Kindern eine Einführung in die

Abbildung 3: Schüler des Max-Ernst-Gymnasiums programmieren ein Spiel für den micro:bit (Foto: JavaLand4Kids)

Abbildung 4: Abschlussfoto mit allen Teilnehmer/innen der JavaLand4Kids (Foto: JavaLand4Kids)

Informatik zu geben und zu zeigen, dass Wissen auch spielerisch vermittelt werden kann, statt in Form von eintönigem Frontalunterricht, dann bist du bei uns genau richtig. Diese Begeisterung teilen auch bekannte JavaLand-Speaker wie Christina Hartmann, Falk Sippach, Kevin Wittek, Lisa Rosenberg oder Thomas Much, um nur ein paar zu nennen. Es wird auch viel Zeit und Gelegenheiten geben, dich mit den anderen Speakern auszutauschen. Die oben erwähnten Workshops werden nicht von einer einzelnen Person betreut, sondern dir stehen zwei bis vier Mentor*innen zur Seite, die dich bei deinem Workshop unterstützen. Wenn du selbst erst einmal nur als Mentor*in unterstützen möchtest, statt selbst einen Workshop zu organisieren, dann bist du natürlich genauso willkommen. Falls dir die JavaLand zu groß ist, kein Problem, schau dich in deiner Gegend einmal um. Es gibt diverse Organisationen, die Workshops für Kinder anbieten. Besuche einmal MINTvernetzt *[6]* oder sieh dir das Angebot zur Code Week *[7]* an, eine zweiwöchige europaweite Veranstaltungsreihe, die jedes Jahr im Oktober stattfindet und bei der praktisch jede MINT-affine Organisation teilnimmt.

Zutaten für einen erfolgreichen Workshop

Je nach Altersgruppen gibt es unterschiedliche Rahmenbedingungen für einen erfolgreichen Workshop. Die von mir durchgeführten Workshops („Roboter bauen und programmieren mit LEGO WeDo 2.0“ und „Zauberschule Informatik“) richten sich an das jüngere Publikum, also Kinder im Alter zwischen 8 und 12 Jahren. Diese Kinder sind sehr leicht zu begeistern, da sie offen und interessiert an allem sind. Allerdings sind sie auch genauso leicht zu frustrieren. Besonders wichtig ist hier, dass sich die Kinder auf die Aufgabe konzentrieren können und die Technik im Hintergrund verschwindet, das heißt, Technik und Software müssen funktionieren. Bei der „Zauberschule“ ist dies kein Problem, denn dieser Workshop kommt gänzlich ohne digitale Technik aus. Klemmbausteine sind bei Kindern sehr beliebt, das Education-Programm von LEGO ist hier mehr als ausgereift. Vergleichbare Produkte von alternativen Klemmbausteinherstellern scheitern hier leider allzu häufig an der Software.

Jugendliche Teinehmende verzeihen schon mal Probleme mit der Technik, aber auch hier ist funktionierende Technik von Vorteil (wie beispielsweise beim Calliope mini *[8]* oder dem micro:bit *[9]*). Die Herausforderung liegt hier eher darin, Teilnehmer*innen mit wenig oder keiner Erfahrung und jene mit viel Erfahrung unter einen Hut zu bringen. Hier empfiehlt sich ein größerer Teilnehmer-Mentoren-Schlüssel, idealweise 2:1.

Wie oben schon erwähnt, nimmt die Zahl der Teilnehmer*innen mit dem Alter deutlich ab. Wie man auch Mädchen für Workshops begeistern kann, hat Christina Hartmann mit einem Beitrag auf der JavaLand vorgestellt. Das Fazit ist jedoch, dass es hierfür kein allgemeines Erfolgsrezept gibt. Probiert es einfach aus!

Und die großen Kinder?

JavaLand4Kidadults

Im Anschluss an die JavaLand4Kids und am Mittwoch während der Konferenz fand die JavaLand4Kidadults (JavaLand für große Kinder) statt. Hier konnten Interessierte die mitgebrachten MINT-Produkte selbst ausprobieren. Neben den bereits bei den Kindern verwen-

deten Produkten, wurden hier weitere präsentiert, auch für noch jüngere Kinder wie etwa dem Bee- und BlueBot, den immer wieder beliebten Ozobots, aber auch den Calliope, Mindstorms und alternative Klemmbausteinsets.

In gemütlicher Runde haben die Erwachsenen Musik programmiert, mit Klemmbausteinen gebaut oder Fährten für die Ozobots gemalt. Dabei ergaben sich hoch-interessante Gespräche mit den Teilnehmer*innen der Konferenz und wir konnten den/die ein oder andere/n Mentor*in gewinnen. Das Ergebnis sind spannende Workshops auf dem kommenden JavaForumNord und den Ablegern der Kids4IT in Darmstadt und Nürnberg.

Ich bedanke mich bei den Workshop-Organisator*innen und Mentor*innen für ihre Beiträge und Rückmeldungen zu diesem Artikel sowie der DOAG für die Ausgestaltung der JavaLand4Kids. Ein besonderer Dank geht an Kathrin Weihe vom RoboLab der Hamburger Öffentlichen Bücherhallen *[10]* für die Leihgabe diverser Roboter.

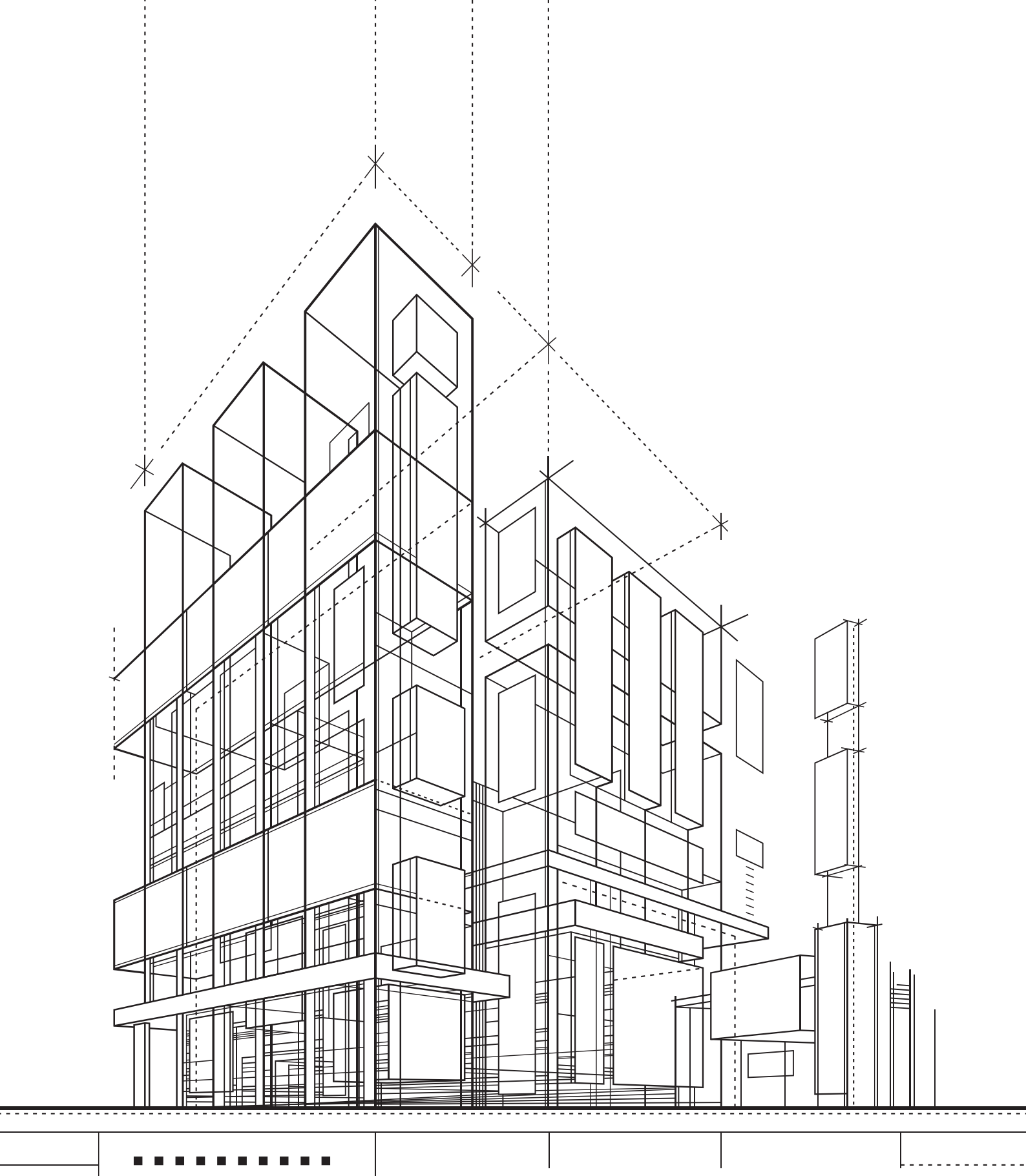
Abbildung 5: Erwachsene erkunden MINT-Produkte (Foto: Fin Labusch)

Referenzen:

[1] Kids4IT: *https://kids4it.de*
[2] Sonic Pi: *https://sonic-pi.net/*
[3] *https://github.com/TobseF/Candy-Crush-Clone*
[4] Replit: *https://replit.com/@kiview/javaland4kids-python*
[5] Python Turtle Graphics: *https://docs.python.org/3/library/turtle.html*
[6] MINT-vernetzt: *https://mint-vernetzt.de/*
[7] Code Week Germany: *https://www.codeweek.de/*
[8] Calliope mini: *https://calliope.cc/*
[9] micro:bit: *https://microbit.org/*
[10] RoboLab Hamburg: *https://robolab.hamburg/*

Fin Labusch
java@labusch.de
https://twitter.com/FutzlFin

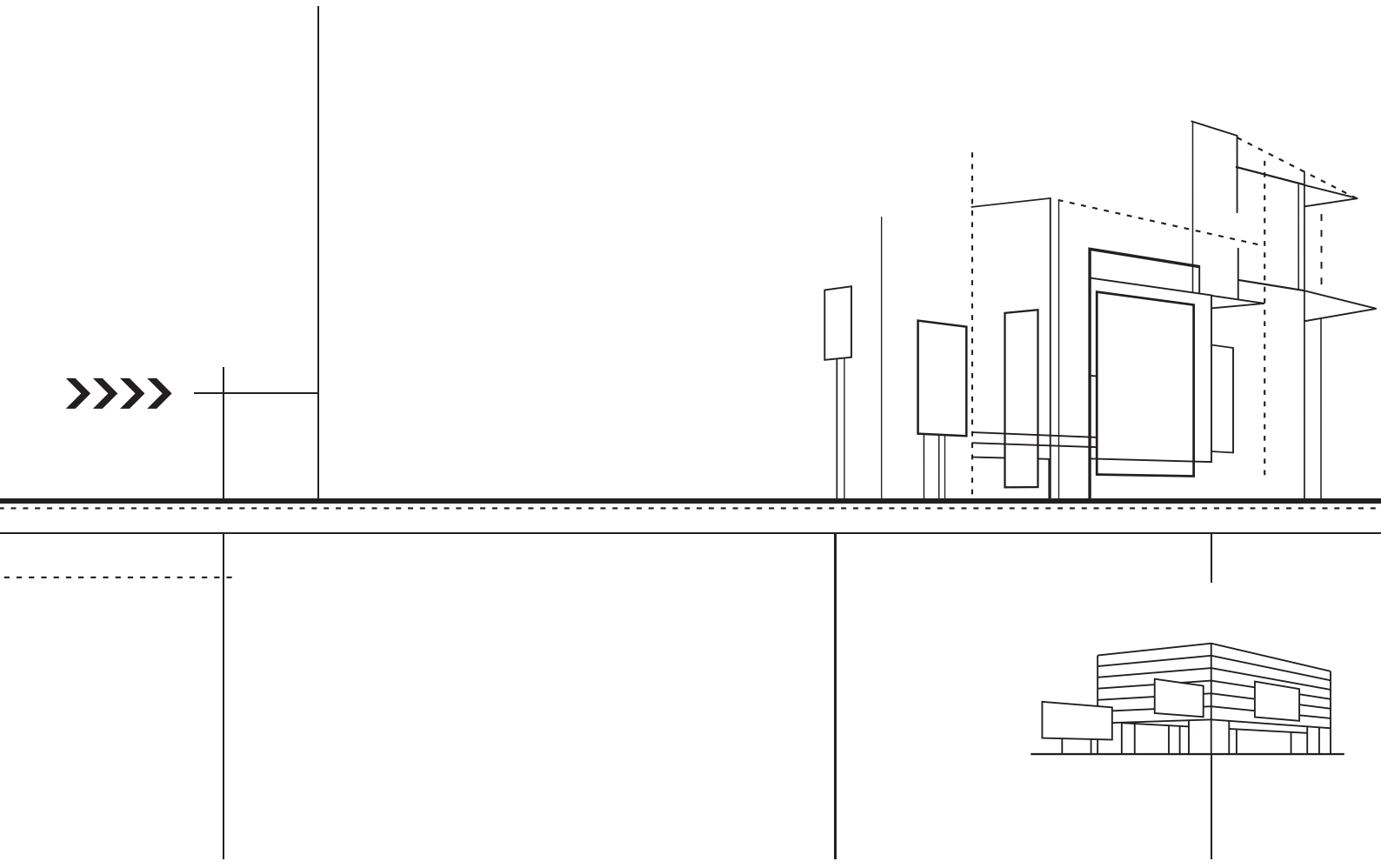
Nach seinem Informatikstudium an der Universität Hamburg war Fin Labusch für diverse IT-Beratungsfirmen tätig und ist seit über zehn Jahren freiberuflich als Fullstack-Softwareentwickler unterwegs. Mit der Lektüre von „Teach Yourself Java in 21 Days" begann 1996 sein Herz für Java zu schlagen. In seiner Freizeit veranstaltet er regelmäßig Workshops für Kinder und ist Mitgesellschafter der Kids4IT e.V.



Maven Daemon

Peter Palaga, Red Hat

Maven ist das wohl bekannteste Build-Tool für Java und verwandte Sprachen, die auf die Java Virtual Machine (JVM) ausgerichtet sind. Sein deklarativer Ansatz zur Definition von Builds gehört zu seinen Hauptstärken. Grundlegende Kenntnisse der Maven-Prinzipien genügen, um praktisch jedes fremde Projekt zu verstehen. Es ist gut etabliert, ziemlich stabil und hat ein reichhaltiges Ökosystem von Plug-ins. Viele Konventionen von Maven haben sich als De-facto-Standard für die Veröffentlichung und Nutzung von Bibliotheken in der JVM-Welt durchgesetzt. Maven Central und seine Artefakt-Metadaten-formate werden in der gesamten JVM-Welt verwendet und respektiert.



Dennoch hat Maven einige bekannte Schattenseiten:

- pom.xml-Dateien werden oft als zu ausführlich kritisiert.
- Die Leistung von Maven ist nicht auf dem Niveau von Gradle, insbesondere in inkrementellen und iterativen Szenarien.
- Parallele Maven-Builds sind schwer zu benutzen, da die Konsolenausgabe ein Mischmasch aus zufällig gemischten Zeilen aus verschiedenen Modul-Builds ist.

Werfen wir gemeinsam einen Blick darauf, wie der Maven Daemon die letzten beiden Punkte angeht.

Grundlagen der JVM-Performanz

Die meisten von uns Java-Entwicklern wissen, dass Java zwar schnell ist, aber eine gewisse Aufwärmzeit benötigt, um sein volles Potential zu entfalten. Das liegt zum einen daran, dass das Booten einer JVM-Anwendung etwas Zeit kostet und zum anderen, an der Art und Weise, wie die Just-in-Time-Kompilierung (JIT-Kompilierung) in der JVM funktioniert.

Maven Startkosten

Werfen wir zunächst einen Blick auf die Startkosten. Im Falle von Maven können wir diese durch ein einfaches Experiment quantifizieren. Im Maven-Quellcode finden wir die Stelle, an der Maven seine eigene Build-Dauer-Messung startet.

Das ist die Basis für die bekannte Meldung `BUILD SUCCESS/Total time`: am Ende eines jeden Maven-Builds. Wir können davon ausgehen, dass dies der Zeitpunkt ist, an dem Maven beginnt, etwas Sinnvolles zu tun. Alles, was davor passiert, kann als Startkosten betrachtet werden.

Wir platzieren ein `System.out.println("Ende von Maven init: " + System.currentTimeMillis())` an dieser Stelle, rufen einfach `mvn install` auf und stellen ihm eine Zeitausgabe in Millisekunden voran, wie in *Listing 1* gezeigt.

```
$ echo $(($(date +%s%N)/1000000)) && mvn install
```

Listing 1: Zeitstempel vom Beginn des Maven Start-Prozesses ausgeben

Es werden zwei Zahlen ausgegeben. Auf unserem Lenovo P1 Gen 3 Testlaptop sind das 1634117024986 und 1634117025765. Subtrahiert man sie, erhält man ein Ergebnis von 779 Millisekunden. Das ist die Zeit, die benötigt wird, um die JVM zu starten und alle Klassen zu laden, die notwendig sind, damit Maven sinnvoll genutzt werden kann. Eine Dreiviertelsekunde. Das ist eine ganze Menge Zeit, nicht wahr?

Just-in-Time-Kompilationskosten

Java-Quelldateien werden zum Zeitpunkt der Erstellung in Bytecode kompiliert.

Das ist jedoch nicht die Art von Code, die auf einem physischen Prozessor ausgeführt werden kann. Der Bytecode muss noch in Anweisungen übersetzt werden, die für einen bestimmten Prozessor verständlich sind.

Dies geschieht zunächst „on the fly", während die JVM die Methoden der Anwendung ausführt. Jede Methode wird zunächst in diesem einfachen Interpreter-Modus ausgeführt. Erst, wenn eine Methode oft genug ausgeführt wird („warm" wird), beschließt die JVM, sie mit dem Quick-Compiler zu kompilieren.

Sobald die betreffende Methode „heiß" wird, das heißt, sehr oft ausgeführt wird, kompiliert sie die JVM mit dem „optimierenden" Compiler. Offensichtlich erfordert die JIT-Kompilierung selbst einige Rechenressourcen. Um die Anwendung schneller zu machen, wird sie also zunächst verlangsamt, indem CPU-Zeit für die Kompilierung und die Optimierungen aufgewendet wird.

Welche Auswirkungen hat dies auf Java-Build-Tools?

Die Auswirkungen sind beträchtlich, denn Builds sind recht kurzlebige Prozesse. Die Boot-Kosten von 0,78 Sekunden können durchaus einen erheblichen Anteil an der Gesamtdauer des Builds ausmachen.

Was die Auswirkungen der JIT-Kompilierung angeht, so läuft der Code eines kurzlebigen Builds entweder im langsamen Interpreter-Modus oder ihr Overhead kann sich kaum amortisieren, bevor der Prozess beendet ist, falls die JIT-Kompilierung überhaupt einsetzt.

Ein Hintergrundprozess (Daemon), der die einzelnen Builds überlebt, ist eine offensichtliche Lösung für diese beiden Probleme. Dieser Hintergrundprozess hostet dann den eigentlichen Build und das Kommandozeilenprogramm ist nur ein Thin-Client für ihn.

Während Gradle diese Funktion schon lange hat, ist sie in Maven ziemlich neu. Lasst uns dies ausprobieren.

Installation

Der Maven Daemon muss separat von Maven installiert werden. Es ist gut zu wissen, dass die Installationen von Maven Daemon und Maven in keiner Weise interagieren. Man kann beide oder nur einen von beiden auf seinem Rechner haben.

Der Maven Daemon kann auf verschiedene Arten installiert werden. Die Verwendung einiger der folgenden Paketmanager ist meist einfacher als das Herunterladen der ZIP-Datei und die manuelle Installation *[1]*:

- SDKMAN! *[2]*
  `sdk install mvnd`
- Homebrew *[3]*
  `brew install mvndaemon/homebrew-mvnd/mvnd`
- MacPorts *[4]*
  `port install mvnd`
- Chocolatey *[5]*
  `choco install mvndaemon`
- Scoop *[6]*
  `scoop install mvndaemon`
- asdf *[7]*
  `asdf plugin-add mvnd && asdf install mvnd latest`

Teile von Maven Daemon

Wie oben skizziert, benötigt man zwei Dinge, um das Konzept des Daemons umzusetzen: den Kommandozeilen-Client und den Hintergrundprozess.

mvnd – der Client

Im Falle von Maven ist der Client ein brandneues Programm namens `mvnd` (oder `mvnd.exe` unter Windows), das kaum etwas mit dem traditionellen `mvn`-Skript teilt.

Obwohl es in Java geschrieben ist, wird es mit GraalVM zu einer nativen ausführbaren Datei kompiliert.

Um genau zu sein, ist dies nur für diejenigen Plattformen der Fall, die von GraalVM unterstützt *[8]* werden. Für die restlichen Plattformen gibt es ein Shell-Skript (`mvnd.sh`), das `mvnd-client.jar` umhüllt.

Doch warum haben wir uns die Mühe gemacht, den Client auf GraalVM zu portieren? Abgesehen davon, dass es Spaß gemacht hat, hat es auch einige Leistungsvorteile gebracht. Die nativen GraalVM-Programme sind bekannt für schnelle Startzeiten und geringen Speicherbedarf. Genau das benötigen wir. Außerdem konnten wir durch die Verwendung einer einzigen ausführbaren Client-Datei, einen weiteren verlangsamenden Prozess in der Ausführungskette eliminieren: das Shell-Skript.

Nun haben wir den `mvnd`-Befehlszeilen-Client, der schnell startet. Doch was macht er eigentlich?

Zuerst prüft er, ob ein Daemon-Prozess läuft und ob er frei ist, um Build-Anfragen zu akzeptieren. Dazu liest er die Daemon-Registry, die in `~/.m2/mvnd/registry/<version>/registry.bin` gespeichert ist. Wenn es keinen freien Daemon gibt, wird ein neuer vom Client gestartet. Dann verbindet sich der Client über einen Netzwerk-Socket mit dem Daemon und übergibt diesem die Befehlszeilenparameter zusammen mit der Umgebung der aktuellen Shell.

Der Daemon nimmt die Anfrage an und beginnt mit der Ausführung. Während des Builds sendet er Benachrichtigungen über den Fortschritt an den Client zurück.

Der Client zeigt den Stand des Builds in seiner textbasierten Benutzeroberfläche an *(siehe Abbildung 1)*.

Daemon

Der Daemon-Prozess ist eine traditionelle JVM. Er bettet eine bestimmte Maven-Version ein. Die Maven-Version kann nur durch den Wechsel zu einer anderen `mvnd`-Version geändert werden.

Der Daemon liest `pom.xml`-Dateien, lädt Plug-ins und Abhängigkeiten herunter und all die anderen Dinge, die Maven normalerweise erledigt. Der Hauptunterschied zu Maven ist, dass der Daemon sich nicht beendet, wenn der Build beendet wird. Er bleibt am Leben, lauscht an seinem Socket und wartet auf neue Anfragen von Clients.

