Java aktuell

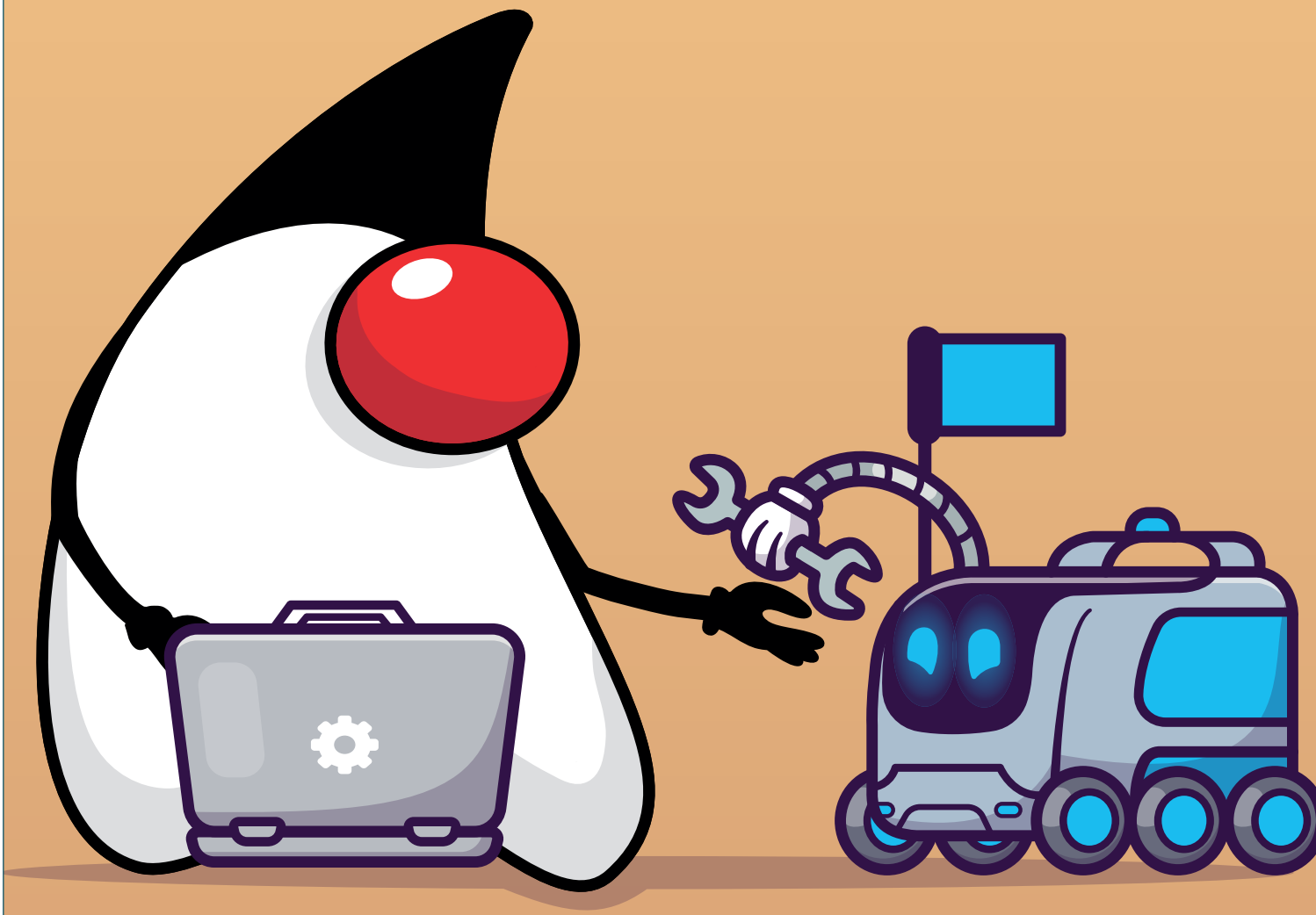
iJUG Verbund
www.ijug.eu

Tools
Shell, GitHub Actions mit Dependabot

Testing
Playwright Java, Test-First-Strategie

Cloud Native
Cloud-Native-Applikationen mit Spring Boot 3 und GraalVM

TOOLS



Ausgabe 03/2024
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 03

10 JAHRE JAVALAND

JavaLand

www.javaland.eu

JAVALAND 2024 VERPASST?

ALLE ON-DEMAND-ANGEBOTE IM TICKETSHOP

JETZT ON-DEMAND-TICKET BUCHEN UND
VORTRAGSAUFZEICHNUNGEN ANSCHAUEN!

Präsentiert von: iJUG Verbund · Heise Medien · DOAG

Veranstalter: JavaLand

Grüezi liebe Leserinnen und Leser,

in dieser Ausgabe begrüßt euch einmal nicht Lisa Damerow. Ende 2018 hat sie die Leitung der *Java aktuell* Redaktion übernommen und seither sehr viel Herzblut in jede Ausgabe gesteckt. Und es werden auch hoffentlich noch ganz viele weitere Ausgaben folgen, in denen sie euch begrüßt! Auch, wenn Lisa als das Gesicht der *Java aktuell* bezeichnet werden kann, wird sie noch von einem kleinen Team unterstützt: dem Redaktionsbeirat.

Alle Mitglieder des Beirats stammen aus Java User Groups, die Mitglied des iJUG e.V. sind. Gemeinsam überlegen wir uns potenzielle Schwerpunkte zukünftiger Ausgaben und sprechen mögliche Autorinnen und Autoren an. Dabei versuchen wir für jede Ausgabe unser Wissen über technische Themen und was die Community gerade beschäftigt, so einzubringen, dass ein möglichst interessanter Themenmix für euch entsteht. Sehr gerne sind wir offen für Feedback von eurer Seite, wie gut wir das mit jeder Ausgabe schaffen, schreibt uns einfach eine E-Mail an redaktion@ijug.eu.

Aktuell besteht unser Redaktionsbeirat aus acht Mitgliedern: Andreas Badelt, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, Bennet Schulz und mir. Vielleicht habt ihr Lust, unseren Beirat zu ergänzen und so an zukünftigen Ausgaben mitzuarbeiten? Es ist schon jedes Mal ein großartiges Gefühl, die neue Ausgabe in Händen zu halten und zu wissen, dass ich daran mitgearbeitet habe und dann die Artikel von den Autorinnen und Autoren zu lesen, die ich vorher angesprochen habe. Die Zeit, die ich dabei dem Redaktionsbeirat widme, darf ich wie jedes andere Mitglied auch selbst bestimmen und der Aufwand darf auch von Ausgabe zu Ausgabe variieren. Es ist also kein Problem, mal mehr und mal weniger beizutragen oder zwischendurch eine Pause einzulegen.

Natürlich dürft ihr jederzeit auf jeden von uns zukommen. Wir erzählen euch gerne mehr und freuen uns auf „frisches Blut" bei uns im Team. Wenn ihr auch zur JavaLand zum Nürburgring kommt, könnt ihr uns auch dort ansprechen. Mich findet ihr zum Beispiel häufig im JavaLand-Studio. Sagt doch mal Hallo! Wir freuen uns alle auf einen kleinen Schnack!

Doch genug über uns und nun zum Thema dieser Ausgabe: Tools. Jeder von uns benutzt sie jeden Tag, meist, ohne viel darüber nachzudenken. In dieser Ausgabe haben wir für euch einige Artikel von Autorinnen und Autoren zusammengestellt, die etwas mehr darüber nachgedacht haben. Hast du beispielsweise schon vom ToolProvider des Java SDK gehört? Nein? Bernd Müller erklärt euch ab Seite 10, was es damit auf sich hat. Viele unserer täglichen genutzten Tools rufen wir auf der Kommandozeile auf. Sandra Parsick führt uns ab Seite 16 sicher per Shell durch die Developer-Galaxie. Jörg Liedl widmet seinen Artikel ab Seite 26 dem oftmals leidigen Thema, wie man Abhängigkeiten in Projekten aktuell halten kann – mit GitHub Actions und Dependabot. Nicht gespielt, aber getestet wird mit Frederik Hahne und Playwright, einem Tool für End-2-End-Testing ab Seite 34, gefolgt von Sascha Selzer mit einem Blick hinter die Bühne mit Backstage.

Abgerundet wird die aktuelle Ausgabe, wie ihr es kennt, von einem Mix vieler weiterer Artikel aus unterschiedlichen Themenbereichen. Wir hoffen, dass für euch viel Interessantes dabei ist, und wünschen euch viel Spaß beim Lesen dieser Ausgabe!

Marcus Filohn
Redaktionsbeirat Java aktuell

INHALT

Einstieg in die Shell: Tipps & Tricks

Backstage: Einführung in das Open-Source-Framework

3 Editorial

6 Java-Tagebuch
Andreas Badelt

8 Markus' Eclipse Corner
Markus Karg

10 Unbekannte Kostbarkeiten des SDK Heute: Der ToolProvider
Bernd Müller

14 Die goldenen Regeln: Teil 1
Andreas Monschau

16 Per Shell durch die Developer-Galaxie
Sandra Parsick

26 Automatische Dependency-Updates mit GitHub Actions und Dependabot
Jörg Liedl

34 End-2-End-Testing mit Playwright Java
Frederik Hahne

42 Backstage – Nur ein besseres Wiki?
Sascha Selzer

Entiwcklung Cloud-nativer Applikationen mit Spring Boot 3 und GraalVM

Story Points: Homöopathie in der Softwareentwicklung?

50 Eclipse Adoptium: Ein Leuchtturmprojekt für sichere und transparente Softwarelieferketten
Scott Fryer & Hendrik Ebbers

56 Cloud-Native-Apps mit Spring Boot 3 und GraalVM
Daniel Krämer

64 Invoke Dynamic
Bernd Müller

70 Test First?
Marco Schulz

76 Anforderungsmanagement in unübersichtlichen Systemen
Stephan Tönnies

80 Story Points und Homöopathie
Susanne Pfeiffer

87 Impressum/Inserenten

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

4. Dezember 2023

Pattern Matching

Wer sich für die Details von Pattern Matching für instanceof und switch statements interessiert, sollte sich den Blogeintrag von Cay Horstmann *[1]* anschauen. Anhand von kleinen Programmierrätseln werden die Grundprinzipien erklärt und wie sie in die existierende Syntax eingebettet wurden. Cay schreibt auch detailliert, was ihm schon früh an der Spezifikation missfallen hat (ändert jetzt aber auch nichts mehr), welche Fallstricke damit ausgelegt wurden und wie sie zu umgehen sind.

7. Dezember 2023

Kotlin WebAssembly in der Alpha-Phase

Fortschritte bei Kotlin WebAssembly: Das Projekt, um in Kotlin Code zur Ausführung im Browser zu schreiben (und auch darüber hinaus – beispielsweise mit WebAssembly Runtimes für Kubernetes), befindet sich jetzt im Alpha-Status. Nachdem Kotlin Multiplatform vor ein paar Monaten für produktionsreif erklärt wurde, ist dies für eine weitere Target-Platform zumindest ein Stück näher gerückt.
Das hängt jetzt nicht mit dem Thema WebAssembly zusammen, aber auch die generelle Unterstützung von Kotlin durch die großen Cloud-Anbieter wächst weiter. Erst kürzlich hat AWS mit dem „SDK für Kotlin" nachgelegt.

10. Dezember 2023

Babylonische Entwirrung

Nicolai Parlog hat vor kurzem in seinem Newscast ein leicht verständliches und unterhaltsames Einführungsvideo zu Projekt Babylon veröffentlicht (Öffnung von Java für andere Programmiermodelle, insbesondere für die Ausführung von Code auf GPUs). Wem das Lesen der JEP-Dokumentation zu sperrig ist und wer stattdessen nach einem schnellen Überblick sucht (das Video ist knapp 10 Minuten lang), wird hier fündig *[2]*.

13. Dezember 2023

Bring den Müll raus, aber hurtig!

Stefan Johannson vom HotSpot GC Team bei Oracle vergleicht in seinem Blog die Änderungen in der Garbage-Collector-Performance zwischen JDK 8, 17 und 21 – jeweils für den G1, den Parallel und den Z GC (letzteren gab es in JDK 8 noch lange nicht, dafür wird hier der neue Generational ZGC herangezogen). Unter Zuhilfenahme der SPECjbb 2015 Benchmark werden reiner Throughput, Latency und Memory Overhead unter die Lupe genommen. Knapp zusammengefasst hat der G1 über die Releases besonders große Sprünge gemacht, für Applikationen auf Java 21 lohnt sich aber definitiv ein Blick auf den Generational ZGC *[3]*.

30. Dezember 2023

Jahresrückblick mit Spring

Zum Jahresende gibt es auch für das Spring-Projekt einen Rückblick. Mal sehen, was die Highlights aus Sicht von Josh Long waren: Zuallererst steht da tatsächlich Artificial Intelligence. Nichts, was man direkt mit Spring in Verbindung bringen würde. Doch auch hier darf das Thema nicht fehlen – es gibt seit letztem Sommer ein Spring-AI-Projekt (aktuell Version 0.8), das sich um die nötigen Abstraktionen und Prozesse (insbesondere Retrieval Augmented Generation [RAG]) rund um die Nutzung von LLMs kümmert. Hier wird die Fahne im Java-Umfeld aktuell von langchain4j hochgehalten – aber es gibt auf jeden Fall Luft nach oben und eine „konkurrierende" Implementierung mit direkter Integration in Spring klingt schon mal gut.

Was war noch? Ohne Kommentare im Schnelldurchlauf: Die Produktionsreife des Native Image Support für die GraalVM – (Spring Boot 3.0). Java 21 und die Unterstützung in Spring Boot, insbesondere die einfache Aktivierung von virtuellen Threads per Konfiguration in 3.2. Ebenfalls in Boot 3.2: die Unterstützung von CraC (Coordinated Restore at Checkpoint) als Alternative zu den Native Images, wenn es „nur" um schnelle Startzeiten geht. Die verbesserte Unterstützung von „Docker Driven Development" mit automatischer Auswertung von docker-compose oder Testcontainers-Einträgen. Und Spring Modulith, das im August offiziell freigegeben wurde.

10. Januar 2024

Java im Programmiersprachen-Ranking

Java hat im TIOBE-Index 2023 noch mal ordentlich Federn gelassen – TIOBE versucht monatlich, über diverse Suchmaschinen und (Kurs-)Anbieterseiten im Web die Popularität der Sprachen zu messen und im Index vergleichbar zu machen *[4]*. Größter Aufsteiger des letzten Jahres war C#, das seinen Negativtrend wieder umkehren konnte und seit 2020 stark „wächst". Laut TIOBE wird es vermehrt (anstelle von Java) in Web-Backends eingesetzt sowie in Spielen (vor

allem dank des Unity Frameworks). Zum Glück hat Heise direkt eine Übersicht über weitere Rankings (PYPL, GitHub) veröffentlicht, die mich wieder optimistisch stimmt, dass ich auch zukünftig noch genug zu schreiben habe. Der Anteil von Kotlin steigt laut TIOBE zwar kontinuierlich, aber noch auf niedrigem Niveau, zwischen Swift und Rust, während Scala in den letzten Jahren (auf noch niedrigerem Niveau) stagniert.

18. Januar 2024

Java 22 im Plan

JDK 22 ist jetzt plangemäß in „Rampdown Phase 2" und wird dann vermutlich auch pünktlich am 19. März erscheinen. Die im letzten Tagebuch als weitere Kandidaten erwähnten Spezifikationen „Launch Multi-File Source-Code Programs", „Structured Concurrency (Second Preview)" und „Scoped Values (Second Preview)" sind alle noch ins Release aufgenommen worden.

24. Januar 2024

sun.misc.Unsafe

Die „unsupported" Memory-Access-Methoden in Javas berüchtigter Unsafe-Klasse sollen demnächst als deprecated markiert werden (für Java 22 kommt der JEP allerdings zu spät) und dann in einem folgenden Release tatsächlich entfernt werden. Ersatz durch offiziell unterstützte Methoden gibt es seit JDK 9 (On-heap-Memory) beziehungsweise mit JDK 22 (off-heap). Unsafe wird damit noch nicht komplett verschwinden, aber von 87 auf 8 Methoden schrumpfen. *[5]*

28. Januar 2024

AQAvit wird reif

Ja, auch AQAvit kann reifen. Das Teilprojekt von Eclipse Adoptium, mit dem die Java Runtimes vor einem Release komplett durchgecheckt werden, soll den „Inkubator" verlassen. Das heißt aber nicht, dass es bislang technisch noch unausgegoren war, hier geht es mehr um das Projekt als solches, seine Prozesse, Branding und so weiter.

30. Januar 2024

Jakarta EE 11 und darüber hinaus

Jakarta EE 11 soll im Juni oder Juli freigegeben werden, etwas später als ursprünglich mal geplant. Ansonsten scheint aber alles wie geplant zu laufen. Allerdings steht noch ein Review und Votum durch das Spezifikations-Komitee an – ein Restrisiko für den Zeitplan, sollte der erste Wurf abgelehnt werden. Eine signifikante Änderung hat sich kürzlich noch ergeben: Neben Java SE 21 wird jetzt doch auch noch die vorherige LTS-Version 17 unterstützt. Laut „Projektleiter" Ivar Grimstad stellt das aber in der Praxis kein Problem dar, da keine speziellen Features von SE 21 direkt in den APIs Verwendung finden. Virtuelle Threads können trotzdem genutzt werden, wenn die darunterliegende VM sie unterstützt.

Ideen für die folgenden Releases gibt es aber auch schon und irgendwie springen mich da ein paar Stichworte an: Total überraschend ist der Vorschlag zu einer „Jakarta AI"-Spezifikation. Auch CRaC und InstantOn wünscht sich Ivar als „Bürger erster Klasse" in Jakarta, um Startup-Zeiten zu reduzieren. Aber es gibt tatsächlich noch viele weitere Ideen, zum Beispiel rund um Robotik, Edge Computing, „Messaging Lite" und viele Weitere. Mehr Infos stehen im Google Doc, in dem sie gesammelt werden *[6]*.

31. Januar 2024

Java 23

Die offizielle Liste für das JDK 23 ist noch ziemlich leer. Nur „Primitive Types in Patterns, instanceof, and switch (Preview)" sind bislang auf der offiziellen Liste. Aber auf der Wunschliste bei Oracle, beziehungsweise dem OpenJDK-Projekt, stehen wohl unter anderem „Condensers" aus Project Leyden *[7]* für schnellere Startup-Zeiten und geringeren Speicherhunger, Project Liliput, das durch kleinere Objekt-Header ebenfalls den Speicherbedarf senken soll, Value Objects (Instanzen, die nur „final" Instanzfelder haben), und weitere Arbeiten an Project Panama (Interkonnektivität mit nativem Code). Außerdem wird Projekt Babylon mit dem AI-Hype natürlich besonders im Fokus stehen; mal sehen, wie schnell sie liefern können.

Referenzen

[1] *https://horstmann.com/unblog/2023-12-01/index.html*
[2] *https://inside.java/2023/11/16/newscast-58/*
[3] *https://kstefanj.github.io/2023/12/13/jdk-21-the-gcs-keep-getting-better.html*
[4] *https://www.tiobe.com/tiobe-index/*
[5] *https://openjdk.org/jeps/8323072*
[6] *https://docs.google.com/document/d/1tVaCmbXdAS_Ddn7LMP-DelpOfAyozChXL*
[7] *https://openjdk.org/projects/leyden/*

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

Ich hatte euch in der letzten Ausgabe ja versprochen, dass ich (nach der Ausnahme beim letzten Mal) selbstverständlich nun wieder über Jakarta EE meckern werde. Nun denn, so sei es!

Es ist ein unglaubliches Hin und Her mit Jakarta EE 11. In der letzten Ausgabe berichtete ich darüber, dass ich zuversichtlich bin, weil es einerseits ein Diskussionsforum geben soll, in dem sich auch der iJUG (endlich) etwas aktiver einbringen kann (und somit sein nominelles Wahlreicht auch besser nutzen kann), andererseits war EE 11 inklusive Support für Java SE 21 durchaus im (bereits verschobenen) Zeitplan. In dieser Ausgabe nun sieht das Bild aber leider schon wieder ganz, ganz anders aus!

Das Diskussionsforum (das übrigens mittels einer Bridge sowohl mit Slack- als auch mit Matrix-Clients besucht werden kann – der gleichen Technik übrigens, die auch der iJUG betreibt und seinen Mitgliedern bereitstellt) existiert mittlerweile nun schon einige Monate und ist leider kaum besucht. Die Anzahl der Nachrichten lässt sich an einer Hand abzählen. Ein sinnvolles Einbringen des iJUG war bislang noch nicht zu beobachten. Warten wir ab, ob da mal noch „was geht". Im Moment jedenfalls herrscht dort ziemliche Flaute, von allen Seiten. Schade, wieder eine Chance vertan (auch hier: von allen Seiten).

Auf der anderen Seite gab es aber durchaus Hektik bei den Herstellern. Die haben nämlich kürzlich bemerkt, dass es durchaus Arbeit macht (bei Herstellern wie auch bei Kunden), wenn man auf Java 21 wechseln will und natürlich ist auch die CDI-Zwangsunterstützung in JAX-RS nicht „für lau" zu erhalten (wir erinnern uns: *JAX-RS* will seit vielen Jahren @Context loswerden, weil es für Anwender völlig unverständlich ist, dass es zwei unabhängige Injection Frameworks in *JAX-RS* und somit in Jakarta EE gibt). Und weil die Hersteller schon lange wussten, dass das sehr aufwändig (und damit teuer) wird, haben sie natürlich ihre Verantwortung wahrgenommen und rechtzeitig vor vielen Monaten schon genug Personal eingesetzt sowie zumindest mal damit angef… Nein, haben sie natürlich NICHT!

Stattdessen hat man bekanntlich vor einigen Wochen erst den Zieltermin verschoben und war sich nun im Kreis der Hersteller hübsch einig, dass man doch lieber ganz auf Java 21 verzichten möchte. Entsprechend schraubte man die Mindestanforderungen nun wieder auf Java 17 zurück – und nimmt sich selbst die Chance, *Virtual Threads* und andere nette Features auf breiter Front bekannt zu machen. Ein Schritt vor, zwei zurück also.

Im Moment herrscht nun im *JAX-RS*-Forum (JAX-RS ist neben CDI und JSON-B der Dreh- und Angelpunkt der Jakarta-EE-Plattform) ein gewisses Chaos, denn nachdem sich die Hersteller nun darauf verständigt haben, nicht das Gesicht zu verlieren, indem sie Java 21 aus *JAX-RS 4* zu streichen, hat man lieber hinausposaunt, der Markt würde doch eher ein JAX-RS 3.2 auf Basis von Java 17 wollen. Dumm nur, dass das gar nicht geht: JAX-RS 3.x basiert auf Java 11, und da im Hause Jakarta inzwischen *SemVer* angesagt ist, darf man von 11 gar nicht auf 17 hochziehen, da existierender Binärcode nicht mehr geladen werden kann. Dumm gelaufen! Was bleibt, wäre zumindest CDI verbindlich anbieten zu müssen, doch auch hier gab es schon von verschiedenen Seiten Kontra. Und nun versucht man allen Ernstes, binnen weniger Wochen einen @Context-Ersatz aus der Taufe zu heben, der durch CDI bereitgestellt wird. Es sollte jedem klar sein, der ernsthaft komplexe Software entwickelt, dass das unter solchem Zeitdruck zum Scheitern verurteilt ist! Natürlich ist alles derzeit rein theoretisch und vollständig gegen das Credo der EF „Code First": Man macht mal wieder „Spec First" – kein Hersteller und kein Produkt unterstützt das bis zum Zieltermin von Jakarta EE 11!

Mein Blick in die Kristallkugel sagt: Wenn ihr diese Zeilen lest, wird sich sehr wahrscheinlich der Plan von Jakarta EE 11 so geändert haben, dass entweder weiterhin das betagte JAX-RS 3.1 enthalten ist oder ein JAX-RS 3.2, bei dem sich außer der Versionsnummer nichts geändert hat. In beiden Fällen fragt man sich dann aber: Wer braucht das? Außer dem Marketing-Komitee der Jakarta Working Group vermutlich niemand. Was wir als Anwender brauchen, sind Features. Und genau diese wollen die Hersteller aber nicht mehr bezahlen. Nur sagen die euch das nicht. Daher mache ich das!

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.

*Weinberge, Innovation und Vernetzung aller, deren Herz für die IT und Softwareentwicklung schlägt - Auf Sachsens größter IT-Community-Konferenz vom 26. - 27. September 2024 in Radebeul bei Dresden.

Unbekannte Kostbarkeiten des SDK "Heute: Der ToolProvider"

Bernd Müller – Ostfalia

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

Unsere heutige Ausgabe der unbekannten Kostbarkeiten widmet sich dem *ToolProvider*. Ein Interface, das es auf standardisiertem Weg erlaubt, bestehende Kommandozeilenwerkzeuge des JDK, aber auch eigene Werkzeuge, über ein öffentliches API aus einem laufenden Programm heraus aufzurufen. Man erspart sich also eine zweite laufende JVM.

Der ToolProvider

Das Interface *ToolProvider* im Package `java.util.spi` beschreibt sich selbst als: *„An interface for command-line tools to provide a way to be invoked without necessarily starting a new VM."* Doch welche Werkzeuge sind gemeint? Der erfahrene Java-Entwickler weiß, dass das OpenJDK in seinem `bin`-Verzeichnis eine ganze Reihe außerordentlich hilfreicher Werkzeuge bereitstellt. Neben `javac` und `java` etwa auch `jar`, `javap`, `jcmd`, `jconsole`, `jps` und viele, viele mehr.

Im OpenJDK Version 21 zählen wir 29 solcher Werkzeuge. Alle Werkzeuge sind ausführbare Binärprogramme (keine Shell-Skripte), die allerdings eine JVM starten. Über `Runtime#exec()` können Betriebssystemprogramme aus einem Java-Programm heraus gestartet werden. Wird dieser Weg genutzt, um eines der genannten Werkzeuge zu starten, laufen auf dem System jedoch zwei JVMs. Das Interface ToolProvider wurde geschaffen, um dies zu vermeiden und mit nur einer JVM auszukommen.

Das Interface im Überblick

Das Interface ToolProvider wurde mit Java 9 eingeführt und basiert auf dem durch die Klasse `ServiceLoader` definierten Mechanismus zum Laden von Diensten. Das Interface umfasst vier (überladene) Methoden, die in *Abbildung 1* dargestellt sind. Die Default-Methode `description()` liefert ein leeres Optional zurück. Die Default-Methode `run()` ruft die normale `run()`-Methode auf, die eine Instanz des Werkzeugs aufruft. Die ersten beiden Parameter sind für die Ausgaben und Fehlermeldungen des Werkzeugs gedacht. Der VarArgs-Parameter repräsentiert die Parameter der entsprechenden Kommandozeilenversion. Dies ist schon alles. Wir werden das API später an Beispielen verdeutlichen.

Stand der Dinge

Im OpenJDK Version 21 sind acht ToolProvider implementiert, die in *Tabelle 1* aufgeführt sind. Wir haben auch ihre jeweilige Implementierung angegeben, da einige so alt sind, dass ihre Package-Namen interessanterweise noch auf Sun basieren. Bis auf jpackage wurden alle ToolProvider, wie das Interface selbst, mit Java 9 eingeführt. jpackage wurde mit Java 14 nachgeschoben. Das Werkzeug war aber auch erst mit Java 14 released worden.

Der Quellcode des OpenJDK enthält auch die Klasse `JshellToolProvider`. Diese wird aber (im Augenblick noch) nicht als `ToolProvider` verfügbar gemacht. Es gab 2016 eine kurze Diskussion auf der OpenJDK-Mailing-Liste, ob `jshell` verfügbar gemacht werden sollte. Dies führte allerdings nicht zur Veröffentlichung von `jshell`.

Ein Beispiel

Nachdem wir nun wissen, was es mit dem `ToolProvider` auf sich hat, wollen wir ein kleines Beispiel entwickeln. Das Werkzeug `javadoc` wird von uns in der Regel über Maven oder Gradle aufgerufen. Als Kommandozeilenwerkzeug hat es eine ganze Reihe von Optionen beziehungsweise Parametern. Zu den wichtigsten gehören `-d` (Destination) für die Angabe des Zielverzeichnisses sowie `-sourcepath` für die Angabe des Quellcode-Verzeichnisses. Letztendlich ist das Package anzugeben, für das das *Javadoc* erzeugt werden soll. Dies sollen auch unsere Beispielparameter sein. Die Main-Methode in *Listing 1* zeigt beispielhaft die Verwendung. Die Methode `findFirst()` erwartet den Namen des Kommandozeilenwerkzeugs. Der VarArg-Parameter der `run()`-Methode ist identisch zum `args`-Parameter der Main-Methode des Kommandozeilenwerkzeugs. Im Beispiel ist dies zunächst das Parameterpaar zur Angabe des Zielverzeichnisses, gefolgt vom Parameterpaar des Quellcode-Verzeichnisses. Diese beiden Parameterpaare könnten in der Reihenfolge vertauscht werden. Der letzte Parameter ist der Package-Name, für den das *Javadoc* zu erstellen ist *(siehe Listing 1)*.

ToolProvider	Implementierende Klasse
jar	sun.tools.jar.JarToolProvider
javac	com.sun.tools.javac.main.JavacToolProvider
javadoc	jdk.javadoc.internal.tool.JavadocToolProvide
javap	com.sun.tools.javap.Main$JavapToolProvider
jdeps	com.sun.tools.jdeps.Main$JDepsToolProvider
jlink	jdk.tools.jlink.internal.Main$JlinkToolProvider
jmod	jdk.tools.jmod.Main$JmodToolProvider
jpackage	jdk.jpackage.internal.JPackageToolProvider

Tabelle 1: Liste der implementierten ToolProvider

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method	Description
default Optional<String>	description()	Returns a short description of the tool, or an empty Optional if no description is available.
static Optional<ToolProvider>	findFirst(String name)	Returns the first instance of a ToolProvider with the given name, as loaded by ServiceLoader using the system class loader.
String	name()	Returns the name of this tool provider.
default int	run(PrintStream out, PrintStream err, String... args)	Runs an instance of the tool, returning zero for a successful run.
int	run(PrintWriter out, PrintWriter err, String... args)	Runs an instance of the tool, returning zero for a successful run.

Abbildung 1

```
public static void main(String[] args) {
     Optional<ToolProvider> javadoc = ToolProvider.findFirst("javadoc");
     String cwd = System.getProperty("user.dir");
     String src = cwd + "/src/main/java";
     String apidoc = cwd + "/target/site/apidocs";
     javadoc.get().run(System.out, System.err,
        "-d", apidoc, "-sourcepath", src, "de.pdbm.toolprovider");
}
```

Listing 1

```
public class TrickyToolProvider implements ToolProvider {

     @Override
     public String name() {
        return "tricky";
     }

     @Override
     public Optional<String> description() {
        return Optional.of("Runs the very tricky Tool");
     }

     @Override
     public int run(PrintWriter out, PrintWriter err, String... args) {
        return Tricky.execute(args, System.out, System.err);
     }

}
```

Listing 2

Eigene ToolProvider

ToolProvider ist ein allgemein verwendbares Interface, sodass auch eigene Implementierungen erstellt werden können. Wenn Sie also ein in Java geschriebenes Werkzeug haben, das Sie sowohl von der Kommandozeile als auch aus einem laufenden Programm heraus über ein definiertes API aufrufen wollen, ist der ToolProvider das Mittel der Wahl. Nachdem wir bereits in *Abbildung 1* gesehen haben, dass das ToolProvider-API sehr schlank ist, zeigt *Listing 2* eine beispielhafte Implementierung.

Das Werkzeug selbst wird durch die Klasse `Tricky` realisiert, die wir hier aber nicht zum Besten geben. Interessanter ist der Aufruf dieser Klasse über den `ToolProvider`. Der Aufruf erfolgt in der `run()`-Methode in *Listing 2*. Die `execute()`-Methode haben wir strukturell an die im OpenJDK vorhandenen `ToolProvider`-Implementierungen angelehnt. Die Java-Main-Methode halt als einzigen Parameter ein String-Array. Die Kommunikation mit der Umgebung erfolgt über die Variablen `in`, `out` und `err` der Klasse `System`. Wenn wir über den `ToolProvider` ein derartiges Werkzeug aus einer laufenden JVM aufrufen, müssen die Standardausgabe sowie die Fehlerausgabe übergeben werden. Dies erfolgt über die Methode `execute()`, die wir, wie gesagt, an die entsprechenden `ToolProvider`-Implementierungen des OpenJDK angelehnt haben.

`ToolProvider` werden über den `ServiceLoader`-Mechanismus geladen. Daher muss noch im Verzeichnis `META-INF/services` eine Datei `java.util.spi.Tool.Provider` angelegt werden, die als einzige Zeile den Namen der `Tricky`-Klasse enthält. Bei Bedarf können Einzelheiten im *Javadoc* der Klasse `ServiceLoader` nachgelesen werden. Das ist auch schon alles. Aufrufe von der Kommandozeile oder auch aus einem laufenden Programm heraus sind nun möglich und der Aufruf aus einem laufenden Programm heraus startet keine zweite JVM *(siehe Listing 2)*.

Falls es Sie in den Fingern juckt und Sie gerne alles selbst ausprobieren wollen, enthält unser Projekt auf GitHub *[1]* den kompletten Quellcode der Beispiele dieses Artikels.

Zusammenfassung

Mit Java 9 wurde das Interface *ToolProvider* eingeführt. Es erlaubt den einfachen Aufruf von Kommandozeilenwerkzeugen, ohne eine weitere JVM starten zu müssen. Eine Reihe von Werkzeugen, die mit dem JDK oder einem JRE ausgeliefert werden, sind hierüber aufrufbar. Die Implementierung eigener `ToolProvider` ist ebenfalls leicht möglich.

Referenzen

[1] *https://github.com/BerndMuller/toolprovider*

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

Die goldenen Regeln: Teil 1

Andreas Monschau, Haeger Consulting

Stetig bin ich auf der abenteuerlichen Suche nach ihnen – den goldenen Regeln, mit denen man Neulingen, Junioren oder Quereinsteigern den Start möglichst vermiesen kann. Softwareentwicklung ist hart und unfair. So soll es auch bleiben. Du hast gelitten, also sollen alle leiden. Diese Regeln entspringen nicht meiner bunten Fantasie, sie finden sich noch in vielen Projekten, Unternehmen und Organisationen wieder – mit dieser Sammlung möchte ich zum einen erheitern und zum anderen mahnen. Mahnen, diese Regeln nicht mehr anzuwenden, und Menschen, die darunter leiden ermutigen, sich zu wehren. Hast du weitere Beispiele für Regeln und möchtest sie mit mir und anderen teilen? Dann kontaktiere mich gerne!

Deine konkrete Rolle ist da egal: Ob du nun Projektmanager, Teamleiter, Entwickler, PO oder auch Tester bist – solltest du derjenige sein, der den Neuling begrüßt, lade ich dich herzlich ein, die folgenden Hinweise, Tipps und Tricks zu beherzigen. Nicht jeder Tipp wird passen, such dir einfach das für dich Beste raus!

Regel 1: Sorge für ein kaltes Willkommen

Am ersten Tag für einen Neuling gilt es nicht nur, technische und organisatorische Einstiegshürden zu überwinden (was das Nichtvorhandensein von Equipment, Accounts und Zugängen betrifft, darauf gehe ich in einem späteren Teil dieser Serie ein), sondern man kann an dieser Stelle auch schon auf verbaler Ebene sofort die Saat für eine langfristige Demotivation sorgen. Schon ein erster Satz der Art „Naja, hoffentlich kannst du wenigstens etwas", statt jemanden nett zu begrüßen, ist da ein guter Einstieg.

Weise auf jeden Fall darauf hin, wie schlecht doch die Atmosphäre ist, die im Team herrscht. Wenn du mit jemand aus dem Team Beef hast, kannst du hier auch schon mal deine Meinung kundtun und der Unvoreingenommenheit des Neulings entgegenwirken, zum Beispiel so: „Du wirst nachher noch den <hier beliebigen Namen einfügen> kennenlernen – der kann nichts, hält sich aber für den Größten! Das ist eine Diva!".

Du bist natürlich ein Projektveteran und hast schon viele kommen und gehen sehen – und du beobachtest, wie das Projekt immer weiter bergab läuft. Hinweise darauf sind auch wichtig (unerheblich, ob sie Tatsachen entsprechen oder du dein „Gefühl" sprechen lässt), zum Beispiel indem du Aussagen wie „Wir sind völlig out of budget!" (ganz egal, ob du Informationen über die finanzielle Lage hast) oder „Wir fahren gegen die Wand!" (unabhängig davon, ob du das „große Ganze" überhaupt kennst) kommen immer gut an. Und wie gesagt, Softwareentwicklung ist hart! Aussagen der Art „Wir erwarten hier Leistung! Wer nicht liefert, fliegt, nur die Harten bleiben!" entsprechend dem Gedanken *Survival of the fittest* – Darwin würde das gefallen.

Bonus: Sollte es sich bei dem Neuling um einen externen Mitarbeiter handeln, hast du noch viele weitere Instrumente, dich bei der Begrüßung abwertend zu verhalten: „Naja, du bist ohnehin nur eine temporäre und vor allem austauschbare Ressource, über die wir hier frei verfügen können", zählt noch zu den harmloseren Varianten.

Solltest du jetzt so weit inspiriert worden sein, dass du weiter darüber nachdenkst, dann fallen dir mit Sicherheit noch mehr abschreckende Dinge ein, die man einem Neuling schon am ersten Tag entgegenwerfen kann. Dabei ist es egal, ob du lügst, übertreibst oder lediglich deine schlechte Laune beim Neuling loswerden willst – einfach raus damit! Vermiese dieser Person den Einstieg!

Im nächsten Teil dieser Artikelserie beschäftige ich mich mit der Einsamkeit von Neulingen – sei gespannt!