Ein weiterer Unterschied zum Standard-Maven ist, dass der Maven Daemon einen Cache von Classloadern hält, die Referenzen auf Maven-Plug-in-Klassen enthalten. Wenn man also im Laufe des Tages immer wieder das gleiche Projekt mit den gleichen Plug-

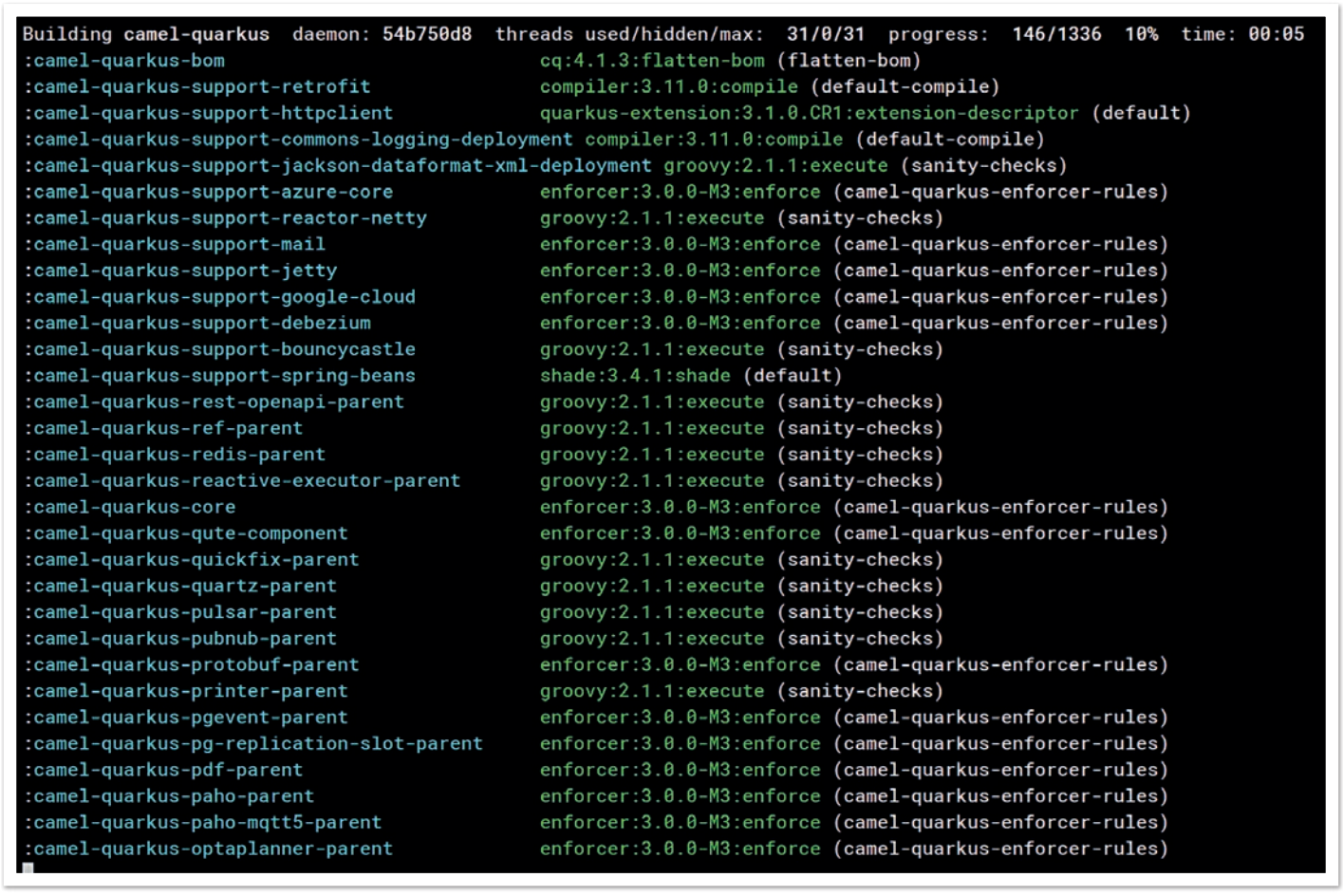


Abbildung 1

```
$ mvnd --status
   ID  PID                Address  Status  RSS  Last activity      Java home
5a12   3434  inet:/127.0.0.1:46675  Idle   721m  2023-05-21T20:12:16.905  ~/java/17.0.5-tem
56fd   3181  inet:/127.0.0.1:34947  Busy   7g    2023-05-21T20:11:57.953  ~/java/17.0.5-tem
```

Listing 2: Status der laufenden Daemons

ins kompiliert, sind die Chancen hoch, dass der optimierte Code ausgeführt wird, der vom JIT-Compiler der zweiten Ebene erzeugt wurde.

Natürlich wird der Daemon zu einem bestimmten Zeitpunkt beendet. Der genaue Zeitpunkt hängt davon ab, ob der betreffende Daemon der Einzige ist, der läuft. In einem solchen Fall beendet er sich nach drei Stunden Leerlaufzeit. Dieser Wert ist konfigurierbar – siehe Abschnitt „Konfiguration".

Wenn andere Daemons laufen, die länger im Leerlauf waren als der aktuelle Daemon, dann fährt sich der aktuelle Daemon viel schneller herunter – der Standardwert ist 10 Sekunden und ist ebenfalls konfigurierbar.

Der Status der laufenden Daemons kann mit dem Befehl `mvnd --status` eingesehen werden. In *Listing 2* ist eine Beispielausgabe zu sehen.

Dabei ist zu beachten, dass in der RSS-Spalte der vom jeweiligen Daemon-Prozess belegte Speicherplatz angezeigt wird.

Alle laufenden Daemons können gestoppt werden durch den Aufruf von `$ mvnd --stop`.

Standardmäßig parallel

Im Gegensatz zum Standard-Maven baut `mvnd` Multimodul-Projekte parallel.

Der Defaultwert für die Anzahl von Build-Threads ist durch den Ausdruck `Runtime.getRuntime().availableProcessors() - 1` gegeben.

```
  A
 / \
B   C
```

Listing 3: Die unteren Module (B, C) hängen vom oberen Modul (A) ab.

Eine weitere Bedingung ist, dass die Abhängigkeitsbeziehungen zwischen den Modulen im aktuellen Quellbaum einen parallelen Build tatsächlich erlauben müssen.

Wenn es zum Beispiel drei Module A, B und C zu bauen gibt und die Abhängigkeiten wie im *Listing 3* gezeigt aussehen, dann wird zuerst das Modul A gebaut. Danach können die Module B und C parallel gebaut werden.

Diese Art der parallelen Ausführung bringt erhebliche Geschwindigkeitssteigerungen in „breiten" Modulgraphen, in denen es viele Geschwister-Module gibt, die nur wenige gemeinsame Abhängigkeiten haben.

Smarte Builder

Standard-Maven verfügt seit der Version 3.2.1 über wählbare Builder. Dies sind Strategien für die Planung und Kompilierung von Modulen. Standardmäßig bietet Maven zwei Implementierungen an:

- singlethreaded (Default)
- multithreaded – verwendet mit der `-T/--threads`-Kommandozeilenoption

Der Maven Daemon verwendet einen dritten Builder namens „Takari Smart Builder" *[9]*.

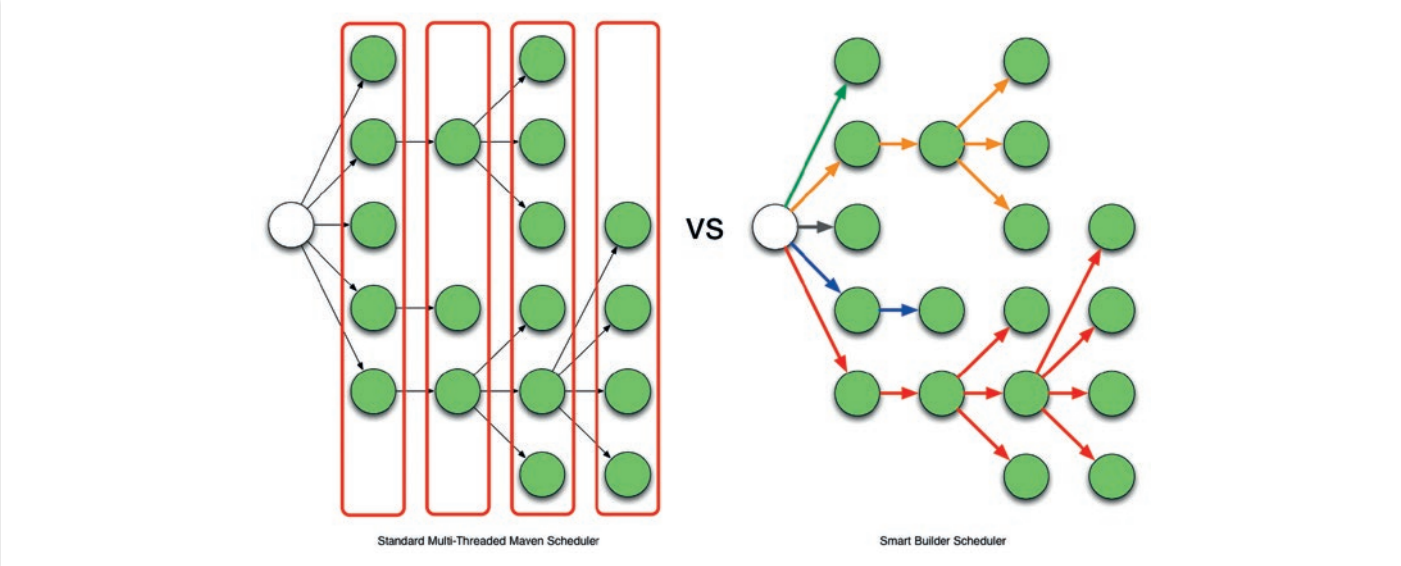


Abbildung 2: Multi-threaded builder vs. Smart builder

Lauf-Nr.	Kommando*	Dauer (Min:Sek)	Beschleunigung
	mvn clean install -Dquickly	2:42	1x (Baseline)
1	mvnd clean install -Dquickly	0:56	2.9x
2	mvnd clean install -Dquickly	0:48	3.4x
3	mvnd clean install -Dquickly	0:46	3.5x

Tabelle 1

Seine Autoren charakterisieren ihn im „Takari Extensions for Apache Maven" Buch *[10]*. Der Hauptunterschied zwischen dem Standard-Multi-Thread-Scheduler in Maven und dem Takari Smart Builder ist in *Abbildung 2* dargestellt.

Der Standard-Multi-Thread-Scheduler verwendet einen recht naiven und einfachen Ansatz zur Verwendung von Informationen über die Abhängigkeitstiefe im Projekt. Er baut alles in einer bestimmten Abhängigkeitstiefe, bevor er mit der nächsten Ebene fortfährt.

Der Takari Smart Builder verwendet einen fortschrittlicheren Ansatz mit Informationen über den Abhängigkeitspfad. Projekte werden aggressiv entlang eines Abhängigkeitspfades in topologischer Reihenfolge gebaut, wenn die vorgelagerten Abhängigkeiten erfüllt sind.

Benchmarks

Camel Quarkus *[11]* ist ein Projekt, bei dem der Geschwindigkeitsgewinn von `mvnd` besonders gut sichtbar ist: Es hat 1336 Maven-Module und sein Modulgraph ist ziemlich flach und breit.

Die CPU der Testmaschine ist ein AMD Ryzen 9 5950X mit 16 Kernen und 32 virtuellen Threads.

Die Build-Zeiten sind in *Tabelle 1* zu sehen.

Der Parameter `-Dquickly` deaktiviert Plug-ins, die nicht essenziell für einen Build sind, der zuvor alle Checks auf der CI bestanden hat, wie beispielsweise Tests, Quelltextformatierung, Enforcer, etc. Für weitere Details darüber, was `-Dquickly` genau macht, empfiehlt sich ein Blick auf dieses Video *[12]* oder diese Folien *[13]*.

Die Auswirkung des parallelen Builds kann man sehr gut sehen, wenn man die Baseline von `mvn` mit dem ersten (auf einer kalten JVM) `mvnd`-Lauf vergleicht. Der `mvnd`-Build ist dank der parallelen Build-Ausführung 2,9-mal schneller.

Die Auswirkungen des Nicht-Neustarts der aufgewärmten JVM lassen sich beim Vergleich der nachfolgenden `mvnd`-Läufe beobachten: Der zweite `mvnd`-Lauf ist 8 Sekunden schneller als der erste und der dritte sogar noch 2 Sekunden schneller. Diese allmähliche Beschleunigung wird durch die Tatsache verursacht, dass mit jeder Iteration weniger Zeit für die JIT-Kompilierung aufgewendet wird und der bereits kompilierte Code schneller läuft.

Einzelne Modul-Builds

Wir haben den Geschwindigkeitsgewinn von `mvnd` für große Multimodul-Builds demonstriert. Doch es gibt auch Entwickler, die überhaupt keine großen Multimodul-Projekte bauen. Es gibt auch kleine und einzelne Modulprojekte. Oder man kann ein einzelnes Modul innerhalb einer Hierarchie bauen. Gibt es in diesem Fall irgendwelche Geschwindigkeitsvorteile?

Schauen wir uns ein Beispiel an: Gizmo *[14]* ist ein Einzelmodulprojekt mit 50 Java-Klassen und 40 Test-Klassen.

Die parallele Ausführung bringt hier keinen Gewinn, da wir nur ein einzelnes Modul bauen. Alles, was wir sehen können, sind die Vorteile der Wiederverwendung der aufgewärmten Daemon-JVM *(siehe Tabelle 2)*.

Wir stellen fest, dass der erste `mvnd`-Lauf langsamer ist als die `mvn`-Baseline.

Dies kann durch den Overhead erklärt werden, der durch das Starten des Daemons und die Verbindung zu ihm über einen Netzwerk-Socket verursacht wird.

Aber schon der zweite `mvnd`-Build ist 4,66-mal schneller als die `mvn`-Baseline. 0,61 Sekunden gegenüber 2,86 Sekunden ist ein Unterschied, der von einem Menschen klar wahrgenommen werden kann. Der dritte Lauf ist sogar noch schneller: 0,52 Sekunden sind ziemlich schnell für diese Art von Build.

Konfiguration

Das Verhalten des Maven Daemon kann auf viele Weisen konfiguriert werden. Die Optionen können entweder über die Kommandozeile übergeben werden oder dauerhaft an einem der folgenden

Lauf-Nr.	Kommando	Dauer (Sek)	Beschleunigung
	mvn clean install -DskipTests	2.86	1x (Baseline)
1	mvnd clean install -DskipTests	3.24	0.88x
2	mvnd clean install -DskipTests	0.61	4.66x
3	mvnd clean install -DskipTests	0.52	5.53x

Tabelle 2

Orte gespeichert werden (in absteigender Reihenfolge der Priorität):

- `${maven.multiModuleProjectDirectory}/.mvn/mvnd.properties`
- `${user.home}/.m2/mvnd.properties`
- `${mvnd.home}/conf/mvnd.properties`

Alle Konfigurationsoptionen und Kommandozeilenparameter können über `mvnd --help` angezeigt werden. Es macht wenig Sinn, sie hier alle aufzulisten. Lasst uns nur ein paar interessante herausgreifen:

Alle gängigen Maven-Pameter, wie `-am/--also-make`, `-B/--batch-mode`, `-D/--define`, `-P/--activate-profiles`, `-v/-version`, etc. werden auch von „mvnd" unterstützt.

`--completion bash` – die Vervollständigung für die Bash-Shell. Wir sollten `source <(mvnd --completion bash)` in unsere `~/.bashrc` oder `~/.bash_profile` hinzufügen.

`-Dmvnd.serial/-1/--serial` befiehlt dem Daemon nur einen Thread mit dem Standard-Singlethreaded-Builder zu verwenden und keine Log-Pufferung durchzuführen. So verhält sich `mvnd` ähnlich wie Standard-Maven. Voreinstellung: `false`.

Einschränkende Startparameter

Bei der Beschreibung der Art und Weise, wie der Client nach einem freien Daemon sucht, haben wir ein wichtiges Detail ausgelassen: die einschränkenden Startparameter. Diese definieren die wesentlichen Eigenschaften des Daemons, wie beispielsweise den Java-Installationspfad, die maximale Heap-Größe und weiteres, die ihn für bestimmte Build-Tasks exklusiv machen.

Wenn wir zum Beispiel JAVA_HOME auf unser Java-11-Installationsverzeichnis in der aktuellen Shell setzen, möchten wir nicht, dass `mvnd` einen Daemon auswählt, der auf einer anderen Java-Version läuft, selbst wenn dieser im Leerlauf wäre. Wir möchten vielmehr, dass `mvnd` immer einen Daemon auswählt, der auf genau dieser Java Version läuft, auch um den Preis, einen neuen Daemon starten zu müssen.

Hier sind einige einschränkende Startparameter zusammen mit einer kurzen Beschreibung, was sie tun:

- `-Djava.home=<Pfad>` Java Home für den Start des Daemons. Umgebungsvariable: `JAVA_HOME`
- `-Dmvnd.idleTimeout=<Dauer>` eine Zeitspanne, nach der sich ein unbenutzter Daemon von selbst beendet. Voreinstellung: 3 Stunden
- `-Dmvnd.duplicateDaemonGracePeriod=<Dauer>` Zeitspanne, nach der ungenutzte doppelte Daemons abgeschaltet werden. Doppelte Daemons sind Daemons mit dem gleichen Satz von einschränkenden Startparametern. Voreinstellung: 10 Sekunden
- `-Dmvnd.maxHeapSize=<Speichergröße>` der -Xmx-Wert, der an den Daemon übergeben wird. Diese Option hat Vorrang vor den in `-Dmvnd.jvmArgs` angegebenen Optionen.

UI-Tastenkürzel

Die textbasierte Benutzeroberfläche (UI) von `mvnd` unterstützt ein paar Tastenkürzel:

Die Taste + zeigt mehr rollende Logzeilen für die einzelnen Builder-Threads an, während die Taste - die Anzahl der Protokollzeilen reduziert. *Abbildung 1* zeigt einen Zustand der Benutzeroberfläche ohne

```
Building camel-quarkus  daemon: de8659ff  threads used/hidden/max: 3/0/3  progress:  434/1336  32%  time: 00:23
:camel-quarkus-vertx-http                    quarkus-extension:3.1.0.CR1:extension-descriptor (default)
   [INFO] Compiling 1 source file with javac [debug deprecation release 17] to target/classes
   [INFO]
   [INFO] --- quarkus-extension:3.1.0.CR1:extension-descriptor (default) @ camel-quarkus-vertx-http ---
:camel-quarkus-validator-deployment          compiler:3.11.0:compile (default-compile)
   [INFO] --- compiler:3.11.0:compile (default-compile) @ camel-quarkus-validator-deployment ---
   [INFO] Changes detected - recompiling the module! :dependency
   [INFO] Compiling 1 source file with javac [debug deprecation release 17] to target/classes
:camel-quarkus-twitter                       quarkus-extension:3.1.0.CR1:extension-descriptor (default)
   [INFO] No sources to compile
   [INFO]
   [INFO] --- quarkus-extension:3.1.0.CR1:extension-descriptor (default) @ camel-quarkus-twitter ---
```

Abbildung 3

```
[camel-quarkus-integration-test-syslog] [INFO] Changes detected - recompiling the module! :dependency
[camel-quarkus-integration-test-syslog] [INFO] Compiling 2 source files with javac [debug deprecation release 17] to target/classes
[camel-quarkus-integration-test-telegram] [INFO]
[camel-quarkus-integration-test-telegram] [INFO] --- jar:3.3.0:jar (default-jar) @ camel-quarkus-integration-test-telegram ---
[camel-quarkus-integration-test-telegram] [INFO] Building jar: /home/ppalaga/orgs/cq/camel-quarkus/integration-tests/telegram/target/
camel-quarkus-integration-test-telegram-3.0.0-SNAPSHOT.jar
[camel-quarkus-integration-test-telegram] [INFO]
[camel-quarkus-integration-test-telegram] [INFO] --- jar:3.3.0:test-jar (default) @ camel-quarkus-integration-test-telegram ---
[camel-quarkus-integration-test-telegram] [INFO] Building jar: /home/ppalaga/orgs/cq/camel-quarkus/integration-tests/telegram/target/
camel-quarkus-integration-test-telegram-3.0.0-SNAPSHOT-tests.jar
[camel-quarkus-integration-test-telegram] [INFO]
[camel-quarkus-integration-test-telegram] [INFO] --- install:2.5.2:install (default-install) @ camel-quarkus-integration-test-telegra
```

Abbildung 4

rollende Zeilen (Standard). In *Abbildung 3* sieht man die UI im Zustand mit drei rollenden Zeilen pro Builder-Thread.

Mit der Tastenkombination `CTRL+B` kann man zwischen der Thread-Ansicht (Standard, siehe oben) und der Ansicht mit rollenden Logzeilen umschalten. Die rollende Ansicht ist in *Abbildung 4* zu sehen.

Zu beachten ist, dass jeder Zeile der Name des Moduls vorangestellt ist, aus dem sie stammt.

Allgemeine Probleme

Wenn man beginnt, `mvnd` in einem Projekt zu verwenden, das nie parallel gebaut wurde, kann man auf einige Probleme stoßen.

Versteckte Abhängigkeiten

Nehmen wir an, dass die Module in unserem Projekt voneinander abhängen, wie im *Listing 3* dargestellt ist. Solange die Module den „singlethreaded" Builder verwenden, werden sie immer in der gleichen Reihenfolge gebaut und jedes Modul wird vollständig gebaut, bevor der Build des nachfolgenden Moduls anfängt. Die Reihenfolge ist also immer A, B, C.

Es ist sehr einfach, sich auf diese konstante und streng serielle Reihenfolge zu verlassen. Zum Beispiel könnte der Build von Modul C eine Datei im `target`-Ordner von B lesen. Oder die Tests von C könnten dynamisch ein Artefakt lesen, das von B in das lokale Maven-Repository geschrieben wurde.

Alle diese Annahmen gelten nicht mehr, wenn man zu einem parallelen Builder wechselt. B kann parallel zu C gebaut werden. Als Konsequenz können sich seltsame Ausnahmen ergeben. Der Build von C kann eine `FileNotFoundException` auslösen, wenn die gewünschte Datei noch nicht im `target`-Verzeichnis von B vorhanden ist. Oder es kann eine `ClassNotFoundException` auftreten, wenn der Build von C ein unfertiges Artefakt öffnet, das von B zur gleichen Zeit erzeugt wird.

Eine einfache und unkomplizierte Abhilfe ist es, einen seriellen Build zu erzwingen, indem man den Parameter `-1/--serial` verwendet.

Eine andere Möglichkeit ist, die Abhängigkeit explizit zu machen. Wenn die Abhängigkeit nicht in die Laufzeit propagieren werden soll, kann man den Scope `test` verwenden und alle transitiven Abhängigkeiten wie im *Listing 4* gezeigt ausschließen. Dies fügt keine wirkliche Abhängigkeit zu C hinzu, aber es garantiert, dass B vollständig vor C gebaut wird.

```
<dependency> <!-- Add this in C -->
     <groupid>org.my-group</groupid>
     <artifactid>B</artifactid>
     <version>${project.version}</version>
     <type>pom</type>
     <scope>test</scope>
     <exclusions>
        <exclusion>
           <groupid>*</groupid>
           <artifactid>*</artifactid>
        </exclusion>
     </exclusions>
</dependency>
```

Listing 4: Explizite Abhängigkeit um sicherzustellen, dass Modul B vor C gebaut wird.

Kaputte Plug-ins

Plug-ins können alle möglichen schlechten Dinge verursachen, die in parallelisierten Umgebungen vermieden werden sollen. Die Aufrechterhaltung eines veränderlichen globalen Zustands (über ein statisches Feld oder eine System Property) ist ein typisches Beispiel. Unweigerlich führt es zu Problemen, sobald auf die gemeinsam genutzte Ressource gleichzeitig aus mehreren Threads zugegriffen wird.

In solchen Situationen kann der Kommandozeilenparameter `-1/--serial` wieder helfen. Die Meldung des Problems an die Plug-in-Maintainer könnte jedoch langfristig zu besseren Ergebnissen führen.

Zusammenfassung

In diesem Artikelwurde Maven Daemon vorgestellt – eine relativ neue Implementierung einer älteren Idee, die von anderen Build-Tools bekannt ist. Sein Hauptzweck ist es, die Builds zu beschleunigen, indem die JVM des Builders über mehrere nachfolgende Builds hinweg „warm" gehalten wird.

Quellen/Referenzen

[1] *https://downloads.apache.org/maven/mvnd[https://downloads.apache.org/maven/mvnd*
[2] *https://sdkman.io/sdks#mvnd*
[3] *https://brew.sh/*
[4] *https://www.macports.org/*
[5] *https://community.chocolatey.org/packages/mvndaemon*
[6] *https://scoop.sh/*
[7] *https://github.com/joschi/asdf-mvnd#install*
[8] *https://www.graalvm.org/latest/docs/introduction/#features-support*
[9] *https://github.com/takari/takari-smart-builder*
[10] *http://takari.io/book/30-team-maven.html#takari-smart-builder*
[11] *https://github.com/apache/camel-quarkus*
[12] *https://www.youtube.com/watch?v=Gwmmz_T6THA*
[13] *https://peter.palaga.org/presentations/221010-maven-my-life-is-short*
[14] *https://github.com/quarkusio/gizmo*

- Quellcode, Dokumentation, Issue-Tracker: *https://github.com/apache/maven-mvnd*
- Downloads: *https://downloads.apache.org/maven/mvnd*

Peter Palaga

ppalaga@redhat.com

Peter ist Principal-Software-Ingenieur bei Red Hat. Er beschäftigt sich hauptsächlich mit der Portierung von Apache Camel Komponenten auf Quarkus. Er bastelt gerne an Build-Tools herum, vor allem an dem Maven-Daemon. In der Vergangenheit hat er an Red Hat Fuse, JBoss EAP und anderen Red Hat Middleware Produkten gearbeitet.

Mit dem Maven-Enforcer-Plug-in Standards einhalten und durchsetzen

Roland Weisleder

In komplexen Projekten ist oft „schwarze Magie“ nötig, um überhaupt einen erfolgreichen Build zu erhalten. Mit etwas Glück sind die Vorbedingungen sogar in den Tiefen des Wikis dokumentiert. Selbst ein erfolgreicher Build kann weiter hinten in der Pipeline zu Fehlern führen. Ein Beispiel sind die „Dependency-Hölle“ und die transitiven Abhängigkeiten von Maven.

In diesem Artikel schauen wir uns das Maven-Enforcer-Plug-in genauer an. Einerseits können wir Vorbedingungen für ein erfolgreiches Build und Anforderungen an Abhängigkeiten explizit definieren. Andererseits lässt sich damit auch die „Dependency-Hölle“ zähmen. Da das Maven-Enforcer-Plug-in direkt zu Beginn des Builds läuft, erhalten wir bei unerwünschten Abweichungen sehr schnell wertvolles Feedback.


MAVEN

Apache Maven ist zweifellos eines der am häufigsten verwendeten Build-Management-Tools im Java-Ökosystem. Ein Grund dafür ist, dass die ersten Schritte dank des „Convention over Configuration"-Ansatzes relativ einfach sind. Wir können ein neues Projekt ohne weitere Konfiguration direkt kompilieren, testen, packen und verteilen. Auch das Hinzufügen neuer Abhängigkeiten ist mit einem Eintrag in die pom.xml schnell erledigt.

Im Laufe der Zeit scheint jedoch der Wartungsaufwand für ein Maven-Projekt zu steigen. Beim Einrichten einer neuen Build-Umgebung, sei es in einem automatisierten Build-System oder bei neuen Teamkollegen, stolpern wir von Fehler zu Fehler. Oder der Build schlägt scheinbar zufällig fehl, weil es Probleme in der Build-Umgebung gibt. Auch externe Abhängigkeiten können zu Problemen führen, da diese sich gegenseitig beeinflussen oder sogar inkompatibel sein können. Das muss nicht im selben Projekt passieren, sondern kann auch an anderen Stellen auftreten, an denen unser Projekt als Bibliothek verwendet wird. Willkommen in der gefürchteten „Dependency-Hölle".

Hier kommt das Maven-Enforcer-Plug-in *[1]* ins Spiel. Es hilft uns, Standards für die Definition von Maven-Projekten zu etablieren und durchzusetzen. Dies können einerseits Rahmenbedingungen für die Build-Umgebung sein, andererseits können dies Anforderungen an die definierten Abhängigkeiten sein.

Plug-in einrichten und testen

Um das Maven-Enforcer-Plug-in zum Projekt hinzuzufügen, genügt es, in der pom.xml einen Eintrag in den Abschnitt *plugins* aufzunehmen. In der Plug-in-Konfiguration müssen wir außerdem die auszuführenden Regeln und gegebenenfalls deren Konfiguration hinterlegen. *Listing 1* zeigt ein vollständiges Beispiel.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>3.3.0</version>
  <executions>
     <execution>
        <id>enforce-rules</id>
        <goals>
           <goal>enforce</goal>
        </goals>
        <configuration>
           <rules>
              <alwaysFail/>
           </rules>
        </configuration>
     </execution>
  </executions>
</plugin>
```

Listing 1: Beispiel-Konfiguration für das Maven-Enforcer-Plug-in

In diesem Beispiel verwenden wir die eingebaute und für Demo-Zwecke hilfreiche Regel `alwaysFail`, die den Build immer fehlschlagen lässt. Wenn wir nun ein Maven-Build mit dieser Konfiguration starten, sollte es sofort fehlschlagen.

Da das Maven-Enforcer-Plug-in in der Phase `validate` direkt am Anfang des Builds ausgeführt wird, wird es immer ausgeführt, egal ob wir das Maven-Projekt mit `mvn package`, `mvn install` oder `mvn deploy` bauen.

Sobald das Maven-Enforcer-Plug-in konfiguriert ist, wird es bei jedem Build ausgeführt und lässt es bei unerwünschten Abweichungen sofort fehlschlagen. Manchmal ist das jedoch nicht sinnvoll, zum Beispiel, wenn es eine Abweichung gibt, die wir nicht sofort beheben können. In diesem Fall können wir das Plug-in überspringen, beispielsweise über die Kommandozeile:

```
mvn package -Denforcer.skip
```

Umgebung und Konfiguration testen

Nachdem die Einrichtung des Maven-Enforcer-Plug-ins abgeschlossen ist, können wir die Regel `alwaysFail` durch sinnvolle Regeln für unser Projekt ersetzen. Wir können nun zunächst definieren, wie die Build-Umgebung für unser Maven-Projekt aussehen soll.

Eine der ersten Fragen, die sich beim Einrichten eines Projekts in einer neuen Umgebung stellt, ist die Frage nach der zu verwendenden Java-Version. Wir können zunächst eine beliebige Java-Version ausprobieren und schauen, ob es Kompilierfehler gibt, die auf eine falsche Java-Version zurückzuführen sind. Oder wir definieren eine Enforcer-Regel `requireJavaVersion`, die die verwendete Java-Version prüft. *Listing 2* zeigt eine Plug-in-Konfiguration, mit der Java 11 oder neuer erzwungen wird. Würden wir versuchen, einen Build mit Java 8 auszuführen, würde das Plug-in sofort mit einem Hinweis auf die zu verwendende Java-Version fehlschlagen. Als Version können wir entweder eine feste Version oder einen Versionsbereich angeben.

```
<rules>
  <requireJavaVersion>
     <version>[11,)</version>
  </requireJavaVersion>
</rules>
```

Listing 2: Plug-in-Konfiguration zum Erzwingen von Java 11 oder neuer

In vielen Projekten gibt es zudem eine Vorgabe, welche Maven-Version für die Builds verwendet werden soll. Normalerweise wird der Maven-Wrapper *[2]* verwendet, um für einen bestimmten Projektstand eine definierte Maven-Version in allen Umgebungen bereitzustellen. Das hindert jedoch niemanden daran, das Projekt mit einer lokalen Maven-Installation in einer anderen, möglicherweise älteren Version zu bauen. Mit der Enforcer-Regel `requireMavenVersion` können wir dies zusätzlich absichern. *Listing 3* zeigt eine Plug-in-Konfiguration, mit der Maven 3.8 oder neuer erzwungen wird.

```
<rules>
  <requireMavenVersion>
     <version>[3.8,)</version>
  </requireMavenVersion>
</rules>
```

Listing 3: Plug-in-Konfiguration zum Erzwingen von Maven 3.8 oder neuer

Auch in den späteren Phasen eines Maven-Builds kann die Umgebung und die Konfiguration Einfluss darauf haben, ob der Build

erfolgreich ist oder fehlschlägt. Eine typische Anforderung ist, dass Artefakte vor der Veröffentlichung signiert werden müssen. Die Lösung könnte so aussehen, dass dafür eine Schlüsseldatei mit einem privaten Schlüssel in der Umgebung vorhanden sein muss. Der Maven-Build soll das Kennwort dafür über eine Umgebungsvariable erhalten. Das Vorhandensein dieser Informationen können wir mit den Regeln `requireFilesExist` und `requireEnvironmentVariable` prüfen, siehe *Listing 4*.

```
<rules>
  <requireFilesExist>
     <files>
        <file>${user.home}/.ssh/id_ed25519.pub</file>
     </files>
  </requireFilesExist>
  <requireEnvironmentVariable>
     <variableName>KEY_PASSWORD</variableName>
  </requireEnvironmentVariable>
</rules>
```

Listing 4: Plug-in-Konfiguration zum Erzwingen einer vorhandenen Schlüsseldatei und deren Kennwort

Gerade dieses Beispiel zeigt, warum der Einsatz des Maven-Enforcer-Plug-ins sinnvoll ist. Angenommen, das Kompilieren und die Testausführung nehmen mehrere Minuten in Anspruch. Wenn nun im Anschluss bei der Signierung der Build fehlschlägt, weil in der Build-Umgebung der notwendige Schlüssel nicht vorhanden ist, dann ist das aufgrund der unnötig verschwendeten Zeit und Ressourcen sehr ärgerlich. Mit dem Hinzufügen des Maven-Enforcer-Plug-ins erhalten wir direkt zu Beginn des Builds wertvolles Feedback, wenn die Umgebung nicht richtig konfiguriert ist.

Abhängigkeiten testen

Nicht nur eine falsche Build-Umgebung kann für fehlschlagende Builds sorgen, ebenso können die definierten Abhängigkeiten zu unerwarteten Effekten führen. Auch hier kann das Maven-Enforcer-Plug-in helfen, die gefürchtete „Dependency-Hölle" zu zähmen.

Ein Beispiel für unerwartete Effekte sind transitive Abhängigkeiten, das heißt, Abhängigkeiten von Abhängigkeiten, die wir deklariert haben. Diese Abhängigkeiten können selbst auch wieder Abhängigkeiten haben, sodass ein nahezu beliebig großer Abhängigkeitsbaum entstehen kann. Das führt dazu, dass bei manchen Drittanbieter-Bibliotheken nicht auf den ersten Blick ersichtlich ist, welche transitiven Abhängigkeiten im Projekt landen. Außerdem passiert es dann sehr schnell, dass Entwickler im Abhängigkeitsbaum vorhandene Java-Klassen verwenden, die nicht als direkte Abhängigkeit definiert wurden. Als Konsequenz gibt es in einigen Projekten die Regel, dass generell alle transitiven Abhängigkeiten zu entfernen sind. Mit der Regel `banTransitiveDependencies` können wir das sicherstellen, wie in *Listing 5* gezeigt.

```
<rules>
  <banTransitiveDependencies/>
</rules>
```

Listing 5: Plug-in-Konfiguration zum Unterbinden von transitiven Abhängigkeiten

Dann gibt es Szenarien, dass ein Projekt generell keine Abhängigkeit zu einer bestimmten Bibliothek oder zu einer bestimmten Version einer Bibliothek haben soll. Ein möglicher Grund kann sein, dass dort eine Sicherheitslücke bekannt ist. Beispielsweise enthält die Bibliothek Log4j bis einschließlich Version 2.17.0 eine Sicherheitslücke, die als Log4Shell *[3]* bekannt ist. Diese Lücke wurde mit Version 2.17.1 behoben. Mit der Regel `bannedDependencies`, wie in *Listing 6* gezeigt, können wir nun alle Versionen vor 2.17.1 aus dem Abhängigkeitsbaum unseres Projektes verbannen. Sollte nun doch jemand eine direkte oder transitive Abhängigkeit zu einer vulnerablen Version von Log4j hinzufügen, wird das Maven-Enforcer-Plug-in den Build direkt fehlschlagen lassen.

```
<rules>
  <bannedDependencies>
     <excludes>
        <exclude>org.apache.logging.
log4j:*:[,2.17.1)</exclude>
     </excludes>
  </bannedDependencies>
</rules>
```

Listing 6: Plug-in-Konfiguration zum Unterbinden bestimmter Abhängigkeiten

Weitere unerwartete Effekte können auftreten, wenn eine Abhängigkeit mehrfach im Abhängigkeitsbaum und mit unterschiedlichen Versionen vorhanden ist. In Java ist es nämlich standardmäßig nicht möglich, dass eine Java-Klasse in mehreren Versionen aus dem Classpath geladen werden kann. Maven entscheidet daher, welche Version der Abhängigkeit in den finalen Classpath aufgenommen wird. Dafür gibt es zwar definierte und dokumentierte Regeln, die jedoch für Überraschungen sorgen können. Mit dem Maven-Enforcer-Plug-in können wir nun die unerwarteten Überraschungen reduzieren, die durch die Versionsauswahl entstehen können.

In größeren Projekten kann es aus Unachtsamkeit vorkommen, dass jemand in der pom.xml eine Abhängigkeit doppelt definiert, wie in *Listing 7* dargestellt. Aus Sicht von Maven stellt das zunächst keinen Fehler dar, sondern sorgt lediglich für eine Warnung. Nun können alle kurz überlegen, welche Version im Fall von *Listing 7* wohl in den Classpath aufgenommen wird. Fertig? Die Regeln besagen, dass die zuletzt definierte Version genommen wird, in diesem Fall Version 5.3.22. Da diese Konstellation an sich jedoch nicht vorkommen sollte, weil sie nur für Verwirrung sorgt, können wir das mit `banDuplicatePomDependencyVersions` unterbinden, wie in *Listing 8* gezeigt.

```
<dependencies>
  <dependency>
     <groupId>org.springframework</groupId>
     <artifactId>spring-context</artifactId>
     <version>5.3.23</version>
  </dependency>
  <!--
     ... 500 Zeilen weitere Abhängigkeiten ...
  -->
  <dependency>
     <groupId>org.springframework</groupId>
     <artifactId>spring-context</artifactId>
     <version>5.3.22</version>
  </dependency>
</dependencies>
```

Listing 7: Beispiel für doppelt definierte Abhängigkeiten

```
<rules>
  <banDuplicatePomDependencyVersions/>
</rules>
```

Listing 8: Plug-in-Konfiguration zum Unterbinden doppelt definierter Abhängigkeiten

Ein weiteres Beispiel, das häufiger auftritt, ist die Inkompatibilität zwischen definierten beziehungsweise transitiven Abhängigkeiten. *Listing 9* zeigt einen Ausschnitt aus den definierten Abhängigkeiten eines Projektes auf Basis von Spring Boot. Auf den ersten Blick gibt es scheinbar kein Problem mit den Abhängigkeiten. Wenn wir nun berücksichtigen, dass Spring Boot 2 auf Spring Framework 5 basiert, dann ergibt sich schnell eine Inkompatibilität, weil eine ältere Version des Spring Frameworks definiert ist, bezehungsweise im finalen Classpath vorhanden sein wird. Diese Inkompatibilität wird umso schwerer zu entdecken, je mehr Abhängigkeiten dazwischen in der pom.xml definiert sind. Mit der Regel `requireSameVersions` können wir definieren, dass eine Gruppe von Abhängigkeiten im gesamten Abhängigkeitsbaum die gleiche Version hat. *Listing 10* zeigt, wie das Inkompatibilitätsproblem in diesem Fall automatisch gefunden werden kann.

```
<dependencies>
  <dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-web</artifactId>
     <version>2.7.5</version>
  </dependency>
  <!--
     ... 500 Zeilen weitere Abhängigkeiten ...
  -->
  <dependency>
     <groupId>org.springframework</groupId>
     <artifactId>spring-context</artifactId>
     <version>4.3.18.RELEASE</version>
  </dependency>
</dependencies>
```

Listing 9: Beispiel für Inkompatibilität zwischen Abhängigkeiten

```
<rules>
  <requireSameVersions>
     <dependencies>
        <dependency>org.springframework:*</dependency>
     </dependencies>
  </requireSameVersions>
</rules>
```

Listing 10: Plug-in-Konfiguration zum Erzwingen gleicher Versionen für mehrere Abhängigkeiten

All diese Beispiele zeigen, dass eine mögliche Ursache für die „Dependency-Hölle" darin liegt, dass Abhängigkeiten mehrfach und mit unterschiedlichen Versionen im Abhängigkeitsbaum vorkommen und, dass am Ende nur genau eine Version im finalen Classpath sein kann. Dieses Problem hätten wir nicht, wenn im Abhängigkeitsbaum je Abhängigkeit nur genau eine Version vorhanden wäre. Genau das können wir mit der Regel `dependencyConvergence` sicherstellen, wie in *Listing 11* gezeigt. Diese Regel schlägt dann fehl, wenn eine Abhängigkeit mit unterschiedlichen Versionen im Abhängigkeitsbaum vorhanden ist. Eine beispielhafte Fehlerausgabe ist in *Listing 12* zu sehen.

```
<rules>
  <dependencyConvergence/>
</rules>
```

Listing 11: Plug-in-Konfiguration zum Erzwingen einheitlicher Versionen von Abhängigkeiten

Eigene Regeln implementieren

Das Maven-Enforcer-Plug-in bringt bereits sehr viele nützliche Regeln mit. Doch jedes Projekt ist anders und hat gegebenenfalls auch andere Anforderungen. Im Unternehmensumfeld ist es beispielsweise häufig so, dass während eines Release-Zyklus zunächst Snapshot-Artefakte erstellt und in einem internen Snapshot-Repository zur Verfügung gestellt werden. Dieses Snapshot-Repository muss dann in jeder Build-Umgebung separat konfiguriert werden. Um das Vorhandensein sicherzustellen, können wir wieder eine Regel für unser Projekt definieren. Da es für diese Anforderung keine vorgefertigte Regel gibt, müssen wir eine eigene Regel schreiben. Durch das bereitgestellte Java-API ist es sehr einfach, eigene Regeln selbst zu implementieren. *Listing 13* zeigt eine Beispiel-Regel, die das Vorhandensein eines Snapshot-Repositorys sicherstellt.

Durch das bereitgestellte Java-API können wir die Regeln sehr flexibel definieren und implementieren. Selbst wenn die Anforderung kommt, dass am Wochenende keine Builds ausgeführt werden dürfen, weil der zentrale interne Repository-Server regelmäßig abgeschaltet wird, ist das durch die Erweiterbarkeit schnell implementiert und ausgerollt.

Fazit

Das Maven-Enforcer-Plug-in ist ein nützliches Werkzeug, das in keinem Werkzeugkoffer fehlen sollte, sobald man in einem Maven-Projekt arbeitet. Richtig konfiguriert erhalten wir sehr schnell Feedback, wenn die Build-Umgebung nicht dem entspricht, was wir er-

```
Dependency convergence error for org.springframework:spring-core:jar:4.3.18.RELEASE:compile paths to dependency are:
+-de.rweisleder.example:module-c:jar:1.0-SNAPSHOT
  +-de.rweisleder.example:module-a:jar:1.0-SNAPSHOT:compile
     +-org.springframework:spring-core:jar:4.3.18.RELEASE:compile
and
+-de.rweisleder.example:module-c:jar:1.0-SNAPSHOT
  +-de.rweisleder.example:module-b:jar:1.0-SNAPSHOT:compile
     +-org.springframework:spring-context:jar:5.3.23:compile
        +-org.springframework:spring-aop:jar:5.3.23:compile
           +-org.springframework:spring-core:jar:5.3.23:compile
```

Listing 12: Beispiel für Fehlermeldung der dependencyConvergence-Regel

```
public class RequireSnapshotRepositoryRule implements EnforcerRule {

  public void execute(EnforcerRuleHelper helper) throws EnforcerRuleException {
     MavenProject project = getProject(helper);

     for (Repository repository : project.getRepositories()) {
        if (repository.getSnapshots().isEnabled()) {
           return;
        }
     }