Andreas Monschau
Haeger Consulting
amonschau@haeger-consulting

Andreas Monschau ist seit über 10 Jahren als Senior IT-Consultant mit den Schwerpunkten Softwarearchitektur- und Entwicklung sowie als Teamleitung bei Haeger Consulting in Bonn tätig und aktuell als Solution Designer im Kundenprojekt unterwegs. Neben seinen Projekten leitet er das umfangreiche Traineeprogramm des Unternehmens und ist als Sprecher und Autor unterwegs.

Per Shell durch die Developer-Galaxie

Sandra Parsick

Die Shell ist oft die schnellere Antwort auf viele Probleme im Developer-Alltag. Dennoch meiden viele Techies die Shell aus unterschiedlichen Gründen. Dieser Artikel gibt einen schnellen und opinionated Shell-Einstieg, angefangen mit einem Überblick über Terminals und Shells bis hin zu Programmen für die Kommandozeile, die im Entwicklungsalltag helfen können.

Terminal, die Benutzeroberfläche

Allgemein gesprochen sind Terminals grafische Benutzeroberflächen, um textbasierte Programme, wie zum Beispiel eine *Shell*, zu verwenden. Strenggenommen, müssten Terminals „Terminalemulation" genannt werden, da sie die Funktion eines Computer-Terminals nachbilden *[1]*.

> Ein Computer-Terminal ist ein Endgerät zur Ein- und Ausgabe von Daten und dient als Benutzerschnittstelle, unter anderem für Großrechner. Heutzutage gibt es noch branchenspezifische Terminals wie zum Beispiel Kreditkartenterminals oder Geldautomaten.

Moderne Terminals bieten Komfort in der Bearbeitung der Eingabe (zum Beispiel durch einfaches Kopieren und Einfügen von Befehlen), Anordnung der Eingabefenster, Definition von Shortcuts, Plug-in-Erweiterung, Anpassung des Aussehens (Schriftgröße, Farbe usw.) und vieles mehr.

Mittlerweile gibt es für jedes Betriebssystem Terminals, die diese Funktionalitäten mitbringen. Wenn eine Entwicklerin auf einem Linux-System unterwegs ist, bringt die eingesetzte Desktop-Umgebung ihr eigenes Terminal mit. KDE hat oft *Konsole [2]* vorinstalliert, GNOME das *Gnome Terminal [3]*. Diese Terminals sind stark in der jeweiligen Desktop-Umgebung verankert. Das führt dazu, dass wenn ein Entwickler die Desktop-Umgebung wechseln will, aber seine Terminalsoftware aus der alten Umgebung weiterverwenden möchte, die Terminalsoftware eine Menge an Pakete von der alten Desktop-Umgebung benötigt und diese mitinstalliert werden. Mit *Tilix [4]* gibt es ein Terminal, dass von der Desktop-Umgebung unabhängig ist, aber dennoch die wichtigsten Funktionalitäten eines modernen Terminals mitbringt. Wenn eine Entwicklerin einen Mac benutzt, dann ist als Standard-Terminal *Terminal [5]* installiert. Viele Mac-User bevorzugen aber *iterm2 [6]*. Auf Windows sah es lange Zeit nicht sehr gut im Hinblick auf komfortable Terminals aus. Mittlerweile bietet Microsoft allerdings das *Windows Terminal [7]* an.

Abseits dieser eher klassischen Terminals gibt es zum Beispiel mit *fig [8]*, *warp [9]* und *hyper [10]* Terminals mit neuen Ansätzen, die sich zum Teil wie eine IDE anfühlen. Die Auswahl ist groß und für jeden Geschmack ist etwas dabei.

Shell, die Kommandozeile

Im allgemeinen Sprachgebrauch versteht eine Entwicklerin unter dem Begriff *Shell* die *Befehlszeilenschnittstelle im Textmodus* des Betriebssystems. Laut Definition der Wikipedia ist eine *Shell* die allgemeine Schnittstelle zum Betriebssystem *[11]*. Sie kann auch grafisch sein (GUI). Der Fokus des Artikels liegt aber auf der textbasierten Form.

Allgemein gesprochen, bietet eine Shell eine Umgebung an, in der Benutzer über eine Kommandozeile Befehle eingeben können, die ein Interpreter ausführt. Ein Befehl besteht aus dem Programmnamen gefolgt von *Argumenten* (Beispiel: *siehe Listing 1*).

```
ls -l /home
```

Listing 1: Beispiel für einen Shell-Befehl

Argumente können Optionen (beginnen oft mit - oder --) sein oder Eingabewerte für das Programm. Jede Shell bringt sogenannte Built-in-Befehle mit, das heißt, es handelt sich um Programme für die Kommandozeile, die mit der Shell installiert werden. Es können Programme für die Shell nachinstalliert werden, indem die Programme im sogenannten *Suchpfad* installiert werden.

> Der **Suchpfad** ist eine Liste mit Verzeichnissen, in der die Shell nach Programmen sucht. Der Wert der Shell-Variable PATH definiert diese Liste.

Auf unix-ähnlichen Systemen wie Linux und Mac kommen oft Shells wie *Bash [12]* und *Z shell (zsh) [13]* zum Einsatz. Windows bietet *CMD* und *Powershell* an. Dank dem Windows-Linux-Subsystem (WSL) können Entwickler auch die unix-basierten Shells unter Windows verwenden.

Unter Entwicklern ist die Shell *fish [14]* recht beliebt, da sie Funktionalitäten von Haus aus mitbringt, die bei anderen Shells nachinstalliert werden müssen, wie zum Beispiel Textautovervollständigung oder automatisierte Befehlvorschläge basierend auf den schon benutzten Befehlen. *Bash* und *zsh* sind recht ähnlich in der Benutzung und eine von beiden ist immer die Standardshell auf unix-ähnlichen Systemen. *zsh* bietet mehr Konfigurationsmöglichkeiten an als *Bash* und beim Scripting verhält sich *zsh* im Detail anders als Bash. Im Alltag macht sich der Unterschied kaum bemerkbar.

In der Community um die Shells haben sich Frameworks gebildet, um die Konfiguration der Shell besser zu verwalten (*oh-my-zsh [15]*, *oh-my-bash [16]*, *oh-my-posh [17]* um ein paar zu nennen). Darüber hinaus bieten sie an, die eigene Shell mit *alias*, Plug-ins und Themes zu erweitern. Wie sich dies im Alltag auswirkt, schauen wir uns am Beispiel *oh-my-zsh* an.

Entwickleralltag mit oh-my-zsh

oh-my-zsh wird mithilfe von *curl [18]* und *git [19]* installiert. Die Installation sieht so aus, dass die Installationsroutine das Framework mit allen gebündelten Plug-ins und Themes unter `~/.oh-my-zsh` herunterlädt und die zsh-Konfigurationsdatei `.zshrc` neu angelegt, in der später definiert werden kann, welche Plug-ins und welche Themes benutzt werden.

Ein *alias* ist eine Möglichkeit in der Shell, mehrere Befehle oder einen Befehl mit mehreren Optionen zu einem neuen Befehl zusammenzufassen. *alias* ist eine Art Shortcut, *oh-my-zsh* bringt eine eigene Liste von *aliases* mit, die sich über Plug-ins erweitern lässt. Diese Alias helfen beispielsweise dabei, schneller im Dateisystem zu navigieren *(siehe Listing 2)*.

oh-my-zsh lässt sich mithilfe von Plug-ins erweitern. Es bringt eine Vielzahl an Plug-ins mit, die eine Entwicklerin in der Konfigurations-

datei aktivieren kann. Sie kann externe Plug-ins nachinstallieren *(siehe Listing 3)*.

Plug-ins sind wiederum eine Sammlung von *alias* und Shell-Funktionen zu einem bestimmten Werkzeug. Das Plug-in *git* bietet zum Beispiel fertige *alias* für Git an, um schneller Git-Befehle einzugeben und nützliche Shell-Funktionen rund um Git, um bestimmte Anwendungsfälle schneller auszuführen (Beispiel: *siehe Listing 4*).

Darüber hinaus bietet *oh-my-zsh* mithilfe von *Themes* an, das Aussehen der *Shell Prompt* anzupassen. Es bringt eine Vielzahl an Themes mit, die ein Entwickler in der Konfigurationsdatei aktivieren kann *(siehe Listing 5)*. Auch hier können externe Themes nachinstalliert werden.

> Mit Shell Prompt ist die Eingabeaufforderung in der Shell gemeint.

Die Theme-Konfiguration wirkt auf den ersten Blick wie eine Spielerei. Auf den zweiten Blick bewirkt die Anpassung der Prompt, dass man nützliche Informationen für die tägliche Arbeit direkt im Blick hat (zum Beispiel auf welchem Git Branch man sich aktuell befindet.) Themes wie *Starship [20]* (muss nachinstalliert werden) erweitern die Prompts um nützliche Informationen, so kann eine Entwicklerin beispielsweise sehen, welche Version von einer Runtime aktuell benutzt wird.

```
# Switching directory
<directory name>='cd <directory name>'
..='cd ..'
...='cd ../..'

# Create directory
md='mkdir -p'

# List directory content
l='ls -lah'
ll='ls -lh'
```

Listing 2: Beispiel für alias-Definitionen in oh-my-zsh

Shellwerkzeuge, die den Developer-Alltag vereinfachen können

Die Shell kann ihr volles Potenzial ausspielen, wenn auch die richtigen Shell-Werkzeuge für die bevorstehende Aufgabe installiert sind. Doch welche Werkzeuge können wann und wie im Alltag helfen? Es folgt eine kleine Auflistung von nützlichen Werkzeugen.

Werkzeugverwaltung vereinfachen

Je nach Projekt-Setup muss die Java-Entwicklerin mit unterschiedlichen Java- und Buildwerkzeugs-Versionen hantieren. Die einmalige

```
$ cat ~/.zshrc

# Which plugins would you like to load? (plugins can be found in ~/.oh-my-zsh/plugins/*)
# Custom plugins may be added to ~/.oh-my-zsh/custom/plugins/
# Example format: plugins=(rails git textmate ruby lighthouse)
# Add wisely, as too many plugins slow down shell startup.
plugins=(git mvn)
```

Listing 3: Plug-in-Konfiguration in oh-my-zsh

```
#alias
g='git'
gcl='git clone'
gl='git pull'
gsw='git switch'
gaa='git add -A'
gcmsg='git commit -m '
gp='git push'

#Functions
grename old new # reanmes branch old to new, including on the origin remote
gbda # deletes all merged branches
```

Listing 4: Beispiele für git alias und Shell-Funktionen

```
$ cat ~/.zshrc

# Set name of the theme to load.
# Look in ~/.oh-my-zsh/themes/
# Optionally, if you set this to "random", it'll load a random theme each
# time that oh-my-zsh is loaded.
#ZSH_THEME="agnoster"
ZSH_THEME="simple"
```

Listing 5: Theme-Konfiguration in oh-my-zsh

```
→ sdk list java # listet verfügbare Java Versionen auf (Ausschbnitt)
================================================================================
Available Java Versions for Linux 64bit
================================================================================
 Vendor        | Use | Version      | Dist    | Status     | Identifier
--------------------------------------------------------------------------------
 Temurin       | >>> | 21.0.1       | tem     | installed  | 21.0.1-tem
               |     | 17.0.9       | tem     | installed  | 17.0.9-tem
               |     | 11.0.21      | tem     |            | 11.0.21-tem
               |     | 8.0.392      | tem     |            | 8.0.392-tem


→ sdk install java 21.0.1-tem # installiert JDK Eclipse Temurin in Version 21.0.1
→ sdk default java 21.0.1-tem # setzt Eclipse Temurin in Version 21.0.1 als Default-JDK
→ sdk use java 17.0.9-tem # setzt Eclipse Temurin in Version 17.0.9 als JDK für die aktuelle Session
```

Listing 6: Java-Versionen verwalten mit SDKMAN!

```
→ cat .sdkmanrc
# Enable auto-env through the sdkman_auto_env config
# Add key=value pairs of SDKs to use below
java=17.0.9-tem
maven=3.9.6
```

Listing 7: Beispiel für .sdkmanrc

```
→ cd myproject
Using java version 17.0.9-tem in this shell.

Using maven version 3.9.6 in this shell.
```

Listing 8: Beispiel für Autodetection

Installation der Versionen ist in der Regel schnell erledigt, doch das Wechseln zwischen diesen ist oft recht mühselig. Dieses Problem möchte *SDKMAN!* *[21]* lösen.

Es bietet eine Schnittstelle an, um Werkzeuge aus dem JVM-Ökosystem (Java, Scala, Kotlin und Groovy, Ant, Gradle, Grails, Maven, SBT, Spark, Spring Boot, Vert.x und viele weitere) zu installieren und zu verwalten. Den Workflow für die Verwaltung von Java-Versionen zeigt *Listing 6*.

Nutzen alle im Team SDKMAN! Als Verwaltungswerkzeug, kann das Team die zu benutzende JDK-Version auch im Projekt definieren, indem sie eine `.sdkmanrc`-Datei im Root-Verzeichnis ablegen, die die JDK-Version definiert *(siehe Listing 7)*. SDKMAN! kann diese Datei auch mit aktuell genutzten Versionen mit dem Befehl `sdk env init` anlegen.

Wenn die Autodetection von SDKMAN! Eingeschaltet ist, dann wechselt SDKMAN! Automatisch auf die richtige Version, beziehungsweise schlägt vor, sie zu installieren *(siehe Listing 8)*. Die Entwickler können die Installation aber auch manuell mit `sdk env install` anstoßen.

Ähnliche Werkzeuge gibt es auch für andere Ökosysteme. *nvm* *[22]* verwaltet beispielsweise Node-Versionen, *asdf* *[23]* verwaltet Werkzeuge aus verschiedenen Ökosystemen.

Arbeiten mit Dateien

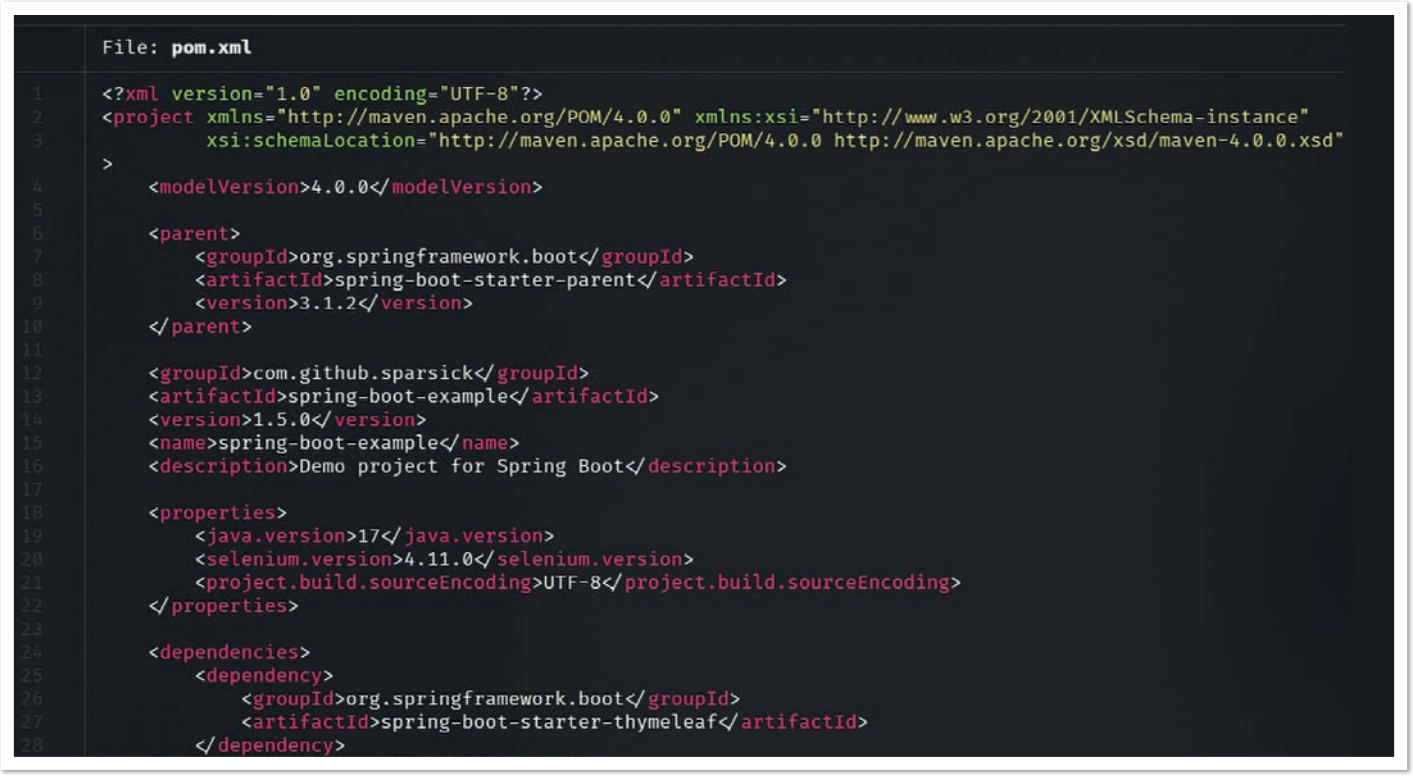
Wenn eine Entwicklerin schnell auf den Inhalt einer Datei zugreifen möchte, dann wird gerne auf cat (ist oft in der Standardinstallation eines Systems dabei) *[24]* verwiesen *(siehe Listing 9)*.

`cat` ist super geeignet, wenn der Inhalt einer Datei mithilfe von Pipes *[25]* mit anderen Werkzeugen weiterverarbeitet werden soll oder mehrere Dateien zusammengeführt werden sollen.

`cat` ist nicht sehr hilfreich, wenn man den Inhalt nur anschauen möchte und dafür Syntaxhervorhebung und Zeilenangaben braucht. Hier hilft das Werkzeug bat *[26]* weiter. Es ist in der Benutzung

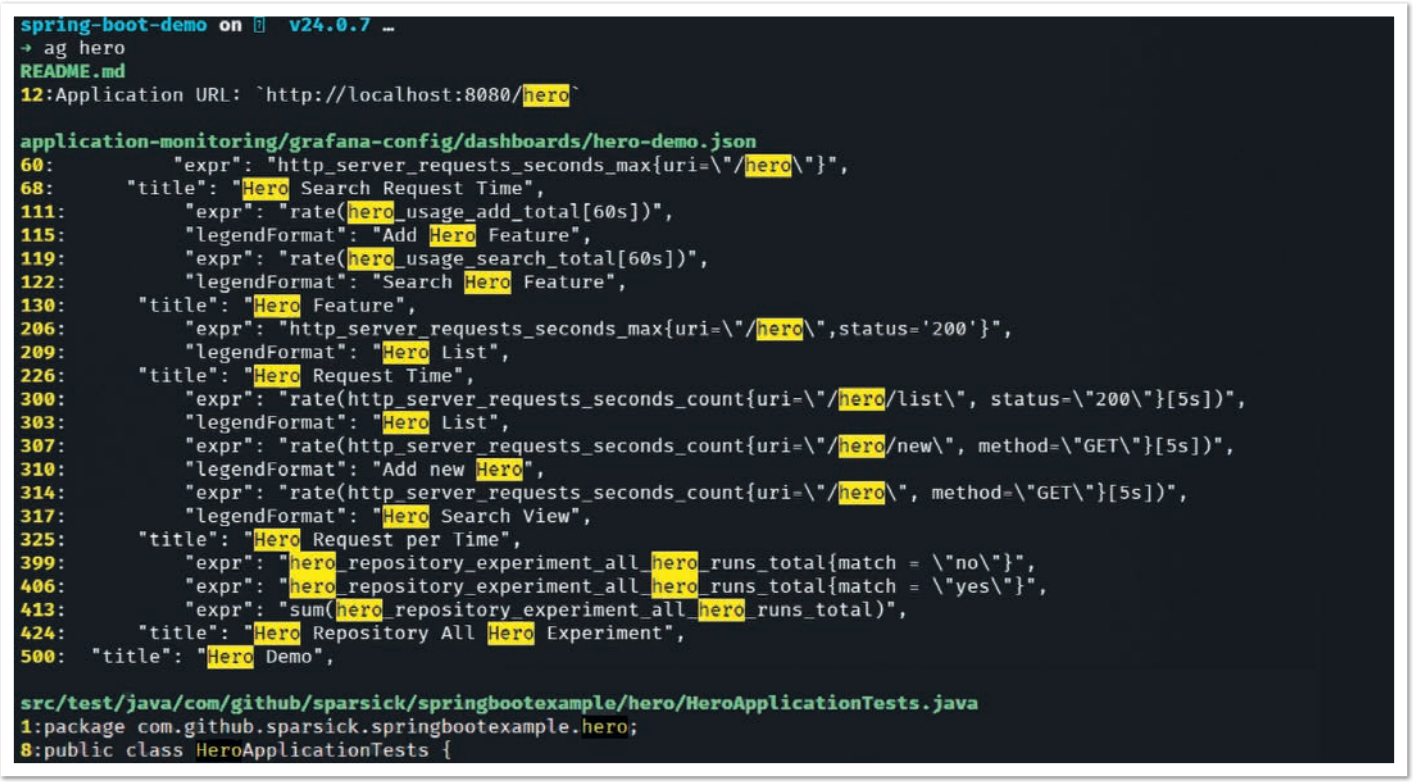
```
→ cat pom.xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>

     <parent>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-parent</artifactId>
           <version>3.1.2</version>
     </parent>

     <groupId>com.github.sparsick</groupId>
     <artifactId>spring-boot-example</artifactId>
     <version>1.5.0</version>
     <name>spring-boot-example</name>
     <description>Demo project for Spring Boot</description>
```

Listing 9: Beispiel mit cat



```
File: pom.xml

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
>
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.2</version>
    </parent>

    <groupId>com.github.sparsick</groupId>
    <artifactId>spring-boot-example</artifactId>
    <version>1.5.0</version>
    <name>spring-boot-example</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>17</java.version>
        <selenium.version>4.11.0</selenium.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
```

Abbildung 1: Dateien-Anzeige mit bat (© bat, Screenshot von Sandra Parsick)



```
spring-boot-demo on  v24.0.7 …
➜ ag hero
README.md
12:Application URL: `http://localhost:8080/hero`

application-monitoring/grafana-config/dashboards/hero-demo.json
60:          "expr": "http_server_requests_seconds_max{uri=\"/hero\"}",
68:      "title": "Hero Search Request Time",
111:          "expr": "rate(hero_usage_add_total[60s])",
115:          "legendFormat": "Add Hero Feature",
119:          "expr": "rate(hero_usage_search_total[60s])",
122:          "legendFormat": "Search Hero Feature",
130:      "title": "Hero Feature",
206:          "expr": "http_server_requests_seconds_max{uri=\"/hero\",status='200'}",
209:          "legendFormat": "Hero List",
226:      "title": "Hero Request Time",
300:          "expr": "rate(http_server_requests_seconds_count{uri=\"/hero/list\", status=\"200\"}[5s])",
303:          "legendFormat": "Hero List",
307:          "expr": "rate(http_server_requests_seconds_count{uri=\"/hero/new\", method=\"GET\"}[5s])",
310:          "legendFormat": "Add new Hero",
314:          "expr": "rate(http_server_requests_seconds_count{uri=\"/hero\", method=\"GET\"}[5s])",
317:          "legendFormat": "Hero Search View",
325:      "title": "Hero Request per Time",
399:          "expr": "hero_repository_experiment_all_hero_runs_total{match = \"no\"}",
406:          "expr": "hero_repository_experiment_all_hero_runs_total{match = \"yes\"}",
413:          "expr": "sum(hero_repository_experiment_all_hero_runs_total)",
424:      "title": "Hero Repository All Hero Experiment",
500:  "title": "Hero Demo",

src/test/java/com/github/sparsick/springbootexample/hero/HeroApplicationTests.java
1:package com.github.sparsick.springbootexample.hero;
8:public class HeroApplicationTests {
```

Abbildung 2: Suchergebnisse mit ag unter der Verwendung von Silversearcher (© ag, Screenshot von Sandra Parsick)

leichtgewichtig wie `cat`, bietet aber Syntaxhervorhebung und Zeilenangaben an, bei Wunsch zeigt es auch Git-Änderung pro Zeile an *(siehe Abbildung 1)*.

Wenn der Entwickler eine Menge an Dateien durchsuchen und dafür gerne die Shell verwenden möchte, wird er oftmals auf `find` *[27]* und `grep` *[28]* verwiesen. Das sind mächtige Werkzeuge, aber nicht intuitiv zu bedienen und es fehlen bei den Ergebnissen Kontextinformationen, die für einen Entwickler interessant sind. Das Werkzeug *Silversearcher [29]* liefert genau diese Funktionalität.

In der Standardbenutzung ag `suchbegriff` *(siehe Abbildung 2)* listet *Silversearcher* alle Stellen inklusive Dateipfad und Zeilenangaben auf, in denen der gesuchte Begriff vorkommt.

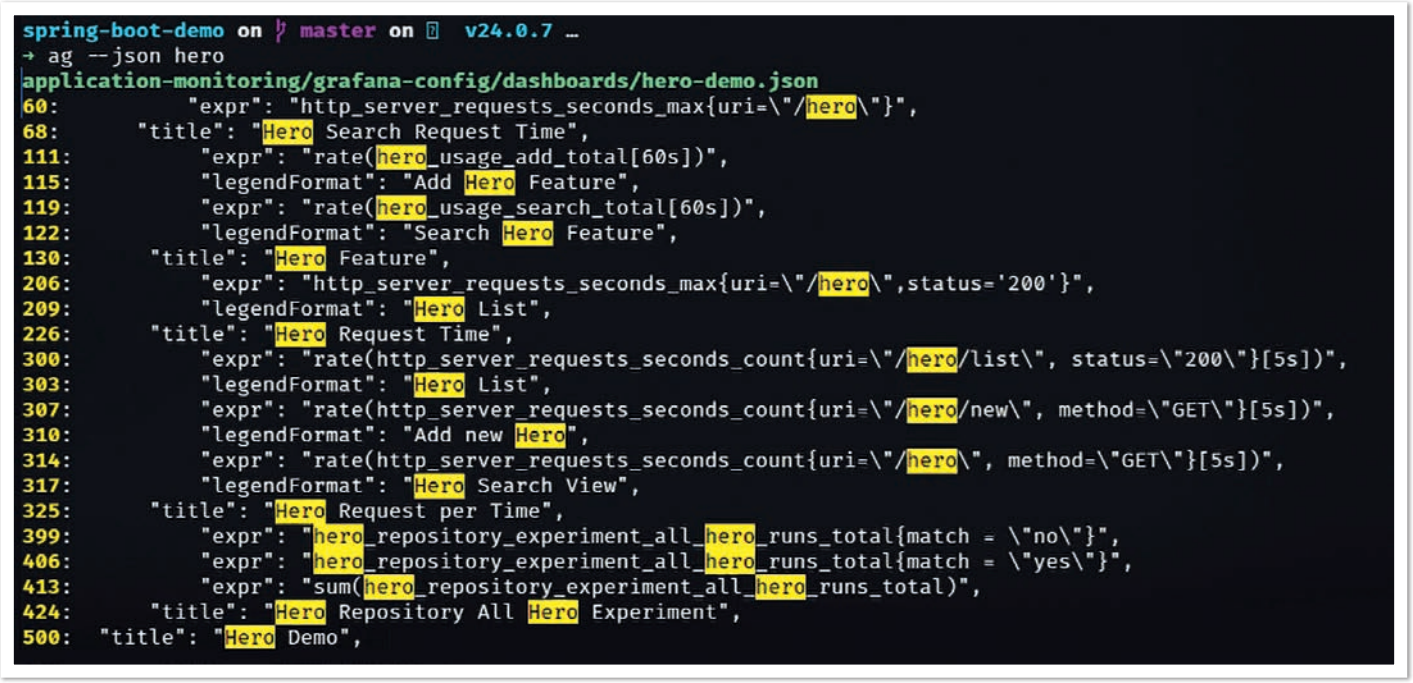


Abbildung 3: Gefilterte Suchergebnisse mit ag (© ag, Screenshot von Sandra Parsick)

Möchte eine Entwicklerin die Suche auf bestimmte Dateitypen einschränken, kann sie es über die Option `--datentyp` (zum Beispiel `--json` für eine Einschränkung auf JSON-Dateien, *siehe Abbildung 3)*. Die Option `--list-file-types` listet alle unterstützen Datentypen auf.

Braucht ein Entwickler doch nur eine Auflistung aller Dateien, in der ein Suchbegriff auftaucht, kann er die Ausgabe auf diese Information mit der Option `--files-with-matches` einschränken *(siehe Abbildung 4)*.

Silversearcher bietet noch weitere Optionen an, um die Suche und Ausgabe auf die eigenen Bedürfnisse anzupassen. Besonders wenn Entwickler JSON oder YAML genauer durchsuchen wollen oder einfach nur Kontext-basiert parsen wollen, stößt auch *Silversearcher* an seine Grenzen. Hier bietet es sich an, darauf spezialisierte Werkzeuge zu benutzen, wie zum Beispiel *jq* (für JSON) *[30]* oder *yq* (für YAML) *[31]*. Es sind zwei verschiedene Werkzeuge, deren Benutzung jedoch ähnlich gehalten ist.

In dem folgenden Beispiel nehmen wir an, dass Entwickler eine JSON-Datei *(siehe Listing 10)* durchsuchen möchten.

Das erste Problem ist, dass die Datei nicht formatiert ist und somit für eine Entwicklerin schwer zu lesen ist. Mit `` `cat starships.json | jq.` `` lässt sich die Datei formatieren *(siehe Listing 11)*.

Möchte der Entwickler aus dem JSON nur die Werte, die unter dem Schlüssel `results` liegen, gibt er `cat starships.json| jq .results` ein. Möchte er es weiter einschränken, zum Beispiel nur die Namen der Sternenschiffe innerhalb des Arrays, gibt er `cat starships.json| jq '.results.[].name'` ein.

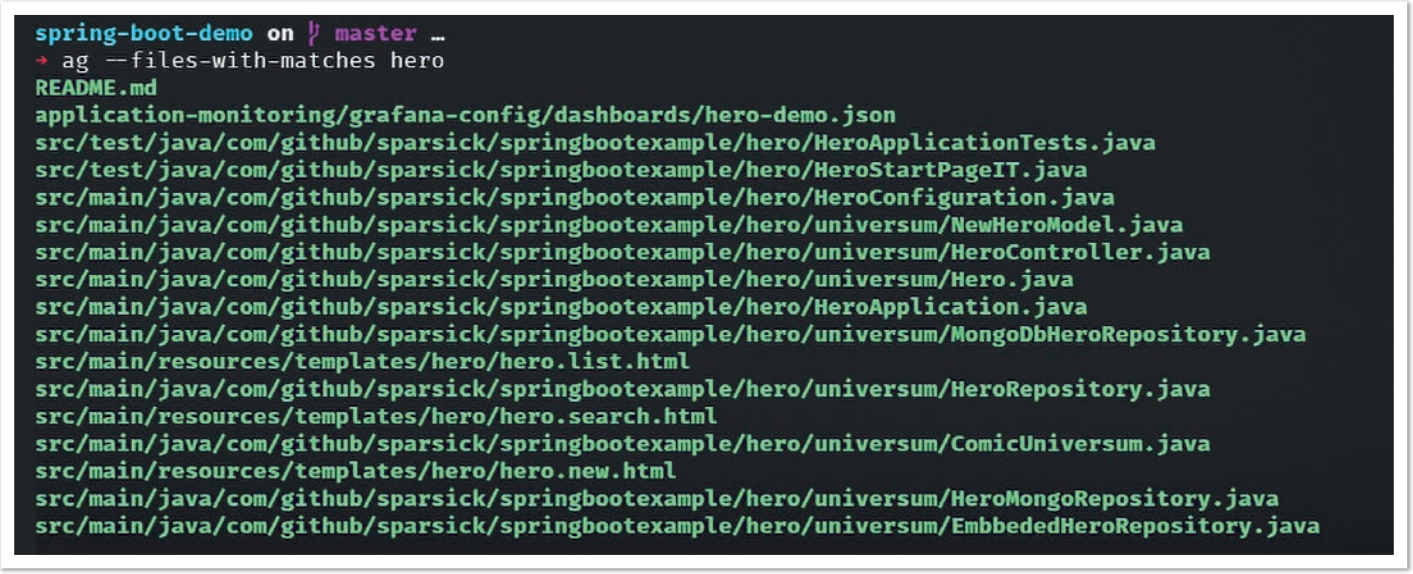


Abbildung 4: Auflistung der Dateinamen (© ag, Screenshot von Sandra Parsick)

```
{"count":36,"next":"https://swapi.dev/api/st
arships/?page=2","previous":null,"results":
[{"name":"CR90 corvette","model":"CR90 corv
ette","manufacturer":"Corellian Engineering
Corporation","cost_in_credits":"3500000","length"
:"150","max_atmosphering_speed":"950","crew":"30-
165","passengers":"600","cargo_capacity":"3000000",
"consumables":"1 year","hyperdrive_rating":"2.0","M
GLT":"60","starship_class":"corvette","pilots":[],"
films":["https://swapi.dev/api/films/1/","https://
swapi.dev/api/films/3/","https://swapi.dev/api/
films/6/"],"created":"2014-12-10T14:20:33.369000Z","e
dited":"2014-12-20T21:23:49.867000Z","url":"https://
swapi.dev/api/starships/2/"}]}%
```

Listing 10: Unformatierte, schlecht lesbare JSON-Datei (Ausschnitt)

Das Werkzeug yq ist in der Benutzung ähnlich, nur dass es sich auf YAML-Dateien spezialisiert hat.

HTTP-Schnittstellen aufrufen

Wenn es darum geht HTTP-Schnittstellen aufzurufen, dann wird gerne auf `curl` oder `wget` *[32]* verwiesen. Diese Werkzeuge sind sehr mächtig, jedoch nicht sehr intuitiv zu bedienen. Oft braucht es etwas leichtgewichtigeres, um zum Beispiel ein REST-API zu testen. Für diesen Anwendungsfall gibt es *httpie* *[33]*. Es besitzt eine intuitive Schnittstelle und liefert alle wichtigen Informationen auf einen Blick *(siehe Listing 12)*.

Der Aufruf erfolgt nach dem Muster `http HTTP-METHOD url`. Möchte eine Entwicklerin nicht alle Informationen, kann sie diese über Optionen (`--headers`, `--meta`, `--body`, *siehe Listing 13)* einschränken.

Über weitere Optionen kann man auch die SSL- und Authentisierungseinstellungen steuern.

Tipps und Tricks

Die letzten Abschnitte haben einen kleinen Einblick gegeben, wie die Shell bei alltäglichen Entwickleraufgaben helfen kann. Doch gerade Anfänger sind mit vielen Sachen, die in der Shell passieren können, etwas überfordert. Daher nachfolgend ein paar Tipps, die das Arbeiten auf der Shell vereinfachen.

1. **Frag deine Kollegen:** Schaue beim Pair-Programming darauf, wie deine Kollegen bestimmte Aufgaben auf der Shell (vielleicht nicht nur dort) lösen und frage nach, was sie gerade gemacht haben.
2. **Benutze Cheat Sheets:** Gerade bei mächtigen Werkzeugen verliert man schnell den Überblick, was alles möglich ist. Cheat Sheets helfen dabei, den Überblick zu behalten.
3. **Benutze Man Pages oder die –help-Option:** Zu jedem Werkzeug gibt es die Hilfsoption oder eine Man Page (`man werkzeug`), die die Benutzung des Werkzeuges und ihre Optionen erklärt.

Es gibt noch weitere Quellen, die gut erklären, wie Befehle auf der Shell funktionieren: Die Webseite *Explain Shell* *[34]* erzeugt zum Beispiel eine genaue Erklärung für jeden Befehl, den man dort eingibt *(siehe Abbildung 5)*.

Die Webseite *tldr pages* *[35]* erklärt Befehle über die Man Page hinaus anhand von Beispielen.

Die Techniker

Sorry, kennen wir uns?

Nein? Dann haben wir uns wohl auf der JavaLand verpasst. Aber keine Sorge – das Kennenlernen holen wir einfach nach! Und vielleicht finden wir ja dann ganz zueinander.

Denn die Techniker Krankenkasse hat als Arbeitgeber einiges zu bieten:

- familienfreundliche, flexible Arbeitszeiten
- 35,5 Stunden/Woche
- flexible Home-Office Regelung
- und vieles mehr!