     throw new EnforcerRuleException("No snapshot repository configured");
  }

  // ...
}
```

Listing 13: Beispiel-Regel, um das Vorhandensein eines Snapshot-Repositorys sicherzustellen

warten. Durch die vielfältigen vorgefertigten Regeln ist es möglich, den Wildwuchs von direkten und transitiven Abhängigkeiten einzudämmen. Durch das bereitgestellte Java-API können wir eigene Regeln implementieren, die individuell auf die Projektanforderungen zugeschnitten sind. Nicht umsonst bezeichnet sich das Maven-Enforcer-Plug-in selbst als „liebenswerte eiserne Faust Mavens".

Quellen

[1] *https://maven.apache.org/enforcer/maven-enforcer-plugin/*
[2] *https://maven.apache.org/wrapper/*
[3] *https://de.wikipedia.org/wiki/Log4Shell*

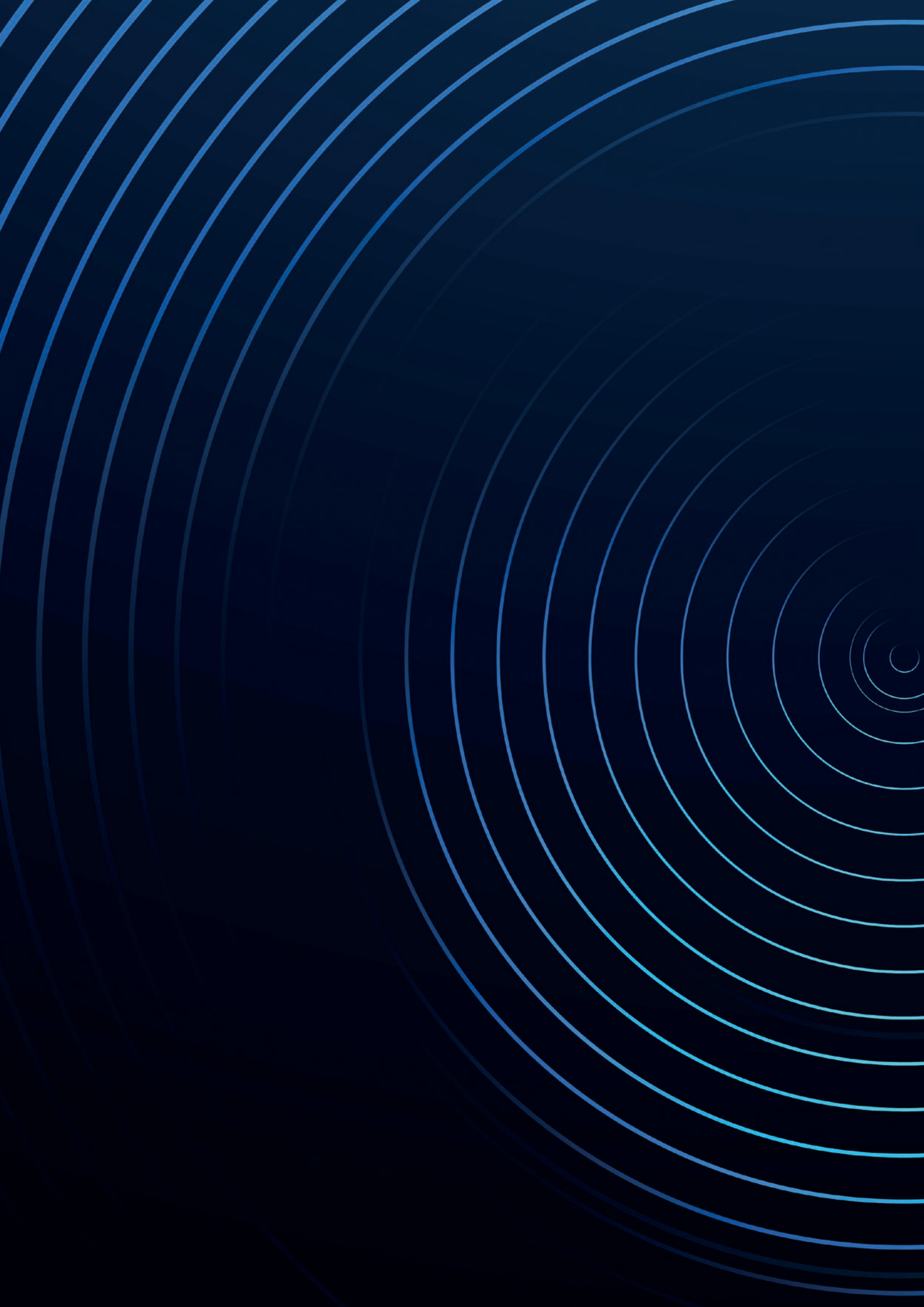
Roland Weisleder
roland@rweisleder.de

Roland Weisleder ist freiberuflicher Softwareentwickler und seit 2009 im Java-Ökosystem unterwegs. Er unterstützt Entwicklungsteams dabei, ihre Legacy-Java-Systeme in die Zukunft zu bringen und insbesondere die Struktur und die Testautomatisierung zu verbessern. Seine Erfahrungen teilt er in Workshops, Artikeln und Vorträgen im In- und Ausland.

Anwendungen mit Java-Modulen und Gradle elegant strukturieren

Dr. Jendrik Johannes

Die Themen Modularisierung und Abhängigkeiten-Verwaltung sind Reizthemen für viele Java-Entwickler. In kaum einem Projekt kommt man an ihnen vorbei, doch oft scheinen sie eher Probleme zu erzeugen als sie zu lösen. In Java-Projekten fallen beide Themenbereiche größtenteils in die Zuständigkeit eines Build-Tools wie Gradle. Seit Java 9 kann Java Modul-Beschreibungen direkt abbilden und bringt dabei noch zusätzliche Features wie private Packages mit. Das macht es erst mal nicht einfacher: Was benutze ich denn nun, wenn sowohl Java als auch das Build-Tool zur Beschreibung von Modulen genutzt werden sollen? Gewinne ich dabei außer Redundanz überhaupt noch was? Dieser Artikel beschreibt, wie man die Stärken des Java-Modulsystems mit dem richtigen Gradle- und Gradle-Plug-in-Setup für eine übersichtlichere und wartbarere Java-Projektstruktur einsetzt.



Bei einer modernen Java-Anwendung wird die Struktur/Architektur der Software typischerweise im Source-Code abgebildet, indem man die Anwendung auch dort in mehrere Module aufteilt *(siehe Abbildung 1)*. In der Praxis bedeutet das, dass man mehrere `src/main/java`-Ordner hat, in denen jeweils der Source-Code eines Moduls liegt *(siehe Abbildung 2 (A))*. Damit die Module aufeinander Bezug nehmen können, um am Ende eine integrierte Software zu bilden, müssen Abhängigkeiten zwischen den eigenen Modulen sowie Abhängigkeiten zu Third-Party-Modulen irgendwo beschrieben werden.

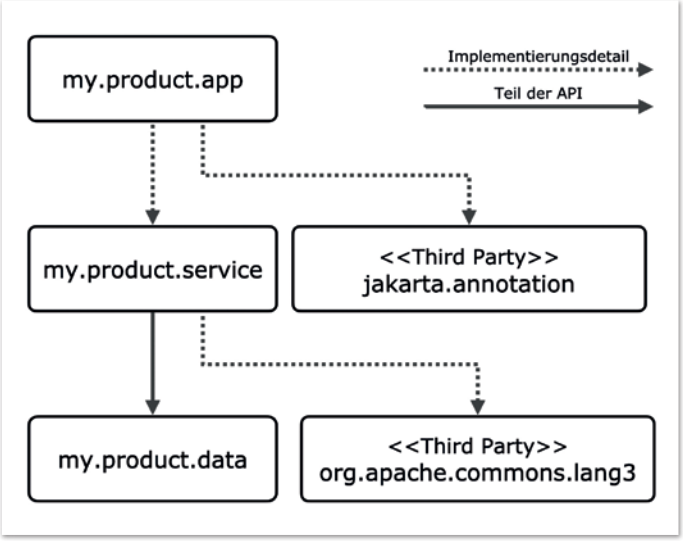


Abbildung 1: Struktur/Architektur einer modularen Software

Haben wir Java als Sprache für die Entwicklung gewählt, aber ansonsten noch keinerlei Tooling festgelegt, stellen sich folgende Fragen:

- **Frage 1**: Wie und wo definiere ich die Abhängigkeiten und Sichtbarkeiten der Module untereinander?
- **Frage 2**: Wie und wo definiere ich, wie die Module gebaut werden? (Mit welchem JDK wird kompiliert? Mit welchem Test-Framework wird getestet?)
- **Frage 3**: Wie sorge ich dafür, dass Third-Party-Module automatisiert bereitgestellt werden und wo verwalte ich deren Versionen?

Antwort 1: Das Java-Modulsystem – Module definieren ohne Build-Tool

In vielen modularen Java-Anwendungen werden die drei oben genannten Aspekte durch Funktionalität im Build-Tool (in der Regel Gradle oder Maven) abgedeckt. Dadurch wird die Abbildung der Struktur *(siehe Abbildung 1)* in der Software mit Build-Tool spezifischer Notation nötig (zum Beispiel Gradle-Kotlin-DSL oder Maven-POM-XML) und es kommt oft zu Vermischungen zwischen der Definition der Software-Struktur und anderen Build-Tool-Konfigurationen.

Mit dem Java-Modulsystem können wir die Software-Struktur in Java direkt definieren. Dafür hat jedes Modul eine `module-info.java`-Datei *(siehe Abbildung 2 (B))* mit eigens dafür eingeführter Syntax und Keywords *(siehe Listings 1-3)*. Diese werden vom Java-Compiler genutzt, um Sichtbarkeiten von Klassen bei der Kompilierung zu prüfen. Außerdem werden sie zu `module-info.class`-Dateien kompiliert, sodass jedes Jar einen dieser Modul-Deskriptoren enthält *(siehe Abbildung 3)*. Dadurch steht die Information über Struktur und Sichtbarkeit von Modulen auch zur Java-Laufzeit zur Verfügung.

Zur Beschreibung der Modul-Abhängigkeiten sind zwei Features in der `module-info.java` von Bedeutung. Erstens bekommt jedes Modul einen eindeutigen Namen, um es referenzieren zu können – `module my.product.service`. Zweitens drückt man über sogenannte `require`-Direktiven aus, auf welche anderen Module ein Modul Zugriff benötigt – `requires my.product.data`. Die Direktiven enthalten eine Sichtbarkeitsangabe: `requires` definiert eine Abhängigkeit auf ein Implementierungsdetail, `requires` transitive definiert eine Abhängigkeit, die nach außen weitergegeben

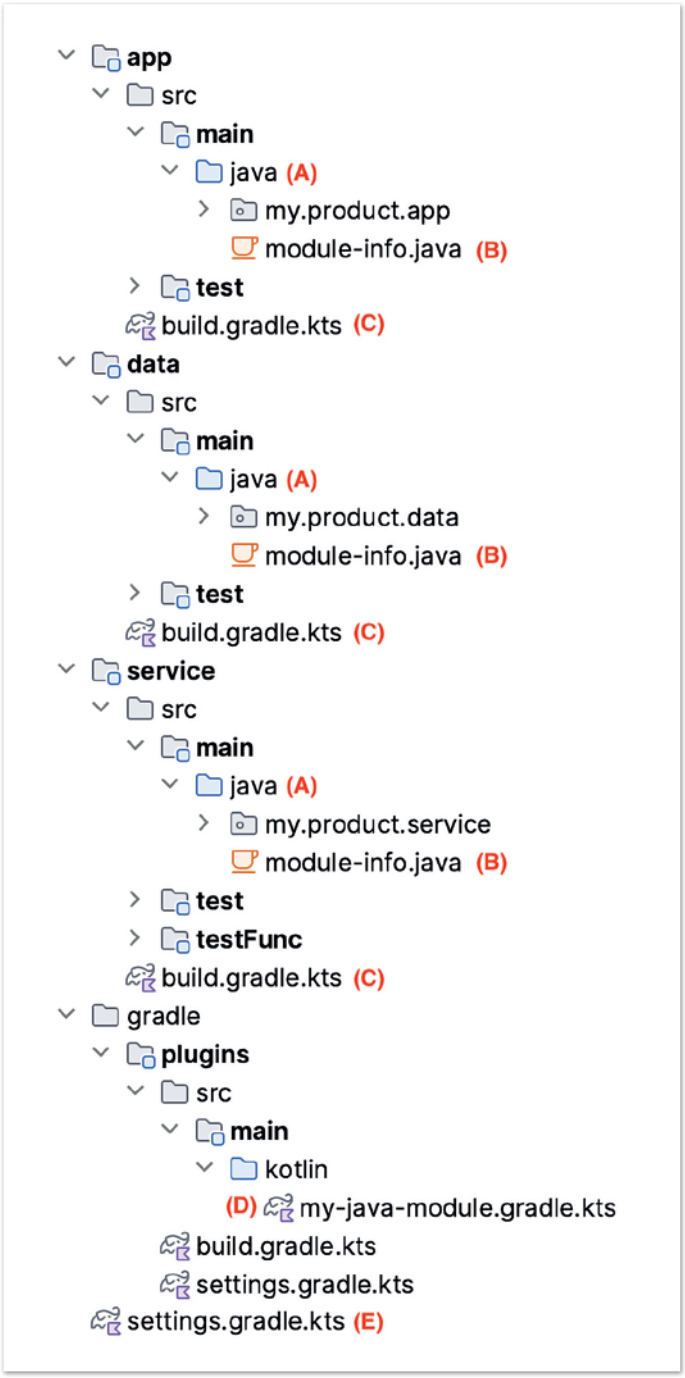


Abbildung 2: Ordner und Dateien für die Strukturierung der Software

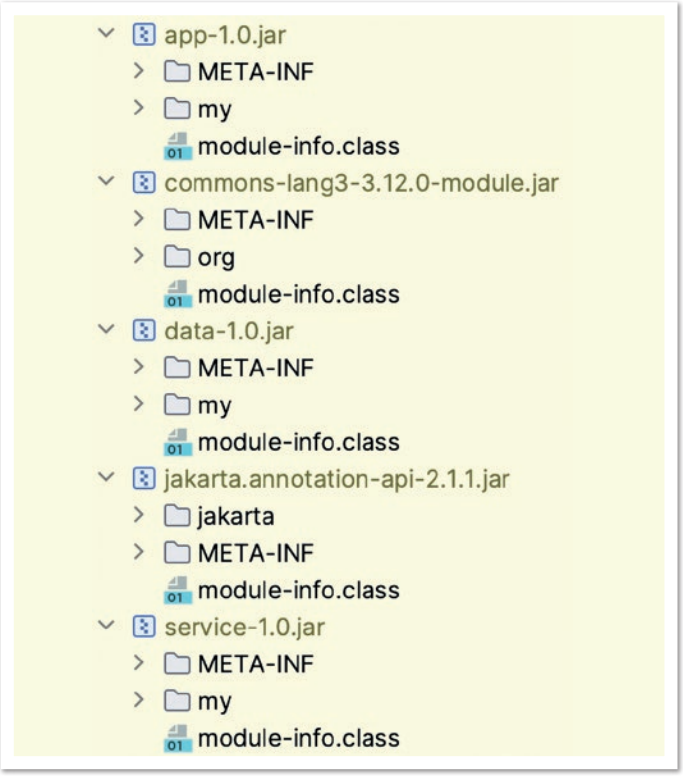


Abbildung 3: Jede Modul-Jar enthält eine module-info.class

wird (also Teil der API des Moduls ist) und `requires static` wird genutzt, wenn etwas nur zur Compile-Zeit benötigt wird (ein eher seltener Fall).

Zusätzlich zu den Abhängigkeiten zwischen Modulen, sind in den `module-info.java`-Dateien die exportierten Packages definiert. Dies ist ein interessantes neues Sprachfeature, das nur das Modulsystem bietet: Packages in einem Modul, also in einem Jar, sind erst mal nur für Klassen in diesem Modul sichtbar – unabhängig davon, ob die Klassen in den Packages `public` sind. Dadurch kann man Packages für die interne Struktur eines Moduls frei nutzen und beispielsweise nur ein Package mit `public`-Interfaces exportieren. Sowohl Java-Compiler als auch -Laufzeit beachten diese Sichtbarkeiten.

Alle Details der `module-info.java` werden in diesem Artikel nicht beschrieben. Es lässt sich beispielsweise noch detaillierter kontrollieren, welches andere Modul auf welches Package zugreifen darf und was per Reflection sichtbar ist (open Keyword). Außerdem werden Services und Service-Provider nun hier definiert, die man ohne Modulsystem via `META-INF`-Dateien kontrolliert (dies ist quasi eine neue Syntax für ein existierendes Feature).

Zusammengefasst bietet das Java-Modulsystem, verglichen mit einer Modul-Definition über das Build-Tool, folgende Vorteile:

1. Build-Tool-unabhängige Java-Standard-Notation
2. Eingeschränkte Sichtbarkeit zur Compile-Zeit (in Gradle auch möglich)
3. Präzise Definition von Package-Sichtbarkeiten
4. Modul-Struktur-Wissen ist in den Java-Tools (zum Beispiel `javac` und `java`) vorhanden

Antwort 2: Zentralisierte (und versteckte) Gradle-Konfiguration

Wenn wir die Struktur der Software nun in Java direkt abbilden, wozu brauchen wir dann noch ein Build-Tool? Wir erinnern uns, dass wir anfangs drei Fragen formuliert haben, die zum Bauen einer modularen Anwendung beantwortet werden wollen. Wenn wir das Modulsystem mit `module-info.java`-Dateien als Antwort auf Frage 1 (Definition der Modul-Abhängigkeiten) hernehmen, sind die Antworten auf Frage 2 und Frage 3 noch offen.

In Frage 2 geht es darum, Dinge zu konfigurieren, die das generelle Setup aller unserer Module betreffen. Gradle bietet hier für fast alles Konventionen (Convention-over-Configuration), sodass es genügt, zu sagen, dass es sich bei unseren Modulen um Java-Projekte handelt. Dann findet Gradle zum Beispiel automatisch Source-Code in `src/main/java` und erkennt an der Existenz der `module-info.java`, dass wir das Java-Modulsystem nutzen. Dennoch gibt es einige Dinge, die man in der Regel selbst festlegen möchte – beispielsweise, welches JDK und welches Test-Framework genutzt wird. Hier möchte man seine eigenen Konventionen, die für alle eigenen Module gelten, über die von Gradle vorgegeben Konventionen legen.

In der Praxis funktioniert dies in modernem Gradle über das Gradle-Plug-in-System. Jedes Modul enthält eine `build.gradle.kts`-Datei *(siehe Abbildung 2 (C))*, in die wir eine Plug-in-ID eintragen – `plugins { id("my-java-module") }` *(siehe Listing 4)*. Wenn wir nur die vorgegebenen Gradle-Konventionen nutzen wollen, können wir direkt ein von Gradle bereitgestelltes Plug-in nutzen – `id("java-library")`. Wenn wir eigene Konventionen hinzufügen, wie in diesem Beispiel, nutzen wir eine eigene ID – `id("my-java-modu-`

```
module my.product.app { // Modul-Name
  requires my.product.service; // Abhängigkeit
  requires jakarta.annotation; // Abhängigkeit (Third-Party)
}
```

Listing 1: app/src/main/java/module-info.java

```
module my.product.service {
  requires transitive my.product.data;
  requires org.apache.commons.lang3;
  exports my.product.service; // Exportiertes Package
}
```

Listing 2: service/src/main/java/module-info.java

```
module my.product.data {
  exports my.product.data;
}
```

Listing 3: data/src/main/java/module-info.java

le")`. Ein solches Convention-Plug-in definieren wir als Erweiterung eines existierenden Plug-ins zentral in einer eigenen Datei, – `my-java-module.gradle.kts` *(siehe Listing 5)* – die wir in einem gesonderten `gradle/plugins`-Ordner ablegen *(siehe Abbildung 2 (D))* – also außerhalb der einzelnen Module. Welche Module es gibt, und wo die Convention-Plug-ins abgelegt sind, wird zentral in der `settings.gradle.kts`-Datei konfiguriert *(siehe Abbildung 2 (E)/ Listing 6; Zeilen 1/3)*.

Schauen wir uns dieses Setup noch einmal in der Übersicht an *(siehe Abbildung 2)*, stellen wir fest, dass in den einzelnen Modul-Ordnern (`data`, `service`, `app`) nur die `build.gradle.kts`-Dateien, die jeweils nur eine Zeile mit der Plug-in-ID enthalten, als Gradle-spezifisches Artefakt liegen. Weitere Konfigurationen sind zentralisiert im `gradle/plugins`-Ordner verschwunden. Dies ist auch bei Java-Projekten, die das Modul-System nicht nutzen, ein sinnvolles Setup, da es die Basis-Build-Konfiguration (alles, was nichts mit Abhängigkeiten zu tun hat), die während der täglichen Entwicklung selten angepasst werden muss, aus den eigentlichen Projekten/Modulen herauszieht und zentralisiert.

Antwort 3: Katalog für Versionen von Third-Party-Module

Die letzte offene Frage 3 ist, wie wir an Third-Party-Module, die in einem Repository wie Maven Central liegen, gelangen und deren Versionen verwalten. Dieses Thema führt oft zu Missverständnissen im Zusammenhang mit dem Java-Modulsystem. Das Modulsystem bietet hierfür keinerlei Funktionalität. Es kennt zwar das Konzept einer Version, die (optional) in die `module-info.class` hineincodiert werden kann, kann aber nicht mit mehreren Versionen eines Moduls gleichzeitig umgehen oder automatisch eine Version (zum Beispiel die höhere) auswählen.

```
plugins { id("my-java-module") }
```

Listing 4: service/build.gradle.kts

Das Beziehen von Abhängigkeiten aus Repositorys ist unabhängig vom Java-Modulsystem. Ob man ein Jar mit `module-info.class` oder ohne Maven Central lädt ist, von außen betrachtet, erst einmal egal. Darum kann man hier Gradle, das viele Features, insbesondere zum Erkennen und Behandeln von Konflikten in großen Abhängigkeitsgraphen mitbringt, gut einsetzen.

Um ein Modul aus einem Repository zu beziehen, benötigt man, anders als bei lokal gebauten Modulen, eine Version. Um diese Versionsverwaltung zu zentralisieren, gibt es in neueren Gradle-Versionen den so genannten Dependency-Version-Catalog, den man in der `settings.gradle.kts`-Datei definieren kann *(siehe Listing 6; Zeilen 9-11)*. Im Gegensatz zur versteckten Build-Konfiguration im Convention-Plug-in *(siehe Listing 5)* werden Versionen während der Entwicklung regelmäßig angepasst.

Anzumerken ist, dass Gradle noch andere Möglichkeiten zur Zentralisierung von Versionen bietet (Platforms/BOMs oder Dependency-Locking), die je nach Projektart und Entwicklungsprozess besser geeignet sein können. Wichtig ist, dass die Versionsverwaltung, ähnlich wie die Build-Konfiguration, zentralisiert ist und nicht über die einzelnen Module verteilt wird. In der integrierten Software kann ohnehin immer nur eine Version eines Moduls verwendet werden.

Für unser Java-Modulsystem-Setup interessiert uns von den Features des Version Catalog nur die Definition von Versionen. Hierbei

```
 1 plugins {
 2    id("java-library")
 3    id("org.gradlex.extra-java-module-info")
 4    id("org.gradlex.java-module-dependencies")
 5    id("org.gradlex.java-module-testing")
 6 }
 7
 8 group = "my.product"
 9 version = "1.0"
10
11 java {
12    toolchain.languageVersion = JavaLanguageVersion.of(17)
13 }
14
15 javaModuleDependencies {
16    moduleNameToGA.put("org.apache.commons.lang3",
17                                "org.apache.commons:commons-lang3")
18 }
19
20 extraJavaModuleInfo {
21    module("org.apache.commons:commons-lang3",
22                "org.apache.commons.lang3") {exportAllPackages()}
23 }
24
25 javaModuleTesting {
26    whitebox(testing.suites.getByName<JvmTestSuite>("test") {
27       useJUnitJupiter("")
28    }) {
29       requires.add("org.junit.jupiter.api")
30       opensTo.add("org.junit.platform.commons")
31    }
32    blackbox(testing.suites.create<JvmTestSuite>("testFunc"))
33 }
```

Listing 5: gradle/plugins/src/main/kotlin/my-java-module.gradle.kts

```
1 pluginManagement { includeBuild("gradle/plugins") }
2
3 include("data", "service", "app")
4
5 dependencyResolutionManagement {
6    repositories.mavenCentral()
7
8    versionCatalogs.create("libs") {
9       version("jakarta.annotation", "2.1.1")
10      version("org.apache.commons.lang3", "3.12.0")
11      version("org.junit.jupiter.api", "5.8.2")
12   }
13 }
```

Listing 6: settings.gradle.kts

gibt man jeweils einen Alias (einen frei gewählten Namen) und die gewünschte Versionsnummer an. Wir nutzen direkt die Modul-Namen als Alias, um die Verbindung zum Modulsystem zu ziehen und entscheiden uns so für Versionen der Third-Party-Libraries, die wir verwenden möchten *(siehe Listing 6; Zeilen 9-11)*.

Das Modulsystem und Gradle miteinander verbinden

Damit haben wir unsere drei Fragen beantwortet:

- **Antwort 1**: Zur Definition der Module mit Abhängigkeiten und Sichtbarkeiten nutzen wir die `module-info.java` des Java-Modulsystems und haben damit – abgesehen von einer einzeiligen `build.gradle.kts`-Datei – keinerlei Gradle-spezifische Notation in den Modulen.
- **Antwort 2**: Details zum Bauen aller Module (JDK, Test-Framework) sind zentral in einem Gradle-Convention-Plug-in definiert und müssen während der täglichen Entwicklung nur selten angepasst werden.
- **Antwort 3**: Versionen definieren wir zentral im Version-Catalog in der `settings.gradle.kts` und nutzen dabei die Modul-Namen als Versions-Aliase, um Versionen zu Modulen zuzuordnen.

Um unsere Module mit Gradle zu bauen oder unser Projekt über Gradle in eine IDE zu importieren, muss Gradle die Modul-System-Artefakte als Teil der Projektbeschreibung interpretieren. Diese Verbindung zieht das `org.gradlex.java-module-dependencies`-Plug-in *[1]*, das wir in unser `my-java-module.gradle.kts`-Convention-Plug-in aufnehmen *(siehe Listing 5, Zeile 4)*. Dieses Plug-in liest die Inhalte der `module-info.java`-Dateien und übersetzt diese in Gradle-Dependencies. Dabei wird die Sichtbarkeit der Abhängigkeiten erhalten (`requires` entspricht `implementation`, `requires transitive` entspricht `api`). Außerdem sucht das Plug-in für jedes Modul die entsprechende Version aus dem Version-Catalog heraus und integriert diese ebenfalls in die entsprechende Gradle-Dependency. Dadurch fällt in unserem Setup der `dependencies {}`-Block weg, der in sich in traditionellen Java-Projekten in jeder `build.gradle.kts`-Datei finden lässt.

Für das korrekte Weitergeben der Dependencies an Gradle, benötigt das Plug-in noch eine wichtige Information. Dependencies werde normalerweise über die `group:artifact`-Koordinaten (GA-Koordinaten) referenziert (zum Beispiel `org.apache.commons:commons-lang3`). Darüber weiß Gradle, wo ein Jar und zugehörige Metadaten physisch in einem Repository zu finden sind. In den `module-info.java`-Dateien verwenden wir jedoch die Modul-Namen zur Identifikation (zum Beispiel `org.apache.commons.lang3`). Beides sind eindeutige Identifier für das gleiche Modul, die sich in der Regel zwischen Versionen nicht ändern. Das heißt, wenn die beiden Identifier bekannt sind, können sie ineinander übersetzt werden. Leider sind bislang die Modul-Namen nicht in den Metadaten in den Repositorys indiziert (sie sind in den Jars versteckt). Darum kann man ein Repository nicht nach `org.apache.commons.lang3` fragen, sondern muss stattdessen `org.apache.commons:commons-lang3` verwenden.

Das `org.gradlex.java-module-dependencies`-Plug-in löst diese Problematik, indem es Mappings zwischen Modul-Name auf GA-Koordinaten für viele auf Maven Central verfügbare Module enthält. Diese Liste wird mit neuen Releases des Plug-ins erweitert. Sollte das Plug-in einen Modul-Namen noch nicht kennen, informiert es darüber in einer Warnung. Das fehlende Mapping kann dann zentral im Convention-Plug-in ergänzt werden *(siehe Listing 5; Zeilen 16/17)*.

Durch dieses zentrale Mapping von Modul-Namen auf GA-Koordinaten kann man an allen anderen Stellen – also in `module-info.java`-Daten *(siehe Listings 1-3)* und bei der Versions-Definition in der `settings.gradle.kts` *(siehe Listing 6; Zeilen 9-11)* – mit den Modul-Namen arbeiten und muss sich bei der täglichen Entwicklung nicht mit zwei unterschiedlichen Identifiern für das gleiche Modul auseinandersetzen.

Mit diesem Setup können wir jetzt unsere Java-Module mit Gradle kompilieren und unsere Anwendung ausführen. Alle Features der Abhängigkeiten-Verwaltung von Gradle können nun auf die in den `module-info.java`-Dateien deklarierte Abhängigkeiten angewendet werden. Insbesondere kann das `com.autonomousapps.dependency-analysis`-Plug-in *[2]* verwendet werden, um zu prüfen, ob die `requires`-Detektiven dem entsprechen, was tatsächlich vom Source-Code im entsprechenden Modul benötigt wird.

Nachtrag 1 (von 2): Third-Party Libraries ohne module-info verwenden

Eine aktuelle Herausforderung ist, dass nicht alle Third-Party-Jars bereits `module-info.class`-Dateien beinhalten. Java selbst bietet hier Mechanismen (Automatic-Modules), um auch diese Jars als Zwischenlösung in einem Modul-Setup zu nutzen. Die Erfahrung zeigt jedoch, dass dadurch viele Mechanismen des Modul-Systems ausgehebelt werden, was oft zu unerwartetem Verhalten führt. Außerdem sind einige der neueren Java-Tools (beispielsweise `jlink`) dann nicht nutzbar.

Hier hilft Gradle mit dem `org.gradlex.java-module-dependencies`-Plug-in *[3]* aus. Dies ermöglicht, die `module-info.class`-Datei nachzurüsten, nachdem ein Jar aus einem Repository geladen wurde. Dabei kann man bei übersichtlichen Modulen ohne Abhängigkeiten – wie `commons-lang3` – alle Packages exportieren lassen *(siehe Listing 5; Zeilen 21/22)*, ohne die `module-info.class` im Detail beschreiben zu müssen – was aber auch möglich ist, wenn nötig. Zusätzlich kann man das Ecosystem unterstützen, indem man bei Open-Source-Projekten hilft, die `module-info` in zukünftigen Releases zu ergänzen. Jars mit `module-info.class` können übrigens weiterhin als normale Jars in Java-Projekten ohne Module verwendet werden. Java ignoriert die Datei dann.

Nachtrag 2 (von 2): Testen von Modulen

Aber auch beim Testen im Java-Modulsystem sind ein paar Dinge zu beachten. Traditionell werden beim Testen weitere Bibliotheken hinzugefügt – etwa JUnit. Die Klassen, die dort mitkommen, sind für die normale Java-Laufzeit nicht von Produktionscode-Klassen zu unterscheiden und können theoretisch Verhalten beeinflussen. Im Modulsystem können Tests in separaten Java-Modulen mit eigener `module-info.java` definiert werden. Diese Test-Module sind dann für die Produktions-Code-Module nicht sichtbar, wodurch die Test-Laufzeitumgebung näher an der Produktions-Laufzeitumgebung ist.

Dies kann aber auch zu Problemen führen, wenn man Unit-Tests schreiben möchte, die direkt auf Methoden zugreifen, die sich in private Packages befinden. Für solche Fälle erlaubt Java das sogenannte Module-Patching, bei dem ein existierendes Modul um weitere Klassen erweitert wird. Dabei nimmt man bewusst in Kauf, dass das getestete Modul nicht exakt dem ausgelieferten Modul entspricht. Das Patching muss sowohl beim Kompilieren als auch Ausführen der Tests beachtet werden. Das `org.gradlex.java-module-testing`-Plug-in *[4]* erlaubt es, dies in Gradle übersichtlich zu konfigurieren *(siehe Listing 5; Zeile 26-32)*.

Zusammenfassung

Obwohl das Java-Modulsystem schon seit einigen Jahren existiert, wird es erst jetzt stärker in der Praxis eingesetzt. Dies hängt auch mit einer eher unbefriedigenden Tool-Landschaft zusammen. Benutzt man das Modulsystem mit reinem Gradle (oder Maven) ohne zusätzliche Plug-ins, wird die Projektstruktur oft noch unübersichtlicher als zuvor und die Vorteile des Modulsystems verpuffen gegenüber neuen Problemen bei der Abhängigkeitenverwaltung und mehr Maintenance-Overhead.

Das in diesem Artikel beschriebene Projekt-Setup verbindet das Modulsystem so mit Gradle, dass deren Stärken kombiniert werden. So entsteht ein eleganteres Setup mit weniger Build-Tool-Spezifika und präzisen Modul-Sichtbarkeiten. Dies kann man nutzen, um ein Java-Projekt mit klar isolierten Modulen und Schnittstellen aufzusetzen. Das gezeigte Setup folgt den aktuellen Standards für gute Gradle-Projektstrukturen. Die konkrete Anbindung an das Modulsystem ist weniger verbreitet, wurde jedoch bereits von einigen Projekten, die ich begleite, erfolgreich eingeführt. Unter *[5]* findet sich ein komplettes Beispielprojekt und Links zu weiteren Ressourcen zum Thema.

Quellen

[1] *https://github.com/gradlex-org/java-module-dependencies*
[2] *https://github.com/autonomousapps/dependency-analysis-android-gradle-plugin*
[3] *https://github.com/gradlex-org/extra-java-module-info*
[4] *https://github.com/gradlex-org/java-module-testing*
[5] *https://github.com/jjohannes/java-module-system*

Dr. Jendrik Johannes
jendrik@onepiece.software

Dr. Jendrik Johannes macht seit vielen Jahren Dinge mit Software – meist im Java-Umfeld. Besonderes Interesse hat er an Technologien, die effizientere Softwareentwicklung fördern. Er war für mehrere Jahre Teil des Entwicklerteams von Gradle. Jetzt unterstützt er Teams dabei, ihre Produktivität durch Automatisierung zu erhöhen und teilt seine Learnings auf YouTube und GitHub.

LUFTHANSA GROUP
DIGITAL HANGAR

Your code reaches 12.000 meters.

_You are a digital pioneer? Perfect!

We develop solutions for over 145 million passengers that travel on our 5 airline brands every year to enrich the entire travel experience. Sounds good? Join us now!

Apply now! → digitalhangar.aero/careers

Eigenständige Java-Applikationen erstellen

Christian Heitzmann, SimplexaCode AG

Java-Applikationen werden normalerweise direkt als Bytecode (also .class-Dateien) paketiert und verteilt. Das ist aber oft nicht wünschenswert. Zum einen lässt sich der Quellcode – und damit die Programmlogik – relativ einfach aus dem Bytecode rekonstruieren, zum anderen benötigt der Endanwender immer eine installierte Java Virtual Machine. Dieser Artikel zeigt auf, welche Möglichkeiten es heute gibt, Java-Applikationen zu erstellen, die eigenständig lauffähig sind.


+
</ />
JAVA
jlink
CONNECTED

Die Motivation zu diesem Artikel hat sicher ein Stück weit mit meiner persönlichen Java-Geschichte zu tun: Vor ziemlich genau 20 Jahren habe ich in der damaligen Firma, in der ich während eines Zwischenjahres angestellt war, nach ausgiebigen Recherchen den „kühnen" Vorschlag unterbreitet, für künftige Lösungen doch besser auf Java zu setzen (damals noch in der Version 1.4). Im Bereich der Verkehrstechnik wurden in neuen öffentlichen Ausschreibungen nämlich zunehmend Internettechnologien gefordert. Für ein Unternehmen, das bislang seine Verkehrsrechner auf „Bit- und Byte-Ebene" in Assembler und in proprietären Intel-Hochsprachen programmiert hat, war das natürlich absolutes Neuland.

Schnell musste der Vergleich her, ob Java-Programme gleich gut performen wie C/C++-Programme – eine unnötige Diskussion, die mich in meiner Laufbahn noch weitere fünf Jahre verfolgen sollte, heute aber Gott sei Dank obsolet geworden ist. Die Gründe dafür erfahren Sie im Laufe dieses Artikels.

Ebenfalls war der Wunsch, Java-Programme gleich „bequem" verpacken zu können, wie es C/C++-Compiler tun – im Idealfall in Form einer einzigen `.exe`-Datei. Dieses Anliegen war durchaus berechtigt, nicht nur in der damaligen Firma, sondern auch in vielen anschließenden Projekten, in denen ich mir so etwas gewünscht hätte. Umso überraschender ist es, dass es weitere 15 Jahre dauerte, ehe sich Oracle dieses Anliegens offiziell angenommen hat.

Geschichte

In den Jahren 2003 und 2004 habe ich mich also schon intensiv mit der Erstellung eigenständiger Java-Applikationen beschäftigt.

Marktführer dazumal war ganz klar Excelsior JET, ein Native Compiler, der aus Java-(SE-)Code direkt ausführbare Programme für Windows, Mac und Linux generieren konnte *[1]*. 2018 besaß ich noch eine Lizenz, ehe 2019 die russische Firma aus Nowosibirsk nach 20 Jahren ganz überraschend und kurzfristig ihre Tätigkeit einstellte. Über die Gründe wurde offiziell nie etwas bekannt. Es kann nur spekuliert werden, dass die GraalVM – um die es nachher in diesem Artikel natürlich auch geht – deren Geschäft den Todesstoß verpasst hat.

Als freien nativen Compiler gab es den GNU Compiler for Java (GCJ) *[1]*, der nach meiner Erinnerung aber nie richtig funktioniert hat, was vorrangig seiner unvollständigen (eigenen!) Implementierung der Standardklassenbibliotheken (GNU Classpath) geschuldet war. 2016 wurde der GCJ komplett eingestellt.

Persönlich sehr überzeugt hat mich dazumal hingegen Borland JBuilder X, der in der Developer- und Enterprise-Variante über einen Wrapper verfügte, mit dem sich Java-Applikationen in Windows, Mac und Linux (und sogar Solaris) via Doppelklick auf eine einzige Executable-Datei ausführen ließen. Das hat auch immer sehr gut funktioniert. Etwa im gleichen Zeitraum wurde das freie Eclipse zunehmend populärer, das aber leider nie ein derartiges Feature bot. Spätere JBuilder-Versionen basierten dann auf Eclipse, und Borland wurde von Embarcadero Technologies aufgekauft. Die letzte JBuilder-Version erschien 2009.

Der Fairness halber muss erwähnt werden, dass der JBuilder-Wrapper lediglich ein kleines Executable enthielt, das den `java`-Aufruf für das jeweilige Betriebssystem tätigte. Java musste beim Anwender also bereits installiert sein. Der Wrapper hat dann die in seinem Executable-Archiv enthaltenen `.class`-Dateien temporär im Arbeitsspeicher extrahiert und der JVM beim Aufruf mitgegeben. Wahrlich keine Hexerei, da sich eine solche Lösung mit überschaubarem Aufwand auch selber realisieren ließe, aber durchaus bequem, stabil und damals seiner Zeit voraus.

Eigenständige Java-Applikationen

Unter eigenständig lauffähigen Java-Applikationen soll verstanden werden, dass der (End-)Benutzer keine eigene Java Virtual Machine installiert haben muss, um sie auszuführen.

An sich mag eine solche Installation kein großes Problem (gewesen) sein. In den ersten Java-Versionen (1.4 bis 8) war es trivial, wenn auch wegen des einnistenden Java Web Start und ständiger Hintergrund-Updates meines Erachtens nie ganz nebenwirkungsfrei. Ebenfalls haftete Java dazumal immer noch leicht der Ruf eines „Sicherheitsrisikos" an, was wohl den früheren Java Applets geschuldet war (die aber heute zum Glück alle aus dem Verkehr gezogen sind). Mit der Umstellung des Oracle-Lizenz- und Versionierungsmodells ab Java 9, bei der technisch wie politisch kein Stein mehr auf dem anderen blieb, wurde das Auffinden und Installieren eines korrekten JDKs geradezu lästig. Mittlerweile scheint sich das mit einem halben Dutzend unkomplizierter OpenJDK-Builds, wie zum Beispiel Adoptium Eclipse Temurin (früher AdoptOpenJDK), und dem sich eingespielten Releasesystem wieder gelegt zu haben.

Das ändert aber nichts daran, das ausgelieferte Java-Applikationen oftmals auf eine ganz bestimmte JDK-Version setzen. Die Abwärtskompatibilität von Java funktionierte zwar rückblickend betrachtet immer ganz gut, aber nicht perfekt. Das mag wohl der Grund sein, wieso jede nennenswerte Java-Desktop-Applikation, die ich kenne, praktisch immer auch ihre eigene JDK-Runtime mitliefert.

Nicht Teil des vorliegenden Artikels soll die gesamte (Native-)Cloud-Thematik sein. Es leuchtet ein, dass sich die Anforderungen an eine Microservice-Architektur, namentlich minimaler Speicherplatzbedarf, hochoptimierter Code sowie schnelles Aufstarten und Herunterfahren, bevorzugt mit nativ vorkompilierten Java-Applikationen umsetzen lassen *[2]*.

Application Images mit jlink

Mit Java 9 wurde das gesamte JDK modularisiert. Gleichzeitig erhielt das Kommandozeilentool `jlink` *[3][4][5]* Einzug in ebendieses. Damit ist es möglich, ein eigenes Application Image zu erstellen (in der Fachsprache: zu linken), das *nur* die für die Ausführung notwendigen Module des JDKs enthält. Eine typische JDK-Installation umfasst heute zwischen 300 und 400 MB. Die meisten Module, wie AWT, Swing oder CORBA, werden nicht benötigt, sodass es mehr als nachvollziehbar ist, wenn zur Ausführung beim Endbenutzer nicht jeweils der ganze Koloss mitgeliefert werden soll. Dank des Modulsystems sind ja alle (transitiven) Abhängigkeiten klar definiert. *Listing 1* zeigt einen typischen `jlink`-Aufruf.

- Mit `--module-path` werden alle Modulpfade angegeben; hier die Module des Eclipse Temurin Java-19-JDKs sowie – mit `:` getrennt (unter Windows mit `;`) – die eigenen Applikationsmodule,

```
jlink --module-path ~/Library/Java/JavaVirtualMachines/temurin-19/Contents/Home/jmods/:./out/production/
      --add-modules slideshow
      --launcher play=slideshow/ch.simplexacode.slideshow.SlideShowMain
      --output slideshow_executable
```

Listing 1

die sich hier im Beispiel im Ausgabeverzeichnis `out/production` befinden.
- `--add-modules` gibt die eigentlichen Namen der einzubindenen Applikationsmodule an. In diesem Beispiel handelt es sich einzig um das Modul `slideshow`.
- Mit `--launcher` wird der Linker angewiesen, einen expliziten Launcher anzulegen, also eine ausführbare Datei, die sich einfach auf der Kommandozeile oder mit Doppelklick ausführen lässt. In diesem Beispiel soll dieser Launcher `play` heißen und auf die Main-Klasse `SlideShowMain` im Package `ch.simplexacode.slideshow` des Moduls `slideshow` verweisen. Fehlt ein solcher Launcher, muss die Applikation inkl. aller Pfade separat auf Kommandozeile mit `java` gestartet werden, womit sich das Pfad-Gepfriemel zum Aufrufer verlagert.
- `--output` gibt den Namen des Verzeichnisses an, in dem das Application Image erzeugt wird; hier `slideshow_executable`.

Das Artefakt von `jlink` ist ein Verzeichnis, bestehend aus vielen Dutzend Dateien und diversen Unterverzeichnissen. Wurde ein Launcher erzeugt, so befindet sich dieser im Unterverzeichnis `bin/`. Die Applikation lässt sich also via Doppelklick auf beziehungsweise Kommandozeilenaufruf von `slideshow_executable/bin/play` starten, selbst wenn auf dem ausführenden Rechner kein JDK installiert ist. Das JDK ist ja Teil des Application Images (oder vielmehr des Application Directories).

Mit den Optionen `--compress=2`, `--strip-debug`, `--no-header-files` und `--no-man-pages` lassen sich bezüglich des Speicherplatzbedarfs noch deutliche Optimierungen erzielen. 40 % Einsparung sind je nach Anwendung problemlos möglich.

`jlink` ermöglicht es auch, Application Images für andere Betriebssysteme als das eigene zu erzeugen. Letztlich muss hierzu nur das JDK des Zielsystems heruntergeladen und im `--module-path` eingebunden werden. `jlink` erkennt, dass es sich um das JDK eines anderen Betriebssystems handelt, und erstellt die dazugehörigen Binärdateien *[6]*.

Für weitergehende Informationen sei vor allem an *[4]*, *[6]* und die vielen einfach auffindbaren Online-Quellen verwiesen.

Installationspakete mit jpackage

Unsere Anforderung, beim Anwender kein separat installiertes JDK zu verlangen, können wir jetzt mit `jlink` erfüllen. Nicht schön hingegen ist das Deployment unserer Applikation als ganze Verzeichnisstruktur statt einer einzigen Datei.

Mit `jpackage` steht seit Java 16 ein Kommandozeilentool zur Verfügung, das Installationspakete in den üblichen Formaten für die Betriebssysteme Windows, Mac und Linux erstellt *[7]*. Für Windows sind das wahlweise `.exe` oder `.msi`, für Mac `.pkg` oder `.app` innerhalb eines `.dmg`, und für Linux `.deb` oder `.rpm`. Ein „Cross-Packaging" ist aber nicht möglich, das heißt, `jpackage` muss – im Gegensatz zu `jlink` – immer auf dem Zielbetriebssystem ausgeführt werden.

Ob die so deployten Applikationen beim Anwender wirklich *installiert* werden müssen, ist nicht sicher. Zumindest unter Mac (meinem Heimsystem) kann ich die so erzeugten Applikationen auch direkt aus dem `.dmg`-Image starten. Vorsicht ist geboten, wenn die Anwendung Konsolenausgaben produziert oder Aufrufparameter erfordert. Zumindest auf Mac ist ein derartiger „Low-Level-Zugriff" deutlich harziger als der direkte Umgang mit der Launcher-Binärdatei, wie sie `jlink` aus dem vorherigen Abschnitt erzeugt.

Der Aufruf von `jpackage` mit Standardoptionen ist relativ unspektakulär und selbsterklärend, wie *Listing 2* an unserer bereits bekannten hypothetischen Slideshow-Applikation zeigt. Für weitere Details sei auf *[7]* verwiesen, das auch die verschiedenen Varianten bei modularer und nicht modularer Applikation sowie gegebenen `.jar`-, `.jmod`- und reinen Source-Dateien behandelt.

Native Images mit GraalVM

Der „heilige Gral" in Sachen eigenständiger Java-Applikationen ist natürlich die neue GraalVM. Diese muss zuerst separat von *[8]* heruntergeladen und installiert werden. Es gibt sie in einer kostenlosen Community und einer kostenpflichtigen Enterprise Edition. Letztere soll für Java zirka 20 % und für dynamische Sprachen etwa 50 % schneller sein, einen nur halb so großen Speicherbedarf aufweisen und diverse Sicherheitsmerkmale haben *[2]*. Am interessantesten dürfte aber die Profile Guided Optimization (PGO) sein, auf die im Abschnitt *Performanzaspekte* kurz eingegangen wird.

Nach der Installation der eigentlichen GraalVM muss mittels `gu install native-image` noch die Erweiterung für die Erzeugung nativ kompilierter Images hinzugefügt werden. `gu` ist der hauseigene **G**raalVM **U**pdater. Am Schluss sollte auch der Kommandozeilenbefehl `native-image` zur Verfügung stehen.

Der Aufruf von `native-image` erfolgt analog einem Aufruf von `java`

```
jpackage --module-path ~/Library/Java/JavaVirtualMachines/temurin-19/Contents/Home/jmods/:./out/production/
         --module slideshow/ch.simplexacode.slideshow.SlideShowMain
         --name slideshow_installer
         --app-version 2023.06
```

Listing 2

[9]. Bleiben wir bei unserer SlideShow-Applikation, dann können wir mit dem Befehl wie in *Listing 3* gezeigt ein natives Kompilat erzeugen.

- In `--module-path` genügt es, nur noch auf die Module der eigenen Applikation zu verweisen. Die GraalVM enthält quasi ihr eigenes JDK, das implizit zur Verfügung steht. Allerdings arbeitet GraalVM (Stand: Juni 2023) noch mit Java 17 und akzeptiert keine Applikationsklassen, deren Bytecodes für eine neuere Java-Version kompiliert wurden.
- `-m` gibt die Main-Klasse an.
- `-o` bestimmt den Namen der einen und einzigen ausführbaren Datei.

Seien Sie nicht überrascht, wenn der Kompiliervorgang sehr lange dauert. Eine überschaubare Swing-Applikation hat bei meinen Tests bereits 30 Sekunden Zeit benötigt.

Stichworte Swing und AWT: Hier offenbart sich bereits ein erster Bug. Eine AWT- (und damit auch Swing-)Applikation stürzt beim Aufruf direkt ab, was offenbar mit den Schriften und wohl auch etwas mit Mac zu tun hat *[10][11]*.

Über die GraalVM und ihre Native Images wurde schon viel geschrieben, vor allem hier in Java aktuell *[12][13][14]*, aber auch in JavaSPEKTRUM *[15][16][17]*. Mit *[2]* ist über sie sogar ein ganzes Buch erschienen. Von daher nur ganz kurz die wichtigsten Punkte:

GraalVM ist – anders als ihr Name vermuten lässt – nicht nur eine Virtual Machine, sondern auch ein Compiler (Graal Compiler), der nebst Java nicht nur andere JVM-Sprachen, sondern (via Truffle) sogar ganz andere Programmiersprachen wie Python oder JavaScript unterstützt. Genau genommen kommt Graal nicht nur mit einer, sondern sogar mit zwei virtuellen Maschinen daher: die zweite heißt SubstrateVM.

Der Graal Compiler kann eine sogenannte Ahead-of-Time-Kompilierung (AOT) durchführen. Dazu analysiert er ausgehend von der `main`-Methode alle erreichbaren Codestellen der eigenen Applikation, des JDKs und aller Drittmodule, und entfernt alles, was nicht aufrufbar ist (Dead Code Elimination). Den Rest übersetzt er inklusive SubstrateVM in Maschinensprache. Es ist ebenso möglich, Static Native und Shared Libraries zu bauen.

Mit der statischen Kompilierung kommt es zu einigen Einschränkungen des an sich hochdynamischen Javas. Es ist theoretisch nicht möglich, Klassen dynamisch nachzuladen, was vor allem einen Einfluss auf Spracheigenschaften wie Reflection, Classloader und dynamische Proxies hat – und damit verbunden auch auf sie basierende Frameworks wie zum Beispiel Spring (Boot) (wobei mit Spring Framework 6 neu auch Native Images unterstützt werden *[18]*). Mehr zu den Hintergründen ist unter anderem in *[13]* zu finden.

Decompiler und Obfuscator

Javas `.class`-Dateien lassen sich prinzipiell sehr einfach dekompilieren. Nicht selten lässt sich daraus der gesamte Quellcode rekonstruieren *[19]*. Wer also auf den Schutz seines geistigen Eigentums bedacht ist, sollte unbedingt Vorkehrungen treffen.

Die Artefakte aus `jlink` und `jpackage` (das ja auf `jlink` basiert) enthalten eine große Datei namens `modules`, in der die Klassendateien der Module im JIMAGE-Format gespeichert sind *[5][20]*. Diese lassen sich einfach extrahieren und bieten daher *keinen* Schutz vor Dekompilierung.

Native Kompilate aus der GraalVM sehen da schon besser aus *[21]*. Dekompilierung ist immer ein Abwägen zwischen Aufwand und Nutzen, und das kippt bei ausführbaren Binärdateien ganz klar auf die Seite des Aufwands.

Mit einem sogenannten Obfuscator lässt sich der Java- und damit der Byte-Code „verschleiern". Ein Obfuscator benennt z. B. alle (nicht öffentlichen) Klassen, Methoden, Variablen etc. in mühsame Namen wie `C1`, `C2`, `C3`, `m1`, `m2`, `m3`, `v1`, `v2`, `v3` etc. um, entfernt sämtliche Debug-Informationen und verkompliziert teilweise einfache Sachen wie String-Konstanten durch sie generierende Methodenaufrufe *[19]*. Der wohl bekannteste Obfuscator ist ProGuard *[22]*.

Performanzaspekte

Die Java Virtual Machine hat sich über die letzten 28 Jahre als gelungener Schachzug herausgestellt und mehr als bewährt *[23]*. Obwohl der Code „halb" interpretiert wird (via Zwischenstufe Bytecode), haben Compiler-Konzepte wie Just-in-time (JIT) oder HotSpot für einen Geschwindigkeitszuwachs gesorgt, der denen von statisch kompilierten Sprachen streckenweise überlegen ist. Die dynamische JVM kann nämlich während der Laufzeit eines Programmes erkennen, welche Teile des Programms besonders oft durchlaufen werden, und diese Teile („Ausführungspfade") dann besonders hoch optimieren. Außerdem kennt die JVM die Hardware, auf der sie läuft, und kann so eine Just-in-Time-Kompilierung genau auf diese Hardware zugeschnitten durchführen, zum Beispiel, um so von diversen Befehlssatzerweiterungen zu profitieren *[2][24]*.

Solche dynamischen Optimierungen können die statischen Kompilate der GraalVM nicht vornehmen. Die Meinungen gehen auseinander, ob Native Images schneller oder langsamer sind *[2]*. Die wahrscheinlich einzig korrekte Antwort darauf ist: „Es kommt darauf an." Wem Performanz wirklich sehr wichtig ist (Ich darf an den Spruch von Donald Knuth erinnern: „Premature optimization is the root of all evil."), der sei an meinen Fachartikel *[25]* zu den Grundzügen der Laufzeitmessung in Java verwiesen.

Die GraalVM *Enterprise Edition* bietet eine sogenannte Profile-Guided Optimization (PGO) an. Dort wird zuerst ein instrumentiertes AOT-Kompilat erzeugt, das Messroutinen für das Ausführungsverhalten