Jetzt weitere Benefits entdecken und bewerben: tk.de/IT

Beste Arbeitgeber Deutschland Great Place To Work. 2023

```
{
  "count": 36,
  "next": "https://swapi.dev/api/starships/?page=2",
  "previous": null,
  "results": [
     {
        "name": "CR90 corvette",
        "model": "CR90 corvette",
        "manufacturer": "Corellian Engineering Corporation",
        "cost_in_credits": "3500000",
        "length": "150",
        "max_atmosphering_speed": "950",
        "crew": "30-165",
        "passengers": "600",
        "cargo_capacity": "3000000",
        "consumables": "1 year",
        "hyperdrive_rating": "2.0",
        "MGLT": "60",
        "starship_class": "corvette",
        "pilots": [],
        "films": [
           "https://swapi.dev/api/films/1/",
           "https://swapi.dev/api/films/3/",
           "https://swapi.dev/api/films/6/"
        ],
        "created": "2014-12-10T14:20:33.369000Z",
        "edited": "2014-12-20T21:23:49.867000Z",
        "url": "https://swapi.dev/api/starships/2/"
     }
  ]
}
```


Listing 11: Beispiel für eine formatierte JSON-Datei (Ausschnitt)


```
→ http GET https://swapi.dev/api/starships/9/
HTTP/1.1 200 OK
Allow: GET, HEAD, OPTIONS
Connection: keep-alive
Content-Type: application/json
Date: Fri, 12 Jan 2024 10:08:02 GMT
ETag: "058c95fce38484128f1c3f2e5dd04d50"
Server: nginx/1.16.1
Strict-Transport-Security: max-age=15768000
Transfer-Encoding: chunked
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN

{
     "MGLT": "10",
     "cargo_capacity": "1000000000000",
     "consumables": "3 years",
     "cost_in_credits": "1000000000000",
     "created": "2014-12-10T16:36:50.509000Z",
     "crew": "342,953",
     "edited": "2014-12-20T21:26:24.783000Z",
     "films": [
           "https://swapi.dev/api/films/1/"
     ],
     "hyperdrive_rating": "4.0",
     "length": "120000",
     "manufacturer": "Imperial Department of Military Research, Sienar Fleet Systems",
     "max_atmosphering_speed": "n/a",
     "model": "DS-1 Orbital Battle Station",
     "name": "Death Star",
     "passengers": "843,342",
     "pilots": [],
     "starship_class": "Deep Space Mobile Battlestation",
     "url": "https://swapi.dev/api/starships/9/"
}
```


Listing 12: Beispiel httpie

Quellen

[1] Wikipedia zu Terminal (Computer) Webseite: *https://de.wikipedia.org/wiki/Terminal_(Computer)*

[2] Terminal Konsole Webseite: *https://konsole.kde.org/*

[3] Terminal GNOME Terminal Webseite: *https://wiki.gnome.org/Apps/Terminal*

[4] Terminal Tilix Webseite: *https://gnunn1.github.io/tilix-web/*

[5] Terminal Terminal für den Mac Webseite: *https://support.apple.com/de-de/guide/terminal/welcome/mac*
[6] Terminal iterm2 Webseite: *https://iterm2.com/index.html*
[7] Terminal Windows Terminal Webseite: *https://learn.microsoft.com/de-de/windows/terminal/*
[8] Terminal fig Webseite: *https://fig.io/*
[9] Terminal warp Webseite: *https://www.warp.dev/*
[10] Terminal hyper Webseite: *https://hyper.is/*
[11] Wikipedia zu Shell (Betriebssystem) Webseite: *https://de.wikipedia.org/wiki/Shell_(Betriebssystem)*
[12] Bash Webseite: *https://www.gnu.org/software/bash/*
[13] Zsh Webseite: *https://www.zsh.org/*
[14] fish Shell Webseite: *https://fishshell.com/*
[15] Oh My Zsh Webseite: *https://ohmyz.sh/*
[16] Oh My Bash Webseite: *https://ohmybash.nntoan.com/*
[17] Oh My Posh Webseite: *https://ohmyposh.dev*
[18] cUrl Webseite: *https://curl.se/*
[19] Git Webseite: *https://git-scm.com/*
[20] Theme Starship Webseite: *https://starship.rs/*
[21] SDKMAN! Webseite: *https://sdkman.io/*
[22] nvm Webseite: *https://github.com/nvm-sh/nvm*
[23] asdf Webseite: *https://asdf-vm.com/*
[24] cat Webseite: *https://wiki.ubuntuusers.de/cat/*
[25] Pipes / Umleitungen Webseite: *https://wiki.ubuntuusers.de/Shell/Umleitungen/*
[26] bat Webseite: *https://github.com/sharkdp/bat*
[27] find Webseite: *https://wiki.ubuntuusers.de/find/*
[28] grep Webseite: *https://wiki.ubuntuusers.de/grep/*
[29] The Silver Searcher Webseite: *https://geoff.greer.fm/ag/*
[30] jq Webseite: *https://jqlang.github.io/jq/*
[31] yq Webseite: *https://mikefarah.gitbook.io/yq/*
[32] wget Webseite: *https://www.gnu.org/software/wget/*
[33] httpie Webseite: *https://httpie.io/*
[34] Webseite: *https://explainshell.com/*
[35] Webseite: *https://tldr.sh/*


```
→ http GET https://swapi.dev/api/starships/9/ --headers
HTTP/1.1 200 OK
Allow: GET, HEAD, OPTIONS
Connection: keep-alive
Content-Type: application/json
Date: Fri, 12 Jan 2024 10:18:34 GMT
ETag: "058c95fce38484128f1c3f2e5dd04d50"
Server: nginx/1.16.1
Strict-Transport-Security: max-age=15768000
Transfer-Encoding: chunked
Vary: Accept, Cookie
X-Frame-Options: SAMEORIGIN
```

Listing 13: Beispiel httpie mit Optionen

Sandra Parsick

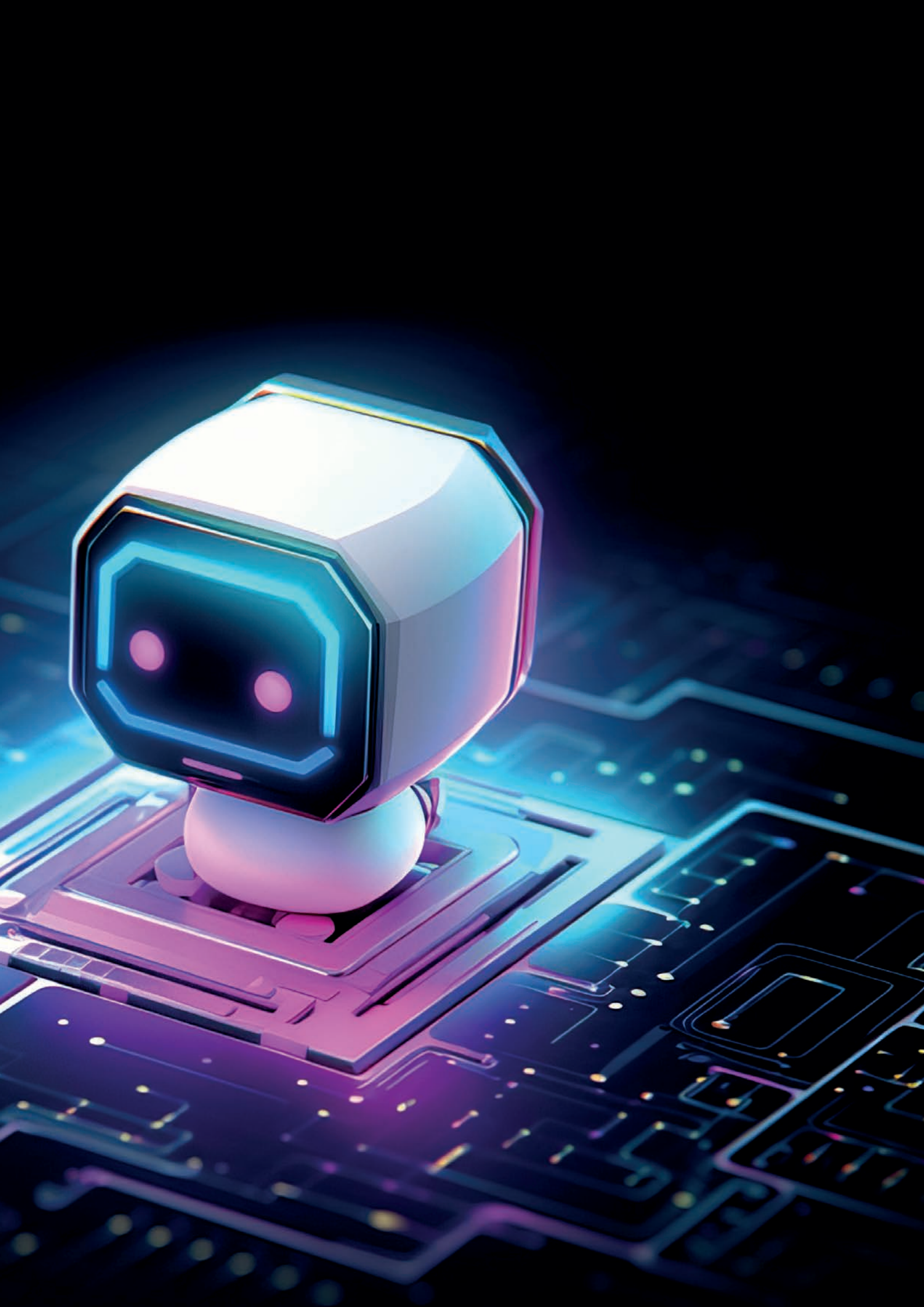
Sandra Parsick ist Java Champion und arbeitet als freiberufliche Softwareentwicklerin und Consultant im Java-Umfeld. Seit 2008 beschäftigt sie sich mit agiler Softwareentwicklung in verschiedenen Rollen. Ihre Schwerpunkte liegen im Bereich Java Enterprise, Cloud, Software Craftsmanship und in der Automatisierung von Entwicklungsprozessen. Darüber schreibt sie gerne Artikel und spricht auf Konferenzen. In ihrer Freizeit engagiert sie sich in verschiedenen Programmkomitees und Community-Gruppen.

Abbildung 5: Beispielausgabe in explainshell.com (© explainshell.com, Screenshot von Sandra Parsick)

Automatische Dependency-Updates mit GitHub Actions und Dependabot

Jörg Liedl, SpringerNature AG

In den meisten Firmen ist es immer das Gleiche: Ein bis zwei Personen kümmern sich um die Dependencies, während sich viele Kolleg:innen davor drücken und hoffen, dass irgendwer das aktuell hält. Daher: Lieber gleich den ganzen Prozess mit GitHub Actions und Dependabot automatisieren.



Das Leid mit veralteten Dependencies

Wer nachhaltige Software schreiben will, sollte diese immer aktuell halten. Wir erinnern uns noch alle an die unzähligen Projekte, die wir im Dezember 2021/Januar 2022 aktualisieren mussten, als die Log4j-Sicherheitslücke (CVE-2021-44228) bekannt wurde. In vielen Projekten wurde Log4j in einer verwundbaren Version direkt oder über Dependencies eingesetzt, diese galt es, damals zu aktualisieren. Und plötzlich stellte man fest: Mist, die Dependency, die man nutzt, hat eine Log4j-Abhängigkeit, lässt sich aber aufgrund von *Breaking Changes* nicht upgraden. Deswegen mussten Workarounds eingesetzt werden, um sicherzugehen, dass nur die gepatchte Version von Log4j verwendet wurde. Deshalb versuchen gute Softwareentwickler:innen die Abhängigkeiten immer auf dem neuesten Stand zu halten. Mein Kollege Dennis Hörsch und ich haben uns daher eingehend mit Dependency-Management und entsprechenden Prozessen beschäftigt.

In dieser Schritt-für-Schritt-Anleitung werde ich euch einen Einblick geben, was die Möglichkeiten sind, was es für Probleme gab, und was es noch immer für Probleme gibt.

Für die einzelnen Schritte habe ich im Beispielprojekt *[1]* jeweils einen Branch angelegt, um alle einzelnen Schritte nachvollziehbar zu machen. In der Anleitung schreibe ich immer dazu, in welchem Branch der beschriebene Stand ist. Der main-Branch ist das fertige Endprodukt, dieser ist auch für Dependabot aktiviert.

Wir starten mit dem Spring Initializer *[2]*, hier wähle ich ein Gradle-Kotlin-Projekt mit JDK 21, das Stable-Release von Spring Boot und als Sprache Kotlin. In meinem Beispiel-Repository ist dies der Branch ***step 0***. Wenn das Projekt aus dem Initializer kommt, sind die Dependencies in der build.gradle.kts definiert. Gerade in größeren Projekten sieht man immer wieder, dass die Dependencies in der `settings.gradle.kts` mittels *Versions Catalogs* definiert werden. Dies hat den Vorteil, dass es übersichtlich für alle Module definiert werden kann. Um möglichst nahe an der Realität zu bleiben, habe ich dies auch im Beispielprojekt so abgeändert.

```yaml
name: Test Demo 1

on:
  pull_request:
     paths:
        - 'demo1/**'
     branches: [ main ]

  workflow_dispatch: {}

jobs:
  test-demo-1:
     runs-on: ubuntu-latest
     defaults:
        run:
           working-directory: ./demo1
     steps:
        - uses: actions/checkout@v4
        - name: Set up JDK 21
           uses: actions/setup-java@v4
           with:
                 distribution: 'liberica'
                 java-version: '21'
        - run: ./gradlew clean test
```

Listing 1: testing-demo1.yml, um den Code zu testen

Pull Requests mit GitHub Actions automatisch testen

Mit *GitHub Actions* gibt es jetzt, ähnlich wie bei GitLab Pipelines, die Möglichkeit, seine eigenen automatischen Prozesse abzubilden. So kann man komplette CI/CD-Pipelines oder automatische Tests bei einem Pull-Request (PR) darüber abbilden. Deswegen legen wir jetzt im Basisverzeichnis des Repository einen Ordner namens `.github` an, in dem sich ein weiterer Ordner namens `workflows` befindet.
In den `workflows`-Ordner kommt jetzt eine Workflow-Datei mit dem Inhalt aus *Listing 1*. Bei jedem neuen Pull-Request auf den main-Branch, in dem Änderungen im Pfad `demo1` stattfinden, wird der Test ausgeführt. Hier wird ein Ubuntu-Container (`runs-on: ubuntu-latest`) gestartet, in dem erst der Code ausgecheckt wird (`uses: actions/checkout`) und dann Java (`uses: actions/setup-java`) mit der JDK 21 (`java-version: '21'`) verwendet wird. Bei der Java-Distribution habt ihr die Wahl zwischen den gängigen Distributionen.

Dies alles findet ihr im Branch ***step_1***.

Zum Testen fügen wir dem Quellcode einen Kommentar hinzu, pushen dies auf einen Extra-Branch und erstellen einen PR. Daraufhin kann man sehen, dass der Test läuft. Allerdings zeigt euch GitHub schon an, dass ihr den PR mergen könnt, ohne dass der Test durchgelaufen ist *(siehe Abbildung 1)*. Dies wollen wir natürlich nicht, weswegen wir uns jetzt um die Branch-Protection kümmern müssen.

Klickt in dem Projekt auf *Settings → Branches → Add branch protection rule*. Wir wollen den `main`-Branch schützen, also geben wir bei *Branch name pattern* „`main`“ an. Dann wählen wir *Require status checks to pass before merging* aus. Das blockiert das Mergen, falls die Tests noch nicht gelaufen oder fehlgeschlagen sind. Ein User mit Adminrechten kann dies dennoch weiterhin, es gibt aber auch die Möglichkeit, den Admin dazu zu zwingen, die Regel einzuhalten – mittels *Do not allow bypassing the above settings* in den Branch Protection Rules. *Require branches to be up to date before merging* wählen wir

Abbildung 1: Obwohl die Tests laufen, könntet ihr mergen – das ist schlecht. ([Screenshot GitHub])

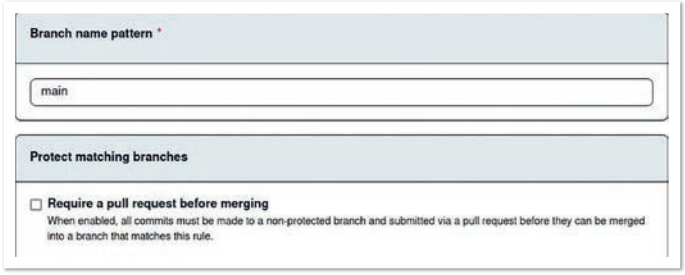


Abbildung 2: So stellt ihr die Branch Protections ein. ([Screenshot GitHub])

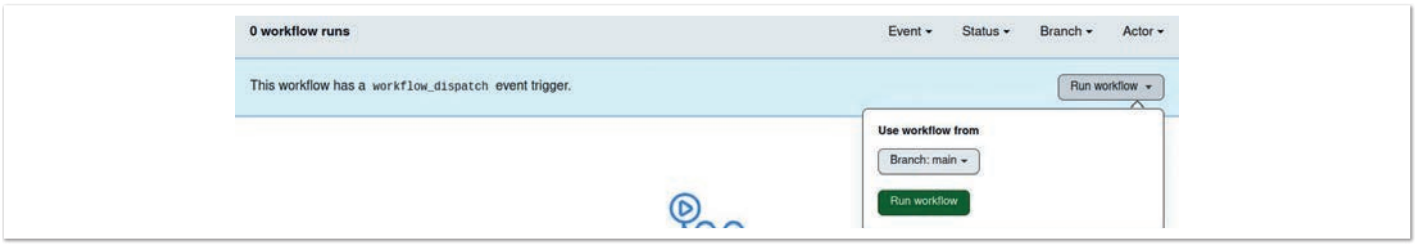


Abbildung 3: So triggert ihr einen Workflow. ([Screenshot GitHub])

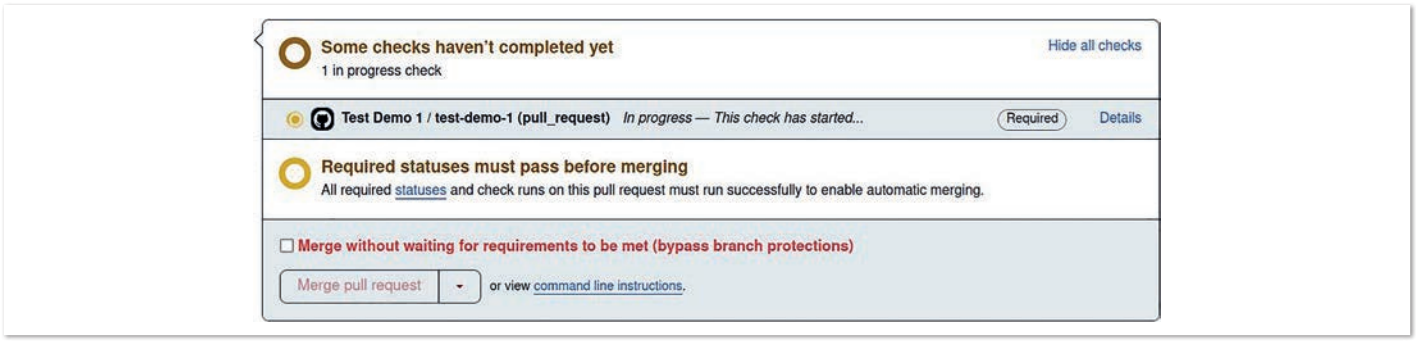


Abbildung 4: Nur Admins können jetzt noch mergen, solange der Test läuft. ([Screenshot GitHub])

ebenfalls aus, um zu garantieren, dass immer der neueste Stand von `main` in einem PR verwendet wird. Dies garantiert uns, dass die Tests auch mit dem aktuellen Stand von `main` funktionieren. Normalerweise sollten Entwickler:innen dies immer selbst machen, aber so ist man auf der sicheren Seite *(siehe Abbildung 2)*.

Solltet ihr in euren Branch Protection Rules eure Tests nicht finden, liegt das daran, dass diese zuvor mindestens einmal laufen müssen. Hierfür geht ihr auf GitHub auf den Actions Buttons, wählt euren Workflow aus und klickt auf *Run workflow*, wie ihr es in *Abbildung 3* seht.

Jetzt könnt ihr als Nicht-Admin nicht mehr einen Branch mergen, bevor der Test grün ist *(siehe Abbildung 4)*. Damit sind wir jetzt bestens vorbereitet, um Dependabot einzubauen.

Dependabot kümmert sich um eure Dependencies

Alle Änderungen hierzu befinden sich im Branch ***step_2***.

Dependabot war früher ein eigenes Unternehmen, bevor es im Mai 2019 von GitHub gekauft wurde. Seitdem wird es immer mehr in GitHub integriert, was wir uns hier zu Nutze machen. Dependabot analysiert Dependencies, prüft auf CVEs und veraltete Versionen unabhängig von der Sprache. Ihr könnt damit sowohl eure JVM-Dependencies, aber auch Abhängigkeiten in npm, GitHub Actions, Docker, PIP, Go, Terraform und noch einige weitere tracken *[3]*. Hierzu fügt ihr im Ordner `.github` eine Datei namens `dependabot.yml` hinzu. Diese sieht aus wie in *Listing 2* gezeigt. Wir nutzen in diesem Beispiel nur das gradle-Ecosystem. Wir wollen das Ganze täglich um 9:00 Uhr laufen lassen. Derzeit ist es nicht möglich, das Scheduling auf Montag bis Freitag zu beschränken. Ihr könnt es aber alternativ auch nur einmal wöchentlich, während eines Arbeitstages, laufen lassen.

Die Labels *dependencies*, *gradle*, *kotlin* und *automatic-merge* gibt es noch nicht und müssen erst erstellt werden. Diese werden zu einem

```yaml
version: 2
updates:
  - package-ecosystem: "gradle"
    directory: "/demo1/"
    schedule:
       interval: "daily"
       time: "09:00"
    open-pull-requests-limit: 100
    labels:
       - "dependencies"
       - "gradle"
       - "kotlin"
       - "automatic-merge"
```

Listing 2: dependabot.yml

```
$ gh label create LABELNAME
```

Listing 3: Erstellen eines Labels mit der GitHub CLI

von Dependabot erstellten PR hinzugefügt. Ebenso benötigen wir die Labels *patch*, *minor* und *major*, die die verschiedenen Versionsänderungen darstellen sollen. Labels erstellt ihr am einfachsten mit der GitHub-CLI *[4] (siehe Listing 3)*. Die Labels *dependencies*, *gradle*, *kotlin* sind nur für euch in einem Projekt, um dieses bei vielen PRs übersichtlich zu halten, benötigt werden sie für das Beispiel nicht.

Um Dependabot zu testen, downgraded ihr in der `build.gradle.kts` eine Version und pusht das Ganze in den main-Branch. Auf eurer GitHub-Projektseite auf *Insights → Dependency Graph → Dependabot* wählt ihr dann eure `build.gradle.kts` aus und startet Dependabot. Nach einiger Zeit werdet ihr dann einen PR von Dependabot sehen. In meinem Beispiel wurden gleich mehrere Sachen zum Updaten gefunden *(siehe Abbildung 5)*.

Nun ändern wir in der extra angelegten `settings.gradle.kts` eine Version, zum Beispiel die `version("lombok", "1.18.30")` zur

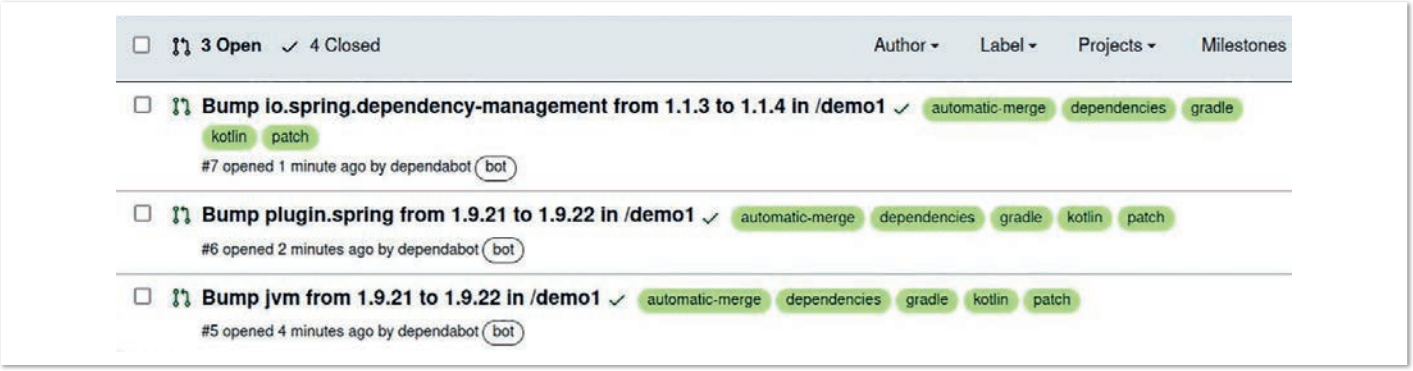


Abbildung 5: Von Dependabot erstellte PRs ([Screenshot GitHub])

```
[versions]
jacksonKotlin = "2.16.1"
junit-jupiter = "5.10.1"
hamkrest = "1.8.0.1"
kotlin = "1.9.22"
kotlinReflect = "1.9.22"
lombok = "1.18.28"
springboot = "3.2.1"
springDependencyManagementVersion = "1.1.1"

[libraries]
springBootStarterWeb = { module = "org.springframework.boot:spring-boot-starter-web", version.ref = "springboot" }
springBootStarterTest = { module = "org.springframework.boot:spring-boot-starter-test", version.ref = "springboot" }
jacksonKotlinModule = { module = "com.fasterxml.jackson.module:jackson-module-kotlin", version.ref = "jacksonKotlin"
}
kotlinReflect = { module = "org.jetbrains.kotlin:kotlin-reflect", version.ref = "kotlinReflect" }
lombok = { module = "org.projectlombok:lombok", version.ref = "lombok" }

[plugins]
kotlin-jvm = { id = "org.jetbrains.kotlin.jvm", version.ref = "kotlin" }
kotlin-spring = {id ="org.jetbrains.kotlin.plugin.spring", version.ref = "kotlin"}
springBoot = { id = "org.springframework.boot", version.ref = "springboot" }
springDependencyManagement = { id = "io.spring.dependency-management", version.ref = "springDependencyManagementVer-
sion" }
```

Listing 4: Das neu erzeugte libs.versions.toml File, inklusive der Plug-ins.

Vorgänger-Version 1.18.28. Dann pushen wir das zu `main` und starten wie oben beschrieben Dependabot. Wie ihr sowohl in den Logs (diese sind dort zu finden, wo ihr Dependabot startet) als auch auf GitHub unter Pull Requests sehen könnt, ist nichts passiert. Dependabot ignoriert leider die Änderungen in der `settings.gradle.kts`.

Version Catalogs mit Hilfe eines libs.versons.toml File

Alle Änderungen findet ihr hierzu im ***step_3*** Branch.

Da die `settings.gradle.kts` nicht von Dependabot ausgewertet wird, müssen wir anders vorgehen. Seit Neuestem unterstützt Dependabot auch Versionskataloge, allerdings muss es hierfür als `.toml` File abgelegt werden. Dies ist laut Gradle-Dokumentation entweder in der höchsten Ebene des Projekts möglich (in diesem Fall also in `demo1`) oder aber im gradle-Ordner, wie ich es im Beispiel zeige. Wir löschen jetzt also komplett den `dependencyResolutionManagement`-Teil in der `settings.gradle.kts` und erstellen die `libs.versions.toml`-Datei wie in *Listing 4* gezeigt.

Die bis dato noch vorhandenen Plug-ins schreiben wir auch in das `.toml` File, `build.gradle.kts` verweist dann, ähnlich wie die Dependencies, nur noch auf die libs.plugins.* *(siehe Listing 5)*.

Wenn wir jetzt, wie vorher beschrieben, den Dependabot-Prozess manuell anstoßen, wird ein PR für die veraltete `lombok-Dependency` erstellt.

Die PRs automatisch mergen

Alle Änderungen findet ihr hierzu im ***step_4*** Branch.

Die PRs werden jetzt alle automatisch erstellt, wie wir uns das vorgestellt haben. Jetzt fehlt nur noch der entscheidende Schritt: Das automatische Mergen der Dependency-Pull-Requests. Um dies vorzubereiten, müssen wir noch zwei Punkte in den General Settings ändern *(siehe Abbildung 6)*.

```
plugins {
      alias(libs.plugins.kotlin.jvm)
      alias(libs.plugins.kotlin.spring)
      alias(libs.plugins.springDependencyManagement)
      alias(libs.plugins.springBoot)
}
```

Listing 5: So sieht jetzt die Plug-in-Sektion in der build.gradle.kts aus.

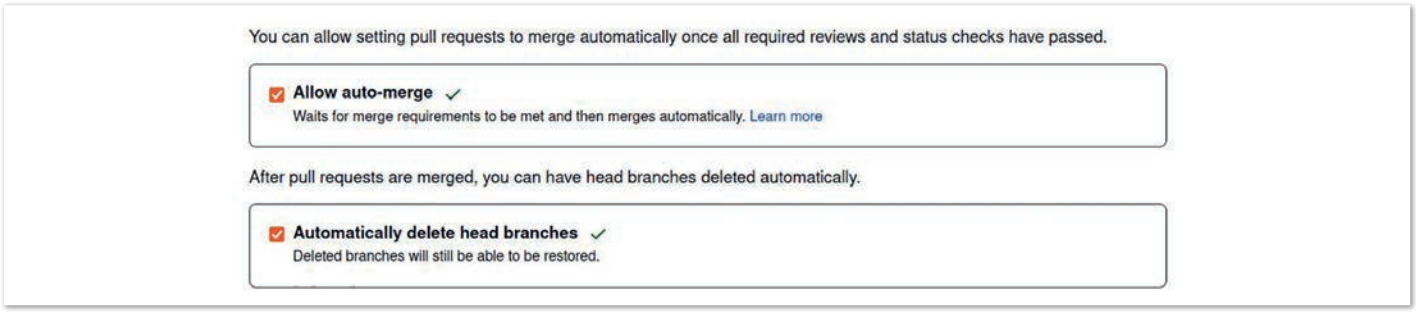


Abbildung 6: Allow auto-merge und Automatically delete head branches sollten aktiviert sein. ([Screenshot GitHub])

```yaml
name: Enables auto-merge on dependency pull requests
"on":
  pull_request:
     branches:
        - main
     paths:
        - 'demo1/**'
  workflow_dispatch: {}

jobs:
  enable-automerge:
     if: |
        contains(github.event.pull_request.labels.*.name, 'automatic-merge')
        && (
           contains(github.event.pull_request.labels.*.name, 'minor')
           || contains(github.event.pull_request.labels.*.name, 'patch')
        )
     runs-on: ubuntu-latest
     permissions:
        contents: write
        pull-requests: write
     steps:
        - name: Enable auto-merge
           run: |
              gh pr merge --merge --auto "$PR_URL"
           env:
              PR_URL: ${{github.event.pull_request.html_url}}
              GITHUB_TOKEN: ${{secrets.GITHUB_TOKEN}}
```

Listing 6: Der enable-auto-merge-Workflow.

Wir erstellen eine neue Datei `enable-auto-merge.yml`, diese wird ebenfalls in `workflows` abgespeichert *(siehe Listing 6)*.

Um den Workflow in den Branch Protection Rules auswählen zu können, müssen wir ihn jetzt noch manuell auslösen.

Findet Dependabot mehr als nur einen PR, würde beim zweiten PR, nachdem der erste gemerged wurde, Folgendes stehen: „This branch is out-of-date with the base branch." Aus diesem Grund macht Dependabot ein Force-Push von `main` in den jeweiligen PR *(siehe Abbildung 7)*. Danach laufen dann die Tests von vorne los und der Code wird bei Erfolg automatisch gemerged.

Manchmal benötigt ein Force-Push etwas mehr Zeit, man kann dann das Ganze auch per Button selbst erledigen.

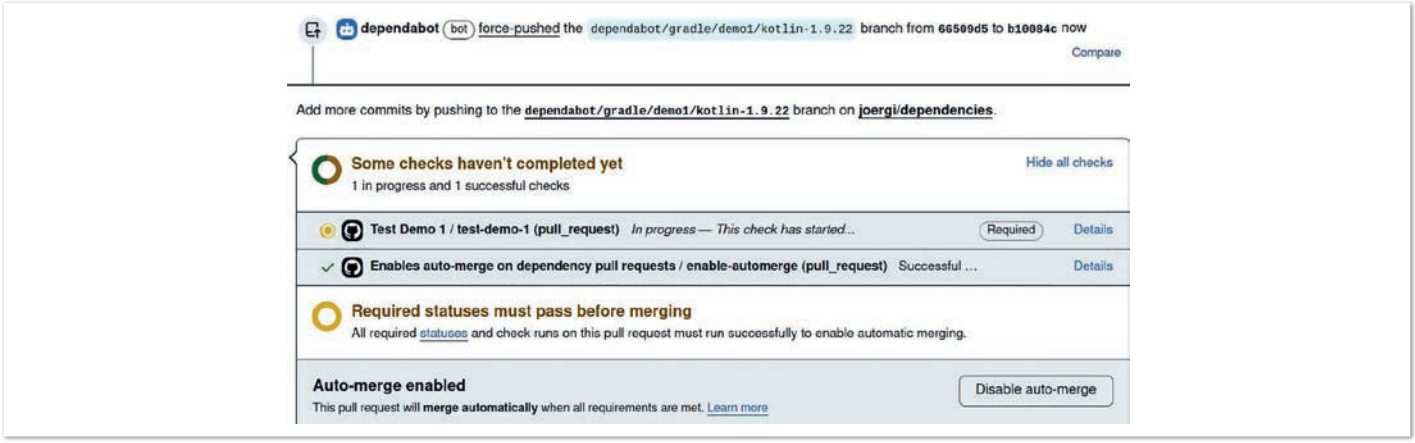


Abbildung 7: So sieht ein automatischer Ablauf aus. ([Screenshot GitHub])

```
name: Test Demo 1

on:
pull_request:
paths:
- 'demo1/**'
- '.github/**'
branches: [ main ]

  workflow_dispatch: {}

jobs:
test-demo-1:
if: |
!contains(github.event.pull_request.labels.*.name, 'automatic-merge')
runs-on: ubuntu-latest
```

Listing 7: So sieht der Beginn von testing-demo1.yml aus.

```
name: Auto-Merge Test Demo 1

on:
pull_request:
paths:
- 'demo1/**'
- 'demo2/**'
- 'demo3/**'
- '.github/**'
branches: [ main ]
workflow_dispatch: {}

jobs:
auto-merge-test-demo-1:
if: |
contains(github.event.pull_request.labels.*.name, 'demo1') ||
contains(github.event.pull_request.labels.*.name, 'automatic-merge')
runs-on: ubuntu-latest
```

Listing 8: So sieht der Beginn von auto-merge-testing-demo1.yml aus.

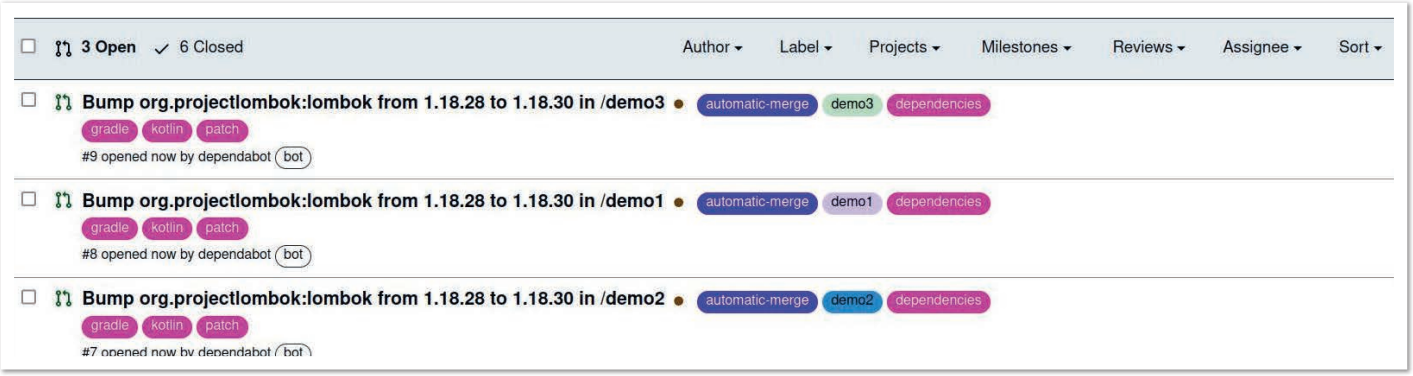


Abbildung 8: In allen drei Projekten wurde ein PR getriggert. ([Screenshot GitHub])

Besonderheiten in einem Monorepo

Und jetzt noch ein Monorepo-Beispiel, das fertige Monorepo-Projekt findet ihr hier.*[5]*

Wenn Teams nicht ausschließlich mit *pairing* und *trunk-based*, also nur auf dem main-Branch arbeiten, sondern klassisch mit Branches und Pull-Requests, sollen nur die eigenen Tests laufen und nicht die Tests der anderen Projekte.

Um dies zu erreichen, habe ich alle Workflows dupliziert: einmal einen normalen Test mit `testing-demo1.yml` und dann den gleichen noch einmal Test mit `auto-merge-testing-demo1.yml`. Der Unterschied zwischen den Workflows ist nur, wann dieser laufen soll, und wann er übersprungen (skipped) werden soll. `testing-demo1.yml` soll nur bei Änderungen in `demo1` getriggert werden *(siehe Listing 7)*. `auto-merge-testing-demo1.yml` hingegen sollbei allen Projektänderungen getriggert werden *(siehe Listing 8)*. Um das Skipping zu erreichen, überprüfen wir in beiden Workflows noch, ob das Label `automatic-merge` gesetzt ist oder nicht und, ob es sich um das richtige Projekt handelt.

Wenn wir jetzt testweise in jedem Projekt eine Dependency downgraden, sehen wir in *Abbildung 8* schön, dass alle drei Dependencies in den Projekten geupdated werden.

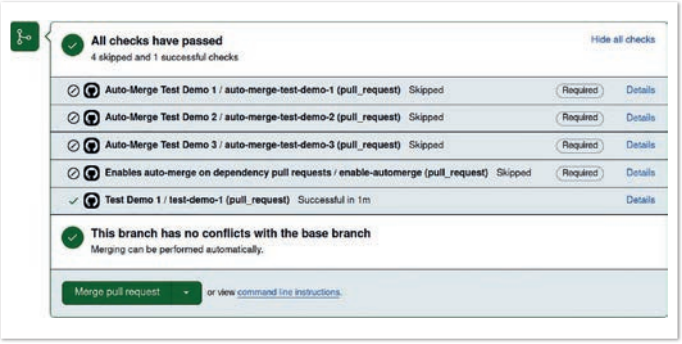


Abbildung 9: Nur die benötigten Tests laufen, der Rest wird geskipped. ([Screenshot GitHub])

Abbildung 10: Leider gab es einen merge-conflict. ([Screenshot GitHub])

Erstellt man selbst einen PR mit einer kleinen Änderung in einem Projekt, so sieht man zwar, dass alle *auto-merge*-Tests gestartet werden, aber da die Bestimmungen nicht erfüllt sind, werden diese geskipped *(siehe Abbildung 9)*.