```
native-image --module-path ./out/production/
             -m slideshow/ch.simplexacode.slideshow.SlideShowMain
             -o slideshow_native
```

Listing 3

enthält. Anschließend wird die Anwendung (eventuell für längere Zeit) laufen gelassen. Abschließend wird erneut ein AOT-Kompilat erzeugt, diesmal unter Einbezug der vorher ermittelten Profildaten, sodass der Graal-Compiler letztendlich ein hochoptimiertes Native Image erstellen kann *[2][9]*.

Fazit

Es gibt heute mehrere Möglichkeiten, eigenständige Java-Applikationen zu erstellen: mittels `jlink`, `jpackage` oder *GraalVM*. Alle Lösungen haben gemeinsam, dass sie ihre eigene Ausführungsumgebung mitbringen, und der Benutzer so keine separate JVM installieren muss. Unterschiede gibt es bezüglich des Deployment-Komforts (von ganzen Verzeichnisbäumen über Launcher und Installer bis hin zu einer einzigen Executable-Datei), aber auch hinsichtlich des Schutzes vor Dekompilierung des darunterliegenden (Byte-)Codes. Performanz sollte kein ausschlaggebendes Kriterium sein, sondern bedarf im Einzelfall einer genauen, systematischen Untersuchung. Ebenfalls sind die benutzten Frameworks zu berücksichtigen, und mit welchen Kompiliermöglichkeiten sich diese vertragen. Nicht ohne Grund sind ja neue Frameworks wie Quarkus entstanden, die dem Native-Ansatz vor allem auf der Cloud besser gerecht werden.

Referenzen

[1] C. Ullenboom, Java ist auch eine Insel, 10. Auflage, Galileo Computing, 2010

[2] A B V. Kumar, Supercharge Your Applications with GraalVM, Packt Publishing, 2021

[3] M. Inden, Java – Die Neuerungen in Version 9 bis 14, dpunkt.verlag, 2020

[4] M. Hunger, Der Java-Linker jlink, in: JavaSPEKTRUM, 2/2022, SIGS DATACOM, 2022

[5] Oracle University, Java SE: Exploiting Modularity and Other New Features, Oracle, 2018

[6] Dev.Java, Creating Runtime and Application Images with JLink, *https://dev.java/learn/jlink/*

[7] Packaging Tool User's Guide, *https://docs.oracle.com/en/java/javase/20/jpackage/*

[8] GraalVM, *https://www.graalvm.org*

[9] GraalVM Native Image Quick Reference v2, *https://www.graalvm.org/latest/docs/quick-references/*

[10] GitHub, Using Java AWT Fonts gives unsatisfiedlinkerror while running native image, *https://github.com/oracle/graal/issues/2729*

[11] GitHub, no awt in java.library.path, *https://github.com/oracle/graal/issues/2842*

[12] K. Vardanyan, Eine JVM für die Cloud: die GraalVM, in: Java aktuell, 1/2020, DOAG, 2020

[13] B. Müller, Native Images mit GraalVM, in: Java aktuell, 2/2020, DOAG, 2020

[14] B. Müller, Native Images mit GraalVM: Reflection, in: Java aktuell, 1/2021, DOAG, 2021

[15] M. Hunger, Polyglot Operations in Java mit GraalVM, in: JavaSPEKTRUM, 1/2019, SIGS DATACOM, 2019

[16] M. Hunger, Vorcompilierte Programme mit GraalVM AOT, in: JavaSPEKTRUM, 2/2019, SIGS DATACOM, 2019

[17] M. Vitz, Cloud native Java-Anwendungen mit Quarkus, in: JavaSPEKTRUM, 3/2019, SIGS DATACOM, 2019

[18] M. Simons, Native Image in Spring Framework 6, in: iX, 12/2022, Heise Medien, 2022

[19] C. Ullenboom, Java SE 9 Standard-Bibliothek, 3. Auflage, Rheinwerk Verlag, 2018

[20] M. Debnath, How Modules Are Packaged in Java 9, *https://www.developer.com/design/how-modules-are-packaged-in-java-9/*

[21] GitHub, Is it possible to decompile the executable and see the original code?, *https://github.com/oracle/graal/issues/4003*

[22] GitHub, Guardsquare ProGuard, *https://github.com/Guardsquare/proguard*

[23] M. Stal, Java rocks! – Eine Liebeserklärung, in: JavaSPEKTRUM, 4/2020, SIGS DATACOM, 2020

[24] C. Ullenboom, Java ist auch eine Insel, 16. Auflage, Rheinwerk Verlag, 2022

[25] C. Heitzmann, Performanzanalysen in Java – Teil 1: Java Microbenchmarks, in: JavaSPEKTRUM, 4/2020, SIGS DATACOM, 2020, *https://link.simplexacode.ch/cj8e*

Christian Heitzmann

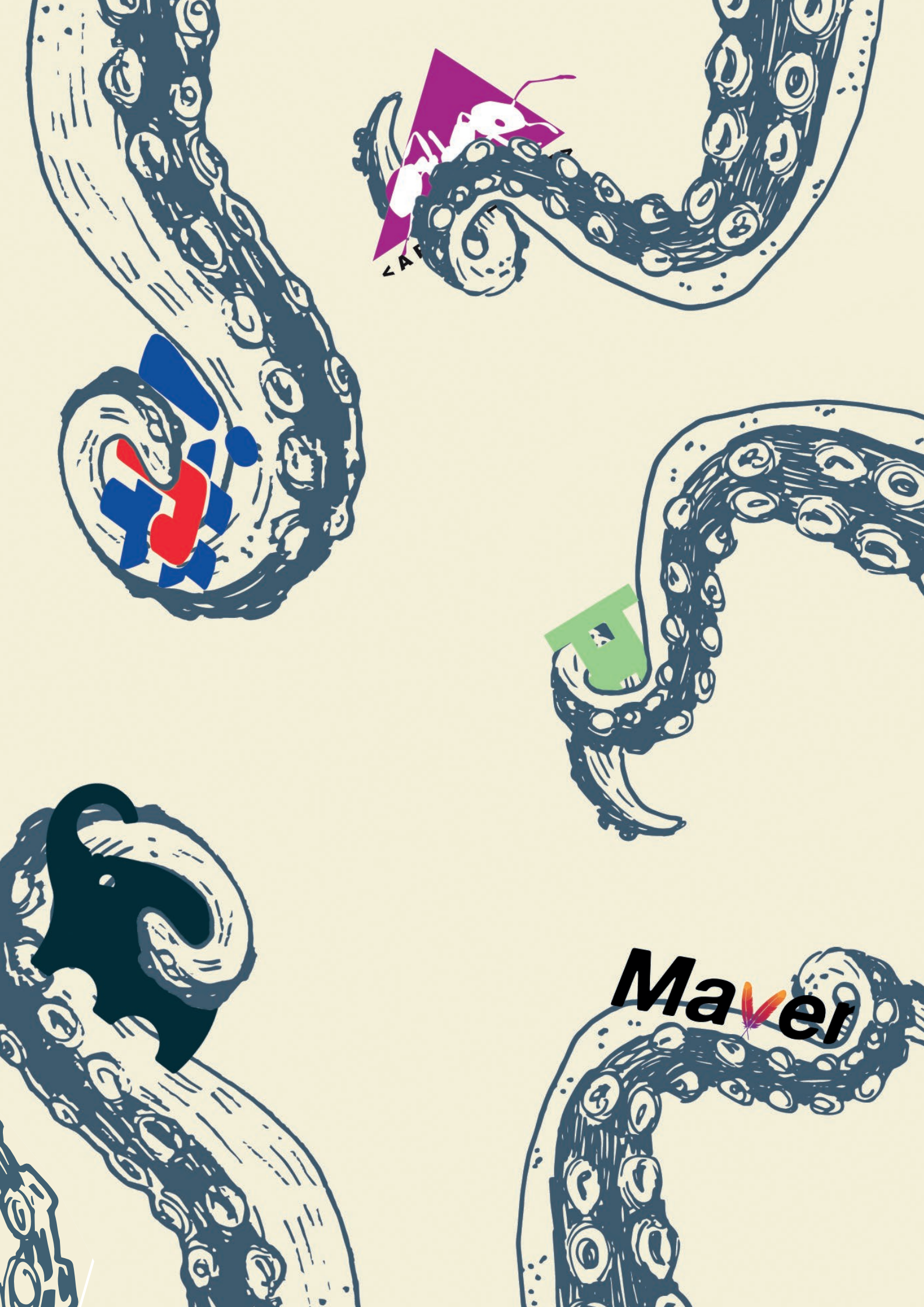
christian.heitzmann@simplexacode.ch

Christian Heitzmann ist Java-, Python- und Spring-zertifizierter Softwareentwickler mit einem CAS in Machine Learning und Inhaber der SimplexaCode AG in Luzern. Er entwickelt seit über 20 Jahren Software und gibt seit über 12 Jahren Unterricht und Kurse im Bereich der Java- und Python-Programmierung, Mathematik und Algorithmik. Als Technical Writer dokumentiert er Softwarearchitekturen für Unternehmen.

Gum

Marcus Fihlon

*Mittlerweile gibt es fast schon so viele Build-Tools wie JavaScript-Frameworks. Wer kann sich da schon so genau merken, bei welchem dieser Tools welcher Kommandozeilenparameter welche Art von Build startet? Besonders, wenn man viel mit Open-Source-Projekten zu tun hat, die unterschiedliche Build-Tools einsetzen, kommt es schnell zu einem geistigen Durcheinander. Passiert es dir auch manchmal, dass du in einem Gradle-Projekt ein Maven-Kommando absetzt oder versehentlich die Optionen von Maven und Gradle vertauscht? Hierbei hilft Gum! Ursprünglich für **G**radle **u**nd **M**aven stehend, erkennt Gum mittlerweile automatisch, ob das Projekt auf Gradle, Maven, Bach, JBang oder Ant basiert und führt den entsprechenden Befehl aus.*


Maven

Eigentlich könnte dieser Artikel schon mit dem vorherigen Absatz zu Ende sein. Viel mehr gibt es nicht über Gum zu erzählen. Es ist ein einfaches, kleines, sehr nützliches Hilfsmittel, um sich weniger im Dschungel der Build-Tools zu verfangen. Im Folgenden zeige ich dir, wie du in wenigen raschen Schritten Gum installieren, konfigurieren und einsetzen kannst.

Installation

Gum ist ein Open-Source-Projekt und wurde in Go entwickelt. Der Quelltext steht auf GitHub *[1]* zur Verfügung. Auf der Release-Seite *[2]* gibt es fertig kompilierte Artefakte für Windows, macOS und Linux. Diese kannst du einfach herunterladen, auspacken und die ausführbare Datei an einen Ort deiner Wahl kopieren. Es macht Sinn, dass der Ort, für den du dich entscheidest, im Suchpfad für Anwendungen enthalten ist, sodass du Gum an der Kommandozeile von überall aufrufen kannst.

Unter macOS steht Gum auch über den Homebrew-Paketmanager *[3]* zur Verfügung, dessen Einsatz ich allen Mac-Nutzern wärmstens empfehlen kann. Hier reicht ein einfacher Aufruf von `brew install kordamp/tap/gum` für die komplette Installation.

Wenn du Linux verwendest, kannst du Gum über Snapcraft *[4]* installieren. Dies funktioniert bei Ubuntu und vielen seiner Derivate. Bei anderen Distributionen muss die Unterstützung von Snapcraft gegebenenfalls erst separat aktiviert werden. Für die Installation von Gum reicht ein einzelner Befehl aus: `sudo snap install --classic gum`.

Falls du Go *[5]* auf deinem System installiert hast, steht dir auch die Möglichkeit zur Verfügung, Gum direkt aus dem Quelltext zu kompilieren und automatisch zu installieren. Das lässt sich ebenfalls mit einem Einzeiler bewerkstelligen: `go install github.com/kordamp/gm`.

Für Windows wird Gum aktuell per Scoop *[6]* verteilt (an der Unterstützung von Chocolatey *[7]* wird gearbeitet): `scoop install main/gum`.

Konfiguration

Um mit Gum zu starten und es einzusetzen, ist keine besondere Konfiguration nötig. Die Standardeinstellungen sind in den meisten Fällen ausreichend. Daher gehe ich hier nur auf die wichtigsten Informationen zur Konfiguration ein.

Gum verwendet eine Konfigurationsdatei im TOML-Format *[8]*, das stark an die früher bekannten Windows-INI-Dateien erinnert. Diese Datei kann entweder im Home-Verzeichnis des Anwenders liegen (`%APPDATA\Gum\gm.toml` unter Windows beziehungsweise `$HOME/.gm.toml` unter macOS und Linux) oder im Projektverzeichnis mit dem Namen `.gm.toml`. Dabei haben alle Einstellungen im Projektverzeichnis Vorrang vor den Einstellungen im Home-Verzeichnis, die die Standard-Einstellungen überschreiben.

Die wichtigste Konfigurationsoption ist die Reihenfolge, in der das verwendete Build-Tool ermittelt wird. Falls mehrere verwendet werden, gewinnt das erste gefundene. Standardmäßig lautet die Reihenfolge: Gradle, Maven, Ant, Bach, JBang. Möchtest du diese Reihenfolge dauerhaft anpassen, modifiziere die Option `discovery` im Abschnitt `[general]`, wie in *Listing 1* gezeigt. Dort habe ich beispielhaft Maven eine höhere Priorität als Gradle eingeräumt.

```
[general]
discovery = ["maven", "gradle", "ant", "bach", "jbang"]
```

Listing 1: Anpassen der Reihenfolge zur Erkennung des Build -Tools

Im GitHub Repository *[1]* befindet sich in der Datei `README.adoc` eine komplette Konfigurationsdatei mit erklärenden Kommentaren aller zur Verfügung stehenden Konfigurationsoptionen.

Einsatz

Um Gum zu verwenden, musst du an Stelle der bisher verwendeten Befehle, wie beispielsweise `mvn` oder `gradle`, einfach nur `gm` schreiben. Das ist das Kommandozeilenprogramm von Gum. Es schaut im aktuellen Projekt nach, welches Build-Tool zum Einsatz kommt und „übersetzt" den Parameter (Goal beziehungsweise Task) entsprechend. So kannst du in einem Gradle-Projekt einfach den Befehl `gm verify` schreiben und Gum erkennt das Gradle-Projekt, übersetzt das Maven-spezifische `verify` zum Gradle-Equivalent `check` und führt es aus. Du brauchst keinen Gedanken mehr daran verschwenden, welches Tool zum Einsatz kommt und wie der Parameter dafür heißt.

Falls du ein Projekt hast, bei dem mehrere Build-Tools zum Einsatz kommen und die automatische Auswahl von Gum das falsche wählt, kannst du mittels eines Kommandozeilenparameters das zu verwendende Tool auch vorgeben:

- `gg` für Gradle
- `gm` für Maven
- `gb` für Bach
- `gj` für JBang
- `ga` für Ant

Ein weiterer großer Vorteil von Gum ist, dass es in jedem Unterverzeichnis des Projektes funktioniert. Gum verwendet dabei immer das Build-File aus dem Root-Verzeichnis des aktuellen Projektes. Falls dein aktuelles Projekt Build-Files auf mehreren Ebenen enthält, kannst du Gum mit dem Parameter `-gn` anweisen, das „örtlich nähere" Build-File zu verwenden. Befindest du dich gerade innerhalb eines Submodules, ist das Build-File des Submoduls näher am aktuellen Verzeichnis. So kannst du explizit nur dieses Submodul bauen. Selbstverständlich wird auch die direkte Angabe des Build-Files mit den von Gradle, Maven und Ant bekannten Parametern verstanden und an diese Build-Tools durchgeschleift.

Gum unterstützt ebenfalls die Verwendung der Wrapper von Gradle und Maven. Setzt dein Projekt einen dieser Wrapper ein, bekommt er höhere Priorität als das installierte Build-Tool. Auch hier ist so sichergestellt, dass sich ein mit Gum gestarteter Build stets so verhält, als hättest du den korrekten Befehl selbst eingetippt.

Probleme

Während meiner Arbeit mit Gum bin ich bisher nur auf ein Problem in Zusammenhang mit der *Z Shell* in Verbindung mit *Oh My Zsh* ge-

stoßen, wenn das *Git-Plug-in* aktiviert wird. Dieses registriert den Alias `gm` für `git merge` und ein Alias hat in der Shell immer Priorität. Es gibt zwei einfache Möglichkeiten, das Problem zu lösen:

1. Verzicht auf das Git-Plug-in
2. Umbenennen des Kommandozeilenprogramms von Gum

In meinem Fall habe ich mich für Variante 2 entschieden und das Kommandozeilenprogramm `gm` von Gum in `gum` umbenannt.

Fazit

Die Arbeit mit unterschiedlichen Build-Tools wird durch Gum deutlich vereinfacht. Du musst keinen Gedanken mehr daran verschwenden, wie die unterschiedlichen Goals/Tasks der einzelnen Tools heißen und welches im aktuellen Projekt im Einsatz ist. Wenn du, wie ich früher, beispielsweise versehentlich `gradle` statt `gradle check` oder `mvn build` statt `mvn verify` eingetippt hast, spielt das nun dank Gum keine Rolle mehr: Sowohl `gm verify` als auch `gm check` werden zur Laufzeit automatisch in die korrekte Syntax übersetzt.

Auch die Möglichkeit, den Build von jedem Unterordner aus zu starten, ist sehr nützlich. Gum gehört für mich damit zur Sammlung der vielen kleinen Helferlein, die den Alltag als Entwickler ein kleines bisschen angenehmer machen. Mein Dank geht an Andres Almiray nach Basel, der Gum vor gut drei Jahren ins Leben gerufen hat und seither aktiv betreut.

Quellen

[1] GitHub Repository: *https://github.com/kordamp/gm*
[2] Gum Releases: *https://github.com/kordamp/gm/releases*
[3] Homebrew Paketmanager: *https://brew.sh/*
[4] Snapcraft Universal Linux Packages: *https://snapcraft.io/*
[5] Go Download: *https://go.dev/dl/*
[6] Scoop Command-Line Installer: *https://scoop.sh/*
[7] Chocolatey Package Manager: *https://chocolatey.org/*
[8] Tom's Obvious, Minimal Language: *https://github.com/toml-lang/toml*

Marcus Fihlon
marcus@fihlon.swiss

Marcus arbeitet hauptberuflich als Agile Coach in Luzern und beschäftigt sich seit seiner Jugend mit der Entwicklung von Software nach Open-Source-Prinzipien. Er engagiert sich leidenschaftlich in verschiedenen Communitys und organisiert Vorträge, Workshops, Konferenzen und andere Veranstaltungen. Als Ausgleich findet man ihn oft auf einem seiner vielen Fahrräder in fernen Ländern die Welt erkunden.

Kotlin im Backend

Andreas Voigt, adesso SE

Kotlin ist eine der beliebtesten JVM-Sprachen nach Java selbst. Wenn man aber genauer hinsieht, handelt es sich dabei meist um Anwendungen im Android-Umfeld. Im Enterprise-Umfeld bei Backend-Applikationen ist Kotlin eher noch eine Randerscheinung. Dabei kann es auch gerade für solche Anwendungen eingesetzt werden.


Kotlin

Kotlin bietet viel Raum, in welchem Stil man konkret Programme gestaltet, da die Sprache sowohl funktionale als objektorientierte Programmierung unterstützt. Jedes Projektteam muss seinen eigenen Weg definieren, in welcher Ausprägung sie die Projektziele erreichen. Die Beschreibung meines eigenen Projekts soll dazu beisteuern, wie Teams ihren eigenen Programmierstil finden können.

Im Folgenden berichte ich über meine Erfahrungen mit Kotlin anhand eines Projekts, mit dem ich selbst Kotlin kennengelernt habe. Dies war ein langlaufendes Projekt, das sich über vier Jahre gestreckt hat. Ich hatte zu der Zeit bereits langjährige Java-Backend-Erfahrung. Bei der zu entwickelnden Anwendung handelt es sich um eine Verwaltungssoftware für Produkte und Kataloge, die als Plattform für Eigenentwicklungen bei verschiedenen Kunden eingesetzt wird. Das vorrangige Ziel war daher, leicht verständlichen und gut zu wartenden Code zu erstellen. Neue Entwickler mit und ohne FP-Hintergrund sollen sich schnell zurechtfinden. Technisch gesehen handelte es sich um eine klassische Cloud-Software mit REST-API. Wir orientierten uns bei der Entwicklung an den DDD-Prinzipien und verwendeten eine hexagonale Architektur *(siehe Abbildung 1)*. Dies ist ein Architekturmuster (von Alistair Cockburn), dass das Ports- und Adapter-Entwurfsmuster verwendet, um das System lose mit seinem Umfeld verbinden zu können. Hierbei wird ein System in drei Ringe aufgeteilt:

- Die innere Schicht (Domäne) enthält die Geschäftslogik der Anwendung
- Darüber liegen die Services, mit denen man auf die Domäne zugreift
- Die Adapter-Schicht verbindet das System mit der Außenwelt (DB, REST-Controller etc.)

Indem Drittsoftware nur in der äußeren Schicht verwendet wird, wird der eigentliche Geschäftslogikanteil frei von diesen Anhängigkeiten gehalten.

Die Programmiersprache Kotlin ist aus der mobilen Welt sehr bekannt und wird in Büchern und Artikeln bereits ausführlich beschrieben. Deshalb ist dieser Beitrag keine Einführung in die Programmiersprache an sich, sondern stellt einige Stärken von Kotlin, speziell im Bereich der Backend-Entwicklung, vor und soll zudem Interesse bei klassischen Java-Entwicklern wecken.

Der Tech-Stack

Oft ist das Erste, was man über ein Projekt erfährt, welche Technologien verwendet wurden. Natürlich richtet sich der Tech-Stack nach den Projekterfordernissen und der Architektur sowie den Kenntnissen der beteiligten Personen aus.

In unserem Projekt waren wir bei dem Einsatz von Drittsoftware sehr zurückhaltend. Denn auch Open-Source-Software ist nicht nur geschenkter Code, sondern es fallen in den meisten Fällen dennoch Kosten an. Beispielsweise muss jede eingebundene Bibliothek verstanden und gewartet werden. Es kann auch vorkommen, dass Open-Source-Software von ihren Entwicklern aufgegeben und nicht weiterentwickelt wird. In diesem Fall muss zeitnah Ersatz gefunden werden, ohne, dass das eigene System gefährdet wird. Teams sollten folglich darauf achten, dass nur ausgereifte und gut erprobte Software eingesetzt wird. Demnach ist weniger oftmals mehr. Hierbei hilft uns die hexagonale Architektur – da sich alle diese Frameworks in der äußersten Schicht befinden, können sie mit überschaubarem Aufwand ersetzt werden.

Die Kotlin-Website *[1]* selbst bietet eine sehr gute Übersicht über Bibliotheken für Backend-Anwendungen. Sie bietet zudem einen guten Ausgangspunkt für alle Fragen rund um Kotlin.

Für eine Cloudanwendung kommt man um ein etabliertes Applikations-Framework nicht herum. Wir haben uns für Spring entschieden, da es dem gesamten Team vertraut war, eine gute Kundenakzeptanz hat sowie stabil und zukunftssicher ist. Ein weiterer Vorteil ist, dass es mittlerweile solide Kotlin-Unterstützung bietet. Auch die Kotlin-Vertreter wie KTor oder http4k oder die Vertreter von Java-Seite Quarkus und Vert.x können problemlos verwendet werden, wenn man ausreichend Know-how besitzt oder sich dieses erarbeiten will.

Im Projekt wurde ebenfalls auf JPA/Hibernate verzichtet, stattdessen verwendeten wir das JDBC-Template von Spring. Es zeigte sich, dass bei geringem Mehraufwand erheblich an Transparenz gewonnen wird *(siehe Listing 1)*.

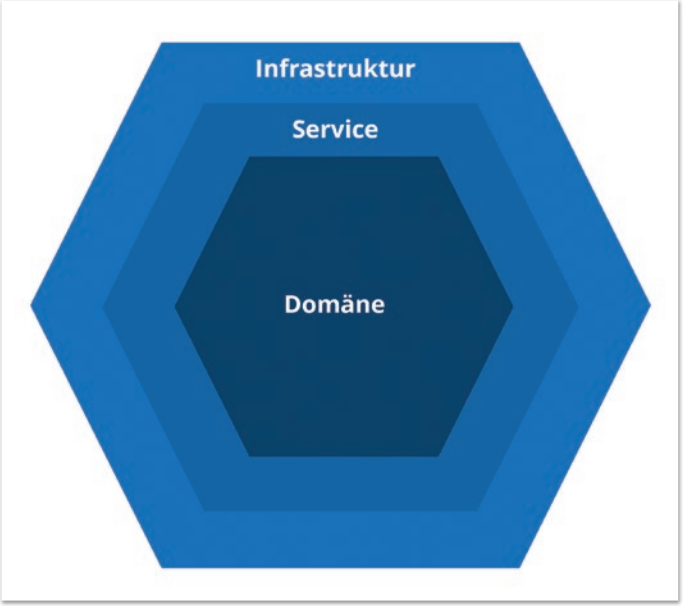


Abbildung 1: Hexagonale Architektur (© adesso SE)

```
fun findById(catalogId: CatalogId): CatalogRecord? = query(
  """SELECT *
      FROM $TABLE_NAME
      WHERE id = ?""", catalogRowMapper, catalogId
)
```

Listing 1: Eine einfache Query auf Basis von JDBCTemplate

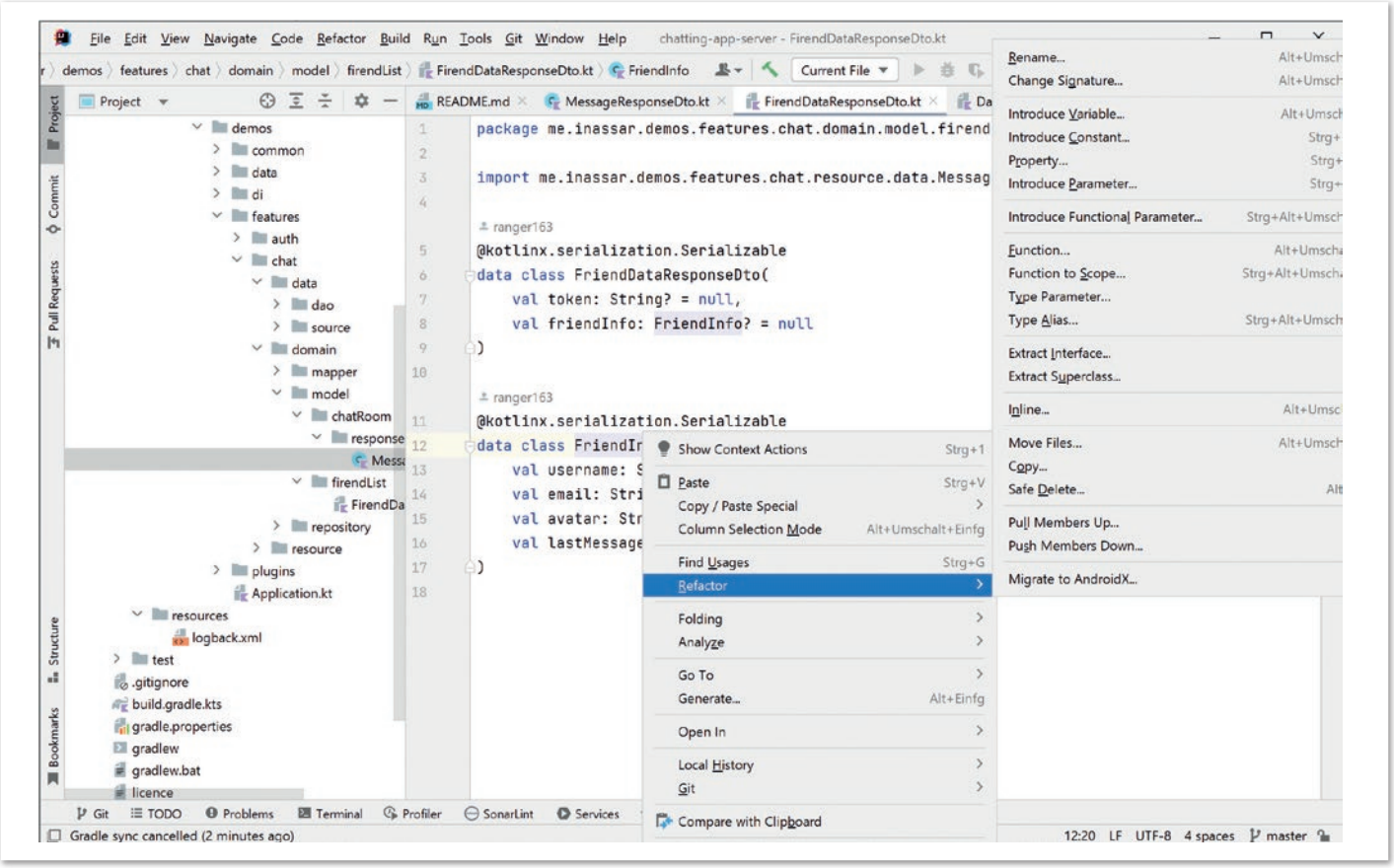


Abbildung 2: Entwicklungsumgebung IntelliJ

Der Code ist sehr kompakt und lesbar. Man erkennt auch einige der praktischen Sprachfeatures: String Templates und mehrzeilige Strings.

Im Backend spielt Nebenläufigkeit eine wichtige Rolle. Anstatt nur Java-vertraute Mechanismen zu verwenden, bietet Kotlin mit Koroutinen eine interessante Alternative. Diese erlauben unter anderem, die aus anderen Sprachen bekannten Async/Await-Konstrukte mit unterbrechbaren Funktionen zu verwenden. Allerdings ist das ein so umfangreiches Thema, das den Beitrag sprengen würde. Alternativ kann ich jedoch den Artikel aus der vorherigen Ausgabe von Java aktuell *[2]* empfehlen sowie diese Präsentation *[3]* von Google.

Auf der Qualitätsseite kann man die gewohnten Java-Tools für Testen, Mocks, Testcontainers, SonarQube etc. einsetzen, teilweise haben diese schon Kotlin Bindings (hier findet man zahllose Tutorials wie beispielsweise kotlintesting *[4]*).

Interoperabilität

Die Integration von Java-Code in Kotlin ist problemlos und auch umgekehrt lassen sich von Java aus Kotlin-Klassen reibungslos aufrufen. Der Code-Mix ist deutlich einfacher, als man dies bei anderen Sprachen bisher gewohnt war.

Die Firma JetBrains, die hinter der Kotlin-Entwicklung steht, entwickelt auch eine der meistverbreiteten Java-Entwicklungsumgebungen IntelliJ *(siehe Abbildung 2)*.

Durch dieses Zusammenspiel erhält man eine sehr gute Developer-Experience, auch wenn der Compiler etwas träger ist, als wir es von Java gewohnt sind.

IntelliJ erlaubt eine direkte Konvertierung zwischen beiden Sprachen: In Kotlin kopierter Java-Code wird automatisch umgewandelt *(siehe Abbildung 3)*. Trotz der guten Integration von Java in Kotlin wäre es bei einigen Bibliotheken angenehmer, wenn diese sich idiomatischer zu Kotlin verhalten würden. Dies ist leicht möglich mithilfe von Extension-Funktionen.

Kotlin im Backend

Die Unterschiede von Kotlin-Code zu Java sind nicht weltbewegend. Es sind vielmehr viele kleine nützliche Konzepte, die ineinandergreifen und auch bereits aus anderen Sprachen bekannt sind. Insgesamt wird damit der eigentliche Geschäfts-Code

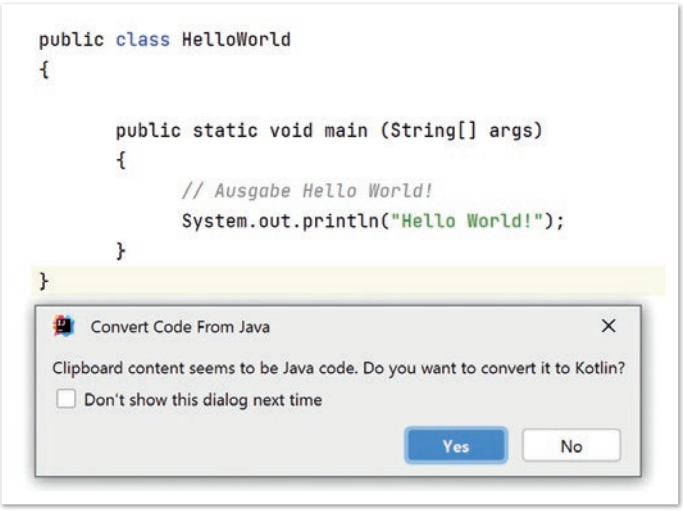


Abbildung 3: Konvertierung von Java-Code in Kotlin-Code durch IntelliJ

leichter zu erkennen und der „Happy-Path" tritt in den Vordergrund.

Zwei der wichtigen Beispiele hierfür sind die Nullsicherheit und Smart Casts. Um einer Variable einen Null-Wert zuweisen zu können, muss dies explizit in dem Typen definiert werden. Die Syntax hierzu ist recht kompakt, man hängt an den Typen ein Fragezeichen an (A) *(siehe Listing 2)*.