Und was ist mit Maven?

In einem Maven-Projekt funktioniert es exakt genauso. Der einzige Unterschied ist, dass ihr in eurem Testworkflow statt `./gradlew clean build` nur `mvn clean test` verwendet. Ein fertiges Beispielprojekt könnt ihr euch hier auf GitHub *[6]* anschauen.

Probleme, Tricks & Fazit

Ab und an kann es passieren, dass es zu einem `merge-conflict` kommt *(siehe Abbildung 10)*. Manchmal löst sich so ein Problem von selbst, ähnlich wie bei „This branch is out-of-date with the base branch", da Dependabot die offenen PRs regelmäßig forcepusht, wenn es etwas Neueres gibt. Ansonsten genügt bei einem Konflikt oft ein `@dependabot recreate` im Kommentarfeld des PRs und Dependabot erstellt es neu.

Manchmal verwenden Projekte private oder öffentliche, aber non-Standard Registries (wie beispielsweise die Confluent Registry), die ihr in `dependabot.yml` angeben müsst *(siehe Listing 9)*. Die Secrets hierfür speichert ihr in GitHub unter Secrets ab.

Solltet ihr in einem Monorepo arbeiten, empfehle ich, dass ihr die Dependabot-Zeiten der einzelnen Projekte auf jeden Fall versetzt, dadurch entstehen weniger Merge-Konflikte.

Wie zuvor bereits erwähnt, wäre es wünschenswert, wenn Dependabot die `settings.gradle.kts` auswerten würde. Dann ließe sich die Anzahl der Files in einem Monorepo mit drei Projekten von drei `.toml` Files auf ein File reduzieren. Hierfür gibt es schon Dependabot-Tickets *[7][8]* auf GitHub.

Dies sind Möglichkeiten, wie ihr mit Dependabot die Verwaltung eurer Dependencies automatisieren könnt und so der Arbeitsalltag im Team erleichtert wird.

```
version: 2
registries:
 secret-private-registry:
     type: maven-repository
     url: https://joergi.io/registry/libs-release/
     username: ${{ secrets.MY_REGISTRY_USERNAME }}
     password: ${{ secrets.MY_REGISTRY_PASSWORD }}
  confluent-repo:
     type: maven-repository
     url: https://packages.confluent.io/maven/
updates:
  - package-ecosystem: "gradle"
     directory: "/my-project/"
        - secret-private-registry
        - confluent-repo
    labels:
       ......
```

Listing 9: Das Dependabot File mit einer privaten und einer öffentlichen Registry.

Quellen

[1] Beispiel-Projekt: *https://github.com/joergi/dependencies*

[2] Spring initializer: *https://start.spring.io/*

[3] Dependabot-Ecosystem: *https://docs.github.com/en/code-security/dependabot/dependabot-version-updates/configuration-options-for-the-dependabot.yml-file#package-ecosystem*

[4] Github CLI *https://cli.github.com/*

[5] Monorepo-Beispiel: *https://github.com/joergi/dependencies-monorepo*

[6] Maven-Beispielprojekt *https://github.com/joergi/dependencies-with-maven*

[7] Dependabot Ticket: *https://github.com/dependabot/dependabot-core/issues/6831*

[8] Dependabot Ticket: *https://github.com/dependabot/dependabot-core/issues/1164*

Jörg Liedl
SpringerNature AG
joerg.liedl@gmail.com

Jörg Liedl arbeitet als Senior Softwareentwickler bei dem Wissenschaftsverlag SpringerNature AG. Das Java-Ökosystem ist seine Heimat, wo mittlerweile aber Kotlin das gute alte Java verdrängt hat. Er bevorzugt es, Dinge zu automatisieren, um den eigenen Arbeitsalltag angenehmer zu gestalten.

End-2-End-Testing mit Playwright Java

Frederik Hahne, adesso SE

Während in letzter Zeit „Fullstack"-Entwicklung mit Java und Server-Side-Rendering wieder in Mode gekommen ist, muss man für End-2-End-Tests entweder auf Javascript basierende Frameworks zurückgreifen oder Selenium verwenden, mit allen Vor- und Nachteilen. In diesem Artikel werden wir Playwright Java als moderne Alternative zur Browserautomatisierung und zum End-2-End-Testing vorstellen, sodass jeglicher Code mit Java geschrieben werden kann.

Playwright Java im Überblick

Playwright [1] ist eine von Microsoft entwickelte Open-Source-Bibliothek zur Browserautomatisierung und zum End-2-End-Testing. *Playwright* unterstützt verschiedene Browser und übernimmt auch die Installation auf jeder unterstützten Plattform (Windows, Mac, Linux). Es existieren Implementierungen für *Node.js* (*TypeScript, JavaScript*), *Python*, *.NET* und *Java*, wobei die Implementierung für *Node.js* die führende Variante ist. Eine Reihe von Tools erleichtert sowohl das Erstellen von Tests als auch die Fehlersuche.

Mithilfe des Codegen-Tools können Klickpfade aufgezeichnet werden und in jeder unterstützten Sprache abgespeichert werden. Der *Inspector* erlaubt es, Tests schrittweise zu durchlaufen und dabei zum Beispiel den *Locator* zu ändern. Mit dem *Traceviewer [3]* kann man die Timeline eines ausgeführten Tests betrachten und somit im Fehlerfall das Problem leichter identifizieren. *Playwright* verspricht, dass „flaky" Tests der Vergangenheit angehören, unter anderem durch *Auto-wait [4]* und automatische Retries, sodass keine künstlichen Timeouts oder Wartezeiten in Tests eingebaut werden müssen. Neben Browsertests kann *Playwright* auch verwendet werden, um REST-APIs zu testen. Diesen Anwendungsfall werden wir im Rahmen dieses Artikels aber nicht näher betrachten. Es ist ebenfalls möglich, Anfragen und Antworten abzufangen und zu verändern, um zum Beispiel das Verhalten der Anwendung im Fehlerfall zu prüfen

Playwright ohne Test Runner

Um *Playwright* in seinem Projekt zu verwenden, genügt es, `com.microsoft.playwright:playwright` als Abhängigkeit hinzuzufügen (Beispiel mit *Gradle* siehe *Listing 1*). Danach kann *Playwright* im eigenen Programm verwendet werden. Um das *Playwright*-Kommandozeilenwerkzeug direkt verwenden zu können, definieren wir einen neuen *Gradle Task*, der mit `gradlew playwright --args="help"` aufgerufen werden kann. In einem größeren Projekt ist es sinnvoll, Unit-, Integration- und End-2-End-Tests getrennt ausführen zu können. Der neue Task e2e ist ein Test-Task und führt nur Tests aus, die auf `E2E` enden. Alternativ kann man auch ein weiteres `SourceSet` definieren, aber die Namenskonvention ist ausreichend.

Am schnellsten kann man *Playwright* ohne Test Runner verwenden *(siehe Listing 2)*. Das *Playwright*-API besteht im Kern aus folgenden Elementen:

- Playwright
- Browser
- Context
- Page

`Playwright` ist der zentrale Einstiegspunkt des API. Mit einem erstellten `Playwright`-Objekt kann dann ein `Browser` gestartet werden. Im Standard wird ein *headless* Browser gestartet. Ein `Context` dient zur Isolation von verschiedenen Browser-Sessions, sodass zum Beispiel verschiedene Benutzer voneinander unabhängig simuliert werden können. Das `Page`-Objekt ist die elementare Klasse, um mit dem Browser zu interagieren. Auf einer Page können HTML-Elemente

```
dependencies{
     testImplementation "com.microsoft.
playwright:playwright:1.41.0"
}

tasks.register("playwright", JavaExec) {
            classpath sourceSets.test.runtimeClasspath
            mainClass = 'com.microsoft.playwright.CLI'
}

tasks.register("e2e", Test) {
            include '**/*E2E.*'
            exclude '**/*Spec.*', '**/*Test.*', '**/*IT.*'
          description 'Execute all E2E with playwright java.'
            group 'Verification'
            useJUnitPlatform()
}
```

Listing 1: Playwright-Integration in build.gradle

```
public class PlaywrightOhneTestRunner {
     public static void main(String[] args) {
         try (Playwright playwright = Playwright.create();
                        Browser browser = playwright.firefox()
                                       .launch(new BrowserType.LaunchOptions().setHeadless(true))) {
                BrowserContext context = browser.newContext();
                page.navigate("https://www.ijug.eu/de/home/");

                assertThat(page).hasTitle(Pattern.compile("Home: iJUG"));
                page.locator("#nav-primary").getByRole(AriaRole.LINK, new Locator.GetByRoleOptions().setName("Java
aktuell")).click();
                assertThat(page).hasTitle("Java aktuell: iJUG");

                page.getByRole(AriaRole.LINK, new Page.GetByRoleOptions().setName("Weitere Informationen zum Maga-
zin")).click();
                assertThat(page).hasTitle("Zeitschrift: iJUG");

                page.locator(".article.news-badge").first().screenshot(new Locator.ScreenshotOptions().
setPath(Paths.get("build", "aktuelle-ausgabe.png")));
                String header = page.locator(".news-badge__content h3").first().innerText();
                System.out.println(header.split(":")[1].trim()); // --> Java aktuell 1/24: Java 21
         }
     }
}
```

Listing 2: Playwright-Skript, um das Titelthema der aktuellen Java aktuell herauszufinden

selektiert werden, Assertions geprüft oder Screenshots der aktuellen Seite angelegt werden. Als Benutzer muss man beachten, dass alle Objekte und Methoden des *Playwright*-API nicht thread-safe sind.

Wie man in *Listing 2* erkennen kann, gibt es keine synthetischen *waits* oder *sleeps* innerhalb des Skriptes, um zum Beispiel auf das vollständige Laden der Seite oder auf die Initialisierung von Elementen zu warten. *Playwright* stellt sicher, dass beispielsweise die click-Operation nur ausgeführt wird, wenn das Element auch klickbar ist. Sollte es innerhalb einer konfigurierbaren Zeit nicht klickbar sein, schlägt die Operation endgültig fehl *[4]*. Dadurch sind *Playwright*-Tests gut lesbar und verständlich.

Als Nutzer wird man am meisten mit dem `Locator` interagieren. `Locator` dienen dazu, Elemente auf der Webseite zu finden. *Playwright* empfiehlt in der Regel, sichtbare Attribute oder Elemente mit einer festen Semantik zu verwenden *[5]*. Ein Beispiel eines solchen *Locator* ist der `getByRole`-Locator in *Listing 2*. Die Verwendung des generischen *Locator* sollte im Idealfall vermieden werden, stattdessen sollten die spezifischen `getByXYZ`-Methoden verwendet werden, um Elemente zu selektieren, da diese besser widerspiegeln, wie Benutzer (oder auch Screenreader) die Seite wahrnehmen.

Es lässt sich natürlich nicht immer vermeiden, technische *Locator* zu verwenden, insbesondere, wenn kein semantisches HTML verwendet wird, sondern verschachtelte *divs* mit Klassen. Der spezielle Locator für „test ids" vereinfacht die Verwendung von `data-testid`-Attributen, die in der Regel stabiler sind als Texte einer Webanwendung.

Der gesamte Code ist auf GitHub verfügbar *[6]*.

Erstellen von Traces

Im Fall von fehlerhaften Tests ist es notwendig, herauszufinden, warum der Test gescheitert ist. *Playwright* bietet mit der Trace-

```
public class PlaywrightOhneTestRunner {
     public static void main(String[] args) {
         try (Playwright playwright = Playwright.create();
                         Browser browser = playwright.firefox()
                                 .launch(new BrowserType.LaunchOptions().setHeadless(true))) {
                 BrowserContext context = browser.newContext();
                 context.tracing().start(new Tracing.StartOptions()
                         .setScreenshots(true)
                         .setSnapshots(true)
                         .setSources(true));

                 page.navigate("https://www.ijug.eu/de/home/");

                 // ...
                 context.tracing().stop(new Tracing.StopOptions()
                                 .setPath(Paths.get("build/ohneRunner/trace.zip")));

         }
     }
}
```

Listing 3: Erstellen einer Trace-Datei

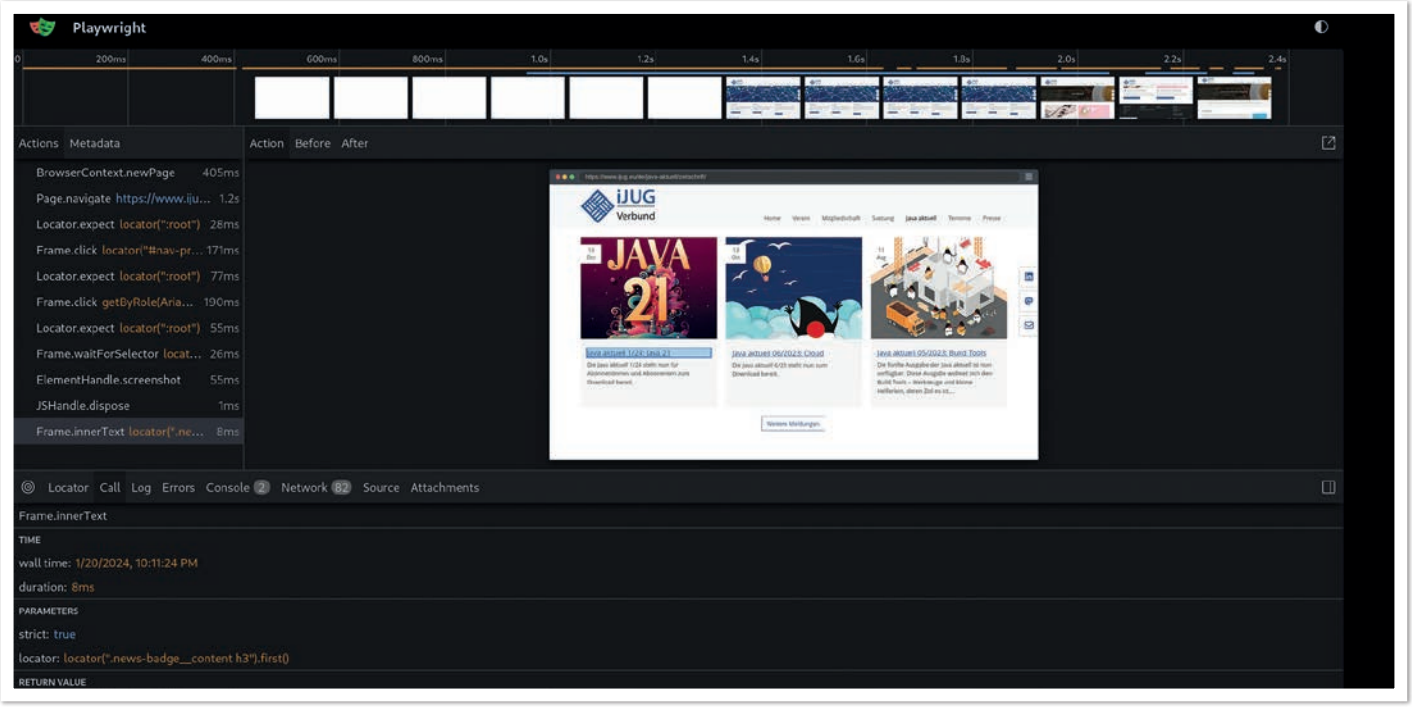


Abbildung 1: Ansicht eines Trace (© Screenshot von Frederik Hahne)

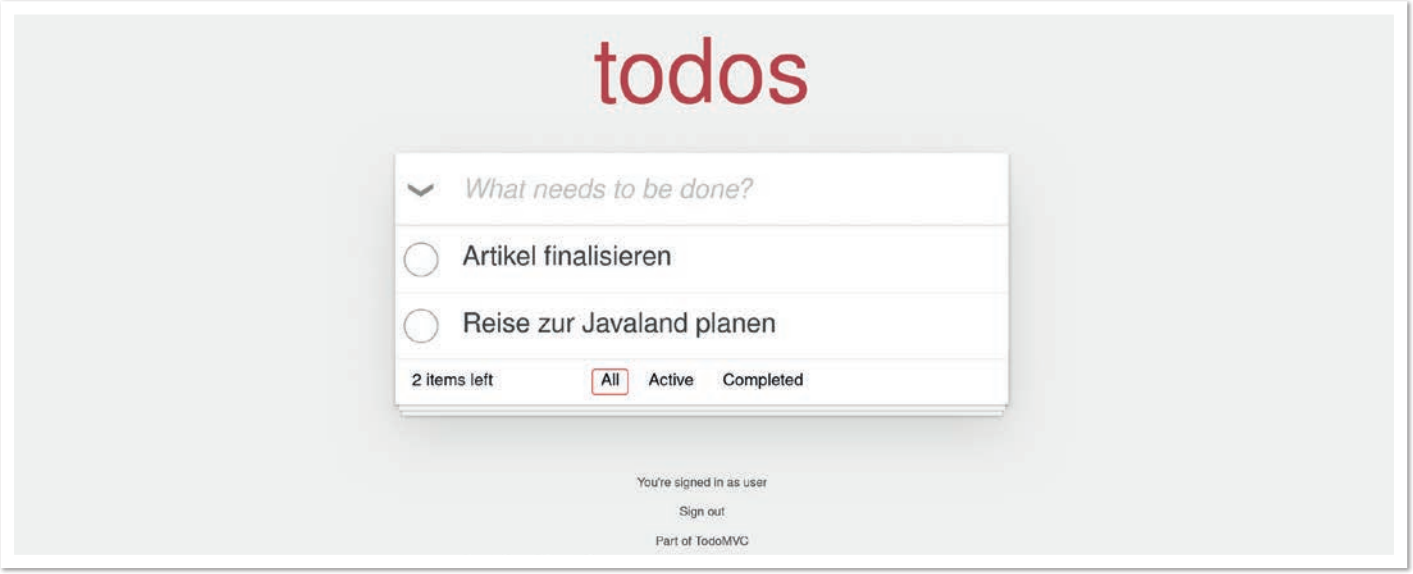


Abbildung 2: TodoMVC auf Basis von Spring Boot und Thymeleaf (© Screenshot von Frederik Hahne)

Funktion *[3]* eine Alternative zu Screenshots, Videos oder der manuellen Beobachtung des Testablaufs im Browser.

Um einen Trace-Datei zu erzeugen, muss die Aufzeichnung des Trace zunächst gestartet, am Ende gestoppt und die Datei gespeichert werden *(siehe Listing 3)*. Die Datei kann nun entweder lokal mit der *Playwright*-CLI oder im Browser via *https://trace.playwright.dev/* geöffnet werden. Um ein Trace lokal zu öffnen, muss der Parameter show-trace und der Pfad zur Trace-Datei übergeben werden (*gradlew playwright –args="show-trace build/ohneRunner/trace.zip"*). Im oberen Bereich sieht man die Zeitleiste des Tests, auf der linken Seite die einzelnen Schritte des Tests. Man kann jeden einzelnen Schritt nachverfolgen und Fehler so nachvollziehen *(siehe Abbildung 1)*.

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class PlaywrightJUnitTestConfiguration {

      Playwright playwright;
      Browser browser;

      @BeforeAll
      void launchBrowser() {
            playwright = Playwright.create();
            browser = playwright.firefox().launch();
      }

      @AfterAll
      void closeBrowser() {
            playwright.close();
      }

      BrowserContext context;
      Page page;

      @BeforeEach
      void createContextAndPage() {
            context = browser.newContext();
            page = context.newPage();

            context.tracing().start(new Tracing.StartOptions()
                        .setScreenshots(true)
                        .setSnapshots(true)
                        .setSources(true));
      }

      @AfterEach
      void closeContext(TestInfo testInfo) {
            String displayName = testInfo.getDisplayName();;

            context.tracing().stop(new Tracing.StopOptions()
                        .setPath(Paths.get("build/ohneRunner/" + displayName +".zip")));
            context.close();
      }
}
```

Listing 4: Playwright-JUnit-Oberklasse

Playwright ohne *Test Runner* zu verwenden, ist eine gute Option, um zum Beispiel innerhalb einer eigenen Anwendung Webseiten zu crawlen. Um aber End-2-End-Tests zu schreiben, bietet sich die Verwendung eines *Test Runners* wie *JUnit* oder *TestNG* an. Im folgenden Abschnitt wird auf die Verwendung von *JUnit 5* näher eingegangen.

Playwright mit JUnit

Zum Redaktionsschluss (Januar 2024) existiert noch keine offizielle *Playwright-JUnit*-Extension, aber es existiert eine Community-Extension *[8]*. Der Autor dieser arbeitet allerdings mit dem *Playwright*-Team daran, die offizielle Extension zu entwickeln, daher sind bei der Community-Extension keine Weiterentwicklungen zu erwarten. Eine erste Version der offiziellen Extension existiert bereits, diese wurde bisher aber noch nicht veröffentlicht *[9]*.

Als Beispielanwendung verwenden wir eine angepasste Variante der in *[2]* erstellten TodoMVC-Anwendung auf Basis von *Spring Boot* und *Thymeleaf* *(siehe Abbildung 2)*.

Generell ist die Integration von *JUnit* und *Playwright* relativ simpel *[7]*. Vor einem Test muss der Browser gestartet werden und nach einem Test wieder geschlossen werden – die einfachste Option ist die Verwendung von `@BeforeAll`, `@AfterAll`, `@BeforeEach` und `@AfterEach`.

In der mit `@BeforeAll` annotierten Methode wird der Browser gestartet, sodass dieser nur einmal pro Testklasse gestartet wird, da diese Operation vergleichsweise teuer ist. Damit jede Testmethode einen eigenen Context verwendet, wird in der mit `@BeforeEach` annotierten Methode jeweils ein neuer Context erzeugt. In den entsprechenden `@After`-Methoden werden die erzeugten Objekte dann wieder geschlossen. Wenn ein neuer Context erzeugt wird, kann bei Bedarf das Tracing eingeschaltet werden. Beim Schließen des Context kann die Trace-Datei gespeichert werden. Mithilfe der

```
public class TodoPage {

      private String baseUrl;
      private Page page;
      private Locator pageHeading;
      private Locator titleInput;
      private Locator todoListItems;
      private Locator completeAllTodosToggle;
      private Locator todoCount;
      private Locator clearCompleted;
      private Locator logoutButton;

      public TodoPage(Page page, String baseUrl) {
            this.baseUrl = baseUrl;
            this.page = page;

            pageHeading = page.getByRole(AriaRole.HEADING, new Page.GetByRoleOptions().setName("todos"));
            titleInput = page.getByPlaceholder("What needs to be done?");
            todoListItems = page.locator(".todo-list");
            completeAllTodosToggle = page.locator("#toggle-all");
            todoCount = page.locator(".todo-count");
            clearCompleted = page.locator(".clear-completed");
            logoutButton = page.getByRole(AriaRole.BUTTON, new Page.GetByRoleOptions().setName("Sign out"));
      }

      public void navigate() {
            page.navigate(baseUrl);
            PlaywrightAssertions.assertThat(pageHeading).isVisible();
      }

      public void createNewTodo(String title) {
            titleInput.fill(title);
            titleInput.press("Enter");

            PlaywrightAssertions.assertThat(todoListItems).isVisible();
      }

      public void completeAllTodos() {
            completeAllTodosToggle.click();

            Assertions.assertThat(todoCount.innerText()).isEqualTo("0 items left");
      }

      public void completeFirstTodo() {
            String todoCountBefore = todoCount.innerText();
            page.locator(".toggle").first().click();

            Assertions.assertThat(todoCount.innerText()).isNotEqualTo(todoCountBefore);
            clearCompleted.click();
      }

      public void logout() {
            logoutButton.click();
            PlaywrightAssertions.assertThat(page.getByRole(AriaRole.ALERT)).isVisible();
      }
}
```

Listing 5: Page Object Model der Todo-Seite

JUnit-`TestInfo` kann man den Namen des Tests verwenden, um einen eindeutigen Dateinamen für die Trace-Datei zu erzeugen. Es bietet sich an, diese Funktionen in eine eigene Klasse auszulagern, von der alle End-2-End-Test dann ableiten *(siehe Listing 4)*.

Um Tests zu strukturieren oder Funktionen in verschiedenen Testmethoden wieder verwenden zu können, bietet es sich an, sogenannte *Page Object Models* zu verwenden *[10]*. Ein *Page Object* benötigt die *Page* und definiert alle relevanten *Locator* als Felder, die dann zum Beispiel im *Constructor* definiert werden. Erst zum Zeitpunkt der Verwendung versucht *Playwright* die Elemente im DOM zu finden *(siehe Listing 5)*.

Zu beachten ist hier, dass der Browser pro Testklasse erzeugt wird. Für jede Testmethode wird ein neuer *BrowserContext* erzeugt. In der Standardkonfiguration führt *JUnit* alle Tests nacheinander in einem Thread aus. Da jede Testmethode durch einen eigenen *BrowserContext* isoliert ist, können die Tests parallel ausgeführt werden. Mit der Annotation `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` teilt man *JUnit* mit, dass diese Klasse nur einmal für alle Testmethoden der abgeleiteten Klasse erstellt wird. Um nun die Tests parallel auszuführen, muss *JUnit* noch entsprechend konfiguriert werden *(siehe Listing 7)*.

Ein abgeleiteter `@SpringBootTest` enthält dann keine Besonderheiten im Vergleich zum Beispiel zu einem Test, der `RestAssured` verwendet. Vor jeder Testmethode wird ein Benutzer eingeloggt und jeder Test verwendet die in den Page-Objekten gekapselten Funktionen, sodass kaum *Playwright*-spezifische Funktionen in den Tests verwendet werden *(siehe Listing 6)*.

Ausblick

Mit der in Entwicklung befindlichen offiziellen *JUnit-Playwright*-Extension *[9]* reduziert sich die Konfiguration auf einige wenige Annotationen. Außerdem wird es möglich sein, Traces nur für fehlerhafte Tests zu schreiben, ohne einen eigenen *JUnit*-Test-Listener schreiben zu müssen. Die notwendigen Objekte des *Playwright*-API können als Parameter injiziert werden, sodass keine gemeinsame Oberklasse mehr notwendig ist. Zudem sorgt die Erweiterung für die korrekte, thread-safe Erzeugung aller *Playwright*-Objekte. Die Verwendung wird sehr wahrscheinlich so aussehen wie in *Listing 8* dargestellt. In der Klasse `DefaultPlaywrightOptions` kann die Konfiguration (zum Beispiel Browser, Viewport) definiert werden.

Fazit

Mit *Playwright Java* steht ein ausgereiftes Framework zur Browserautomatisierung und End-2-End-Testing zur Verfügung. Insbesondere in Projekten, in denen die Verwendung der Node.js-Variante nicht oder nur schwer möglich ist, ist *Playwright Java* eine moderne und nutzerfreundliche Alternative zu etablierten Tools wie *Selenium*. Es hat sich im Projektalltag bei der adesso SE bereits bewährt. Mit der kommenden *JUnit*-Erweiterung wird die Verwendung in Java vereinfacht, sodass einer Verwendung nichts mehr im Wege stehen sollte. Abseits von End-2-End-Testing kann *Playwright* auch im Anwendungscode verwendet werden, um zum Beispiel Webseiten zu crawlen, sofern kein API zur Verfügung steht. Eine Integration in typische Build-Pipelines ist problemlos möglich. Neben einer offiziellen GitHub Action stellt das *Playwright*-Team fertige Container bereit, in denen *Playwright* ausgeführt werden kann.

Allerdings sollte man die Nutzung der Node.js-Variante nicht kategorisch ausschließen, da diese einige Features bietet, die vermutlich niemals in dieser Form mit *JUnit* erreicht werden können. Der erzeugte Report sehr mächtig – man kann zum Beispiel beliebige Dateien anhängen und diese im Report betrachten, was sehr nützlich ist, wenn man zum Beispiel PDF-Dokumente herunterlädt. Außer-

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class PlaywrightJUnitE2E extends PlaywrightJUnitTestConfiguration {

	@LocalServerPort
	private int port;

	private String baseUrl() {
		return "localhost:%s".formatted(port);
	}

	@BeforeEach
	void beforeEach() {
		new LoginPage(page, baseUrl()).doLogin("user", "password");
	}

	@Test
	void createNewTodo() {
		TodoPage todoPage = new TodoPage(page, baseUrl());
		todoPage.navigate();
		todoPage.createNewTodo("Write article");
	}
}
```

Listing 6: Playwright-Test mit JUnit

```
junit.jupiter.execution.parallel.enabled = true.
junit.jupiter.execution.parallel.mode.default = same_thread
junit.jupiter.execution.parallel.mode.classes.default = concurrent
junit.jupiter.execution.parallel.config.strategy=dynamic
junit.jupiter.execution.parallel.config.dynamic.factor=0.5
```

Listing 7: Konfigurationsparameter, um JUnit-Tests parallel auszuführen

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@UsePlaywright(options = DefaultPlaywrightOptions.class)
public class PlaywrightJUnitExtensionsE2E {

      @LocalServerPort
      private int port;

      private String baseUrl() {
          return "localhost:%s".formatted(port);
      }

      @BeforeEach
      void beforeEach(BrowserContext context, Page page) {
            new LoginPage(page, baseUrl()).doLogin("user", "password");
      }

      @Test
      void createNewTodo(Page page) {
            TodoPage todoPage = new TodoPage(page, baseUrl());
            todoPage.navigate();
            todoPage.createNewTodo("Write article");
      }

      @Test
      void completeAllTodos(Page page) {
            TodoPage todoPage = new TodoPage(page, baseUrl());
            todoPage.navigate();
            todoPage.createNewTodo("Write article");
            todoPage.createNewTodo("Javaland");
            todoPage.completeAllTodos();
      }

      @Test
      void completeTodo(Page page) {
            TodoPage todoPage = new TodoPage(page, baseUrl());
            todoPage.navigate();
            todoPage.createNewTodo("Write article");
            todoPage.completeFirstTodo();
      }
}
```

Listing 8: Vorschau auf die JUnit-Playwright-Extension

dem werden Screenshots und Videos an den Report angehängt. Visuelle Regressionstests sind integraler Bestandteil des Node.js-API.

Quellen

[1] Playwright Website: *https://playwright.dev/java/*

[2] Frederik Hahne (2023): Java Aktuell 1/23, Moderne Frontends mit Thymeleaf, Spring Boot und HTMX

[3] Playwright Trace-Viewer: *https://playwright.dev/java/docs/trace-viewer-intro*

[4] Auto-waiting Dokumentation: *https://playwright.dev/java/docs/actionability*

[5] Locator Dokumentation: *https://playwright.dev/java/docs/locators*

[6] Beispiel Code: *https://github.com/atomfrede/java-aktuell-playwright*

[7] Playwright JUnit: *https://playwright.dev/java/docs/test-runners#junit*

[8] Playwright JUnit Community Extension: *https://github.com/uchagani/junit-playwright*

[9] Playwright JUnit Extension: *https://github.com/microsoft/playwright-java/tree/ffe2bd4a966bf109c43eab4a9440d04c58fd4863/playwright/src/main/java/com/microsoft/playwright/junit*

[10] Page Object Models: *https://playwright.dev/java/docs/pom*

Frederik Hahne

frederik.hahne@adesso.de

Frederik entwickelt seit 2007 vorwiegend Webanwendungen, er besitzt aber auch Erfahrung im Bau von Desktopanwendungen, angefangen von JSP-basierten Anwendungen über Wicket bis hin zu Angular und Vue. Seit 2022 bringt Frederik sein Wissen als Senior Software Engineer für die adesso SE am wunderschönen Standort Paderborn in verschiedene, meist Java-basierte Projekte ein. In der knappen freien Zeit organisiert er die Java User Group Paderborn, um den Wissensaustausch, nicht nur zum Thema Java, in und um Paderborn zu fördern. Er ist seit 2015 teil des JHipster Core Teams und kümmert sich dort insbesondere um den Gradle-Support.

Backstage – Nur ein besseres Wiki?

Sascha Selzer, OpenValue Düsseldorf GmbH

ckstage

Dieser Artikel stellt Backstage als Framework für Entwicklerportale vor. Er zeigt, wie es aufgebaut ist, wie man es mit Plug-ins nach den eigenen Wünschen anpassen kann und vermittelt ein Grundverständnis für den Aufbau und die Philosophie hinter Backstage. Abschließend wird erklärt, für wen Backstage ein sinnvolles Tool darstellen kann.

Wir kennen es vermutlich alle: Wir starten mit einem neuen Projekt erstmal klein. Die Codebasis ist noch überschaubar. Dokumentation ist nur sehr rudimentär vorhanden, da das Team noch klein ist und jeder jeden kennt. Über die Zeit wird das Projekt komplexer, es werden mehr Teams dazu geholt, mehr Services und Bibliotheken geschrieben und es wird immer schwieriger, die Übersicht zu behalten. Fragen kommen auf, wie:

- Welche Services und Bibliotheken gibt es überhaupt und werden diese noch benötigt?
- Welchem Team gehört dieser Service oder diese Library?
- Wo kann ich die Dokumentation dazu finden?
- Welche Version von Java verwenden alle?
- Wo ist das API beschrieben, das man verwenden soll?
- Wie erstelle ich am einfachsten einen neuen Service oder eine neue Library?

Je größer das Projekt wird, desto schwieriger wird es, diese Fragen zu beantworten. Viele dieser Informationen sind überall in Wikis verstreut, in veralteten How-tos notdürftig beschrieben oder auch einfach nur in den Köpfen Einzelner gespeichert. *Spotify* stand vor denselben Herausforderungen und hat eine Lösung entwickelt, die eine strukturierte Übersicht über die komplette Entwicklungsabläufe gibt und diese für alle open source zur Verfügung gestellt.

Backstage – Das Tor zur Entwicklungsumgebung

Backstage [1] ist eine zentrale Plattform für das Management von Entwicklungsressourcen. Ein Entwicklungsportal, das Entwicklerinnen und Entwicklern einen zentralen Einstieg bieten soll, um schnell und einfach alle notwendigen Informationen zu finden. Es bringt von Hause aus schon einige Funktionalitäten mit, darunter:

- Einen *Softwarekatalog* als Übersicht über existierende Services, Bibliotheken oder allgemeiner über alle Softwarekomponenten.
- *Software Templates* als einfacher Weg, um neue Softwarekomponenten zu erstellen und in den Katalog aufzunehmen.
- *TechDocs* zur Darstellung und Verwaltung von Dokumentation, die zu einer Softwarekomponente gehört.
- Eine *integrierte Suche*, über den gesamten Datenbestand von Backstage inklusive Dokumentation.

Backstage ist dabei flexibel und auf die eigenen Bedürfnisse anpassbar sowie erweiterbar, entweder um bereits existierende Plug-ins oder um selbstgeschriebene Erweiterungen.

Um die hohe Anpassbarkeit zu gewährleisten, geht Backstage auch beim Aufsetzen einen eigenen Weg.

Wo ist das Docker Image?

Wenn man sich auf *Docker Hub* oder in der Dokumentation von Backstage umschaut, wird man vergebens nach einem fertigen *Container Image* suchen, das man herunterladen und einfach starten kann. Backstage versteht sich weniger als ein fertiges Produkt, das man über Konfigurationen anpassen kann. Es ist vielmehr ein Framework, um ein eigenes, angepasstes Portal zu bauen. Um lokal ein neues Backstage-Projekt aufzusetzen, benötigt man eine aktuelle LTS-Version von Node.js. Dann kann das Projekt mit folgendem Befehl erstellt werden *(siehe Listing 1)*.

```
npx @backstage/create-app@latest
```

Listing 1: Erstellen eines eigenen Backstage-Projekts.

Nach dem Beantworten einiger Fragen, wird ein neuer Ordner mit dem Projekt erstellt. Das Projekt kann dann lokal gestartet werden *(siehe Listing 2)*.

```
yarn dev
```

Listing 2: Lokales Starten von Backstage

Nach kurzer Ladezeit öffnet sich automatisch der Browser und man sieht Backstage in der Standardkonfiguration, das man jetzt nach den eigenen Vorstellungen gestalten kann *(siehe Abbildung 1)*.

Dadurch, dass Backstage nur ein Framework und kein fertiges Produkt ist, hat man die volle Kontrolle über die eigene Instanz, da alles nur Code ist. Man möchte die Navigationsleiste anpassen? Code. Man möchte ein neues Plug-in hinzufügen? Code. Man möchte die Farbe, den Text oder das Logo durch die firmeneigenen Elemente ersetzen? Code. Um das Ganze so einfach wie möglich für die Entwicklerin und den Entwickler zu gestalten, bietet Backstage APIs und eigene UI-Komponenten an. Damit ermöglicht es einen leichten Weg, auf zum Beispiel den ganzen Softwarekatalog oder auch nur auf die Metadaten einer einzelnen Softwarekomponente zuzugreifen.

Es lohnt sich auch ein Blick auf die Architektur, um zu verstehen, wie Backstage als zentrales Portal funktioniert und an alle Daten kommt.

Ein Blick unter die Haube

Die Architektur besteht, vereinfacht gesagt, aus drei Teilen *(siehe Abbildung 2)*:

- Ein um Plug-ins erweiterbares Frontend.
- Ein Backend, das man ebenfalls um Plug-ins erweitern kann, sofern das Frontend Daten benötigt, die nicht im Softwarekatalog vorhanden sind.
- Ein Proxy, der zur Weiterleitung von Anfragen aus dem Frontend an Fremdsysteme dient.

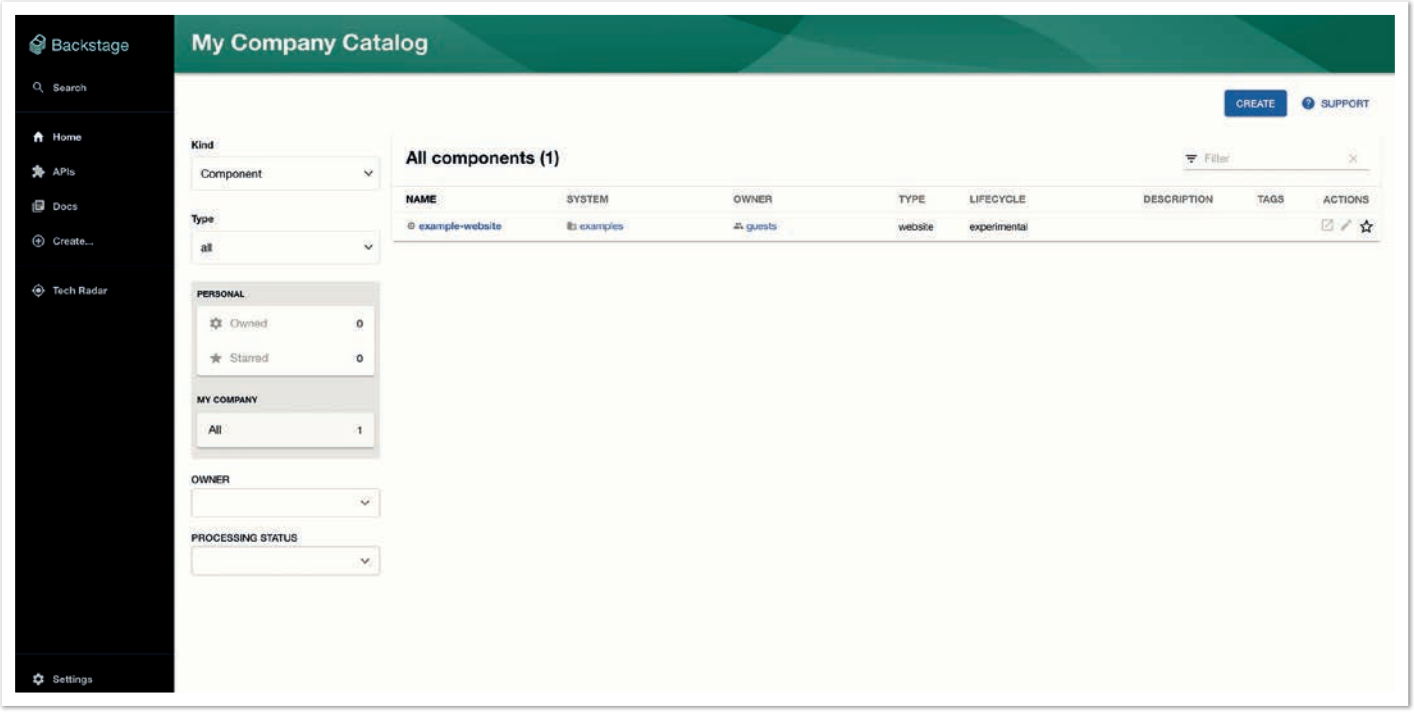


Abbildung 1: Backstage in der Basiskonfiguration (© Sascha Selzer)

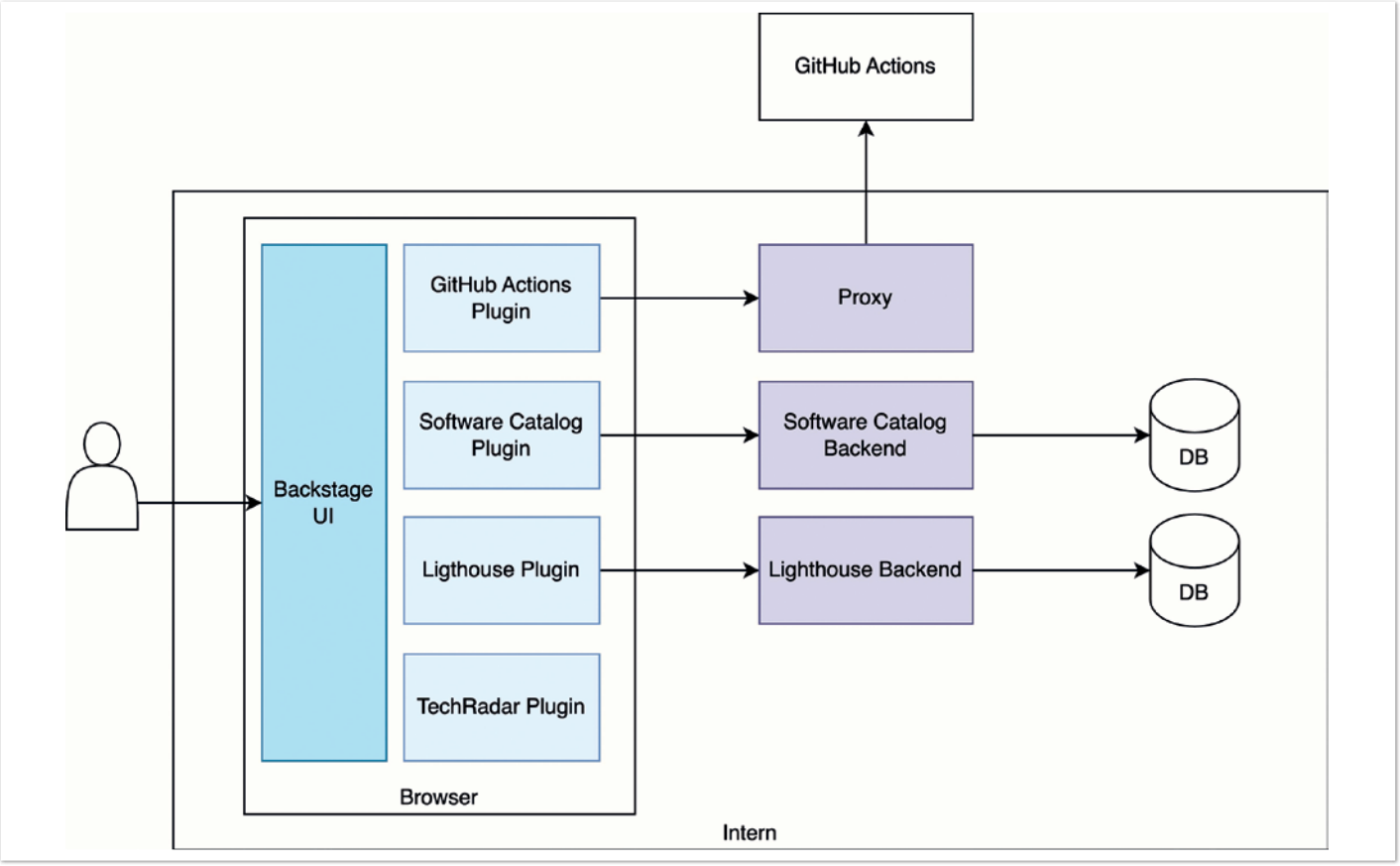


Abbildung 2: Die Backstage-Architektur (© Sascha Selzer)

Das Grundkonzept von Backstage ist, dass es nicht als eine große Datenbank agiert, sondern neben der internen Softwarekatalog-DB auch andere Datentöpfe anzapft oder bestimmte Daten dynamisch von Fremdsystemen anfragen kann. Es zentralisiert nicht alle Daten, macht diese aber leichter auffindbar und integriert Ergebnisse in die eigene UI. Ein Beispiel ist das *GitHub-Actions-Plug-in*, das die Ergebnisse von *GitHub Actions* als Teil der Backstage-UI für eine bestimmte Softwarekomponente anzeigt *(siehe Abbildung 3)*.

So wie bei dem *GitHub-Action-Plug-in* agieren viele Plug-ins nur als eine UI für ein externes System. Diese geben eine bessere Einsicht

in den aktuellen Zustand unserer Softwarekomponente. Doch wie kommen die Komponenten in den Softwarekatalog und wie ist der Katalog aufgebaut?

Wie ist der Softwarekatalog aufgebaut?

Der Softwarekatalog, beziehungsweise das Entitäten-Modell, ist das Herzstück von Backstage und bringt die Softwarekomponenten in Relation zueinander aber auch zu anderen Entitäten. Die wichtigsten Entitäten sind in *Abbildung 4* beschrieben.

- Eine *Domain* beschreibt einen fachlichen Bereich, zu dem mehrere Systeme gehören. Dieser ist gleichbedeutend mit einem *Bounded Context* aus dem DDD-Umfeld und hilft dabei, sehr große Anwendungen zu strukturieren.
- Ein System ist eine Sammlung von *APIs*, *Components* und *Resources*, die eine bestimmte Funktionalität zur Verfügung stellt. Ein Beispiel dafür könnte ein Webshop sein, der vom Kunden als ein System wahrgenommen wird, jedoch aus vielen einzelnen Services bestehen kann.
- Eine *Component* beschreibt eine Softwarekomponente, wie einen Service oder eine Library.
- Ein *API* beschreibt eine Schnittstelle, die im System zu Verfügung gestellt wird. Ein API besteht hauptsächlich aus einer Beschreibung wie beispielsweise OpenAPI, AsyncAPI oder GraphQL und wird meistens von einer Softwarekomponente bereitgestellt.
- Eine *Resource* beschreibt eine Infrastrukturkomponente, die für den Betrieb des Systems oder für einzelne Komponenten notwendig ist, wie zum Beispiel eine Datenbank oder ein Messaging-System.

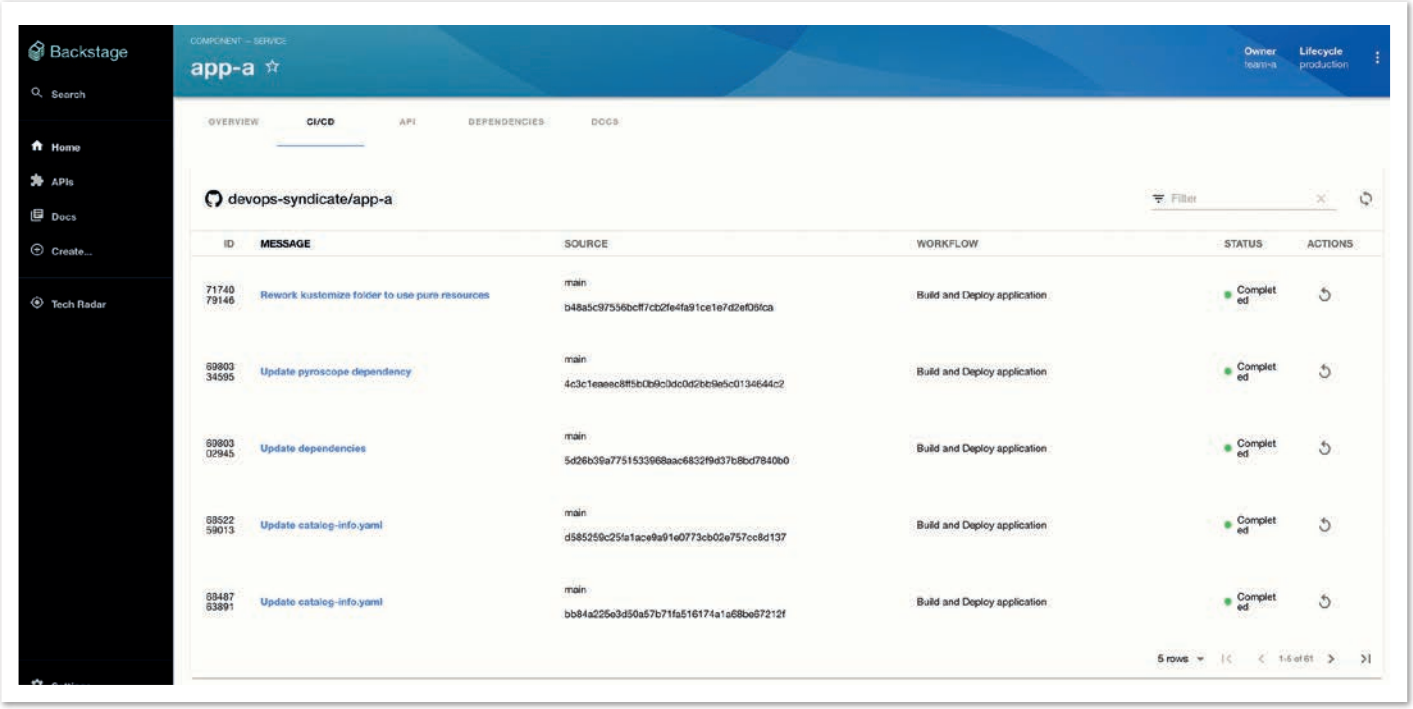


Abbildung 3: GitHub-Action-Jobs in der Backstage-UI (© Sascha Selzer)

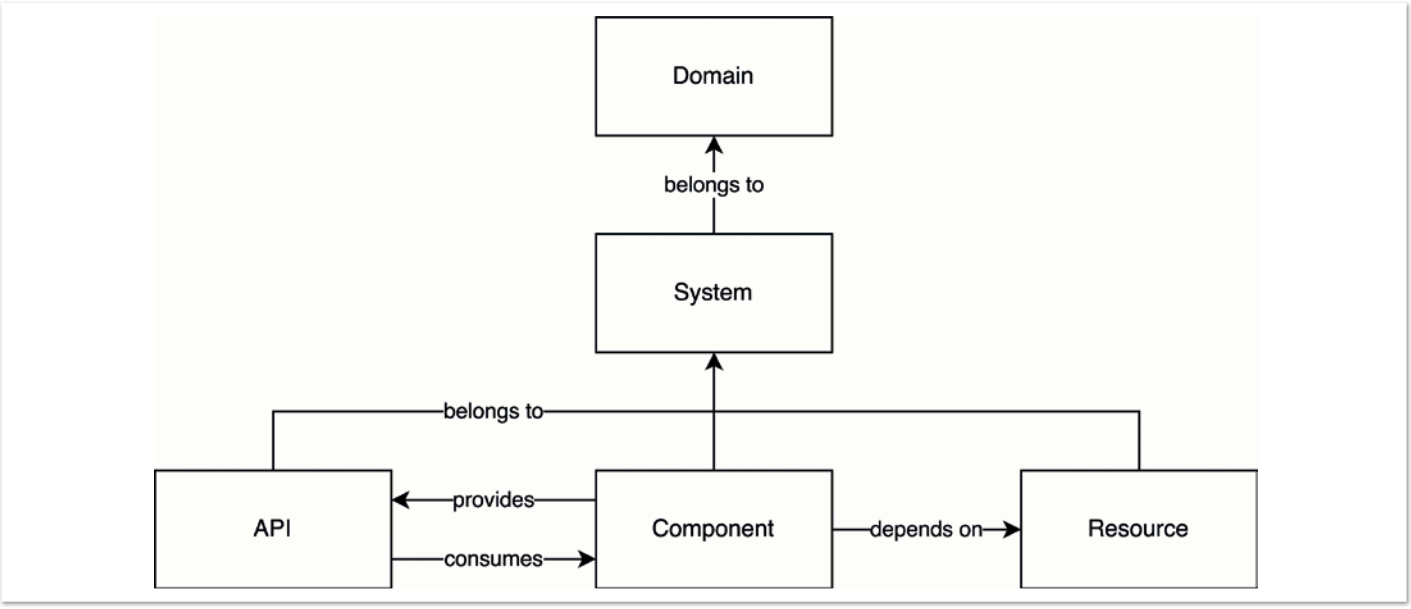


Abbildung 4: Entitäten-Modell in Backstage (© Sascha Selzer)

Entitäten werden formell in YAML beschrieben. Einigen, die schon mit Kubernetes gearbeitet haben, wird die Struktur bekannt vorkommen, da man sich an das *Kubernetes Resource Model* angelehnt hat (*siehe Listing 3* für eine Komponentenbeschreibung).

- `kind` beschreibt den Typ der Entität.
- `metadata` beinhaltet Metainformationen zu der Entität wie Name, Tags oder auch eine Beschreibung. Es kann mit Annotations um eigene Metainformationen beliebig erweitert werden.
- `spec` beinhaltet typenspezifischen Inhalt, der die Entität genauer beschreibt und sie mit anderen in Verbindung setzt. Das `type`-Feld wird verwendet, um Softwarekomponenten feingranularer zu unterteilen, zum Beispiel in Website, Service und Library.

Diese Dateien können statisch erzeugt werden und mit Backstage gemeinsam eingecheckt werden, wie es im Standardsetup beschrieben ist. Das ist aber starr, da neue Komponenten manuell nachgepflegt werden müssen. Deswegen bietet Backstage andere Wege an, um den Katalog automatisch zu befüllen, wenn eine neue Komponente erstellt wird.

Befüllen des Softwarekatalogs

Backstage bietet Integrationen zu verschiedenen Sourcecode-Verwaltungssystem wie unter anderem *GitHub* und *GitLab* an und kann *Repositorys* nach diesen Entitäten durchsuchen. Dabei sucht Backstage eine Datei mit dem Namen `catalog-info.yaml` im Root-Verzeichnis. Wird diese Datei gefunden, werden die Metadateien ausgelesen und in den Softwarekatalog aufgenommen. Das hat den Vorteil, dass die Metainformationen zusammen mit dem Sourcecode im selben *Repository* liegen und vom selben Team gepflegt werden können. Wird nun ein neues Repository für einen Service oder eine Bibliothek erstellt und diese Datei dabei mitangelegt, wird das Repository automatisch in den Softwarekatalog aufgenommen.

Dokumentation als Code

Backstage kann auch Dokumentation zu einem Service oder einer Bibliothek im Markdown-Format rendern. Dabei müssen die Markdown-Dateien im selben Repository liegen wie der Sourcecode. Damit Backstage weiß, dass zu dieser Komponente eine Dokumentation existiert, muss diese Information in die Komponentenbeschreibung aufgenommen werden *(siehe Listing 4)*.

In diesem Fall sucht Backstage in dem Root-Verzeichnis nach einer `mkdocs.yaml`, die Informationen zu Navigation und Inhalt beinhaltet. Eine Beispieldatei könnte wie in *Listing 5* gezeigt aussehen.

```
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: "app-a"
spec:
  type: service
  owner: team-a
  lifecycle: production
  providesApis:
  - app-a-api
  consumesApis:
  - app-b-api
  dependsOn:
  - resource:default/app-a-db
  - component:default/app-b
```

Listing 3: Beschreibung einer Softwarekomponente

```
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: app-a
  annotations:
     backstage.io/techdocs-ref: dir:.
spec:
  ...
```

Listing 4: Verlinkung von Dokumentation zu Komponente

```
site_name: 'architecture'
nav:
- Home: index.md
plugins:
- techdocs-core
```

Listing 5: Beispiel für eine mkdocs.yaml

Das Ergebnis wird dann in Backstage im Reiter „Docs" angezeigt *(siehe Abbildung 5)*.

Da das Rendering auf *MkDocs* basiert, kann man jegliches *MkDocs-Plug-in* integrieren, um die Dokumentation noch weiter an die eigenen Bedürfnisse anzupassen.

Neue Services und Bibliotheken erstellen

Nicht nur bei der Integration von bestehenden Services und Bibliotheken in den Katalog, sondern auch bei der Erstellung von Neuen kann Backstage helfen. Wenn man in einem größeren Projekt neue Services und Bibliotheken erstellt, bedient man sich gerne bei be-

Abbildung 5: Die gerenderte Dokumentation in Backstage (© Sascha Selzer)

stehenden *Repositorys* oder bei sogenannten *Blueprint Repositorys*, die ein Grundskelett für ein neues Repository beinhalten. Am Ende ist es ein Copy-&-Paste-Job und manuelles Anpassen. Das kostet Zeit und kann auch fehlerbehaftet sein.

Einen besseren Weg bietet Backstage mit den sogenannten *Software Templates*, ein Tool, um neue Komponenten zu erzeugen. Ein *Software Template* kann für alles angelegt werden, was man automatisch erzeugen möchte, wie zum Beispiel einen neuen Service, eine neue Library oder anderes, das in dem Projektkontext relevant ist. Ein *Template* wird durch Variablen und eine Folge von Aktionen definiert, die der Nutzer eingeben kann. Standardaktionen sind beispielsweise:

- das Laden von Dateien aus einem bestehenden Repository
- das Überschreiben von Werten in den Dateien mit den manuell eingegebenen Variablenwerten
- das Publizieren gerenderter Dateien und Ordner in ein neues Repository
- das Hinzufügen einer neuen Softwarekomponente zu dem Softwarekatalog

Ein Template wird wie auch eine *Component* in YAML beschrieben *(siehe Listing 6)* und kann über die gleichen Mechanismen von Backstage eingelesen werden.

In diesem Beispiel kann die Nutzerin beziehungsweise der Nutzer einen neuen Service anlegen und muss dabei nur den Service- und Repositorynamen vergeben. Danach werden Dateien von dem Ordner „template" geladen, mit dem Wert der Variable „name" gerendert und dann in ein neues Repository gepusht sowie in den Softwarekatalog aufgenommen. In der UI werden Templates dabei in dem Reiter „Create…" zur Verfügung gestellt *(siehe Abbildung 6)*.

Damit Backstage die folgenden Schritte automatisch ausführen kann, benötigt es Zugriff auf das jeweilige Sourcecode-Verwaltungssystem, um unter anderem ein neues Repository anzulegen. Mithilfe der Templates ist aber das Anlegen von neuen Komponenten einfach und kann durch ein zentrales Team gepflegt werden. Wenn zum Beispiel eine neue Java- oder Spring-Boot-Version veröffentlicht wird, wird das Template angepasst und alle neu angelegten Services würden davon direkt profitieren und man läuft nicht Gefahr, von einem veralteten Repository zu kopieren.

```
apiVersion: scaffolder.backstage.io/v1beta3
kind: Template
metadata:
  name: my-template
  title: My Template
  description: this is a demo template
spec:
  owner: user:guest
  type: service
  parameters:
  - title: Fill in some steps
     required:
     - name
     - repoUrl
     properties:
        name:
           title: Name
           type: string
           description: Unique name of the component
        repoUrl:
           title: Repository Location
           type: string
           ui:field: RepoUrlPicker
           ui:options:
              allowedHosts:
              - github.com
  steps:
  - id: fetch-base
     name: Fetch Base
     action: fetch:template
     input:
        url: ./template
        values:
           name: ${{ parameters.name }}
  - id: publish
     name: Publish
     action: publish:github
     input:
        allowedHosts: ['github.com']
        description: This is ${{ parameters.name }}
        repoUrl: ${{ parameters.repoUrl }}
  - id: register
     name: Register
     action: catalog:register
     input:
        repoContentsUrl: ${{ steps['publish'].output.repoContentsUrl }}
        catalogInfoPath: '/catalog-info.yaml'
```

Listing 6: Die Beschreibung eines neuen Templates für Backstage

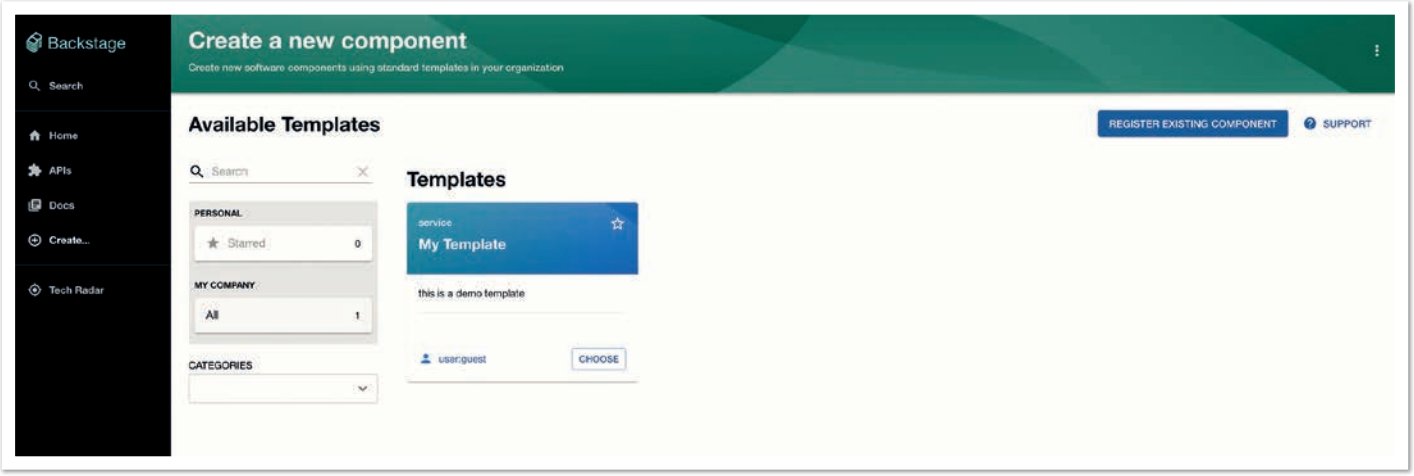


Abbildung 6: Die verfügbaren Software Templates in Backstage (© Sascha Selzer)

Das Ökosystem

Sollten einem die Standardfunktionen von Backstage nicht ausreichen, kann man es mit Plug-ins erweitern. Auf der Website gibt es bereits eine Liste von vorhandenen Plug-ins *[2]*, die Backstage an den verschiedensten Stellen erweitern, wie beispielsweise die Integration von Build-Informationen für ein Projekt von *Circle CI* oder *GitHub Actions*. Oder auch eine Erweiterung von Actions, die wir in unseren Templates nutzen können, wie zum Beispiel das Versenden von Slack-Nachrichten. Die Liste von Plug-ins wächst dabei stetig.

Sollten die Plug-ins nicht ausreichen, da man zum Beispiel ein nicht unterstütztes Sourcecode-Verwaltungssystem verwendet oder ein exotisches Buildsystem hat, hat man immer noch die Möglichkeit, selbst ein Plug-in zu bauen. Durch die bereitgestellten APIs, UI-Komponenten und Anleitungen in der Originaldokumentation, stehen alle Türen offen. Egal wie ausgefallen die eigene Entwicklungsumgebung ist, Backstage lässt sich mit entsprechendem Aufwand an alle Gegebenheiten anpassen.

Fazit

Backstage ist ein gutes Framework, um Ordnung ins Chaos zu bringen und einen zentralen Einstiegspunkt für alle Entwicklerinnen und Entwickler zu schaffen. Es ist flexibel und lässt sich an alle Anforderungen anpassen. Durch die Verwendung von bereits bestehenden oder selbst geschriebenen Plug-ins ergibt sich die Möglichkeit, die Funktionalität einfach nach den eigenen Vorstellungen zu erweitern und auch bestehende Systeme zu integrieren, ist es ein idealer Kandidat für jedes größere Entwicklungsprojekt.

Doch es ist auch nicht alles Gold, was glänzt. Backstage schafft zwar einen schönen Rahmen, doch je mehr integriert wird, desto mehr wird Backstage zum *Single Point of Failure* im System beziehungsweise zum Sicherheitsrisiko. Für jede Integration eines Fremdsystems benötigt Backstage zusätzliche Zugriffsrechte, das heißt, Backstage wird ein potenzielles und lukratives Ziel für Angriffe sein. Doch wenn man diesen Punkt unter Kontrolle hat, bekommt man ein Portal, das keine Entwicklerwünsche offenlässt und einen guten Überblick bieten kann. Besser als jede Wiki.

Quellen

[1] *https://backstage.io*

[2] *https://backstage.io/plugins*

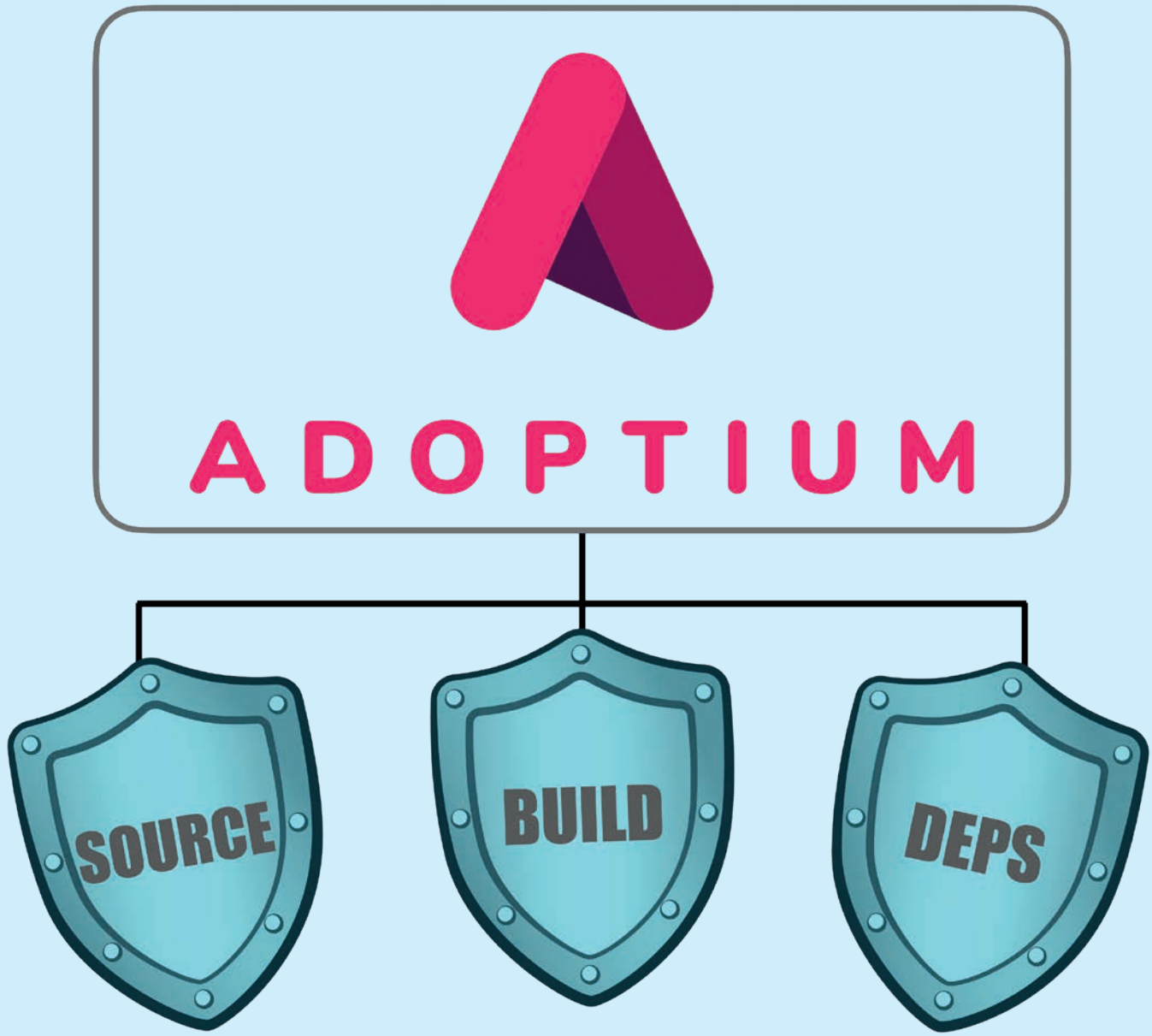
Sascha Selzer

sascha.selzer@openvalue.de

Sascha Selzer ist Senior Software Engineer bei OpenValue und verfügt über umfassende Erfahrung in der Entwicklung mit JVM-basierten Sprachen sowie in der Softwarearchitektur. Er konzentriert sich derzeit auf die Konzeption und Umsetzung von Backend-Architekturen und Continuous-Delivery-/Deployment-Strategien. Darüber hinaus interessiert er sich für Themenbereiche wie Cloud-Monitoring und -Tracing, Microservices und DevOps.

Eclipse Adoptium: Ein Leuchtturmprojekt für sichere und transparente Softwarelieferketten

Scott Fryer, Red Hat, Hendrik Ebbers, Open Elements



In der heutigen digitalen Ära ist die Sicherheit von Software nicht nur ein Anliegen, sondern eine Notwendigkeit. Eclipse Adoptium [1], ein Vorreiter in der Bereitstellung sicherer Java-Laufzeitumgebungen, hat sich dieser Herausforderung gestellt. In diesem Artikel werfen wir einen Blick auf die Initiativen von Adoptium, die darauf abzielen, die Sicherheit in ihrer gesamten Softwarelieferkette (Software Supply Chain) bei dem Bau und der Bereitstellung der Java-Distribution Eclipse Temurin zu revolutionieren.

Sicherheit durch bewährte Frameworks

Adoptium hat sich zum Ziel gesetzt, führende Sicherheitsstandards zu implementieren. Hier haben sich in den letzten Jahren einige wichtige und globale Standardisierungen durchgesetzt. Hierzu zählt das SLSA-Framework (Supply Chain Levels for Software Artifacts) *[2]*, das eine Zertifizierung auf verschiedenen Sicherheitsstufen bietet und, wie in *Abbildung 1* gezeigt, Absicherungen für die gesamte Supply-Chain bietet.

Mit der derzeitigen Zertifizierung auf SLSA-Build-Level 2 und einigen Plattformen, die bereits Level 3 erreicht haben, setzt Adoptium Maßstäbe in der Sicherheit der Softwareproduktion ihrer Java-Laufzeitumgebung Temurin. Adoptium veröffentlicht hierbei den aktuellen Stand der Zertifizierung auf der eigenen Webseite *[3]*. Die Punkte, die hierbei umgesetzt wurden, sind je nach Angriffsvektor *(siehe Abbildung 1)* unterschiedlich:

- Zu Punkt B „Compromise source repo" werden die Quellcode-Repositorys durch eingeschränkte Berechtigungen für alle Benutzer geschützt. Darüber hinaus werden die „main"-Branches gesperrt, um direkte Änderungen zu verhindern, und es wird eine 2-Faktor-Authentifizierung in GitHub für alle Contributors vorausgesetzt.
- Zu den Punkten F, G und H, bei denen es um das Bereitstellen von kompromittierten Artefakten geht, wurden eine ganze Reihe an Sicherheitsmechanismen umgesetzt: Zum Schutz von Release-Paketen hat Adoptium eine Codesignierung mit einer Lösung eines Drittanbieters eingeführt, die die Integrität der Artefakte gewährleistet. Zusätzlich verwendet Adoptium SBOMs, Prüfsummen von Artefakten und GPG-Signaturen, um das Hochladen von geänderten Paketen zu verhindern. Die eigentlichen Prozesse zum Hochladen von Artefakten sind ebenfalls stark eingeschränkt und nur für Personen mit ausreichender Berechtigung verfügbar. Auf der anderen Seite stellt Adoptium Installationsanleitungen zur Verfügung, in denen detailliert beschrieben wird, wie GPG-Signaturen und so weiter zu prüfen sind, um sicherzustellen, dass Endbenutzer Artefakte vor der Verwendung validieren können *[4]*.

Ebenso verpflichtet sich Adoptium, dem *NIST Secure Software Development Framework* (SSDF) *[5]*, das Richtlinien für sichere Softwareentwicklung bereitstellt, zu folgen. Dieses von der US-Regierung definierte Sicherheitsframework wurde mit Fokus auf der Executive Order 14028 *[6]*, dem Memorandum M-22-18 *[7]* und den daraus resultierenden Nachweispflichten *[8]* entwickelt, wodurch nur noch Software, die die Spezifikationen erfüllt, innerhalb der US-Regierung eingesetzt werden darf. Doch auch neue Gesetze der EU, wie der *Cyber Resilience Act* *[9]*, werden solche Nachweise in Zukunft im größeren Rahmen verpflichtend machen. Durch eine Analyse und Umsetzung dieser Frameworks strebt Adoptium danach, die eigenen Sicherheitsstandards kontinuierlich zu erhöhen und somit kommende Regelungen und Nachweispflichten bereits heute zu erfüllen. Hierdurch können Privatpersonen, Firmen und Regierungen bedenkenlos auf Adoptium setzen. Einige der spezifischen Bestandteile dieser Sicherheitsaspekte werden im Folgenden genauer beschrieben.

Aber nun genug der „langweiligen" Gesetze. Schauen wir uns einmal konkrete Punkte an, durch die Adoptium die gewünschte Sicherheit und Transparenz umsetzt.

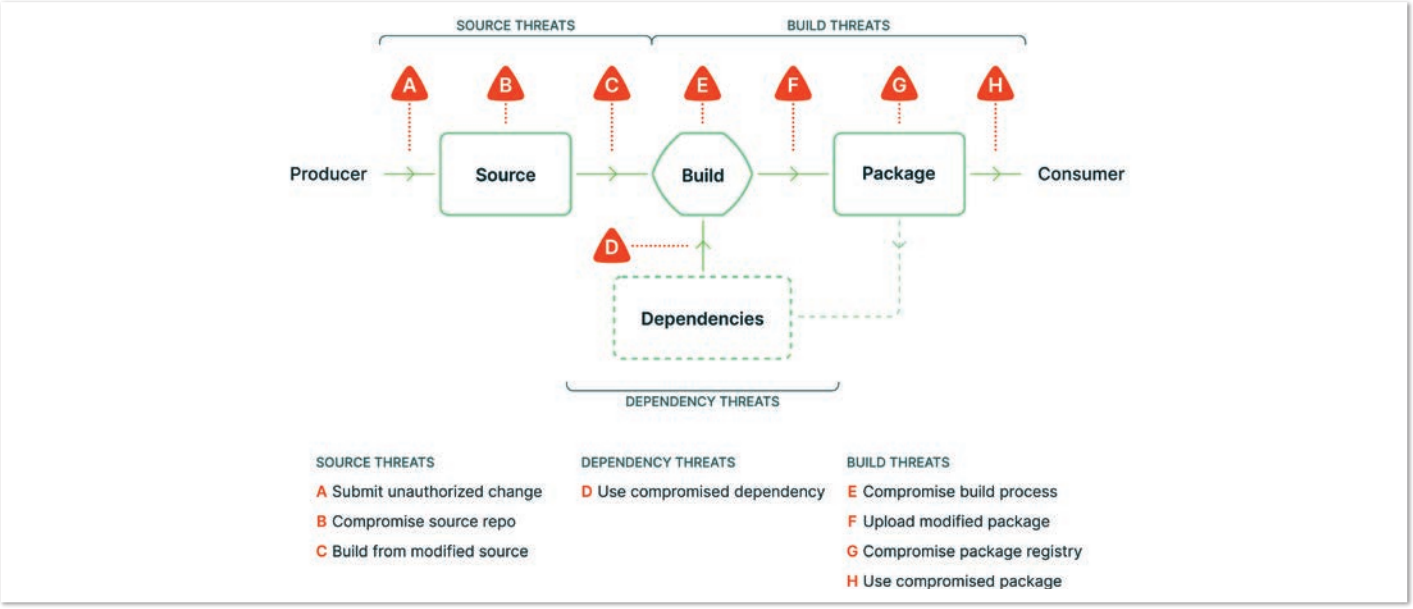


Abbildung 1: SLSA-Framework mit Zertifizierungen auf verschiedenen Sicherheitsstufen

Transparenz durch Software Bill of Materials

Ein weiterer wesentlicher Schritt von Adoptium ist die Einführung einer detaillierten *Software Bill of Materials* (SBOM) *[10]* für jedes Release des Temurin-JDK. Eine SBOM liefert eine umfassende Liste aller Softwarekomponenten und Abhängigkeiten einer Software und ermöglicht somit eine nie dagewesene Transparenz und Sicherheit. Eine SBOM bietet hierbei nicht nur Einblicke in die Zusammensetzung der Software, sondern auch eine verlässliche Basis für Compliance und Sicherheitsprüfungen. Bei Eclipse Adoptium wird das mittlerweile weit verbreitete CycloneDX-Format *[11]* genutzt und die jeweilige SBOM wird als JSON-Datei zu jedem Release bereitgestellt. Wie alle anderen Release-Artefakte von Adoptium sind auch die SBOMs direkt in GitHub-Releases zu finden. Für das jdk-21.0.1+12-Release von Eclipse Temurin, kann man alle Artefakte auf der GitHub-Seite *[12]* finden.

Da Temurin Builds für verschiedene Plattformen (Linux x64, Windows X64, ...) bereitstellt, wird auch für jedes dieser Artefakte eine passende SBOM bereitgestellt (siehe „**OpenJDK21U-sbom***.json"-Dateien im Release). *Listing 1* beinhaltet einen Einblick in eine solche SBOM, die jeweils zu lang sind, um sie komplett abzubilden.