```
// A: A nullable variable
var x: String? = null;

// B: null safe method chaining
catalog?.firstCateogory()?.firstProduct()?.name

// C: Null safe elvis operator as guard
fun setCatalogName(id: CatalogId): String? {
     val catalog = retrieveCatalog() ?: return null
     // …
}

// D: smart casts
when (x) {
     is Int -> print(x + 1)
     is String -> print(x.length + 1)
     is IntArray -> print(x.sum())
}
```

Listing 2: Nullsichere Operatoren und Smart Casts

Wurde von dem Compiler eine erfolgreiche Nullprüfung angestellt, so muss dies im weiteren Code nicht mehr erfolgen. Mit dem Fragezeichen-Operator sind nullsichere Verkettungen (B) möglich. Der nullsichere Elvis-Operator macht beispielsweise Null-Guards am Anfang einer Methode sehr prägnant. Ein ähnlich intelligentes Compiler-Konzept sind Smart Casts (D). Auch hier muss der Compiler ähnlich zu der Nullprüfung nur einmal den Typen eines Wertes feststellen, ein weiterer Cast ist überflüssig.

Klassen

Klassen sind in Kotlin verglichen mit Java sehr leichtgewichtig und können und sollten auch sehr ausgiebig verwendet werden. In *Listing 3* sind einige Beispiele zu finden.

Getter und Setter werden nicht benötigt, durch die Sichtbarkeit und die Unterscheidung von `val` (unveränderlicher Wert) und `var` (Variable) werden von Compiler bereits die Zugriffsmöglichkeiten abgeleitet – ein Framework wie Lombok ist überflüssig. Für viele ist schon dies ein Grund, zu Kotlin wechseln. Man sieht in A) bereits ein vollständige Klassendefinition. Daher macht es Sinn, Klassen ausgiebig zu nutzen, auch wenn man wie in B) nur einen String „wrapt". Der Compiler erkennt durch Typinterferenz an den meisten Stellen den Typ einer Variable aus dem Kontext (C). Dadurch tritt die starke Fokussierung auf Klassen in den Hintergrund. Ein gutes Konzept ist es, große Klassenhierarchien durch Delegation zu vermeiden, hierzu gibt es ebenfalls ein Sprachkonstrukt, der Compiler erzeugt selbst den Glue-Code.

Sehr nützlich sind Data-Klassen (D). Wird das Schlüsselwort `data` einer Klasse vorangestellt, so erzeugt der Compiler `compare`, `hashcode`, `toString` sowie `copy`-Funktionen. Letztere sind besonders interessant für Value-Klassen: In F) wird ein neues Entry-Objekt erstellt, das alte bleibt unverändert. Durch die Konzepte von benannten Parametern und Default-Werten muss nur die Eigenschaft angegeben werden, die tatsächlich verändert wird.

Fehlerkonzept

Ein wichtiger Bestandteil jeder Anwendung ist der Fehlerbehandlungscode. Jedoch lenkt er oftmals stark von der eigentlichen Geschäftslogik ab. Daher macht es Sinn, ein Konzept zu entwickeln, wie der „Happy-Path" möglichst gut im Mittelpunkt behalten werden kann.

In Java sind Exceptions und die dazugehörigen Try/Catch-Blöcke das Mittel der Wahl, um unerwartetes Verhalten zu behandeln. Der Java-Compiler erzwingt auf jeder Ebene zu definieren, wie eine Exception behandelt werden soll. In der Praxis führt dies oft zu unübersichtlichen Try/Catch-Blöcken und Verzweigungen. Der Mehrwert solcher detaillierterer Fehlerbehandlung ist in vielen Fällen den Aufwand nicht wert:

- Wenn es ein technisches Problem wie ein Netzwerkfehler ist, kann man meist nicht direkt etwas machen und auch der Anwender kann nicht viel mehr als informiert werden.
- Handelt es sich um einen fachlichen Fehler, sind Exceptions nicht gut geeignet, um dies zu repräsentieren.

In Java muss jede Exception auf Methodenebene behandelt werden. In Kotlin werden im Kontrast dazu, alle Exceptions nicht nur die `RuntimeExceptions` vom Compiler „ignoriert". In unserer Anwendung haben wir dieses Konzept aufgenommen und weitergeführt: Wir behandeln in der Regel Exceptions nicht, sondern lassen diese bis zum Aufruf im Controller durchgehen. Dort werden diese durch einen

```
class User(val id: Int, public val email: String)  // A

class EmailAddress(val value: String) // B

var x = 33; // C

data class Order(val productId: String, val amount: Int)  // D

val entry = Order("4711", 3); // E

val newEntry = entry.copy(amount = 5); // F

val (nextProductId, _) = entry;  // G
```

Listing 3: Beispiele für Kotlin-Klassen und -Variablen

```
sealed class ServiceResult<T> {
  abstract fun <B> map(block: (a: T) -> B): ServiceResult<B>
  abstract fun <B> success(block: (a: T) -> B): B
  abstract fun error(block: (errors: List<String>) -> B): B

  class Result<T>(val value: T) : ServiceResult <T>() { ..}
  class Error<T>(val errors: List<String>) : ServiceResult <T>() { … }
}
```

Listing 4: Eine an Monaden angelehnte Ergebnisklasse

```
fun create(request: ServerRequest): ServerResponse {
  //…
  return service.createProduct(label, productType).success { product ->
        redirect("${contextForProducts(request)}/${product.id}/edit")
  }.error { errors ->
     val model = …
     error("$TEMPLATE_PATH/new", model)
  }
}
```

Listing 5: Verwendung der Eigenbau-„Monade"

zentralen Exception-Handler behandelt. Dies ist möglich, da unsere Anwendung keine Seiteneffekte hat und Änderungen sich nur in der Datenbank manifestieren. Bei Exceptions wird alles zurückgerollt. Viele Cloudanwendungen mit einem REST-API lassen sich auf diese Weise entwickeln.

Eine Dosis funktionale Programmierung

Dies ist die eine Seite – die technischen Fehler – für fachliche Fehler haben wir uns von einem Konzept der funktionalen Programmierung inspirieren lassen, der Monade – ein in der funktionalen Programmierung oft verwendeter abstrakter Datentyp, der der Klassentheorie der Mathematik entliehen ist. Grob formuliert, ist eine Monade ein Behälter, in den man einen Wert ablegen kann sowie zusätzliche Informationen. Wir kennen das seit Java 8 mit der Optional-Monade.

Diese Art zu programmieren, ist jedoch für die meisten Java-Entwickler eine hohe Einstiegshürde, die verhindert werden sollte. Außerdem haben FP-Konzepte die Tendenz dazu, sich immer weiter auf die gesamte Anwendung auszubreiten. Wenn man dennoch tiefer in diesen „Kaninchenbau" der funktionalen Programmierung absteigen will, empfehle ich das Buch „Functional Programming in Kotlin" *[6]* und die dazu passende Kotlin idiomatische Bibliothek Arrow *[7]*.

Angelehnt an das Konzept der „Monade", haben wir unser anwendungsspezifisches ServiceResult entwickelt *(siehe Listing 4)*.

Ein `ServiceResult` ist ein Generic, über den Ergebnistyp T zwei Unterklassen enthält: `Result` und `Error`. `Result` beinhaltet ein mögliches Ergebnis, `Error` die Liste an Fehlern. Mit der Funktion `map` kann ein `ServiceResult` in ein anderes überführt werden, um Zwischenergebnisse zu kombinieren.

Speziell sind die zwei Funktionen `success` und `error`, die einen Lambda-Block übernehmen können, um dann auf die beiden möglichen Pfade reagieren zu können. Die Belohnung für diese Typakrobatik ist nun ein gut lesbarer Business-Code, zum Beispiel in einem Controller wird ein Service aufgerufen, der ein `ServiceResult` zurückliefert *(siehe Listing 5)*.

Wie man sieht, kann man die beiden möglichen Pfade sehr einfach erkennen. Ein alternatives, in manchen Teilen ähnliches Konzept zur

```
scope.createCatalog("Clothing") {
  addToMarket(market_de)
  createCategory("Men") {
     createCategory("Shirts")
  }
}
```

Listing 6: DSL für Testdaten