```
{
      "bomFormat" : "CycloneDX",
      "specVersion" : "1.4",
      "version" : 1,
      "metadata" : {
      "timestamp" : "2023-10-18T20:35:48Z",
      "tools" : [
              {
                      "name" : "GLIBC",
                      "version" : "2.17"
              },
              . . .]
       . . .
}
```

Listing 1: Einblick in eine SBOM

Konsistenz durch reproduzierbare Builds

Adoptium gewährleistet, dass jede Version des Temurin-JDK binär identisch nachgebaut werden kann. Dies ist seit Java 21 vollumfänglich möglich und Mitglieder von Adoptium haben hierbei auch mehrere Änderungen zum OpenJDK beigetragen, um reproduzierbare Builds zu ermöglichen.

Dieser Ansatz stellt sicher, dass jeder Build nicht nur verifizierbar, sondern auch zuverlässig ist. Durch die Überwindung von Herausforderungen wie stabilen Build-Umgebungen und der Adressierung nicht-deterministischer Elemente, hebt Adoptium die Vertrauenswürdigkeit und Qualität seiner Software auf ein neues Level. Ein Timestamp des Builds, ein systemspezifisches Newline und viele weitere Kleinigkeiten führen beim Großteil der Softwareprojekte dazu, dass Builds nicht wirklich reproduzierbar sind. Adoptium hat daran gearbeitet, all diese Probleme zu beheben und seit kurzem voll reproduzierbare Builds der Temurin-Distribution zu liefern.

Zum Sicherstellen der Reproduzierbarkeit gibt es bei Adoptium öffentlich einsehbare Jenkins-Jobs, die Temurin auf zwei unterschiedlichen Maschinen baut und dann die Artefakte miteinander vergleicht. Sobald es irgendeinen Unterschied in den Bytes der beiden Artefakte gibt, schlägt der Build direkt fehl. Die Jenkins-Builds für die verschiedenen Releases von Adoptium können jederzeit auf dem Dashboard unter ReproducibleBuild der Adoptium-Website eingesehen werden *[13]*.

Fortschritt durch sichere Infrastruktur

Die Basis aller genannten Initiativen bildet eine sichere Infrastruktur. Adoptium nimmt diesen Aspekt sehr ernst und setzt modernste Strategien und Tools ein, um unbefugten Zugriff zu verhindern und die Entwicklungsprozesse zu schützen. Von der Verwaltung der Inf-

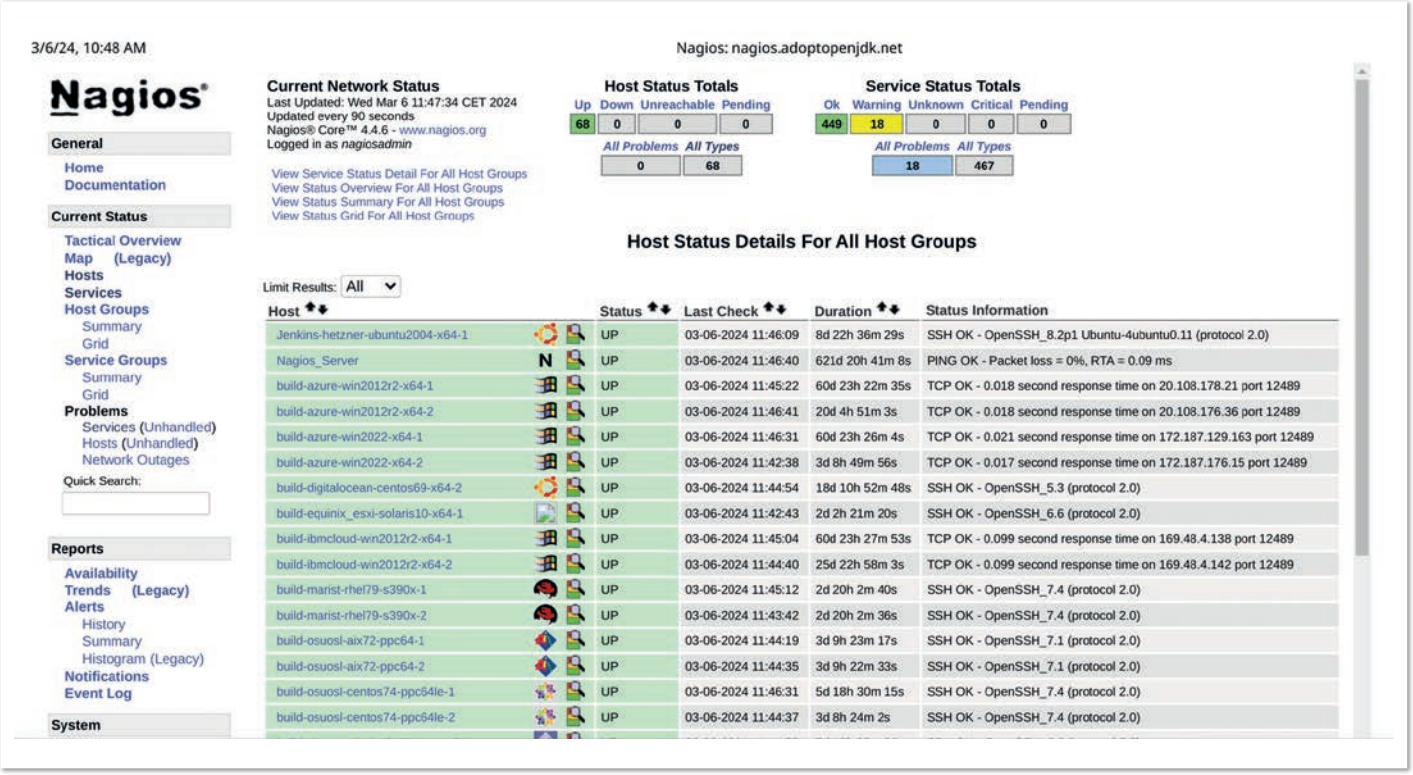


Abbildung 2: Überwachung der Systeme und Services

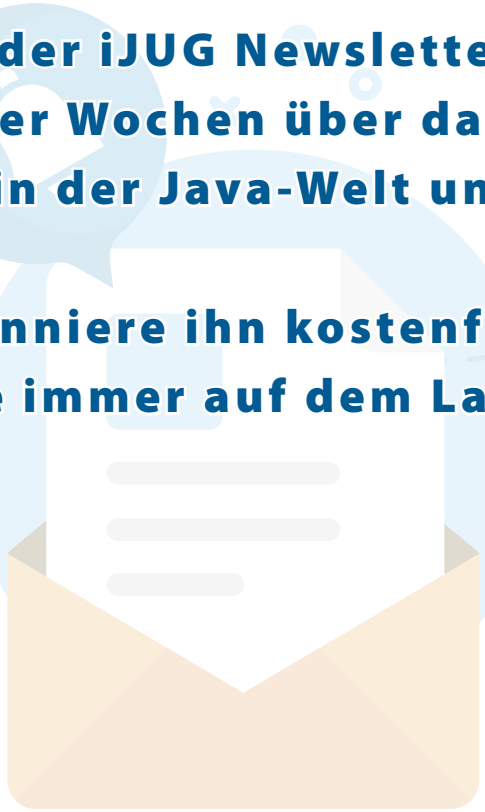
iJUG Verbund

NEWSLETTER

Anmeldung

Java aktuell: der iJUG Newsletter informiert dich alle vier Wochen über das aktuelle Geschehen in der Java-Welt und im iJUG.

Abonniere ihn kostenfrei und bleibe immer auf dem Laufenden!



https://shop.doag.org/newsletter/

oder nach dem Login mit euren Zugangsdaten direkt über euer Profil abonnieren.

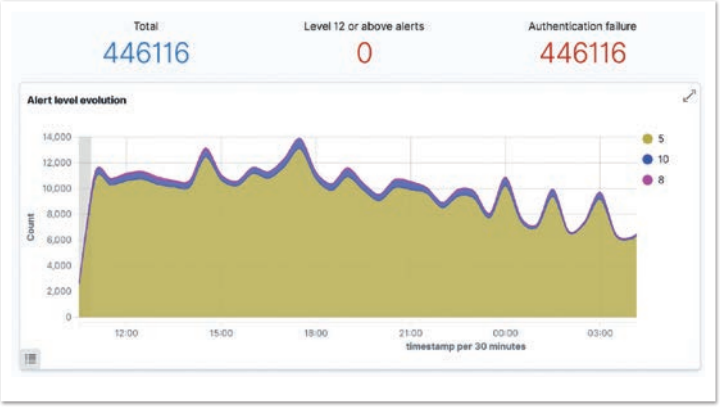


Abbildung 3: Angriffe auf Adoptium-Systeme über einen Zeitraum von 24 Stunden

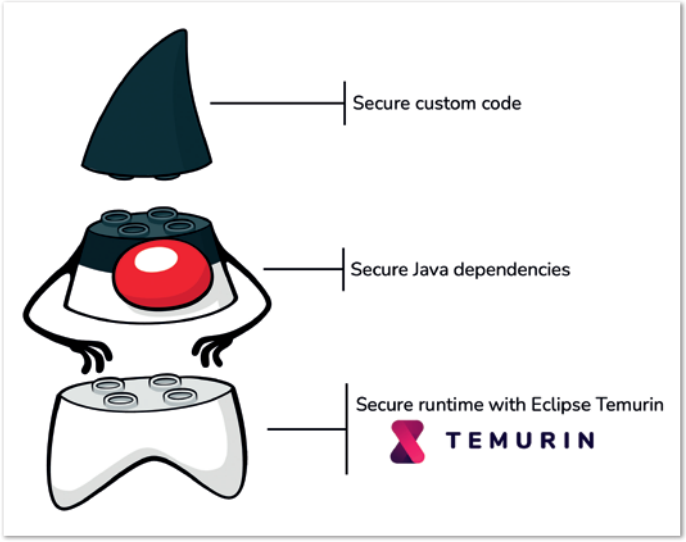


Abbildung 4: Temurin als Sicherheitsbasis für Projekte (© Open Elements)

rastrukturkonfiguration über Ansible-Playbooks bis hin zu strengen Zugriffskontrollen und Sicherheitsaudits, verfolgt Adoptium einen ganzheitlichen Ansatz in Sachen Infrastruktur-Sicherheit. Alle Konfigurationen, Playbooks und Dokumentationen zu dem Thema sind offen auf GitHub im Infrastructure-Repository zu finden *[14]*.

Zur Überwachung aller System wird Open-Source-Software von Nagios *[15]* und Wazuh *[16]* eingesetzt. Leider haben wir hier noch keine frei zugänglichen Dashboards bezüglich der Sicherheitsübersicht aller Systeme. Wie man in *Abbildung 2* erkennen kann, werden aber über 50 Windows, Linux und MaxOS-Systeme mit insgesamt über 400 Services überwacht.

Allein hier gehen täglich unglaublich viele Angriffsversuche ein (Login-Versuche durch gängige Passwörter, etc.), sodass wir von den ersten Ergebnissen dieser Tools wirklich überrascht waren und sich deren Nutzen direkt von Tag 1 an gezeigt hat.

Abbildung 3 zeigt beispielhaft die Angriffe auf die Systeme von Adoptium an einem Tag (18. Januar 2024). Hier kann man sehen, dass es in 24 Stunden insgesamt über 500.000 Angriffe auf das System gab.

Fazit

Adoptium hat in den letzten Jahren immer mehr Engagement in die Sicherheit der Softwareentwicklung investiert. Durch die konsequente Umsetzung von Sicherheitsframeworks, die Einführung von SBOMs und die Verfolgung reproduzierbarer Builds, setzt das Unternehmen teils neue Maßstäbe in Sachen Transparenz und Vertrauenswürdigkeit.

Da all diese Punkte genau wie die Quellen des Open-Source-Projekts so transparent wie möglich betrieben und bereitgestellt werden, können andere (Open-Source-) Projekte sich Adoptium als positives Vorbild nehmen und basierend auf den offenen Beschreibungen die einzelnen Schritte im eigenen Projekt abbilden. Dies wird in naher Zukunft durch die US *Executive Order 14028* und den in der EU geltenden *Cyber Resilience Act* wichtiger als jemals zuvor, da diese Punkte bald schon eine Voraussetzung sein werden, um Software kommerziell zu vermarkten. Da dies natürlich immer für das Softwareprodukt in Gänze (also inklusive Abhängigkeiten, etc.) gilt, bietet Adoptium mit Eclipse Temurin hier die Java-Laufzeitumgebung der Wahl, um sichere Software auf Basis von Java betreiben und vertreiben zu können *(siehe Abbildung 4)*.

Dank der proaktiven Haltung und den immer weiter ausgebauten robusten Sicherheitsmaßnahmen kann Adoptium hier die führende Rolle in der stetigen Weiterentwicklung sicherer Software-Praktiken im Java-Ökosystem beanspruchen. In einer Welt, in der Software-Sicherheit keine Option, sondern eine Notwendigkeit ist, steht Adoptium glücklicherweise an der Spitze dieser Bewegung. Und in diesem Artikel haben wir noch nicht einmal über Eclipse AQAvit, dem offenen Testframework von Adoptium zur Qualitätssicherung von OpenJDK-basierten Distributionen gesprochen.

Referenzen

[1] *https://adoptium.net/*
[2] *https://slsa.dev/*
[3] *https://adoptium.net/de/docs/slsa/*
[4] *https://adoptium.net/de/installation/*
[5] *https://csrc.nist.gov/projects/ssdf*
[6] *https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity*
[7] *https://www.whitehouse.gov/wp-content/uploads/2022/09/M-22-18.pdf*
[8] *https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity*
[9] *https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act*
[10] *https://en.wikipedia.org/wiki/Software_supply_chain*
[11] *https://cyclonedx.org/specification/overview/*
[12] *https://github.com/adoptium/temurin21-binaries/releases/tag/jdk-21.0.1%2B12*
[13] *https://ci.adoptium.net/view/ReproducibleBuild/*
[14] *https://github.com/adoptium/infrastructure*
[15] *https://www.nagios.org/*
[16] *https://wazuh.com/*

Scott Fryer
Red Hat

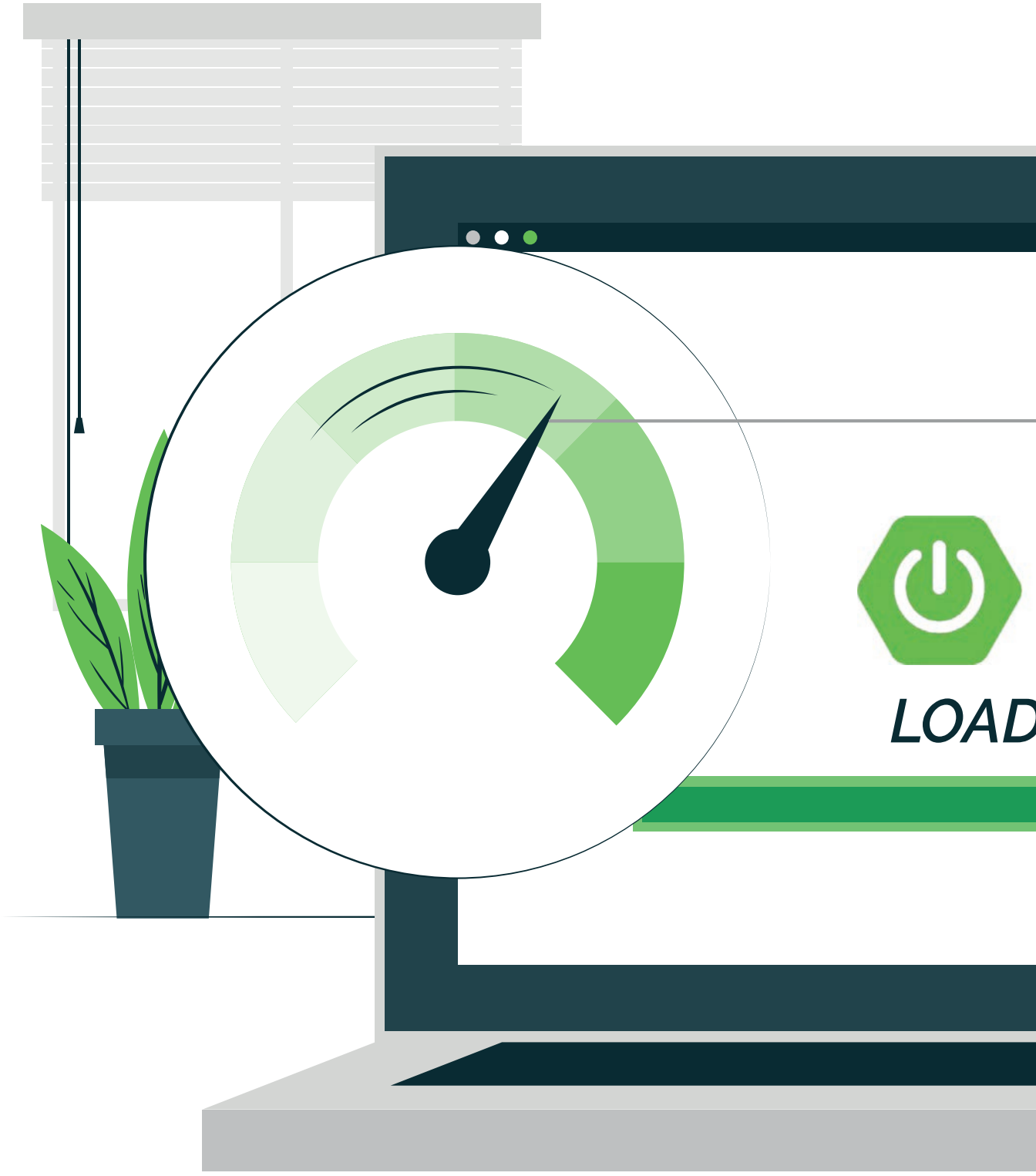
Hat Scott Fryer ist Software-Ingenieur mit einer Gesamterfahrung von über 25 Jahren, hauptsächlich in der Finanzdienstleistungs- und Versorgungsbranche. Er hat sich auf die Automatisierung von Prozessen spezialisiert, die auf das Software-Release- und Konfigurationsmanagement zugeschnitten sind, sowie auf Audits und die Sicherstellung der Einhaltung gesetzlicher Vorschriften. In letzter Zeit war Scott vor allem an der Gewährleistung der Sicherheit der Lieferkette des Adoptium-Temurin-Projekts beteiligt, mit besonderem Schwerpunkt auf der Sicherheit der Infrastruktur und der Erfüllung der SSDF- und SLSA-Rahmenbedingungen. Außerhalb der Arbeit spielt er gerne Brettspiele oder geht mit dem Hund spazieren.

Hendrik Ebbers
Open Elements

Hendrik Ebbers ist Java-Champion, Mitglied von JSR-Expertengruppen und JavaOne-Rockstar. Erreicht hat er das alles durch sein Vertrauen in Open Source und Communitys. Hendrik ist Gründer und Leiter der JUG Dortmund und hält weltweit Vorträge und Präsentationen auf Konferenzen und in ausgewählten User Groups. Er ist Mitglied in den Jakarta EE und Adoptium Working Groups (WG) und Mitglied des Board of Directors (Aufsichtsrat) der Eclipse Foundation. Neben den Eclipse-bezogenen Projekten trägt Hendrik auch zu anderen Open Source Softwares (OSS) bei. Zum Beispiel ist er Core Committer des Hedera Hashgraph, dem einzigen in Java geschriebenen Open Source Public Ledger. Er unterstützt Hedera und Swirlds Labs bei der Entwicklung sicherer, schneller und zuverlässiger Komponenten unter Verwendung von Open-Source-Standards und bekannten Workflows.

Cloud-Native-Apps mit Spring Boot 3 und GraalVM

Daniel Krämer, anderScore GmbH



Spring
Boot 3
DING...

Verglichen mit modernen Cloud-Native-Frameworks wie Quarkus und Micronaut galt Spring Boot zwar stets als ausgesprochen robust und praxiserprobt, konnte hinsichtlich Startzeit und Ressourcenverbrauch allerdings nie mit seiner jüngeren Konkurrenz mithalten. Gerade für hoch verteilte und skalierbare Anwendungen in der Cloud bedeutete dies mitunter einen nicht unwesentlichen Nachteil. Vier Jahre nach der Vorstellung von GraalVM und mehreren Redesigns haben native Images und Ahead-of-time Compilation (AOT) mittlerweile auch Einzug in Spring gehalten. In diesem Artikel wenden wir uns zunächst der Funktionsweise von GraalVM und nativen Images zu. Anschließend werfen wir einen gründlichen Blick unter die Haube des lange erwarteten Spring Boot 3 (vormals Spring Native) und stellen es gegenüber anderen Frameworks auf den Prüfstand.

Einleitung

Bis vor wenigen Jahren galt für Java noch der oft zitierte (und allzu gerne verballhornte) Grundsatz „Build once, run anywhere!" *[1]*. Sourcen wurden stets zu uniformem Bytecode kompiliert und als solcher auf (plattformspezifische) *Java Virtual Machines* geladen. Letztere ermöglichen auf diese Weise eine Ausführung der gleichen Anwendung auf unterschiedlichen Betriebssystemen und Prozessorarchitekturen. Was gerade zur Anfangszeit der Programmiersprache in den 1990er Jahren einen erheblichen Vorteil gegenüber der nativ kompilierenden Konkurrenz (zum Beispiel C++) darstellte, verlor spätestens im Zuge der fortschreitenden Containerisierung rund um Technologien wie *Docker* ab 2013 zunehmend an Bedeutung. Geblieben ist allerdings die Kehrseite der Medaille, die sich in Gestalt (zumindest potenziell) reduzierter Performance und erhöhtem Ressourcenverbrauch zeigt. Vor diesem Hintergrund drängte sich immer mehr die Frage auf, ob der eingangs zitierte Grundsatz überhaupt noch zeitgemäß war.

Einführung in GraalVM

An dieser Stelle betrat im Jahr 2019 mit der von Oracle selbst initiierten *GraalVM* ein neues Java Development Kit die Bühne. Neben der Möglichkeit zur Ausführung von Code nahezu beliebiger Programmiersprachen über abstrakte Syntaxbäume unterstützt es durch die Erweiterung „native-image" *[2]* erstmals auch eine Kompilierung von Sourcen zu nativem Maschinencode, für dessen Ausführung keine virtuelle Maschine mehr benötigt wird.

Was auf den ersten Blick womöglich unspektakulär klingt, stellt für eine stark laufzeitoptimierte Programmiersprache wie Java durchaus eine ernstzunehmende Herausforderung dar. Wie soll etwa Bytecode instrumentiert, über *Class Loader* dynamisch nachgeladen oder mittels *Reflection* analysiert werden, wenn es nun offensichtlich keinen Bytecode mehr gibt?

Eine (teilweise) Lösung des Problems stellt die *Ahead-of-time Compilation* (AOT) dar, deren wesentliches Ziel darin besteht, einen Großteil der sonst zur Laufzeit stattfindenden Dynamik auf den Zeitpunkt der Kompilierung vorzuverlegen. Anstatt beispielsweise mögliche Kandidaten zur *Dependency Injection* erst beim Starten einer Anwendung zu ermitteln, kann dies in vielen Fällen auch schon während des Builds geschehen. Anwendungen, die auf diese Weise vom Compiler optimiert und nativ übersetzt worden sind, besitzen gegenüber ihren Bytecode-Pendants einige nennenswerte Vorteile. Insbesondere zeigen sie eine klare Tendenz zu erheblich reduzierten Startzeiten und deutlich verringertem Speicherverbrauch – Eigenschaften, die in bestimmten Einsatzszenarien von entscheidender Bedeutung sein können.

Gleichwohl kommen diese Vorzüge nicht ohne eine Kehrseite daher. Zwangsläufig zur Laufzeit durchzuführendes Laden oder das Manipulieren von Bytecode ist ausgeschlossen. Mechanismen wie *Reflection* sind zumindest insoweit eingeschränkt, als sämtliche genutzten Klassen und Ressourcen dem Compiler explizit bekannt gemacht werden müssen.

Frameworks

Das Framework *Micronaut [3]* stellte bereits ein Jahr vor Veröffentlichung von *GraalVM* erfolgreich unter Beweis, dass *Ahead-of-time Compilation* (unabhängig von nativen Images) in Java durchaus ein valides Mittel zur Verbesserung von Performance und Ressourcenverbrauch sein kann. Während mit dem 2019 neu entwickelten Konkurrenten *Quarkus [4]* schließlich beide Konzepte miteinander kombiniert und einem breiteren Entwicklerkreis zugänglich gemacht wurden, schritt die Integration in das Spring-Framework, trotz der unmittelbar gegründeten Initiative „Spring Native", nur langsam voran. Ein Java-LTS-Release und diverse Redesigns später, ging diese Initiative im Jahr 2022 schließlich komplett in *Spring 6* sowie *Spring Boot 3 [5]* auf und ist als eigeständiges Projekt seitdem offiziell deprecated.

Quickstart mit Spring Boot

Höchste Zeit also, erste Erfahrungen mit der neuen Technologie zu sammeln! Wie bei *Spring Boot* gewohnt, ist ein frisches Projekt schnell aufgesetzt und in die lokale Entwicklungsumgebung importiert. Hierzu muss lediglich die zusätzliche *Dependency* „GraalVM Native Support" inkludiert werden, die das zusätzliche Maven-Profil „native" und das „native-maven-plugin" aus den *GraalVM Build Tools* mitbringt. In Kombination *(siehe Listing 1)* ist damit bereits ein natives Kompilieren möglich – installierte *GraalVM* inklusive *Native Image Plugin* vorausgesetzt.

```
mvn -Pnative native:compile
```

Listing 1: Maven Command für einen nativen Build

Wie verschiedene Benchmarks später noch zeigen werden, ist dieser Vorgang insbesondere aufgrund der umfangreichen Voranalysen selbst bei einer minimalistischen Anwendung durchaus zeitauf-

wendig und stellt hohe Anforderungen an die genutzte Hardware. Am Ende erhalten wir neben dem gewohnten *Fat Jar* aber schließlich das gewünschte *Native Executable* *(siehe Abbildung 1)* und können es anschließend auf der Kommandozeile (ohne JVM!) starten.

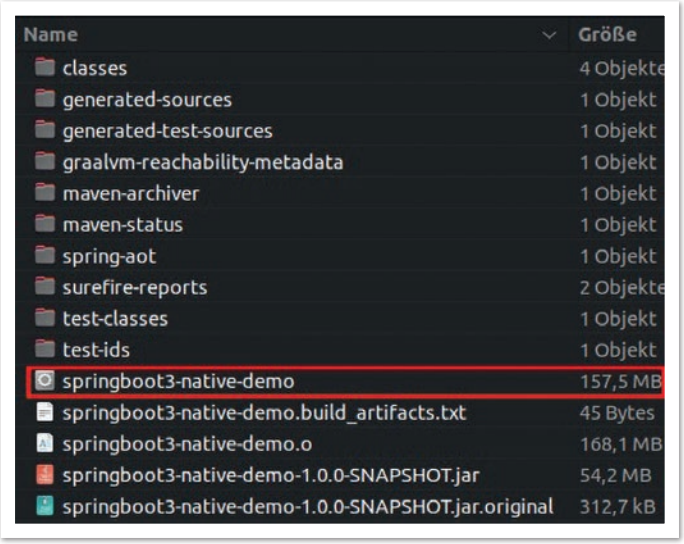


Abbildung 1: Inhalt des target-Verzeichnisses (© Daniel Krämer)

Um das native Image bei dieser Gelegenheit auch gleich in einen geeigneten Container (etwa zum Deployment auf einem Kubernetes-Cluster) zu verpacken, können die vom Spring-Boot-Maven-Plug-in mitgelieferten (und flexibel anpassbaren) Buildpacks genutzt werden *(siehe Listing 2)*.

```
mvn -Pnative spring-boot:build-image
```

Listing 2: Nativer Build (inklusive Docker Image)

Unter der Haube

Im Laufe der dreijährigen Entwicklung *[6]* fanden mehrere grundlegende Neukonzeptionen statt, was die Komplexität des Themas verdeutlicht. Im Gegensatz zu *Quarkus*, das im Rahmen der *Ahead-of-time Compilation* ein Speicherabbild als Ausgangsbasis für spätere Startups erzeugt, hat sich *VMware Tanzu* für eine (automatisierte) Optimierung des *Application Context* entschieden.

Damit *GraalVM* beim nativen Build später geeignete Optimierungen durchführen kann, erledigt das Spring-Boot-Maven-Plug-in zunächst einige Vorarbeiten. Zu Beginn erfolgt eine statische Analyse des *Classpath*, um sämtliche, von der Anwendung **tatsächlich** benötigten Klassen, Methoden, Felder und Ressourcen zu identifizieren. Aus diesen Erkenntnissen generiert das Framework spezielle Hinweise für *GraalVM*, die um zusätzliche Informationen aus dem „GraalVM Reachability Metadata Repository“ *[7]* angereichert werden. Letzteres liefert bereits fertig zusammengestellte Protokolle zu einer Vielzahl häufig genutzter Bibliotheken.

Um die zur Laufzeit existierenden *Spring Beans* zu ermitteln, wird im nächsten Schritt der sonst dynamisch beim Startup (über Mechanismen wie *Component Scan*) aufgebaute *Application Context* hochge-

MITMACHEN UND AUTOR/IN WERDEN!

Sie kennen sich in einem bestimmten Gebiet aus dem Java-Themenbereich bestens aus und möchten als Autor/in Ihr Wissen mit der Community teilen?

Nehmen Sie Kontakt zu uns auf und senden Sie Ihren Artikelvorschlag zur Abstimmung an redaktion@ijug.eu.

Wir freuen uns, von Ihnen zu hören!

iJUG Verbund

fahren – ohne die darin enthaltenen *Beans* tatsächlich zu instanziieren. Letzteres wäre in vielen Fällen auch gar nicht möglich, etwa weil der ausführende CI/CD-Server keinen Zugriff auf die konfigurierten Umsysteme hat und sich dieser Teil der Konfiguration (zum Beispiel über Properties) ohnehin von Umgebung zu Umgebung unterscheidet. Das Ergebnis ist ein stark optimierter *Application Context*, der weitgehend ohne *Reflection* und andere Laufzeitmechanismen auskommt. Die generierten Klassen werden als (AOT-) Sourcecode bereitgestellt, was einen leichten Einblick und bei Bedarf auch Debugging vor dem nativen Build ermöglicht.

Auf Grundlage der zuvor erzeugten, beziehungsweise aus dem Repository geladenen, Metadaten entfernt *GraalVM* während des nachfolgenden, nativen Builds sämtliche Klassen, Methoden, Felder und Ressourcen, die – zumindest gemäß den Erkenntnissen aus der Analyse – zur Laufzeit nicht (mehr) benötigt werden. Danach übersetzt sie zusammen mit den generierten AOT-Sourcen zunächst in Byte- und anschließend in ausführbaren Maschinencode. Sofern angefordert, verpackt Spring diesen abschließend noch über die konfigurierten Buildpacks in ein Docker Image.

Testing

Grundsätzlich spricht nichts dagegen, bestehende Unit- und Integration-Tests im Rahmen des Builds weiterhin auf einer JVM auszuführen und auf diese Weise leicht nachvollziehen und debuggen zu können. Um möglichst frühzeitig zu erfahren, ob und wo der von Spring ausgetauschte *Application Context* gegebenenfalls Probleme bereitet, besteht zudem die Möglichkeit, die AOT-Sourcen bereits in einem konventionellen Build zu erzeugen und die (optimierte) Anwendung anschließend auf der JVM hochzufahren *(siehe Listing 3)*.

```
 mvn -Pnative clean package
java -Dspring.aot.enabled=true -jar target/app.jar
```

Listing 3: Bauen und Starten im AOT-Modus

Da sich eine Anwendung als natives Image aber mitunter anders verhält als sein Bytecode-Pendant (etwa aufgrund wegoptimierter Klassen), empfiehlt sich zumindest bei kritischen Tests zusätzlich auch eine native Ausführung. Spring bietet hierfür Unterstützung über das Maven-Profil „nativeTest", das seinerseits von den *GraalVM Native Build Tools* Gebrauch macht *(siehe Listing 4)*.