```
class CatalogScope(
  private val builder: DataBuilder,
  private val catalogId: CatalogId,
  private val parentId: CategoryId = null
) {
  fun createCategory(name: String, block:   CategoryScope.() -> Unit = {}) {
     val category = categoryService.create(name, catalogId, parentId)
     block(CategoryScope(builder, catalogId, category.id))
  }
  fun addToMarket() // …
}
```

Listing 7: Implementierung eines Builders

Fehlerbehandlung, findet man bei Phillip Hauer *[4]* (sein Blog bietet auch weitere gute Kotlin-Tipps).

Embedded DSL

Ein weiteres wichtiges Konzept, das man erst mit der Zeit zu schätzen lernt, ist die Möglichkeit, in Kotlin Code so zu schreiben, dass man eine in Kotlin eingebettete Anwendungssprache entwickeln kann. Hier spielen einige Kotlin-Features praktisch zusammen, zum einen das Builder-Konzept und zum zweiten Currying, also die Möglichkeit, einen Lambda-Parameter aus dem Parameterblock zu ziehen und hinter die schließende Parameterklammer zu schreiben.

In unserer Anwendung erzeugen wir Testdaten direkt in Kotlin-Code *(siehe Listing 6)*. Der Code, beispielsweise für `createCategory()`, ist dafür recht prägnant *(siehe Listing 7)*.

Man kann den Lambda-Parameter `block` aus dem Aufruf ziehen. Der Lambda-Code wird dann mit der erzeugten Kategorie aufgerufen.

Fazit

Die Community um Kotlin ist stetig gewachsen und man findet für vieles auch neue Frameworks. Dennoch kann es sich in keinen Fall mit Java messen. Im Vergleich zu Java entwickelt sich die Sprache und das Ökosystems rasant weiter. Kotlin ist keine atemberaubend andere Art, zu entwickeln, dennoch ist es für einen Java-Entwickler leicht zu beherrschen und steigert die Entwicklungsperformance (siehe auch *[3]*). Man kann langsam ein- oder umsteigen und mehr und mehr idiomatische Konzepte übernehmen. Meine subjektive Beobachtung ist, dass die meisten Entwickler, die sich auf Kotlin eingelassen haben, eher zukünftig dabeibleiben wollen.

Quellen

[1] *https://kotlinlang.org/docs/server-overview.html#frameworks-for-server-side-development-with-kotlin*

[2] Java aktuell 3/21, S35 Kotlin Koroutinen *https://shop.doag.org/zeitschriften/id.19.java-aktuell-03-2021-jvm-sprachen/*

[3] *https://www.youtube.com/watch?v=o14wGByBRAQ*, Google's Journey to Kotlin for Server Side, Devoxx 2022

[4] *https://www.kotlintesting.com/*

[5] *https://phauer.com/2019/sealed-classes-exceptions-kotlin/*

[6] Functional Programming in Kotlin, Manning 2021, Marco Vermeulen, Rnar Bjarnason, Paul Chiusano

[7] *https://arrow-kt.io/*

Andreas Voigt
adesso SE
Andreas.Voigt@adesso.de

Andreas Voigt ist Softwarearchitekt bei der adesso SE. Er beschäftigt seit mehr als 15 Jahren mit Java-Software als Entwickler und Architekt. Neben Java hat er ein Faible für alle möglichen exotischen und weniger exotischen Programmiersprachen.

Performance-Analyse – Theorie

Dr. Stefan Koch, ORDIX AG

Performance ist, ähnlich wie frische Luft, immer dann ein Thema, sobald ein Mangel daran festgestellt wird: Anwender sind unzufrieden, wenn die IT sie in ihrer Arbeit behindert oder Stapelverarbeitungen benötigen unerwartet viel Zeit für die Ausführung.
Die Performance-Analyse hat die Aufgabe, diesen Mangel zu quantifizieren und Ansatzpunkte zu liefern, um die Situation zu verbessern. Bei Backend-Anwendungen, wie beispielsweise RESTful-Services, eignet sich die Lastkurve zur Quantifizierung der Performance.

In diesem Artikel werden Sie die Ideale Lastkurve kennenlernen. Die damit verbundenen analytischen Zusammenhänge sind hilfreich, um Lastkurven zu interpretieren und erfolgsversprechende Tuning-Maßnahmen vorzuschlagen.

Notwendigkeit der Performance-Analyse

Aus der Perspektive der Software-Entwicklung ist die Performance eine nicht-funktionale Anforderung von vielen. Anwender haben eine konkrete Vorstellung von den Arbeitsabläufen – was die IT dafür umsetzen muss, um genau diese zu unterstützen, steht dabei nicht im Mittelpunkt. Bei der Implementierung steht die Verständlichkeit des Codes im Fokus: Entwickler müssen mit dem Code mitunter jahrelang leben und wollen diesen verstehen und warten können. Performance konkurriert nicht zuletzt mit dem Aufwand und den damit verbundenen Kosten: Performance-Untersuchungen oder -Optimierungen werden oftmals unterlassen, da sie mit einem erheblichen Aufwand verbunden sind.

Die Einstellung zur Performance der technischen Verantwortlichen lässt sich gut mit den englischen Worten *Wishful Thinking* beschreiben: Das wird schon gutgehen – und wenn nicht, werden wir das Problem im Betrieb feststellen und möglichst mit betrieblichen Mitteln beseitigen. Der Betrieb in der Cloud (Container-as-a-Service) ermöglicht eine horizontale Skalierung der Rechner-Ressourcen per Mouse-Click. Erforderliche Änderungen – vielleicht hilft schon ein nettes Gimmick als Sanduhr – sind dank Continuous Delivery auch schnell im Betrieb eingespielt.

So ist eine Performance-Analyse nur für hartnäckige Probleme eine Option: Wieso kann die Antwortzeit auch nach umfangreicher horizontaler Skalierung nicht signifikant verringert werden? Der theoretisch erwartete Durchsatz bleibt aus, obwohl die CPU sich langweilt. Dann sind da noch Instabilitäten des Laufzeitverhaltens: Ein Service muss täglich gestartet werden, um ein Out-of-Memory zu verhindern. Oder das Dead-Lock-Phänomen: Anfragen verenden im Wartezustand oder werden durch Time-Out-Überwachung nach Minuten in einen Fehlerzustand erlöst.

Ideale Lastkurve

Grau und abschreckend ist die Theorie – doch wer sie versteht, wird durch Gewissheiten belohnt.

Zielgrößen Durchsatz und Antwortzeit

Die Zielgrößen der Performance – zumindest für Backend-Anwendungen – sind **Antwortzeit** und **Durchsatz**. Der Begriff Antwortzeit ist selbsterklärend – die Zeit zwischen der Anfrage und der Antwort. Diskussionsbedarf gibt es vielleicht noch, an welcher Stelle (End-to-End oder Eingang der Anfrage und Absenden der Antwort auf dem Backend-Server) die Zeiten gemessen werden. Der **Durchsatz** gibt dagegen an, wie viele Antworten das System pro Zeit geben kann.

Vor Augen führen lassen sich diese Begriffe im weihnachtlichen Gedränge im Supermarkt *(siehe Abbildung 1)*. Sie suchen sich die Schlange aus, die Sie möglichst bald zur Kasse und dann aus dem Supermarkt führt. Der Zeitpunkt, zu dem Sie sich in die Schlange einreihen bis zum Zeitpunkt des Bezahlens ist die **Antwortzeit**. Eine für Sie wichtige Größe, da ja noch andere Einkäufe zu erledigen sind.

Für den Filialleiter ist die Antwortzeit – und damit die Kundenzufriedenheit – sicher auch ein wichtiger Maßstab. Für seinen Umsatz ist jedoch der **Durchsatz** wesentlich: Wie viele Kunden lassen sich pro Zeiteinheit abkassieren. Der Filialleiter wird den optimalen **Betriebspunkt** wählen, indem er die Anzahl der aktiven Kassierer festlegt, um bei zumutbaren Wartezeiten den Durchsatz zu bewältigen.

Abbildung 1: Gedränge an der Kasse im Supermarkt [2]

Theoretische Lastkurve

Im Folgenden wird die Theoretische Lastkurve für eine Backend-Anwendung hergeleitet. Hierbei handelt es sich um ein stark vereinfachtes Modell, das aber die Beschreibung wesentlicher Systemeigenschaften erlaubt.

Die Backend-Anwendung erhält eine Anfrage, der Prozessor berechnet daraufhin eine Antwort. In diesem Modell braucht der Prozessor immer dieselbe Zeit, um die Antwort zu berechnen. In *Abbildung 2* wird die Zeit für die Berechnung der Antwort als Pfeil dargestellt. Drei Prozessoren stehen zur Verfügung. Die **Last** ist definiert als die Anzahl der Anfragen, die gleichzeitig an die Anwendung gerichtet sind. Im **Last-Bereich 1** werden drei Anfragen parallel bearbeitet, jede Anfrage kann von einem eigenen Prozessor bearbeitet werden. Die Antwortzeit ist identisch mit der **Berechnungszeit** – die Zeit, die ein Prozessor benötigt, um eine Antwort zu berechnen.

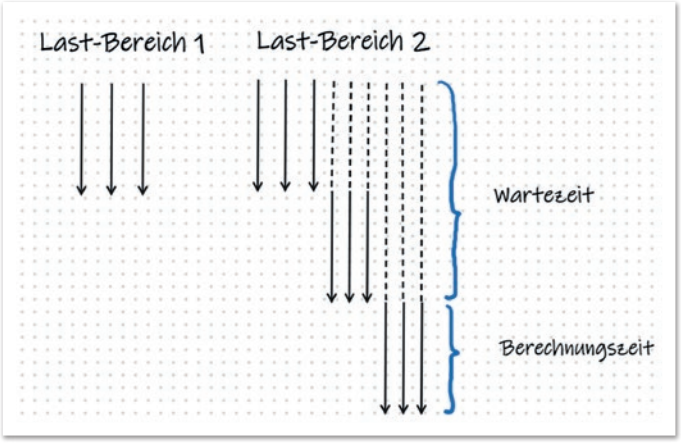


Abbildung 2: Ermittlung der Antwortzeit in unterschiedlichen Lastbereichen (© Dr. Stefan Koch)

Im **Last-Bereich 2** werden mehr als drei Anfragen gleichzeitig gestellt. In *Abbildung 2* werden parallel neun Anfragen bearbeitet. In der Abbildung werden zuerst die ersten drei Anfragen bearbeitet, dann die nächsten drei und schließlich die letzten drei. Die Antwortzeit setzt sich aus Berechnungszeit und Wartezeit zusammen. Die **Wartezeit** kommt durch einen Mangel an Prozessor-Zeit zustande: Die Anfrage wartet auf Prozessor-Zeit. In dem Fall der neun parallelen Anfragen bei drei Prozessoren ist die Antwortzeit daher dreimal

so lang wie die Berechnungszeit. Mit dieser Vorstellung lässt sich die ideale Lastkurve *(siehe Abbildung 3)* ermitteln.

In der Idealen Lastkurve werden Antwortzeit und Durchsatz in Abhängigkeit von der Last (parallele Anfragen) dargestellt. Es gilt die Voraussetzung, dass die Bearbeitung der Anfrage immer dieselbe Berechnungszeit (und nur diese) in Anspruch nimmt. Im ersten Last-Bereich ist die Antwortzeit so lange konstant, wie genügend Prozessoren zur Verfügung stehen, um die Anfragen unmittelbar zu bearbeiten.

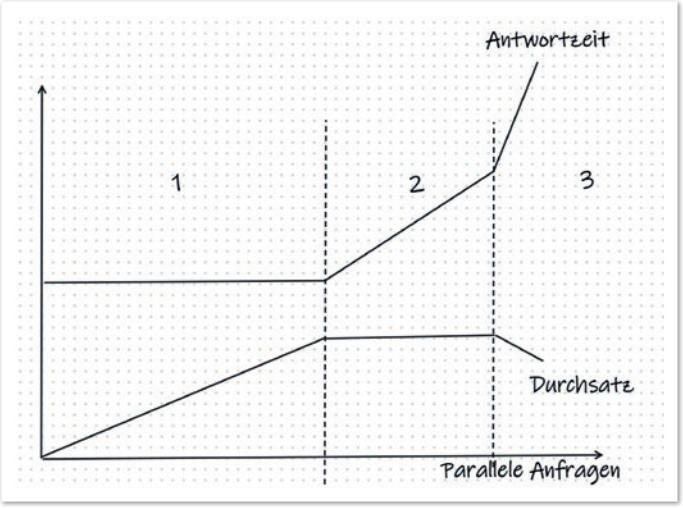


Abbildung 3: Ideale Lastkurve (© Dr. Stefan Koch)

Übersteigt die Last die Anzahl der verfügbaren Prozessoren, kommt es zu Wartezeiten. Die Wartezeiten werden mit zunehmender Last steigen. Zusätzlich ist in *Abbildung 3* ein Last-Bereich 3 eingezeichnet. Dieser berücksichtigt, dass ab einer gewissen Last weitere Wartezeiten auftreten, die mit der Verwaltung der hohen Last in Verbindung stehen. Dadurch erhöht sich die Antwortzeit zusätzlich.

Analytische Berechnung der Idealen Lastkurve

Die analytische Berechnung der Lastkurve wird dadurch vereinfacht, dass in Backend-Anwendung die Prozessor-Zeit in Zeitscheiben vergeben werden. Dadurch wird die verfügbare Prozessor-Zeit gleichmäßig auf alle Anfragen verteilt. Jede Anfrage wird zwar unmittelbar in Angriff genommen, doch kommt es nur zu einem Fortschritt, wenn der Anfrage Prozessor-Zeit zur Verfügung gestellt wird. Die in *Abbildung 2* skizzierte Berechnungs- und Wartezeit wird durch dieses Prinzip in kleine Stücke aufgeteilt.

Für die analytische Berechnung können die Begriffe der Mechanik Leistung P und Arbeit W analog verwendet werden. Als **Leistung P** kann die Anzahl der Basis-Operationen definiert werden, die durch einen Prozessor pro Zeiteinheit ausgeführt werden. Sie ist also von der Bauart und der Taktung des Prozessors abhängig. Ein Prozessor verfügt demnach über eine Leistung. Betrachten wir nur einen Typ von Prozessor, können wir als **Einheit für die Leistung [CPU]** festlegen. Verfügt unsere Backend-Anwendung über eine Hardware mit vier Prozessoren, so ist die verfügbare Leistung 4 [CPU].

Die **Arbeit W** ist die Anzahl der ausgeführten Basis-Operationen, also die Leistung multipliziert mit der Zeit. Für die Leistung haben wir als Einheit [CPU] festgelegt, damit ist die **Einheit der Arbeit CPU-Sekunden, also [CPUs]**. Der Aufwand zur Berechnung einer Antwort ist als Arbeit anzugeben. Die Arbeit ist berechenbar als Produkt aus Leistung und **Zeit t**.

$$W=P*t \quad (1)$$

In dem folgenden Gedanken-Experiment gehen wir davon aus, dass für die Berechnung einer Antwort 0,1 [CPUs] erforderlich sind und die Leistung von 4 [CPU] zur Verfügung steht. Die Anwendung wird unter Last gesetzt, indem ununterbrochen n parallele Anfragen an die Backend-Anwendung gerichtet werden. Benötigt eine Anfrage die **spezifische Arbeit *w***, so benötigen die parallelen Anfragen in der Summe die Arbeit n * w. Die **Antwortzeit *t*** lässt sich damit berechnen.

$$W=n*w=P*t \quad (2)$$
$$t=\frac{(n*w)}{P} \quad (3)$$

Mit Formel (3) ist der Last-Bereich 2 in *Abbildung 3* berechenbar: Die Antwortzeit ist proportional zur Anzahl der parallelen Anfragen. Die Steigung wird bestimmt aus dem Quotienten aus spezifischer Arbeit w und der verfügbaren Leistung P. Für das Beispiel mit w=0,1 [CPUs] und P=4 [CPU] ergibt sich eine Steigung von 0,025 [s]: Wird die Last um eins erhöht, so verlängert sich die Antwortzeit um 0,025 [s].

Der Last-Bereich 1 lässt sich durch Formel (3) nicht erklären! In der *Abbildung 3* steckt die realistische Annahme, dass die Arbeit w einer Antwort nicht auf mehrere Prozessoren aufteilen lässt. Wird also immer nur eine Anfrage parallel erstellt, so kommt dafür auch nur eine CPU zum Einsatz. Die Leistung der übrigen Prozessoren wird dafür nicht verwendet. Der Last-Bereich 1 erstreckt sich auf das Intervall für n von 0 bis N; **N ist die Anzahl der verfügbaren Prozessoren**.

Der **Durchsatz T** – also die Anzahl der Antworten pro Zeiteinheit – wird über die Antwortzeit berechnet. Für nur eine parallele Anfrage ist sie gleich dem Kehrwert der Antwortzeit: $T=1/t$. Der Durchsatz wächst proportional zur Anzahl der parallelen Anfragen.

$$T=\frac{n}{t}=\frac{P}{w} \quad (4)$$

In Gleichung (4) ist unmittelbar zu erkennen, dass der Durchsatz als Quotient aus verfügbarer Leistung und spezifischer Arbeit unabhängig von der Belastung n ist. Genauso, wie für Gleichung (3), muss diese Gleichung aber im geringen Lastbereich n < N, angepasst werden: Hier darf nur die tatsächlich verwendet Leistung eingesetzt werden! Daher kommt es bei geringer Last zunächst zu einem linearen Anstieg des Durchsatzes.

Grenzen der Idealen Lastkurve

Die Ideale Lastkurve wurde hergeleitet für den Fall, dass allein die Rechenleistung für die Antwortzeit verantwortlich ist. Vielfach beinhalten die Antwortzeiten aber auch **Wartezeiten**, in denen auf Antworten von der Datenbank oder auf Datenübertragung von der Festplatte oder aus dem Netzwerk gewartet wird. Darüber hinaus können neben der CPU auch weitere **Ressourcen** wie Verbindungen zur Datenbank oder zu anderen Anwendungen zu Wartezeiten führen.

Beispiel	Dauer	Dauer in [s]	Zeilen Java-Code
Größenordnung für die Ausführung einer Zeile Java-Code	10 [ns]	10 * 0,000 000 001 [s]	1
Größenordnung für die Latenz im Netzwerk	1 [ms]	1 * 0,001 [s]	100.000
Reaktionszeit eines Anwenders	1 [s]		100.000.000

Tabelle 1: Zeitliche Größenordnungen

Die spezifische Arbeit oder aber Wartezeiten sind nicht selten vom System-Zustand und damit von der Historie der Belastung abhängig: So führen Initialisierungs-Vorgänge zu Anfang zu einer Vergrößerung der spezifischen Arbeit. Wartezeiten werden oft mit dem Füllstand einer Datenbank zunehmen.

In der Regel bietet eine Backend-Anwendung mehrere Services an. Jeder Service hat (mindestens eine) eigene spezifische Arbeit.

Die genannten Einflussfaktoren müssen bei der Performance-Analyse berücksichtigt werden. Performance-Analysen lassen sich nur exakt durchführen, wenn die Belastung durch Szenarien definiert wird: Neben den Einflussfaktoren der Idealen Lastkurve sind dabei weitere Einflussgrößen wie Füllstand der Datenbank, Mischung und zeitlicher Ablauf der Service-Aufrufe, Ressourcen-Situation vorzugeben.

Die analytische Betrachtung der Idealen Lastkurve geht von konstanten Zeiten aus. In der Realität werden sowohl Antwort- wie auch Wartezeiten Schwankungen unterworfen sein. Da die Berechnung nur lineare Abhängigkeiten von diesen Zeiten hervorbringt, können anstelle der konstanten Zeiten auch deren Mittelwerte eingesetzt werden, ohne dass sich an den Aussagen etwas ändert.

Wartezeiten

In den meisten Anwendungen kommt es bei der Bearbeitung von Anfragen zu Wartezeiten. Das ist beispielsweise immer dann der Fall, wenn eine Netzwerk-Kommunikation erfolgt. Zeiten, in denen die Anfrage auf Ein- oder Ausgabe-Operationen warten muss, bringen diese Anfrage nicht voran. Auch wenn derartige Wartezeiten kurz erscheinen, sind diese für den Berechnungsfortschritt erheblich. Die folgende Tabelle setzt Wartezeiten in Relation zur Ausführung einer Zeile Java-Code.

Zugegebenermaßen ist die Angabe der Ausführungszeit einer Zeile Java-Code in *Tabelle 1* mit einer erheblichen Unsicherheit verbunden. Diese sehr grobe Schätzung ist aber hilfreich, um zu sehen, dass Wartezeiten viel zu kostbar sind, um nur zu warten.

Betrachten Sie beispielsweise die Verarbeitung einer RESTful-Request in *Abbildung 4*.

Die typische Request besteht aus drei Phasen. Die Request wird interpretiert, ein Datenbank-Zugriff erfolgt, um schließlich die Antwort zu berechnen. Der Datenbank-Zugriff wird in einem anderen Prozess durchgeführt, die Verarbeitung muss auf die Antwort der Datenbank warten. Nur in der ersten Phase, der Interpretation der Request, und in der dritten Phase, der Zusammenstellung der Antwort, ist Prozessor-Zeit erforderlich. Die *Abbildung 4* verdeutlicht, dass der Prozess zu einem wesentlichen Anteil aus der Wartezeit besteht.

Wie groß der Anteil der Wartezeit an der Antwortzeit ist, ist von dem jeweiligen Fall abhängig. Die Wartezeit macht üblicherweise aber einen erheblichen Anteil der Antwortzeit aus. Aus diesem Grund sind Backend-Anwendung so aufgebaut, dass sie mehrere Anfragen gleichzeitig bearbeiten können. Jede Anfrage wird in einem eigenen Thread – einem leichtgewichtigen Prozess in der Anwendung – bearbeitet.

Für die Ausführung von Prozessen ist das Betriebssystem zuständig. Es sorgt dafür, dass mehrere Prozesse gleichzeitig ausgeführt werden, indem die Prozessor-Zeit auf Prozesse aufgeteilt wird. Der Scheduler des Betriebssystems teilt die CPU-Zeit dazu in kleine Teile und stellt der Reihe nach den Prozessen diese CPU-Zeit zur Verfügung. Der Algorithmus des Schedulers berücksichtigt auch, ob ein Prozess sich in einem Wartezustand befindet – in dem Fall bekommt er keine Prozessor-Zeit zugeteilt. So wird die Prozessor-Zeit optimal genutzt. Auch für die leichtgewichtigen Prozesse, die Threads in Backend-Anwendungen, ist der Scheduler zuständig.

An dem qualitativen Aufbau der Lastkurve *(siehe Abbildung 3)* ändert sich wenig. Die skizzierten drei Last-Bereiche werden auch auftreten, wenn die Verarbeitung mit Wartezeiten verbunden ist. Die **Antwortzeit *t*** für eine Anfrage setzt sich aus den folgenden Anteilen zusammen:

- **I/O-Wartezeit *u***: zum Beispiel durch das Warten auf Daten über das Netzwerk
- **Berechnungszeit *v***: Die Zeit, die für die Berechnung der Antwort erforderlich ist

$$t = u + v \quad (5)$$

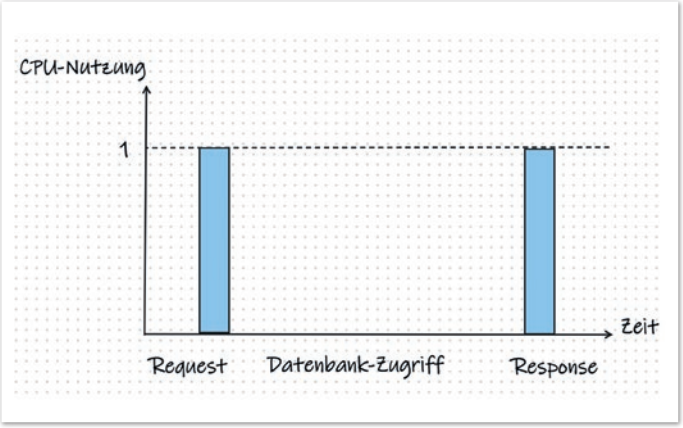


Abbildung 4: Einsatz von Prozessor-Zeit (blau) für die Verarbeitung einer RESTful-Request (© Dr. Stefan Koch)

MITMACHEN UND BEITRAG EINREICHEN!

Du kennst dich in einem bestimmten Gebiet aus dem Java-Themenbereich bestens aus und möchtest als Autorin oder Autor dein Wissen mit der Community teilen?

Nimm Kontakt zu uns auf und sende deinen Artikelvorschlag zur Abstimmung an redaktion@ijug.eu.

Wir freuen uns, von dir zu hören!

iJUG Verbund

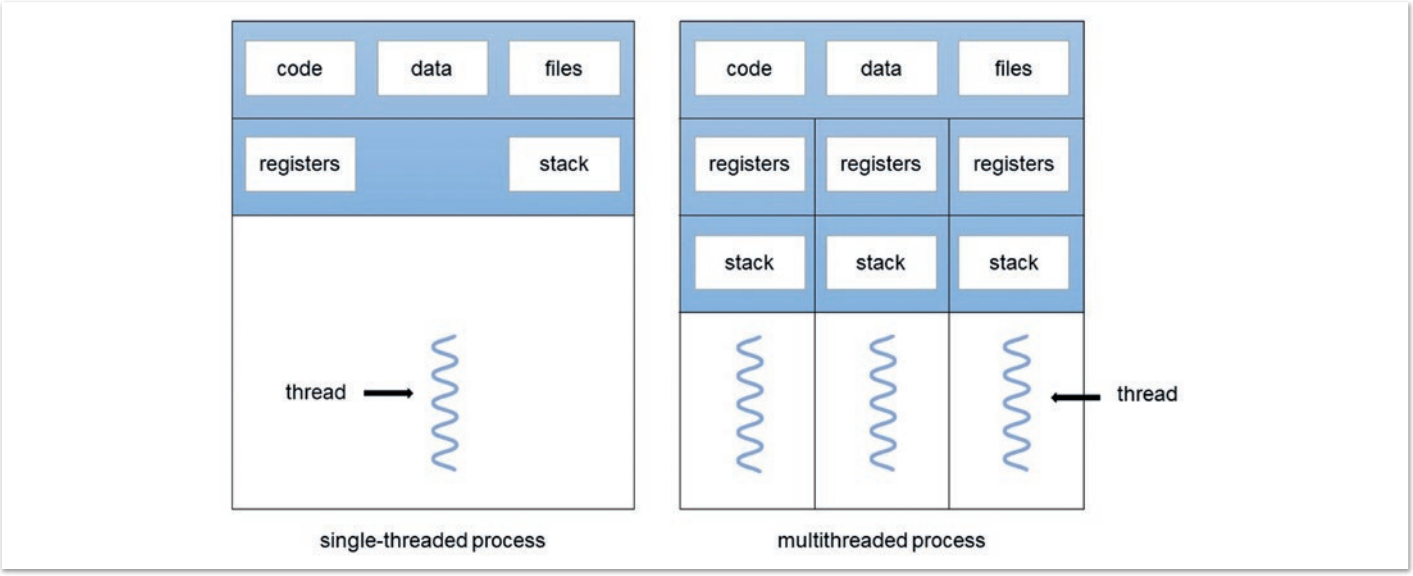


Abbildung 5: Thread im Kontext eines Prozesses (© Dr. Stefan Koch)

Die Antwortzeit *t* wird im Folgenden berechnet. Zunächst betrachten Sie das Intervall von Last-Bereich 1: Wie viele parallele Anfragen n_1 können erfolgen, ohne dass sich die Antwortzeit t_1 verändert? Dazu führen Sie ein Gedankenexperiment durch: Sie bilanzieren die Arbeit, die während der Zeit t_1 durch alle N Prozessoren zur Verfügung gestellt wird.

$$W(t\_1)=P^* t\_1=n\_1^*w \quad (6)$$

Diese Arbeit *W(t1)* wird durch die Last n_1 vollständig konsumiert. Gleichung (6) nach n_1 aufgelöst, liefert die Anzahl der parallelen Anfragen.

$$n_1=\frac{P^* t_1}{w}=\frac{N^*p^*(u+v)}{v^*p}=\frac{N(u+v)}{v} \quad (7)$$

Die Arbeit $W(t_1)$, die die Prozessoren während der Zeit t_1 ausführen, kann genutzt werden, um die Anzahl n_1 parallele Anfragen abzuarbeiten. Im Last-Bereich 2 kommen zusätzliche Anfragen dazu: $n > n_1$. Bemerkenswert ist, dass diese Anfragen dann eine Antwortzeit $t > t_1$ haben werden: Innerhalb der Antwortzeit ist die Wartezeit *u* enthalten. Während dieser Zeit konsumiert der zugehörige Thread keine Prozessor-Zeit. Für den Last-Bereich 2 gilt daher dieselbe Bilanz wie für die ideale Lastkurve:

$$P^*t=n^*w \quad (8)$$

$$t=\frac{n^*w}{P}=\frac{n^*v^*p}{N^*p}=\frac{n^*v}{N} \quad (9)$$

In Gleichung (9) wird die **Leistung eines einzelnen Prozessors** mit ***p*** eingeführt.

Schematisch sind in *Abbildung 6* zwei Lastkurven eingezeichnet. Die schwarze Lastkurve entspricht einer idealen Lastkurve. Die zugehörige Berechnungszeit *v* lässt sich als Achsenabschnitt ablesen. Die blau gestrichelte horizontale Lastkurve gehört zu einer Anfrage mit derselben Berechnungszeit und der I/O-Wartezeit *u*. Die Summe aus I/O-Wartezeit und Berechnungszeit ergibt die Antwortzeit t_1 in Last-Bereich 1. Der Last-Bereich 1 der Lastkurve mit I/O-Wartezeit ist entsprechend Gleichung (7) verlängert. Die Lastkurve mit I/O-Wartezeit fällt in dem Last-Bereich 2 mit der Idealen Lastkurve zusammen.

Für eine Vorstellung, wie stark der Last-Bereich 1 durch die Wartezeit *u* erweitert wird, kann die Wartezeit über die Berechnungszeit ausgedrückt werden. Zu diesem Zweck wird die **Response-Ratio *R*** als Quotient aus Antwortzeit *t* und Berechnungszeit *v* eingeführt.

$$R=\frac{t}{v} \Leftrightarrow u=v^*(R-1) \quad (10)$$

Führen Sie die Response-Ratio in Gleichung (7) ein, so erhalten Sie die Last n_1 als Funktion von *R*.

$$n\_1=N^*R \quad (11)$$

In der Realität nimmt die Response-Ratio zweistellige Werte ein. Die Faustformel, dass mit einem Prozessor eine Last von 50 gleichzeitigen Anfragen bewältigt werden kann, ist mit Gleichung (11) nachvollziehbar.

Ressourcen

Auch Ressourcen rücken erst in den Blick des Betrachters, wenn sie die Performance beeinträchtigen. Dabei lassen sich Ressourcen in

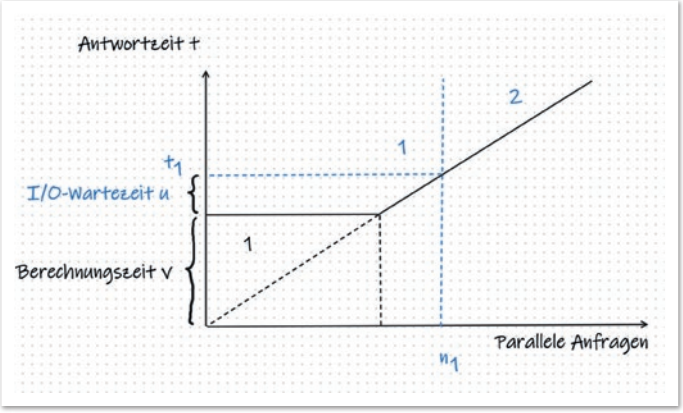


Abbildung 6: Ideale Lastkurve (schwarz) und Lastkurve mit Wartezeit (blau) (© Dr. Stefan Koch)

essenziell notwendige Ressourcen, wie beispielsweise den Hauptspeicher, oder aber in aufteilbare Ressourcen, wie Datenbank-Verbindungen, aufteilen. Fehlen essenzielle Ressourcen, so führt das zu einer Gefährdung der Leistungserbringung. Nach einem Out-of-Memory-Error wird der Dienst einer Backend-Anwendung nicht mehr funktionieren. Fehlen essenzielle Ressourcen, so müssen diese in ausreichendem Maße zur Verfügung gestellt werden. In einigen Fällen führt die Erhöhung der Ressourcen nicht zum Erfolg: die Anwendung verbraucht Ressourcen etwa in Form von Memory-Leaks oder nicht geschlossenen Ressourcen. Derartige Anwendungsfehler müssen für einen stabilen Betrieb beseitigt werden.

Oftmals werden die Verbindungen zu einer Datenbank in einem Connection-Pool verwaltet. Das ist ein weit verbreitets Beispiel für eine Ressource, – Verbindung zur Datenbank – die von allen Anfragen gemeinsam genutzt wird. Ist der Pool zu klein dimensioniert, so müssen Anfragen auf Datenbank-Verbindungen warten, weil der Pool gerade leer ist: Alle Verbindungen sind in Verwendung. Besonders ungünstig sind solche Wartezeiten dann, wenn die Reihenfolge, in der frei gewordene Verbindungen vergeben werden, nicht nach dem Prinzip First Come First Serve verteilt werden. In dem Fall kommt es zu einer erheblichen Streuung der Wartezeiten, weil manche Anfragen bei der Vergabe von Connection einfach öfter übergangen werden. Unter diesen Umständen ist es schwierig, zugesicherte Antwortzeiten einzuhalten.

Einfach ist die Ressourcen-Situation dann, wenn für jede Anfrage genau eine Ressource erforderlich ist: In dem Fall ist es sinnvoll, den Pool mit genau der Anzahl an Connections auszurüsten, wie parallele Anfragen bearbeitet werden können.

Erkenntnisse

Performance-Eigenschaften einer Backend-Anwendung lassen sich durch Lastkurven beschreiben. Die Last-Bereiche geben eine gute Orientierung, um den Betriebspunkt in Abhängigkeit von den Performance-Anforderungen festzulegen. Der Betriebspunkt wird dadurch festgelegt, dass die maximale Last limitiert wird. Der System-Integrator wird die Last durch die Größe des Thread-Pools begrenzen. Ist die reale Last höher, so kann die Antwortzeit nur dadurch garantiert werden, dass überzählige Anfragen abgewiesen werden.

Maßnahmen zur Verbesserung der Perfomance sind abhängig davon, in welchem Last-Bereich Performance-Probleme auftreten: Im Last-Bereich 2 wird die Performance durch die Bereitstellung weiterer Rechner-Ressourcen (etwa horizontale Skalierung) erreicht. Dagegen ist im Last-Bereich 1 eine Performance-Verbesserung nur möglich, indem entweder ein schnellerer Prozessor bereitgestellt wird oder aber die Antwortzeit einer einzelnen Anfrage verringert wird. Im letztgenannten Fall konzentriert sich zunächst die Suche auf die Verringerung von Wartezeiten (zum Beispiel durch Performance-Tuning der Datenbank).

Der Durchsatz einer Anwendung ist oft unabhängig von der Belastung. Er wird durch die verfügbare Leistung P und die spezifische Arbeit w bestimmt. Der Ziel-Durchsatz kann aber nur dann erreicht werden, wenn die Last entsprechend hoch ist! Im Last-Bereich 1 steigt der Durchsatz proportional mit der Anzahl der parallelen Anfragen. Eine Anwendung kann einen Durchsatz nur garantieren, wenn zugleich eine gewisse Belastung eingefordert wird.

Quellen

[1] Titelbild - Free photo bar graph statistics analysis business concept | Bild von *rawpixel.com* von *Freepik.com*

[2] Abbildung 1
Gruppe von Menschen, die Medikamente in der Apotheke kaufen | Bild von Hispanolistic von *iStockphoto.com*

Dr. Stefan Koch
ORDIX AG
info@ordix.de

Dr. Stefan Koch ist als Principal Consultant bei der ORDIX AG beschäftigt. Als Bereichsleiter für Application Development betreut Dr. Stefan Koch eine Vielzahl der Entwicklungsprozesse des Unternehmens. Neben seiner Referententätigkeit im Seminarzentrum der ORDIX AG ist er auch auf Konferenzen als Speaker tätig.

Mitglieder des iJUG

01 BED-Con e.V.
02 Clojure User Group Düsseldorf
03 DOAG e.V.
04 EuregJUG Maas-Rhine
05 JUG Augsburg
06 JUG Berlin-Brandenburg
07 JUG Bremen
08 JUG Bielefeld
09 JUG Bonn
10 JUG Darmstadt
11 JUG Deutschland e.V.
12 JUG Dortmund
13 JUG Düsseldorf rheinjug
14 JUG Erlangen-Nürnberg
15 JUG Freiburg
16 JUG Goldstadt
17 JUG Görlitz
18 JUG Hannover
19 JUG Hessen
20 JUG HH
21 JUG Ingolstadt e.V.
22 JUG Kaiserslautern
23 JUG Karlsruhe
24 JUG Köln
25 Kotlin User Group Düsseldorf
26 JUG Mainz
27 JUG Mannheim
28 JUG München
29 JUG Münster
30 JUG Oberland
31 JUG Ostfalen
32 JUG Paderborn
33 JUG Saxony
34 JUG Stuttgart e.V.
35 JUG Switzerland
36 JSUG
37 Lightweight JUG München
38 SUG Deutschland e.V.
39 JUG Thüringen
40 JUG Saarland
41 JUG Duisburg

iJUG Verbund
www.ijug.eu

Impressum

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

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

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

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Fried Saacke
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, Bennet Schulz

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

Bildnachweis:
Titel: Bild © macrovectorart
https://123rf.com
S. 14 + 15: Bild © yewkeo
https://stock.adobe.com
S. 22 + 23: Bild © Designed by upklyak
https://freepik.com
S. 28 + 29: Bild © pickup
https://stock.adobe.com
S. 36 + 37: Bild © Designed by Freepik
https://freepik.com
S. 40 + 41: Bild © Designed by Freepik
https://freepik.com
S. 46 + 47: Bild © ribkhan
https://stock.adobe.com
S. 54 + 55: Bild © Mykyta
https://stock.adobe.com

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org

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

Druck:
WIRmachenDRUCK GmbH
www.wir-machen-druck.de

Inserentenverzeichnis

JavaLand

www.javaland.eu

IM PHANTASIALAND BEI KÖLN 27. - 29.2.2024

Early Bird bis 21.12.2023

Präsentiert von: DOAG Heise Medien Community Partner: iJUG Verbund