```
mvn -PnativeTest test
```

Listing 4: Native Tests

Analog zu einem regulären nativen Build generiert das Spring-Boot-Maven-Plug-in zunächst AOT-(Test-)Sourcecode, um die eingebundenen *Application Contexts* ohne Laufzeitmechanismen abzubilden. Anschließend kompiliert *GraalVM* sowohl die Anwendung **als auch ihre Tests** zu einem gemeinsamen nativen Image. Führt man dieses aus, startet nun allerdings nicht die Applikation, sondern ein *Test Runner*. Nach Abschluss des Laufs präsentiert dieser alle Ergebnisse in strukturierter Form auf der Konsole und beendet schließlich den Prozess.

Limitierungen

Im Wesentlichen erben native Spring-Applikationen sämtliche Einschränkungen, die sich aus der Funktionsweise von *GraalVM* und *Ahead-of-time Compilation* ergeben. Hierzu zählen insbesondere die Beschränkung von *Classpath*, *Reflection* und *Ressource Loading* auf bereits zur Compile-Zeit bekannte Klassen und Dateien sowie der Verzicht auf sämtliche Mechanismen zum Laden und Manipulieren von Bytecode zur Laufzeit (zum Beispiel *Agents*, *Profiler* und *[Lazy] Class Loader*). Zusätzliche Besonderheiten ergeben sich aus der Generierung des optimierten *Application Context*. Insbesondere Profile und *Conditions* werden nun nicht mehr dynamisch zur Laufzeit ausgewertet, sondern bereits beim Build fest in den Context einkodiert. Weichen sie je nach Umgebung ab, müssen dementsprechend auch unterschiedliche native Images erzeugt werden, was Staging und CI/CD-Pipelines gleichermaßen verkompliziert.

Nativer Migrationspfad

Um bestehende Anwendungen zu nativen Images kompilieren zu können, sind einige Vorarbeiten zu leisten. Da *Spring 6* und *Spring Boot 3* gleichermaßen auf Java 17 ausgelegt sind, ist es sehr empfehlenswert, zunächst eine entsprechende Migration durchzuführen. Anschließend bietet es sich an, die Applikation einmal im AOT-Modus zu bauen, jedoch weiterhin regulär auf der JVM zu starten. Fehler bei der Generierung der optimierten Kontexte fallen so frühzeitig auf. Ist dieser Schritt erfolgreich absolviert, kann eine native Ausführung der Tests schnell Auskunft darüber geben, ob das Binary grundsätzlich funktionsfähig ist oder eventuell doch einzelne Klassen, Methoden, Felder oder Ressourcen versehentlich der Optimierung zum Opfer gefallen sind.

In einem solchen Fall ist es erforderlich, die generierten *GraalVM*-Metadaten händisch zu erweitern oder entsprechende *Runtime Hints* zu implementieren. Letztere geben dem Framework unter anderem explizite Hinweise darüber, welche zusätzlichen Klassen und Methoden über *Reflection* zur Verfügung stehen sollen, welche Ressourcen die Anwendung zu laden gedenkt und welche *Dynamic Proxys* oder Aufrufe über JNI gekapselt werden müssen.

Benchmarks

Inwiefern sich die theoretischen Vorteile nativer Images auch in der Praxis mit *Spring Boot 3* bemerkbar machen, soll im Rahmen mehrerer Benchmarks geklärt werden. Gegenstand der Betrachtung ist ein exemplarischer *RESTful Webservice* zur Verwaltung von Arbeitsauf-

Abbildung 2: GraalVM Native Build mit Spring Boot 3 (© Daniel Krämer)

Abbildung 3: GraalVM Native Build mit Quarkus (© Daniel Krämer)

trägen, der über eine relationale In-Memory-Datenbank (H2) verfügt und bei vergleichbarer Funktionalität sowohl mit *Spring Boot 3* als auch *Quarkus* implementiert ist. Möglich wird somit gleichermaßen der Vergleich eines nativen Images mit seinem Bytecode-Pendant und mit dem nativen Image eines vergleichbaren Frameworks. Zunächst erfolgt eine Gegenüberstellung des Builds von *Spring Boot 3* und *Quarkus (siehe Abbildung 2 und Abbildung 3)*.

Wie in den Abbildungen 2 und 3 zu erkennen ist, benötigt *Spring Boot* für einen Lauf rund 48 % mehr Zeit und etwa 8 % zusätzlichen Arbeitsspeicher. Bei der Startup-Time *(siehe Abbildungen 4, 5 und 6)* setzt sich dieses Bild im Duell der beiden Frameworks weitgehend fort.

Abbildung 4: Startup bei Spring Boot 3 (JVM) (© Daniel Krämer)

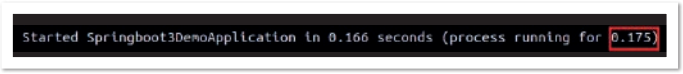


Abbildung 5: Startup bei Spring Boot (nativ) (© Daniel Krämer)

Abbildung 6: Startup bei Quarkus (nativ) (© Daniel Krämer)

Sehr viel deutlicher fallen allerdings die Unterschiede zu einem regulären Start auf der JVM aus. Im Speicherverbrauch *(siehe Abbildungen 7, 8 und 9)* zeigt sich eine ähnliche Abstufung.

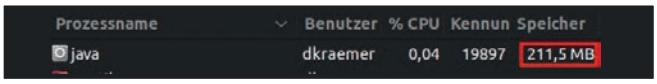


Abbildung 7: Speicherverbrauch bei Spring Boot (JVM) (© Daniel Krämer)

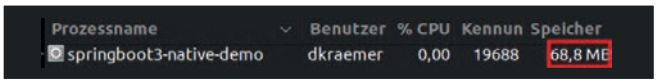


Abbildung 8: Speicherverbrauch bei Spring Boot (nativ) (© Daniel Krämer)

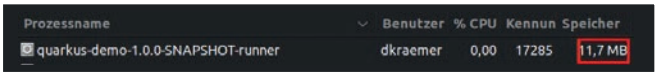


Abbildung 9: Speicherverbrauch bei Quarkus (nativ) (© Daniel Krämer)

Einordnung

Wie fast alle Technologien sind native Images (mit oder ohne *Spring Boot*) kein Allheilmittel und eignen sich für manche Szenarien besser oder schlechter als für andere. So können sie ihre Vorteile insbesondere dort ausspielen, wo Ressourcen wie CPU und Speicher begrenzt, beziehungsweise teuer sind. Sind eine hohe Anzahl von Instanzen oder häufige Startups vorgesehen, geht die Rechnung umso mehr auf. In der Praxis bietet sich ein Einsatz etwa bei Serverless-Architekturen und insbesondere FaaS an. Können einzelne Funktionen in größerem Umfang schnell hoch- und wieder heruntergefahren werden und benötigen sie darüber hinaus nicht mehr Speicher als unbedingt erforderlich, was sich positiv auf die Skalierbarkeit auswirkt. Letztere gehört generell zu den entscheidenden Vorzügen des Serverless-Architekturstils, ist aber oft auch ein gewünschtes Qualitätsmerkmal bei Microservices oder containerbasierten Lösungen im Allgemeinen.

Aufgrund des hohen Ressourcenverbrauches durch das Native-Image-Plug-in sind sie weniger für Projekte mit häufig ausgeführten Builds geeignet, etwa im Umfeld konsequent und durchgängig praktizierter *Continuous Integration*. Auch der notwendige Verzicht auf (dynamisch gesetzte) Profile und *Conditions* sowie Agenten und Profiler ist nicht in jedem Kontext gewünscht oder praktikabel. Veraltete oder hardwarenah (zum Beispiel mittels JNI) operierende Dependencies erweisen sich mitunter als weitere Stolpersteine.

Auf hohen Durchsatz ausgelegte Anwendungen können als nativer Maschinencode in der Praxis zudem eine schlechtere Performance zeigen als ihre Bytecode-Repräsentationen auf der JVM. Was in Anbetracht der Benchmarks zunächst widersprüchlich erscheinen mag, ist schlicht in der Abwesenheit eines Just-in-time-Compilers (JIT) und seiner ausgefeilten Optimierungen zur Laufzeit begründet. Blicken wir noch einmal zurück auf den eingangs zitierten Grundsatz „Build once, run anywhere!“: Wird eine Portierbarkeit jenseits von Containern angestrebt, sind auf spezifische Betriebssysteme und Prozessorarchitekturen zugeschnittene Binaries offensichtlich ebenfalls nicht die erste Wahl. Viele dieser Merkmale finden sich in der Praxis – insbesondere auch in gewachsenen oder stark monolithisch strukturierten Anwendungen. Ungeachtet der konkret vorliegenden Anforderungen ist eine Migration aufgrund des zu erwartenden Aufwandes in solchen Fällen meist weder praktikabel noch wirtschaftlich sinnvoll.

Cloud Native?

Zu klären wäre abschließend noch die Frage, ob *Spring Boot* neben *Quarkus* künftig guten Gewissens als zweites (beziehungsweise drittes) „Cloud Native Java Framework“ bezeichnet werden darf.

Obwohl bis heute keine einheitliche und allgemein verbindliche Definition von „Cloud Native“ existiert, lassen sich aus der Literatur (wie beispielsweise aus dieser Definition auf GitHub *[8]*) dennoch einige wiederkehrende Kriterien ableiten:

- Abgeschlossene Deployment-Units
- Ausrichtung auf den Betrieb in der Cloud
- Containerisierung & Orchestrierung
- Automatisierung
- Lose Kopplung
- Kommunikation über APIs
- Skalierbarkeit
- Resilienz
- Observability

Aspekte wie lose Kopplung, Kommunikation über APIs und Resilienz lassen sich nur begrenzt auf einzelne Application Frameworks anwenden, sondern stellen in erster Linie Qualitätsmerkmale der gewählten Software- beziehungsweise Systemarchitektur und ihrer Implementierung dar. Was die abgeschlossenen Deployment-Units anbelangt, so können monolithische Binaries ohne externe Laufzeitumgebung zweifellos als solche bezeichnet werden. Weil sich native Images vor allem auch für Serverless, FaaS und containerbasierte Lösungen empfehlen, kann ihnen eine gewisse Ausrichtung auf den Betrieb in der Cloud nicht abgesprochen werden.

Mithilfe des Spring-Boot-Maven-Plug-ins lässt sich der gesamte Build mit wenig Aufwand automatisieren und erzeugt auf Wunsch fertige Container – etwa zum Deployment auf ein *Kubernetes Cluster*. Kurze Startzeiten und ein moderater Speicherverbrauch spielen wiederum der Skalierbarkeit in die Karten. *Spring Boot Actuator* hat sich in der Praxis unlängst als gut integriertes (und integrierbares) Werkzeug zur Überwachung von Services bewährt. Zusammenfassend kann *Spring Boot* also spätestens seit der Unterstützung nativer Images als vollwertiges und ernst zu nehmendes Framework zur Realisierung von Cloud-Native-Apps angesehen werden.

Fazit

Mithilfe des Native-Image-Plug-ins ist es mit *GraalVM* schon seit einer Weile möglich, in Java geschriebene Anwendungen als native Binaries zu kompilieren, die sich insbesondere durch eine reduzierte Startzeit und einen geringeren Speicherverbrauch auszeichnen. *Spring 6* und *Spring Boot 3* bieten nun erstmals eine vollwertige und leicht zu verwendende Integration, sodass sich die Einsatzmöglichkeiten des Frameworks erneut erweitern.

Bei Serverless-Architekturen (insbesondere FaaS) und containerbasierten Lösungen schlägt Spring nun in eine Kerbe, die zuvor eher seiner Konkurrenz vorbehalten war. Gleichzeitig bringt der neue Ansatz aber auch Einschränkungen mit sich, etwa hinsichtlich der Verwendung von Profilen und Conditions, sodass eine Entscheidung dafür oder dagegen – wie bei jeder Technologie – vor dem Hintergrund individueller Anforderungen und Rahmenbedingungen abgewogen werden sollte. Abschließend lässt sich festhalten, dass der Kreis der „Cloud Native Java Frameworks" spätestens jetzt um einen interessanten Kandidaten reicher ist.

Quellen

[1] Christopher Tozzi (2023): Why 'Build Once, Run Anywhere' Is a False Promise – And Always Will Be (*https://www.itprotoday.com/software-development/why-build-once-run-anywhere-false-promise-and-always-will-be*)

[2] Oracle (2024): Native Image (*https://www.graalvm.org/native-image*)

[3] Micronaut Foundation (2024): Micronaut Framework (*https://docs.micronaut.io/latest/guide/*)

[4] Red Hat (2024): What is Quarkus? (*https://quarkus.io/about/*)

[5] VMware Tanzu (2024): GraalVM Native Image Support (*https://docs.spring.io/spring-boot/docs/current/reference/html/native-image.html*)

[6] Stéphane Nicoll, Brian Clozel (2022): Ahead Of Time and Native in Spring Boot 3.0 (*https://www.youtube.com/watch?v=TS4DpYSmfXk*)

[7] Oracle (2024): GraalVM Reachability Metadata Repository (*https://github.com/oracle/graalvm-reachability-metadata/blob/master/library-and-framework-list.json*)

[8] Cloud Native Computing Foundation (2018): CNCF Cloud Native Definition v1.0 (*https://github.com/cncf/toc/blob/main/DEFINITION.md*)

Daniel Krämer

daniel.kraemer@anderscore.com

Daniel Krämer ist als Senior Software Engineer für die anderScore GmbH tätig und begeistert sich für bedarfsgerechte Software-Architektur und sauberes Design. In seinen Kundenprojekten setzt er sich überwiegend mit individueller Java- und Web-Entwicklung sowie Fragestellungen rund um Migration und Integration auseinander. In Ergänzung zur Arbeit mit Code macht es Daniel auch Spaß, seine Kenntnisse und Erfahrungen mit anderen Entwicklern zu teilen. Zu den Inhalten regelmäßig abgehaltener, öffentlicher Trainings zählen Themen wie Software-Architektur (CPSA), Microservices und Spring.

CloudLand

WWW.CLOUDLAND.ORG

Eventpartner: Heise Medien

DAS CLOUD NATIVE FESTIVAL

DAS EVENT DER DEUTSCHSPRACHIGEN CLOUD NATIVE COMMUNITY

18. – 21. JUNI

#CLOUDLAND2024

Invoke Dynamic – ursprünglich gedacht für dynamisch typisierte JVM-Sprachen, ist dieser Bytecode-Befehl mittlerweile das Universalwerkzeug für viele Java-Neuerungen

Bernd Müller, Ostfalia

Der Bytecode-Befehl ***`invokedynamic`*** *wurde mit Java 7 als neuer Befehl in die JVM-Spezifikation aufgenommen. Der zugrundeliegende JSR beschreibt ein damaliges wachsendes Interesse an der Ausführung verschiedener, meist dynamisch typisierter Programmiersprachen auf der JVM. Diese Ausführung sollte effizienter als mit den zu dieser Zeit vorhandenen Maschinenbefehlen möglich sein. Der Befehl wurde im JDK für Java 7 tatsächlich selbst nicht verwendet, sondern war ein Angebot an andere Programmiersprachen, die die JVM als Zielplattform hatten. Mittlerweile wird* ***`invokedynamic`*** *aber zur Realisierung sehr vieler Java-Sprach-Features eingesetzt. Von Lambda-Ausdrücken über String-Konkatenationen bis hin zu Records wird mittlerweile* ***`invokedynamic`*** *verwendet. Dieser Artikel beleuchtet die ursprüngliche Motivation des Befehls, die Attraktivität für die heutige Verwendung sowie die nach unserem Wissen aktuell tatsächlichen Verwendungen.*

Was ist invokedynamic und welche Probleme werden damit gelöst?

Java ist eine streng typisierte Sprache. Für jeden Ausdruck ist daher bereits zur Compile-Zeit der Typ, zumindest der Obertyp, bekannt. Bei dynamisch oder schwach (*weak*) typisierten Sprachen ist dies nicht so. Erst zur Laufzeit ist der Typ eines Ausdrucks bekannt und führt damit erst während der Laufzeit zur Bestimmung und Verwendung der dem Typ entsprechenden Operationen. Bei der folgenden Python-Funktion zur Addition werden keine konkreten Typinformationen verwendet:

```
def add(a,b):
     return a + b;
```

Zur Laufzeit können mit dieser Funktion sowohl ganze als auch gebrochene Zahlen addiert werden. Bei einer Laufzeitimplementierung von *Python* durch die JVM muss daher zunächst der Typ der Parameter festgestellt werden, um den korrekten JVM-Befehl aufrufen zu können. Die JVM unterstützt lediglich die Addition von Integer-, Long-, Float- und Double-Werten mit den Maschinenbefehlen `iadd`, `ladd`, `fadd` und `dadd`, aber keinen ungetypten Additionsbefehl.

Da die JVM eine sehr stabile und performante Laufzeitumgebung ist, wird sie von vielen Sprachimplementierungen gerne genutzt. Der Wikipedia-Artikel *[1]* führt hierzu eine ganz Reihe bekannter Sprachen auf. Um die Popularität der Java-Plattform, insbesondere der JVM, weiter zu erhöhen, wurde bereits 2006 der JSR 292 *[2]* ins Leben gerufen, der die Unterstützung dynamisch typisierter Sprachen, insbesondere deren performante und qualitativ hochwertige Umsetzung auf der JVM, zum Ziel hatte. Dieser JSR wurde 2011 mit der Einführung des neuen JVM-Befehls `invokedynamic`, umgangssprachlich auch häufig *indy* genannt, mit dem Release von Java 7 finalisiert. Da `invokedynamic` für Nicht-Java-Sprachen gedacht war, wurde der Befehl vom Java-7-Compiler tatsächlich nicht bei der Erzeugung von Bytecode verwendet *[3]*.

Wie funktioniert nun aber `invokedynamic`? Um dies zu klären, ist ein Blick auf die bisherigen Alternativen für Aufrufe hilfreich. Es gab vier verschiedene Möglichkeiten, Methoden aufzurufen: `invokestatic` für den Aufruf statischer Methoden, `invokeinterface` für den Aufruf von Interface-Methoden, `invokespecial` für den Aufruf von Konstruktoren, von `super()`-Konstrukten und von private-Methoden, sowie `invokevirtual` für Objekt-Methoden.

Diese vier Aufrufarten haben eines gemeinsam: Sie sind sehr eng an die Semantik der Sprache Java gebunden und damit in ihrer Umsetzung sehr stringent festgelegt, so dass sie bereits zur Compile-Zeit vollumfänglich, ihrem jeweiligen Verwendungszweck entsprechend, erzeugt werden können. Sie erlauben somit keinerlei Dynamik zur Laufzeit.

Demgegenüber erlaubt `invokedynamic` ein Bootstrap-Verfahren, um zur Laufzeit beliebigen Bytecode aufrufen zu können. Dazu liefert der Aufruf der Bootstrap-Methode (kurz BSM) eine Instanz der Klasse `CallSite` (Package `java.lang.invoke`) zurück, die ein `MethodHandle` (ebenfalls Package `java.lang.invoke`) enthält, das letztendlich aufgerufen wird. Mit dieser zusätzlichen Indirektion wird die oben angesprochene Dynamik beziehungsweise Flexibilität gewonnen, da die entsprechenden Methoden in jedem neuen Java-Release überarbeitet und verbessert werden können. Diese Flexibilität wird jedoch mit einem gewissen Mehraufwand bezahlt. Dieser muss in der Regel jedoch nur beim ersten Aufruf erbracht werden. Das in der `CallSite` zurückgelieferte `MethodHandle` wird gecacht und beim zweiten sowie weiteren Aufrufen direkt und ohne das Bootstrap-Verfahren verwendet. Das hier etwas simplifiziert dargestellte Verfahren wird im *JavaDoc* des Package `java.lang.invoke` ausführlich dargestellt.

Wir gehen im Folgenden auf einige der Verwendungen des `invokedynamic`-Befehls ein. Da uns selbst Bytecode eher fremd ist, bleiben wir auf der Ebene der Java-Bezeichner beziehungsweise generierter Bezeichner. Der geneigte Leser kann sich aber mit dem `javap`-Befehl und der Option `-v` beziehungsweise `-verbose` recht einfach davon überzeugen, dass die genannten Java-Methoden im Bytecode tatsächlich verwendet werden.

Lambda-Ausdrücke

Wir beginnen mit Lambda-Ausdrücken, die mit Java 8 eingeführt wurden und damit das erste Java-Sprachkonzept waren, das mithilfe des in Java 7 eingeführten `invokedynamic`-Befehls umgesetzt wurde. Ben Evans und Martijn Verburg erwähnen in *The Well-Grounded Java Developer [3]*, dass eine erste prototypische Implementierung von Lambda-Ausdrücken in Java 8 durch anonyme Klassen erfolgt sei, was recht nahe liegt. Diese Implementierungsalternative hatte aber eine ganz Reihe negativer Eigenschaften. So werden eine nicht unerhebliche Anzahl zusätzlicher Klassen erzeugt, die jeweils in die JVM geladen, validiert und gelinkt werden müssen. Zudem steht die Implementierung fest, da der Bytecode erzeugt und unveränderlich ist. Die beschriebene Alternative über ein Bootstrap-Verfahren erlaubt es jedoch einer späteren Implementierung, ein effizienteres Verfahren im Bootstrap-Mechanismus und der Call-Site einzusetzen und so die Performanz zu erhöhen.

Brian Goetz selbst beschreibt in *Translation of Lambda Expressions [4]* das Verfahren, um Lambda-Ausdrücke in Bytecode zu übersetzen. Die verwendete Bootstrap-Methode ist die Methode `metafactory()` der Klasse `LambdaMetafactory` im Package `java.lang.invoke`. Die ersten Zeilen des *JavaDoc* der Klasse erläutern den Einsatzzweck:

> *„Methods to facilitate the creation of simple function objects that implement one or more interfaces by delegation to a provided MethodHandle, possibly after type adaptation and partial evaluation of arguments. These methods are typically used as bootstrap methods for invokedynamic call sites, to support the lambda expression and method reference expression features of the Java Programming Language."*

Neben Lambda-Ausdrücken werden auch Methodenreferenzen mit diesem Ansatz übersetzt. Der Rückgabetyp der Methode `metafactory()` ist `CallSite`. Diese Datenstruktur einhält ein `MethodHandle`, das durch den `invokedynamic`-Befehl aufgerufen wird. Als Ziel des Method-Handles werden Methoden nach den Namensschemata `lambda$classname$0`, `lambda$classname$1` und weiteren erzeugt, die die im Quellcode verwendeten Lambda-Aus-

drücke repräsentieren. Auch hiervon kann man sich leicht mit dem `javap`-Befehl überzeugen.

String-Konkatenation

Scheinbar waren die Erfahrungen bei der Verwendung von `invokedynamic` mit Lambda-Ausdrücken in Java 8 so positiv, dass mit Java 9 ein weiteres Java-Sprach-Feature damit realisiert wurde: die Konkatenation von Strings. Wir sind auf den zugrundeliegenden JEP 280 bereits in der Ausgabe 03/2023 der *Java aktuell* [5] kurz eingegangen. Der JEP 280 [6] trägt den Titel *Indify String Concatenation* und definiert sich in der Zusammenfassung als:

> *„Change the static String-concatenation bytecode sequence generated by javac to use invokedynamic calls to JDK library functions. This will enable future optimizations of String concatenation without requiring further changes to the bytecode emitted by javac."*

Während die Konkatenation von String-Literalen mit dem Plus-Operator bereits zur Compile-Zeit erfolgt, wurden bis Java 9 String-Konkatenationen mit Variablenbeteiligung durch `StringBuilder#append()`-Aufrufe im Bytecode realisiert. Der JEP 280 hatte das Ziel, die Konkatenation von Strings in späteren Versionen der JVM performanter durchführen zu können, als mit direkt im Bytecode festgehaltenen `append()`-Aufrufen. Die Indirektion von `invokedynamic` über eine Bootstrap-Methode, die in einer späteren JVM eventuell optimiert wurde, macht dies möglich.

Die tatsächlich verwendete Bootstrap-Methode ist `StringConcatFactory#makeConcatWithConstants()`. Die Klasse befindet sich ebenfalls im Package `java.lang.invoke` und die ersten Zeilen des *JavaDocs* lauten:

> *„Methods to facilitate the creation of String concatenation methods, that can be used to efficiently concatenate a known number of arguments of known types, possibly after type adaptation and partial evaluation of arguments. These methods are typically used as bootstrap methods for invokedynamic call sites, to support the string concatenation feature of the Java Programming Language."*

Im Gegensatz zu Lambda-Ausrücken, die als neu eingeführtes Sprachkonstrukt mit `invokedynamic` realisiert wurden, wird für die String-Konkatenation erst ab Java 9 `invokedynamic` verwendet. Der Leser kann sich daher mit `javac` und `javap` recht einfach davon überzeugen, dass der erzeugte Bytecode, zum Beispiel für Java 8 und 9, sich in diesem Punkt unterscheidet.

Records

Records wurden im JEP 359 als Preview in Java 14 eingeführt. Nach einem zweiten Preview mit JEP 384 in Java 15 wurden sie schließlich mit JEP 395 in Java 16 endgültig als Feature in die Sprache Java aufgenommen. Die Implementierung der Record-Methoden `equals()`, `hashCode()` und `toString()` erfolgte von Anbeginn an, also bereits mit Java 14, über `invokedynamic`. Die entsprechende Bootstrap-Methode ist `ObjectMethods#bootstrap()` im Package `java.lang.runtime` und dem folgenden *JavaDoc*:

Falls der Leser in das *JavaDoc* schaut, wird er feststellen, dass als erstmalige Version der Klasse 16 angegeben ist („since: 16"). Dies ist kein Widerspruch zum oben genannten JEP 359, der bereits mit Java 14 eingeführt wurde. Alle Klassen eines Previews werden im *JavaDoc* mit der Version des ersten Previews geführt. Wenn eine Klasse dann final in die Sprache Java aufgenommen wird, wird die Versionsnummer auf die finale Java-Version aktualisiert.

Pattern Matching for switch

Das Pattern-Matching für Switch-Ausrücke und -Anweisungen wurde mit JEP 406 als Preview in Java 17 eingeführt. Die JEPs 420, 427 und 433 wurden ebenfalls als jeweils nachfolgender Preview des Sprach-Features für die Java-Versionen 18, 19 und 20 definiert. Mit JEP 441 und Java 21 wurde diese Art des Pattern-Matching schließlich endgültig in die Sprache aufgenommen.

Die entsprechende Bootstrap-Methode ist `SwitchBootstrap#typeSwitch()` im Package `java.lang.runtime`. Der erste Satz des *JavaDoc* lautet:

> *„Bootstrap method for linking an invokedynamic call site that implements a switch on a target of a reference type."*

Die Klasse `SwitchBootstrap` enthält noch eine zweite Methode, nämlich `enumSwitch()`. Laut *JavaDoc* ist sie die Bootstrap-Methode, wenn der Switch-Ausdruck aus Werten eines Aufzählungstyps besteht. Wir konnten dies mit den Java-Versionen 17 bis 21 jedoch nicht nachstellen und können es somit hier nicht bestätigen.

String-Templates

String-Templates wurden mit JEP 430 [7] mit Java 21 als Preview-Feature eingeführt. Sie erlauben durch Template-Prozessoren die Auswertung von syntaktisch speziell geformten Ausdrücken innerhalb von Strings und Textblöcken. Auch hier wurde bei der Implementierung entschieden, diese auf `invokedynamic` zu basieren. Die drei Bootstrap-Methoden `newStringTemplate()`, `newLargeStringTemplate()` und `processStringTemplate()` sind in der Klasse `TemplateRuntime` im Package `java.lang.runtime` enthalten. Die ersten beiden Sätze des *JavaDoc* der Klasse lesen sich wie folgt:

> „Manages string template bootstrap methods. These methods may be used, for example, by Java compiler implementations to create StringTemplate instances."

Invoke Dynamic reloaded

Die Einführung von `invokedynamic` als neuer Bytecode-Befehl war, wie wir am zunächst ungeplanten, jetzt aber breiten Einsatzspektrum gesehen haben, ein Erfolg. Das Konzept, die Übersetzung eines Methodenaufrufs durch eine zusätzliche Indirektion zu flexibilisieren, scheint derart attraktiv, dass es auch für Java-Konstanten übernommen wurde. Der JEP 309, *Dynamic Class-File Constants* [8] hat genau dies zum Ziel und wurde mit Java 11 umgesetzt. Die Zusammenfassung lautet folgt:

„Extend the Java class-file format to support a new constant-pool form, CONSTANT_Dynamic. Loading a CONSTANT_Dynamic will delegate creation to a bootstrap method, just as linking an invokedynamic call site delegates linkage to a bootstrap method."

Hintergrund ist, dass der Bytecode für Konstanten nur einen kleinen Bereich und eingeschränkte Funktionalität vorsieht. Durch die Einführung der Klasse `LambdaMetafactory` verwenden deren Bootstrap-Methoden Parameter, die häufig Konstanten sind, die mit den herkömmlichen Konstrukten aufwendig zu realisieren sind. Mit dem JEP sollen diese Art von Konstanten flexibler erzeugt werden können.

Für die Umsetzung wurde die Klasse `ConstantBootstrap` im Package `java.lang.invoke` geschaffen, deren *JavaDoc*

„Bootstrap methods for dynamically-computed constants."

diese Flexibilität zum Ausdruck bring.

Zusammenfassung

Mit Java 7 wurde der neue Bytecode-Befehl `invokedynamic` eingeführt, um dynamische Sprachen, die auf der JVM ausgeführt werden sollen, einfacher implementieren zu können. Die zunächst nicht intendierte Verwendung zur Implementierung von Java-Sprach-Features erfolgte erstmalig für die mit Java 8 vorgestellten Lambda-Ausdrücke. In späteren Versionen wurden noch weitere neue Sprach-Features, aber etwa auch die String-Konkatenation, mithilfe von `invokedynamic` realisiert. Das Konzept hinter `invokedynamic` sieht vor, dass der aufzurufende Code nicht direkt in den Bytecode übersetzt, sondern über eine sogenannte Bootstrap-Methode die aufzurufende Call-Site zurückgeliefert wird, die dann aufgerufen wird. Diese zusätzliche Indirektion erlaubt in einer späteren Java-Version eine effizientere Ausführung, da der Bibliotheks-Code dieser verwendeten Java-Version eine bessere Umsetzung bereitstellen könnte , als die Java-Version, mit der der Code übersetzt wurde.

Es wurden bisher, Stand Java 21, die folgenden Klassen mit entsprechenden Bootstrap-Methoden implementiert:

- `java.lang.invoke.LambdaMetafactory`
- `java.lang.invoke.StringConcatFactory`
- `java.lang.runtime.ObjectMethods`
- `java.lang.runtime.SwitchBootstrap`
- `java.lang.runtime.TemplateRuntime`
- `java.lang.invoke.ConstantBootstrap`

Alle genannten Klassen und deren Methoden sind öffentlich, können also von Anwendungsentwicklern verwendet werden. Es gibt allerdings keine sinnvollen Verwendungsmöglichkeiten hierfür, da die Klassen einzig und allein als Werkzeugunterstützung für den Compiler entwickelt wurden.

Referenzen

[1] List of JVM languages. *https://en.wikipedia.org/wiki/List_of_JVM_languages*

[2] JSR 292: Supporting Dynamically Typed Languages on the Java Platform, *https://jcp.org/en/jsr/detail?id=292*

[3] Benjamin J. Evans, Martijn Verburg. The Well-Grounded Java Developer, Manning 2013

[4] Brian Goetz. Translation of Lambda Expressions, *https://bit.ly/3sLQxnw*

[5] Bernd Müller. Was jeder Java-Entwickler über Strings wissen sollte. Java aktuell 03/2023

[6] JEP 280: Indify String Concatenation, *https://openjdk.org/jeps/280*

[7] JEP 430: String Templates (Preview), *https://openjdk.org/jeps/430*

Bernd Müller
Ostfalia
bernd.mueller@ostfalia.de

Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

Java aktuell

JAHRESABO

FÜR 29,00 €
BESTELLEN

iJUG
Verbund
www.ijug.eu

Mehr Informationen zum Magazin und Abo unter:

www.ijug.eu/de/java-aktuell

Test First?

Marco Schulz

Testgetrieben Entwicklung klingt ja ganz vernünftig. Zumal anschließend auch voll automatisierte Tests vorhanden sind. Aber muss man wirklich erst mit den Testfällen beginnen, um dann die zugehörige Implementierung zu schreiben? Ich finde, es geht auch anders, denn viele Wege führen nach Rom.

Als ich vor über zehn Jahren begonnen habe, testgetrieben zu programmieren, waren mir sehr viele verschiedene Konzepte theoretisch bekannt. Doch das Vorgehen, erst Testfälle zu schreiben und dann die Implementierung umzusetzen, war nicht der Weg, mit dem ich gut zurechtgekommen bin. Wenn ich ehrlich bin, ist das bis heute der Fall. Daher habe ich eine für mich funktionierende Adaption des TDD-Paradigmas (Test Driven Development) von Kent Beck *[1]* gefunden. Aber immer langsam und der Reihe nach. Vielleicht ist mein Ansatz für den ein oder anderen ebenfalls recht hilfreich.

Ich komme ursprünglich aus einem Umfeld für hoch skalierbare Webanwendungen, auf das sich all die tollen Theorien aus dem universitären Umfeld in der Praxis nicht ohne Weiteres umsetzen lassen. Der Grund liegt vor allem in der hohen Komplexität solcher Anwendungen. Zum einen sind verschiedene Zusatzsysteme wie In-Memory-Cache, Datenbank sowie Identitäts- und Zugriffs-Management (IAM) ein Teil des Gesamtsystems. Zum anderen verstecken viele moderne Frameworks wie OR-Mapper Komplexität hinter verschiedene Zugriffsschichten. All diese Dinge müssen wir als Entwickler heutzutage beherrschen. Deshalb gibt es robuste, praxiserprobte Lösungen, die zwar bekannt sind, aber wenig Verwendung finden. Kent Beck ist eine der wichtigsten Stimmen für den praktischen Einsatz automatisierter Softwaretests.

Wenn wir uns auf das Konzept TDD einlassen wollen, ist es wichtig, nicht jedes Wort zu sehr auf die Goldwaage zu legen. Es ist nicht alles in Stein gemeißelt. Wichtig ist am Ende des Tages das Ergebnis. Aus diesem Grund ist es unabdinglich, sich die Zielvorgabe aller Bemühungen vor Augen zu führen, um dann einen persönlichen Mehrwert zu erzielen. Also schauen wir uns zuerst einmal an, was wir überhaupt erreichen wollen.

Der Erfolg gibt uns Recht

Als ich meine ersten Gehversuche als Entwickler unternommen hatte, benötigte ich stetiges Feedback, ob das, was ich da gerade zusammenbaute, auch wirklich funktioniert. Dieses Feedback habe ich meist dadurch erzeugt, indem ich meine Implementierung einerseits mit unzähligen Konsolenausgaben gespickt habe und andererseits immer versucht habe, alles in eine Benutzeroberfläche einzubinden, um mich dann „manuell durchzuklicken". Im Grunde ein sehr umständliches Test-Setup, das am Schluss wieder zu entfernen ist. Wenn dann noch spätere Bugfixes vorgenommen werden mussten, ging das ganze Prozedere wieder von Neuem los. Dies war sehr unbefriedigend und von einer produktiven Arbeitsweise weit entfernt. Irgendwie musste das verbessert werden, ohne stetiges Neuerfinden.

Schließlich hat mein ursprünglicher Ansatz genau zwei markante Schwachstellen. Die offensichtlichste ist das Ein- und Auskommentieren von Debug-Informationen über die Konsole.

Viel schwerwiegender ist allerdings der zweite Punkt. Denn all das erworbene Wissen zu dieser speziellen Implementierung ist nicht konserviert. Es droht also, über die Zeit zu verblassen und schlussendlich auch verloren zu gehen. Ein solches Spezialwissen ist für viele nachfolgende Prozessschritte in der Softwareentwicklung aber äußerst wertvoll. Damit meine ich explizit das Thema Qualität. Refactoring, Code Reviews, Bug Fixes und Change Requests sind nur einige der möglichen Beispiele, Bei denen tiefgreifendes Detailwissen gefragt ist.

Für mich persönlich kommt auch hinzu, dass mich monoton wiederholbare Arbeiten schnell ermüden und ich diese deshalb sehr gern vermeiden möchte. Sich immer wieder aufs Neue mit der gleichen Testprozedur durch eine Anwendung zu klicken, ist weit von dem entfernt, was für mich einen erfüllten Arbeitstag ausmacht. Ich möchte neue Dinge entdecken. Das kann ich aber nur, wenn ich nicht in der Vergangenheit gefangen gehalten werde.

Die trauen sich aber was

Bevor ich aber darauf eingehe, wie ich meinen Entwicklungsalltag durch TDD aufgepeppt habe, muss ich noch ein paar Worte über Verantwortung und Mut loswerden. Immer wieder wird mir in Gesprächen erklärt, dass ich ja recht hätte, aber man das alles ja nicht selbst umsetzen könne, weil der Projektleiter oder ein anderer Vorgesetzter kein grünes Licht gebe.

Eine solche Einstellung ist in meinen Augen äußerst unprofessionell. Ich frage doch auch nicht den Marketingleiter, welcher Algorithmus am besten terminiert sei. Er hat schlichtweg keine Ahnung, denn es ist auch nicht sein Aufgabengebiet. Ein Projektleiter, der sich gegen das testgetriebene Arbeiten im Entwicklungsteam ausspricht, hat seinen Beruf verfehlt. In der heutigen Zeit sind Testframeworks so gut in die Build-Umgebung integriert, dass die Vorbereitung für TDD selbst für unerfahrene Personen in wenigen Augenblicken sich umsetzen lässt. Es ist also nicht notwendig, das Vorhaben an die große Glocke zu hängen. Denn ich kann versprechen, dass selbst bei den ersten Gehversuchen nicht mehr Zeit benötigt wird, als mit der ursprünglichen Vorgehensweise. Ganz im Gegenteil, sehr schnell wird sich eine merkliche Erhöhung der Produktivität einstellen.

Die erste Stufe der Evolution

Wie bereits erwähnt, ist *Logging* für mich ein zentraler Teil der testgetriebenen Entwicklung. Wann immer es sinnvoll erscheint, versuche ich den Zustand von Objekten oder Variablen auf der Konsole auszugeben. Wenn wir hierfür die aus der verwendeten Programmiersprache zur Verfügung gestellten Mittel nutzen, würde dies bedeuten, dass wir diese Systemausgaben nach getaner Arbeit mindestens auskommentieren müssen und bei späterer Fehlersuche wieder einkommentieren. Ein redundantes und fehleranfälliges Vorgehen.

Nutzen wir hingegen von Beginn an ein Logging-Framework, so können wir die Debug-Informationen im Code stehen lassen und deaktivieren diese später im Produktivbetrieb über den eingestellten Log-Level.

Ich nutze *Logging* aber auch als *Tracer*. Das heißt, jeder Konstruktor einer Klasse schreibt einen entsprechenden Log-Eintrag im Log-Level-Info, während er aufgerufen wird. Damit kann man sehen, in welcher Reihenfolge Objekte instanziiert werden. Hin und wieder bin ich so auch auf die übermäßige Instanziierung eines einzelnen Objektes aufmerksam geworden. Dies ist für Maßnahmen zur Performance und Speicheroptimierung hilfreich.

Fehler, die bei der Ausnahmebehandlung geworfen werden, logge ich je nach Kontext als *Error* oder *Warning*. Das ist später im Betrieb ein sehr hilfreiches Mittel, um Fehlern auf die Spur zu kommen.

Wenn ich also einen Datenbankzugriff habe, schreibe ich eine Log-Ausgabe im Log-Level-Debug, wie das zugehörige SQL zusammen-

gebaut wurde. Führt dieses SQL zu einer *Exception*, weil ein Fehler enthalten ist, so wird diese *Exception* mit dem Log-Level-Error geschrieben. Findet wiederum eine einfache Suchanfrage mit korrekter SQL-Syntax statt und die Ergebnismenge ist leer, wird dieses Ereignis je nach Bedarf entweder als *Debug* oder *Warning* klassifiziert. Handelt es sich beispielsweise um eine Login-Anfrage mit falschem Benutzernamen oder Passwort, neige ich dazu, mich für den Log-Level *Warning* zu entscheiden, da dies im Betrieb eventuell sicherheitstechnische Aspekte enthält.

Im gesamten Kontext konfiguriere ich das Logging für die Testfallausführung eher sehr geschwätzig und beschränke mich auf eine reine Konsolenausgabe. Im Betrieb wiederum werden die Logging-Informationen in ein Logfile geschrieben.

Die Henne oder das Ei

Wenn wir mit dem *Logging* die Voraussetzung für eine zusätzliche Feedbackschleife geschaffen haben, stellt sich im nächsten Schritt die Frage, wie es weitergehen soll. Wie bereits erwähnt, fällt es mir sehr schwer, erst einen Testfall zu schreiben, um dann eine entsprechende Implementierung dafür zu finden. Vor diesem Problem stehen auch viele andere Entwickler, die mit TDD beginnen.

Eine Sache, die ich bereits vorwegnehmen kann, ist das Problem, das man bei einer Implementierung darauf achten muss, diese auch testbar zu halten. Habe ich erst den Testfall, so merke ich umgehend, ob das, was ich gerade erstelle, auch wirklich testbar ist. Erfahrene TDD-Entwickler haben recht schnell in Fleisch und Blut übernommen, wie testbarer Code aussieht. Der wichtigste Punkt hierbei ist, dass Methoden stets einen Rückgabewert haben sollten, der möglichst nicht null ist. So etwas erreicht man beispielsweise, wenn man anstatt null eine leere Liste zurückgibt.

Die Vorgabe, einen Rückgabewert zu haben, liegt an der Art und Weise, wie Unittest-Frameworks arbeiten. Ein Testfall vergleicht den Rückgabewert einer Methode mit einem Erwartungswert. Die Testzusicherung (engl. *Assertation*) kennt verschiedene Ausprägungen und kann entsprechend gleich, *ungleich*, *wahr* oder *falsch* sein. Natürlich gibt es hier auch verschieden Variationen. So ist es durch die Verwendung von *Exceptions* möglich, Methoden zu testen, die keinen Rückgabewert haben. Alle diese Details erschließen sich bei der Anwendung in sehr kurzer Zeit, sodass jeder ohne langwierige Vorbereitungen umgehend loslegen kann.

Bei der Lektüre des Buches *Test-Driven-Development by Example* von Kent Beck *[1]* finden wir auch schnell eine Erklärung, warum die Testfälle zuerst geschrieben werden sollten. Es handle sich hierbei um einen psychologischen Faktor: Denn es helfe uns dabei, den üblichen Stress, der im Projekt entstehe, besser zu bewältigen. Es erzeugt in uns einen mentalen Zustand über den Zustand und Fortschritt der aktuellen Arbeit. So erhalte man Zugang zu einem iterativen Prozess, in dem man vorhandene Lösung schrittweise über die verschiedenen Testfälle weiter ausbauen und verbessern könne.

Für alle, die wie ich aber zu Beginn einer Implementierung noch keine konkrete Vorstellung über das fertige Ergebnis haben, ist dieser Ansatz schwer umzusetzen. Der bezweckte Effekt der Entspannung kehrt sich ins Negative um. Da wir Menschen alle unterschiedlich sind, müssen wir also herausfinden, wie wir funktionieren, um unser Potential bestmöglich ausschöpfen zu können. Ganz so, wie es mit Lernstrategien ist. Manche Menschen verarbeiten Informationen besser visuell, andere eher haptisch und wieder andere extrahieren alles Wichtige aus Gesprochenem. Versuchen wir uns also nicht zu verbiegen, da wir so vermutlich nur mittelmäßige oder schlechte Ergebnis produzieren.

Den ersten Strich zeichnen

Mir erschließt sich ein Thema eben erst, während ich damit arbeite. Also versuche ich mich so lange an einer Implementierung, bis ich ein erstes Feedback benötige. Genau dann schreibe ich den ersten Test. Es ergebenen sich bei diesem Vorgehen automatisch Fragen, bei der jede einzelne einen eigenen Testfall erfordert. Finde ich alle vorhanden Ergebnisse? Was passiert, wenn die Ergebnismenge leer ist? Wie lässt sich die Ergebnismenge eingrenzen? Alles Punkte, die sich auf einem Zettel notieren und Schritt für Schritt abhaken lassen. Die Idee, eine Aufgabenliste auf einem Zettel zu notieren, hatte ich schon sehr lange bevor ich das bereits erwähnte Buch von Kent Beck *[1]* gelesen habe. Es hilft mir, meine Gedanken zu konservieren, sodass ich nicht meinen eigentlichen Fokus verliere. Außerdem vermittelt es am Ende des Tages ein Gefühl, etwas geschafft zu haben.

Da ich nicht warte, bis ich alles umgesetzt habe, um den ersten Test zu schreiben, ergibt sich auch bei diesem Vorgehen ein iterativer Ansatz. Ich merke auch sehr schnell, wenn mein Entwurf nur unzureichend testbar ist, da ich sofort eine Rückmeldung erhalte. Daraus ergibt sich meine eigene Interpretation für TDD, die sich durch den permanenten Wechsel zwischen Implementieren und Tests schreiben auszeichnet.

Als Ergebnis meiner frühen TDD-Versuche habe ich bereits in der ersten Woche eine Beschleunigung meiner Arbeitsweise bemerkt. Ebenfalls bin ich sicherer geworden. Aber auch die Art und Weise, wie ich programmiere, hat sich schon sehr zeitig zu verändern begonnen. Mir ist aufgefallen, dass mein Code kompakter und robuster geworden ist. Dinge, die sich erst mit der Zeit aufgezeigt hatten, ergaben sich bei Tätigkeiten wie Refactoring und Erweiterungen. Fehlgeschlagene Testfälle haben mich vor bösen Überraschungen bewahrt.

Ohne Übereifer beginnen

Wenn wir uns in einem bestehenden Projekt dazu entschließen, TDD einzusetzen, ist es eine schlechte Idee, loszulegen und für bestehende Funktionalitäten Testfälle zu schreiben. Abgesehen von der Zeit, die hierfür eingeplant werden muss, wird das Ergebnis die hohen Erwartungen nicht erfüllen.

Eines der Probleme ist, dass man sich nun in jede Funktionalität neu einarbeiten muss und das ist sehr zeitaufwendig. Die Qualität der so entstandenen Testfälle ist auch unzureichend. Ein weiteres Problem ergibt sich auch aus der Erfahrung. Wird die Erfahrung erst aufgebaut, so ist die Qualität der Testfälle noch nicht ganz optimal und möglicherweise muss auch Code umgeschrieben werden, damit dieser testbar wird. Es entstehen also eine Menge Risiken für das tägliche Projektgeschäft.

Ein bewährtes Vorgehen TDD einzuführen, ist es, TDD einfach für die aktuelle Implementierung einzusetzen, an der gerade gearbeitet wird. Es wird also der Ist-Zustand des aktuellen Problems

durch automatisierte Tests dokumentiert. Da man sich bereits auf vertrautem Terrain befindet, muss man sich nicht erst in eine neue Thematik einarbeiten, sodass man sich voll auf das Formulieren aussagekräftiger Tests konzentrieren kann. Abgesehen davon, dass man ungefragt Verantwortung über fremde Arbeiten übernimmt, wenn man für diese fremde Arbeit Testfälle schreibt.

Bestehende Funktionalität wird nur bei Fehlerkorrekturen entsprechend um Testfälle ergänzt. Für die Korrektur muss man sich ohnehin mit den Implementierungsdetails auseinandersetzen, da hier ein gewisses Maß Wissen notwendig ist, um zu verstehen, wie eine Funktionalität sich verhalten sollte. Die so entstandenen Tests dokumentieren zusätzlich auch die Korrektur und stellen für die Zukunft sicher, dass sich das Verhalten bei Optimierungsarbeiten nicht verändert.

Folgt man dieser Vorgehensweise diszipliniert, verliert man sich nicht in sogenannter hektischer Betriebsamkeit, die das Gegenteil von Produktivität ist. Zudem erwirbt man so recht schnell fundiertes Wissen, wie effektive und aussagekräftige Tests umgesetzt werden können. Erst, wenn ausreichend Erfahrung gesammelt wurde und möglicherweise umfangreiche *Refactorings* geplant worden sind, dann kann man überlegen, wie die Testabdeckung für das gesamte Projekt schrittweise verbessert werden kann.

Qualitätsstufe

Nur weil Testfälle vorhanden sind, bedeutet dies nicht, dass diese auch eine Aussagekraft haben. Genau so wenig beweist eine hohe Testabdeckung, dass ein Programm fehlerfrei ist. Eine hohe Testabdeckung stellt nur sicher, dass sich ein Programm im Rahmen der Tests fehlerfrei verhält.

Wie kann man also sicherstellen, dass die vorhandenen Tests auch wirklich eine Bereicherung sind und eine gute Aussagekraft haben? Der erste und meines Erachtens wichtigste Punkt ist, Testfälle möglichst kurz zu halten. Das heißt im Konkreten, dass ein Test nur eine explizite Fragestellung beantwortet, zum Beispiel: „Was passiert, wenn die Ergebnismenge leer ist?“ Entsprechend der Fragestellung ergibt sich dann auch die Benennung der Testmethode. Der Mehrwert dieser Vorgehensweise ergibt sich in dem Moment, wenn der Testfall fehlschlägt. Ist der Test sehr kurzgefasst, lässt sich oft schon anhand der Testmethode ablesen, worin das Problem besteht, ohne sich erst langwierig in einen Testfall einarbeiten zu müssen.

Ein anderer wichtiger Punkt im TDD-Vorgehen für meine umgesetzte Funktionalität ist es, sowohl die Testabdeckung für Codezeilen als auch die Testabdeckung für Verzweigungen zu überprüfen. Kann ich zum Beispiel in einer IF-Abfrage das Eintreten einer einzelnen Bedingung nicht simulieren, so kann diese Bedingung bedenkenlos gestrichen werden.

Natürlich hat man im eigenen Projekt auch genügend Abhängigkeiten zu fremden Bibliotheken. Nun kann es vorkommen, dass eine Methode aus dieser Bibliothek eine Ausnahme wirft, die durch keinen Testfall simuliert werden kann. Das ist genau der Grund, wieso man zwar eine hohe Testabdeckung anstreben sollte, aber nicht verzweifeln muss, wenn 100 % nicht erreicht werden können. Gerade bei der Einführung von TDD ist eine Testabdeckung von größer als 85 % üblich. Mit wachsender Erfahrung des Entwicklungsteams kann dieser Wert auf bis zu 95 % angehoben werden.

Abschließend ist aber noch anzumerken, dass man sich nicht zu sehr hineinsteigern sollte. Denn es kann auch schnell übertrieben werden und dann sind die ganzen gewonnene Vorteile wieder dahin. Dabei geht es um den Punkt, dass man keine Tests schreibt, die wiederum Tests testen. Hier beißt sich die Katze in den Schwanz. Das gilt auch für Bibliotheken von Fremdanbietern. Für diese werden ebenfalls keine Tests geschrieben. Kent Beck *[1]* äußert sich hierzu sehr klar: Selbst, wenn es gute Gründe gebe, dem Code anderer zu misstrauen, solle man ihn nicht testen. Externer Code erfordere mehr eigene Implementierungslogik.

Lessons Learned

Gerade die Erkenntnisse, die sich bei dem Versuch, eine möglichst hohe Testabdeckung zu erzielen, bilden, wirken sich auf das künftige Programmieren aus. Der Code wird kompakter und robuster. Die Produktivität steigt einfach durch die Tatsache, dass fehleranfällige und monotone Arbeiten durch Automatisierung vermieden werden. Es entstehen keine zusätzlichen Arbeitsschritte, denn alte Gewohnheiten werden durch neue, bessere ersetzt.

Ein Effekt, den ich immer wieder beobachten konnte, wenn sich einzelne Personen aus dem Team für TDD entschieden haben, war, dass deren Erfolge schnell bemerkt wurden. Innerhalb weniger Wochen hat dann das gesamte Team TDD entwickelt. Jeder Einzelne nach seinen eigenen Fähigkeiten. Manche mit *Test First*, andere wiederum so, wie ich es gerade beschrieben habe. Zum Schluss zählt das Ergebnis und das war einheitlich hervorragend. Wenn die Arbeit leichter fällt und am Ende des Tages jeder Einzelne auch noch das Gefühl hat, etwas geleistet zu haben, bewirkt dies im Team einen enormen Motivationsschub, der dem Projekt und dem Arbeitsklima einen gewaltigen Auftrieb verschafft. Also worauf warten Sie noch? Probieren Sie es am besten gleich selbst aus.

Quellen/Referenzen

[1] Kent Beck, Test Driven Development: By Example, ISBN 978-0-321-14653-3

Marco Schulz

Marco Schulz, im Internet auch unter seinem Pseudonym Elmar Dott bekannt, realisiert seit über knapp 20 Jahren als freier Berater in internationalen Projekten große Web-Applikationen. Seine Schwerpunkte sind DevOps, Konfigurationsmanagement, Software-Architekturen & Release Management. Als Trainer teilt er sein Wissen in Schulungen und spricht auch regelmäßig auf Konferenzen über aktuelle Themen. Seine Homepage finden Sie unter: https://elmar-dott.com

DEINE VORTEILE

25 % Rabatt
auf JavaLand-Tickets

Java aktuell
Jahres-Abonnement

Java Community Process
Mitgliedschaft

JETZT MITGLIED WERDEN!

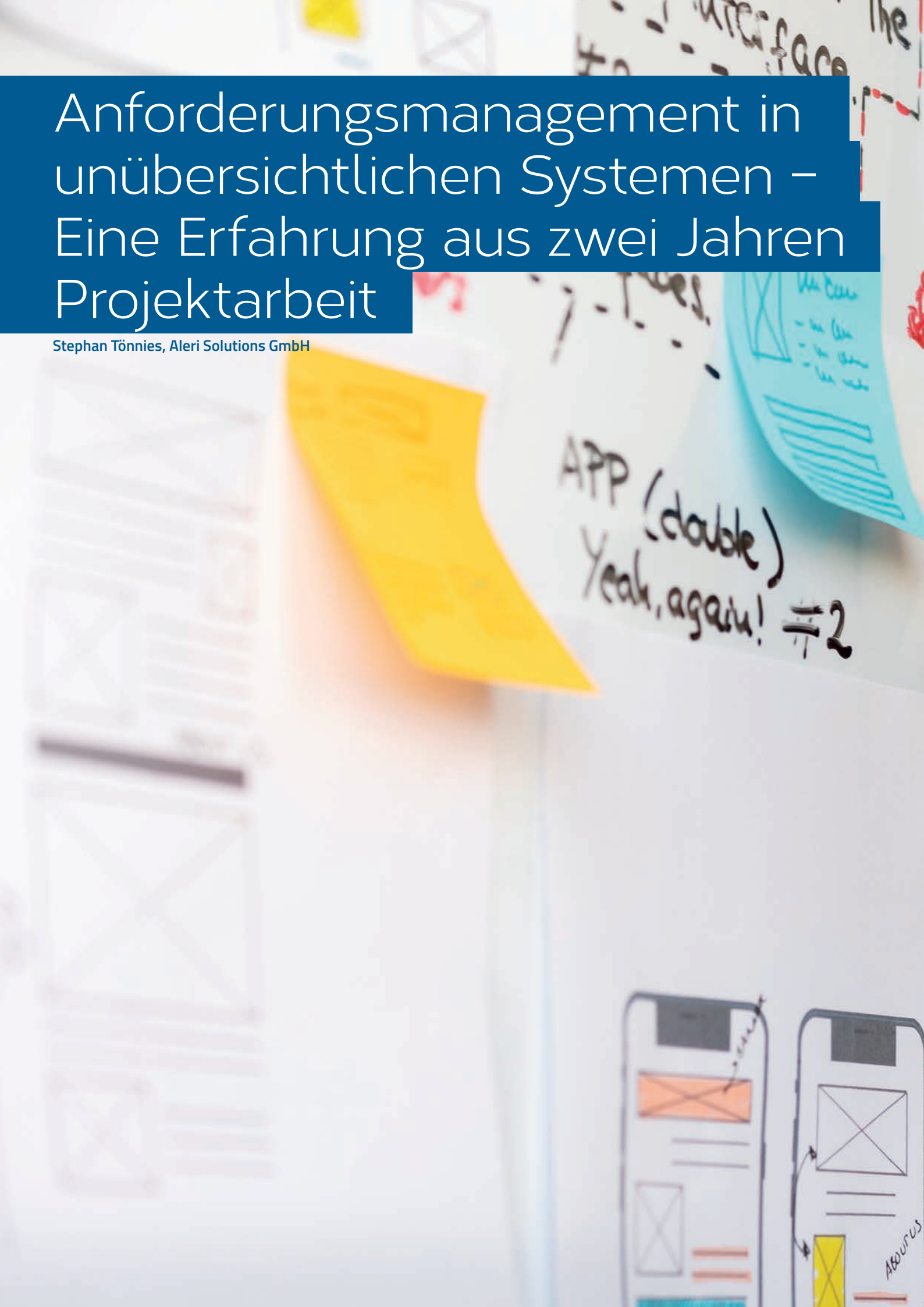
Ab 15 Euro im Jahr

www.ijug.eu

iJUG Verbund

Anforderungsmanagement in unübersichtlichen Systemen – Eine Erfahrung aus zwei Jahren Projektarbeit

Stephan Tönnies, Aleri Solutions GmbH



- Note:
ASAP

Der folgende Artikel ist ein Erfahrungsbericht aus zwei Jahren technischem Requirements Engineering. In diesem Beitrag teile ich Lehren und Einsichten, die ich aus meinem letzten Projekt gewonnen habe.

Als Business Analyst war ich in den letzten zwei Jahre lang im öffentlichen Sektor tätig und hatte mit meinem Team die Herausforderung, unterschiedliche Identitätsmanagementsysteme im Bildungsbereich miteinander zu verknüpfen. Die Aufgabe entpuppte sich in der Praxis als deutlich komplexer als anfänglich angenommen, denn die Herausforderung lag nicht nur in der Technik. Der fachliche Anwendungsfall „Eine Person kann sich anmelden" war ein ständiger Begleiter. Das bildete fast die Gesamtheit der fachlichen Anforderungen ab, mit denen ich arbeiten durfte. Als Business Analyst, zuständig für das Erarbeiten von Mehrwert für den Kunden, befand ich mich oft auf einer Gratwanderung. Es war offensichtlich, dass es sich eher um ein technisch geprägtes Projekt handelte, das in der Anforderungserhebung seine eigenen Herausforderungen mit sich brachte.

Anforderungen werden häufig in User Stories formuliert. Dieses Format bietet den Vorteil, Anforderungen in einer für Entwickelnde verständlichen Form zu präsentieren. Das Ziel dabei ist es, in den User Stories ein gewünschtes Verhalten und den angestrebten Zielzustand zu beschreiben. Zusätzlich sollte man den erwarteten Mehrwert der Story erläutern und klarstellen, welches Problem eigentlich gelöst werden soll. Viele Tickets in der Praxis erreichen diesen Detailgrad jedoch nicht, obwohl dies sinnvoll wäre. Eine präzise Formulierung erweist sich bereits bei fachlichen Anforderungen als herausfordernd, bei technischen Anforderungen sogar noch mehr.

Generell wird empfohlen, Anforderungen und User Stories primär fachlich zu formulieren. Der Kerngedanke dabei ist, dass Software entwickelt wird, um ein fachliches Problem zu beheben. Eine technische Lösung hat das Potenzial, fachliche Herausforderungen zu meistern, sofern sie adäquat eingesetzt wird. Für die Bewertung, ob eine technische Lösung ein fachliches Problem effektiv adressiert, ist eine präzise Beschreibung der fachlichen Zielsetzung unerlässlich. Das fachliche Ziel muss dabei klar definiert sein. Variabel bleibt die technische Realisierung, solange sie die fachliche Anforderung erfüllt. Bei der Formulierung einer technischen Story hingegen neigt man dazu, die technische Lösung zu präskribieren, anstatt den eigentlichen Zweck der Umsetzung in den Vordergrund zu rücken. Eine solche Vorgehensweise kann langfristig zu einem Missverhältnis zwischen Technik und fachlicher Zielsetzung führen; ein Problem, worunter viele Produkte und Projekte leiden. Technische Anforderungen konzentrieren sich oft stärker auf das Erfüllen spezifischer Aufgaben, statt den Nutzen für den Anwender zu priorisieren.

In meinem Projekt lag der Schwerpunkt darauf, technische Herausforderungen im Bereich des Identitätsmanagements im Bildungswesen zu bewältigen. Oft handelte es sich dabei um eine Art *Refactoring* bestehender Lösungen, jedoch mit erweiterten Funktionen in Bezug auf Konnektivität, Interoperabilität und auf die Portabilität von Identitäten. Ziel war es, dass sich am Verhalten der Lösungen nichts ändert, jedoch das Produkt Schnittstellen für den Austausch von Daten bereitstellt. Dies bedingte vor allem die Entwicklung technischer Stories, da sich die fachlichen Aspekte theoretisch nicht veränderten. Wir widmeten uns also primär der Lösung technischer Probleme.

Wir erkannten schnell, dass wir uns mit der Fokussierung auf technische User Stories in eine Sackgasse begaben. Aufgrund unserer Unvertrautheit mit dem Einsatzumfeld starteten wir zunächst mit eher grob umrissenen Tickets, um uns einen ersten Überblick zu verschaffen. Unsere Strategie bestand darin, Herausforderungen zu identifizieren und diese in separaten Tickets weiterzuentwickeln. Nach einer eingehenden Analyse und einem technischen Proof-of-Concept stießen wir auf spezifische Herausforderungen, die bewältigt werden mussten, um das zugrundeliegende Gesamtproblem effektiv anzugehen.

Typischerweise führte die Lösung eines Problems jedoch zur Aufdeckung von zwei weiteren. Dadurch verlor man leicht den Überblick und das ursprüngliche Ziel aus den Augen: die Entwicklung eines Minimal Viable Product (MVP) aus dem initialen Proof-Of-Concept. Die Verlockung war groß, sich in der Lösung von Teilproblemen zu verlieren, anstatt in Betracht zu ziehen, ob es möglicherweise einen alternativen Ansatz geben könnte. Ein wertvoller Rat aus der Unterhaltungsbranche sollte hier Beachtung finden: „Kill your darlings." Dies bedeutet, dass man sich manchmal von einer anscheinend attraktiven Idee lösen und einen anderen Weg einschlagen sollte. Anfangs führte dieser Fokusverlust in meinem Team und mir zu Schwierigkeiten. Doch mit der Zeit lernten wir, die Anforderungen effektiver zu strukturieren und unseren Fokus neu zu justieren. Eine einfache, aber wirkungsvolle Frage half uns dabei: „Welches Problem wollten wir ursprünglich lösen?" Diese Frage brachte uns in manchen Situationen dazu, zu überlegen, ob ein anderer Weg möglicherweise zielführender sein könnte.

Eine weitere signifikante Herausforderung war unser begrenztes Verständnis der oft historisch gewachsenen Netzwerke und Lösungen. Der Einarbeitungsprozess gestaltete sich schwierig, vor allem aufgrund der oftmals schwer auffindbaren Dokumentation. Bevor wir mit der eigentlichen Umsetzung beginnen konnten, war ein umfassendes *Reverse Engineering* aller Komponenten erforderlich. Dies stellte sich als nur bedingt umsetzbar heraus, da wir gleichzeitig in der Umsetzung Fortschritte erzielen wollten. Die Vernachlässigung dieser Schritte führte zu einem erhöhten Aufwand in der Nacharbeit.

Nach einigen Wochen starteten wir mit dem systematischen Ausformulieren aller impliziten Annahmen. Dies erwies sich teilweise als hilfreich, jedoch stellte sich heraus, dass keine unserer Annahmen zutraf. Keine einzige. Diese Erkenntnis führte mich zu einer wichtigen Lektion: Was, wenn wir davon ausgehen, dass keine unserer Annahmen Bestand hat? Wie würde dann die Lösung aussehen?

Mit Annahmen zu arbeiten, erwies sich als kluger Schachzug, auch wenn dies zunächst kontraproduktiv erschien. Unser Ziel war es, ver-

lässliche Erkenntnisse über die bestehende Software zu gewinnen. Dies war nur möglich, indem wir Hypothesen aufstellten und diese dann entweder bestätigten oder widerlegten. Waren die Annahmen korrekt, war das ein Erfolg – keine weiteren Anpassungen notwendig. Waren sie falsch, so hatten wir etwas Wichtiges gelernt und konnten unsere Strategie sowie unser weiteres Vorgehen entsprechend anpassen. Annahmen bieten in einem unsicheren Kontext auch eine gewisse Sicherheit und Stabilität. Ohne sie ist man oft mit den zahlreichen Risiken und der Unwissenheit überfordert. Sie dienen somit als eine Art Selbstüberlistung und Komplexitätsreduzierung.

Trotz solider Strukturen und wohlüberlegter Annahmen standen wir vor einem zentralen Problem: der unterschiedlichen Erwartungshaltung der Beteiligten. Jeder Stakeholder – der Kunde, die Behörde, die Entwickelnden, das Projektteam, das eigene Unternehmen – hat eigene Anforderungen an das Projekt und die anderen Beteiligten. Die Erhebung und das Management dieser Anforderungen ist ein kontinuierlicher Prozess, der oft unbewusst stattfindet. Entwickelnde beispielsweise streben nach hoher Codequalität und der sauberen Umsetzung ihrer Designmuster, mit dem Ziel, technische Schulden in ihrer Codebasis zu vermeiden. Diese Bedürfnisse sind häufig nur implizit formuliert. Um sie angemessen zu berücksichtigen, ist es ratsam, implizite Anforderungen in explizite umzuwandeln. Dieser Transformationsprozess kann durch formelle und informelle Interviews sowie das schriftliche Ausformulieren dieser Erkenntnisse erfolgen.

Zu Beginn eines Projekts ist es entscheidend, dass alle Stakeholder ihre Erwartungen klar formulieren. Dies trägt dazu bei, nachvollziehbar zu machen, warum jemand bestimmte Anmerkungen macht oder warum eine Person mit den erbrachten Leistungen nicht vollends zufrieden ist. Ein solches Vorgehen unterstützt das effektive Management von Erwartungen und hilft, potenzielle Enttäuschungen zu verhindern.

Wir haben diesen Ansatz in unserem Projekt umgesetzt. Bei den Entwickelnden und Architekten verlief dieser Prozess relativ unkompliziert. Je näher wir jedoch an die Kundenseite heranrückten, desto unklarer wurden die Erwartungen, die an uns gestellt wurden. Diese Unklarheit war problematisch, da wir vom Kunden beauftragt waren und dieser keine explizite Erwartungshaltung formuliert hatte. Es war notwendig, iterativ mit dem Kunden zu arbeiten, um seine Vision zu schärfen und ihm zu ermöglichen, konkrete Erwartungen an uns zu stellen.

Warum war dies wichtig? Eine Zusammenarbeit mit einem Kunden, der sich seiner Erwartungen nicht bewusst ist, birgt das Risiko großer Enttäuschungen am Projektende, wenn man den Kunden nicht angemessen einbezieht. Eine solche Enttäuschung wäre bedauerlich für den Kunden, da das Projekt möglicherweise seine Probleme nicht löst. Gleichzeitig wäre es frustrierend für das Entwicklungsteam, das viel Arbeit in eine Lösung investiert hat, die möglicherweise nicht wertgeschätzt wird. Und es wäre äußerst bedauerlich für das Unternehmen, das dadurch das Vertrauen des Kunden verlieren könnte. Aus diesen Gründen ist die frühzeitige Einbindung des Kunden in den Prozess essenziell.

Trotz der anfänglichen Herausforderungen konnten wir im Laufe der Zeit wertvolle Erkenntnisse gewinnen und unser Projekt erfolgreich vorantreiben. Ich hoffe, dass die geteilten Erfahrungen und bewältigten Schwierigkeiten für dich in deinem beruflichen Alltag von Nutzen sein können.

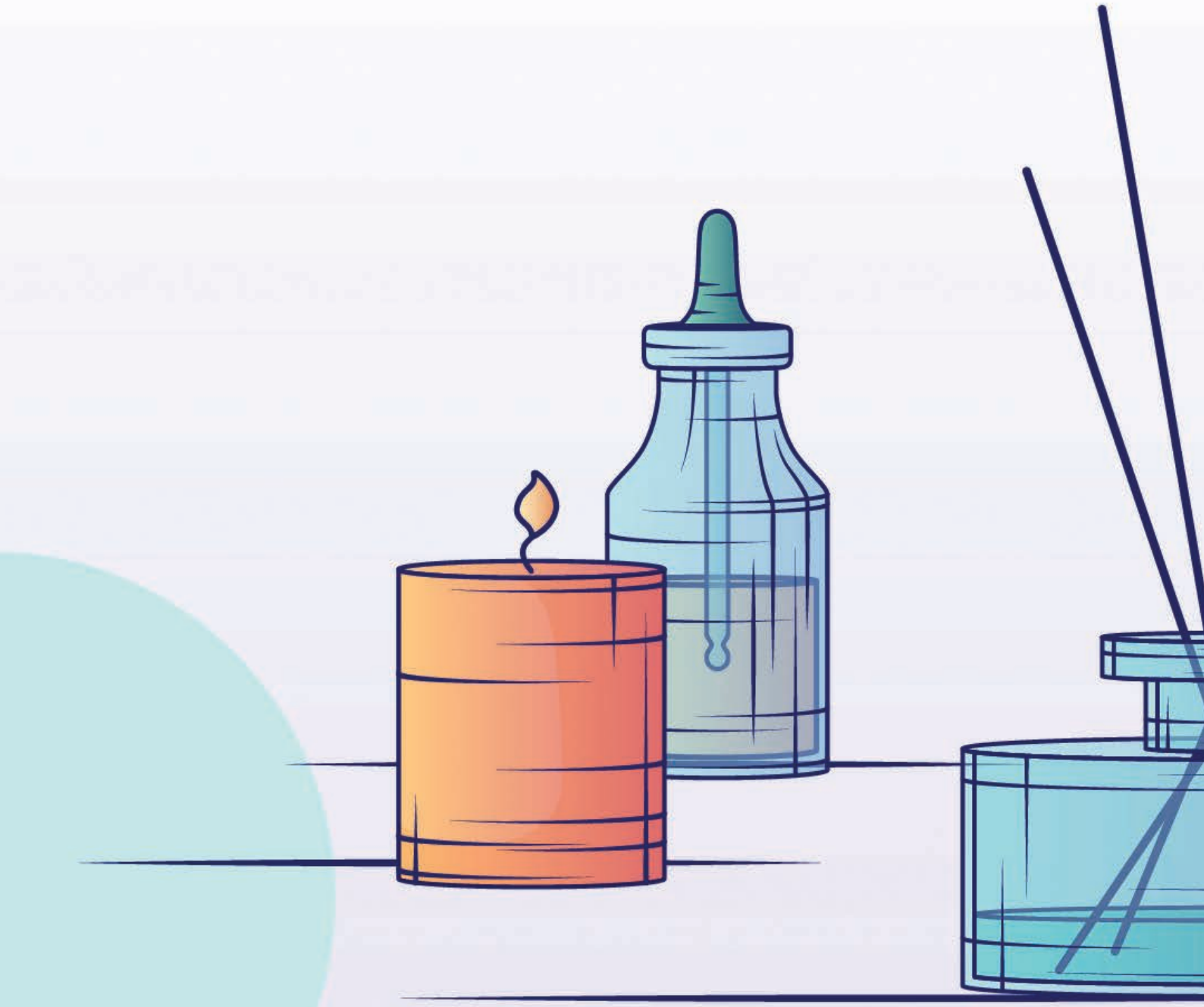
Stephan Tönnies

Stephan.toennies@aleri.de

Stephan Tönnies beschäftigt sich bei der Aleri mit dem Themen Requirements Engineering und agile Organisation. Er tritt als Speaker auf und engagiert sich bei der DOAG. Vormals im Vorstand für das Thema Nachwuchsentwicklung in der DOAG tätig, betätigt sich Stephan als Themenverantwortlicher für digitale Transformation.

Story Points und Homöopathie – Wie sich pseudowissenschaftliche Denkansätze in der agilen Softwareentwicklung etabliert haben

Susanne Pfeiffer, Qvest Digital AG



In vielen Scrum-Teams sind Story Points ein fester Bestandteil des Plannings und der Fortschrittsmessung. Allerdings wirft ihre Verwendung Fragen auf, die wir in diesem Artikel kritisch untersuchen möchten. Story Points stellen ein singuläres Phänomen in der Softwareentwicklung dar, das sonst in keinem anderen Bereich zur Anwendung kommt, die mathematische Grundlage ist fragwürdig. Um diese Probleme zu verdeutlichen, zieht dieser Artikel Parallelen zur Homöopathie. Beide Ansätze teilen auffallende Ähnlichkeiten in ihrer Pseudowissenschaftlichkeit. Wir möchten in diesem Artikel die Verwendung von Story Points kritisch hinterfragen und aufzeigen, wie sie dennoch sinnvoll eingesetzt werden können.

Im Jahr 1790 führte der Mediziner Samuel Hahnemann einen Selbstversuch mit Chinarinde durch, einer Pflanze, die wegen ihres Chinin-Gehalts fiebersenkend wirkt und gegen Malaria eingesetzt wurde. Er nahm über mehrere Tage zweimal täglich etwas Chinarinde zu sich und entwickelte malariatypische Beschwerden, wie zum Beispiel Abgeschlagenheit und Kopfschmerzen, aus denen er das homöopathische Ähnlichkeitsprinzip ableitete. Dieser Selbstversuch stellt nach landläufiger Meinung den Beginn der Homöopathie dar.

Am Beispiel der Homöopathie möchte ich erläutern, was Pseudowissenschaft ist, wie sie sich von empirischer Wissenschaft unterscheidet, und warum das in Scrum häufig eingesetzte Konzept von Story Points und deren Nutzung im Sprint Planning pseudowissenschaftliche Charakterzüge aufweist.

Story Points und Velocity

Ende der 90er Jahre setzte sich Extreme Programming als eine der ersten agilen Softwareentwicklungsmethoden durch. Damit wurden neue Ansätze zur Aufwandsabschätzung erforderlich, insbesondere sahen sich Entwickler:innen vor der Aufgabe, den Aufwand für die Umsetzung ihrer User Storys abzuschätzen. In diesem Kontext bildete sich der Begriff der „idealen Stunde" beziehungsweise des „idealen Tages" heraus.

Eine ideale Stunde ist die Menge Arbeit, die ein Paar von Entwickler:inen innerhalb einer Stunde schafft, wenn man es ungestört arbeiten lässt. Zunächst wurden diese idealen Zeitangaben für die Schätzungen verwendet und an die Stakeholder kommuniziert. Das Problem dabei: Es handelt sich um „ideale Zeitangaben". Um eine realistische Aufwandsschätzung zu erhalten, wird diese ideale Zeit mit einem Load-Faktor multipliziert, der üblicherweise um die drei liegt, das heißt, eine User Story mit dem Umfang eines idealen Tages benötigt also zirka drei reale Tage bis zu ihrer Fertigstellung. Die Angabe des Aufwands in „idealen Tagen" – oft auch nur kurz „Tage" genannt – führte aber zu falschen Erwartungen bei den Stakeholdern. Diese waren immer sehr überrascht, wenn die Umsetzung in der Regel dreimal so lange wie veranschlagt benötigte. Um diesem Kommunikationsproblem zu begegnen, führte Ron Jeffries, einer der Begründer des Extreme Programming, den Begriff der „Story Points" ein – ein Story Point ist danach nichts anders als ein idealer Tag. Die ursprüngliche Idee hinter den Story Points war also eine Abschätzung des Aufwands in Stunden oder Tagen *[1]*.

Die Scrum-Community übernahm die Idee der Story Points. Über die Zeit hat sich das Verständnis allerdings verändert. Während es sich ursprünglich um eine Zeitangabe handelte, fließen nun nach dem üblichen Verständnis verschiedene Aspekte in die Aufwandsschätzung ein. Normalerweise wird hier die Arbeitsmenge, die Komplexität und das Risiko aufgeführt. Wie diese Parameter genau in die Aufwandsschätzung einfließen, ist nicht allgemeingültig festgelegt. Eine direkte Relation zwischen der Einheit Story Point und der für eine geschätzte Arbeit erforderliche Zeit lehnen Scrum-Anhänger:innen in der Regel ab, die Umrechnung von Story Points in Stunden oder Tage wird allgemein als Antipattern betrachtet.

Für die Schätzung von Aufgaben werden nicht beliebige Zahlwerte verwendet, sondern nur bestimmte Werte, als Wertebereich dient dabei meist die sogenannte modifizierte Fibonacci-Folge, beginnend mit 1, 3, 5, 8, 13. Es wird angenommen, dass die Story-Point-Schätzungen mit dem tatsächlichen Aufwand korrelieren – auch, wenn bei der Schätzung häufig die Komplexität im Vordergrund steht. Mike Cohn zum Beispiel geht davon aus, dass innerhalb eines Scrum-Teams jede Story-Punkt-Zahl einer stochastischen Verteilung von Aufwänden entspricht und dass sich diese Verteilungen proportional zu den Schätzungen verhalten, also statistisch gesehen der Aufwand für ein mit 8 Punkten geschätztes Item knapp dreimal so groß ist wie der eines 3-Punkte-Items *[2]*.

Auf den Story Points basiert die Velocity. Generell gesprochen ist die Velocity das Maß für die Geschwindigkeit eines Teams, das heißt, wieviel das Team pro Zeiteinheit umsetzt. Typischerweise betrachtet man hier im Scrum-Kontext die Anzahl der Story Points, die pro Sprint umgesetzt werden. Aus der Entwicklung der Velocity über mehrere Sprints wird die Team-Velocity ermittelt, die durchschnittliche Velocity über die letzten Sprints. Diese Team-Velocity wird in der Regel im Planning herangezogen, um den Sprintumfang für den nächsten Sprint festzulegen. Die mittlere Velocity wird also fortgeschrieben und als Prognose für den kommenden Sprint angesetzt.

Sehen wir uns einmal an, was beim Sprint Planning üblicherweise passiert. Das Vorgehen im Planning unterscheidet sich stark von Team zu Team, aber in der Regel fließt eine Auswahl der folgenden Parameter ein, um einen Erwartungswert für die im kommenden Sprint zu erreichende Velocity zu berechnen:

- die Team-Velocity aus den vergangenen Sprints
- die zu erwartende Kapazität auf der Basis von Teamstärke und geplanten Abwesenheiten
- Pufferzeiten für ungeplante Aufgaben

Das Scrum-Team wählt die Backlog-Items für den kommenden Sprint so aus, dass die Summe der Story Points in etwa der kalkulierten Velocity entsprechen. Bei den ausgewählten Items kann es

sich um geschätzte Storys, ungeschätzte Bugs oder Tasks handeln. Als Ergebnis erhalten wir den Sprintumfang, mit dem das Team in den Sprint startet, mit der Erwartung, die auf diese Weise eingeplanten Items bis zum Sprintende abschließen zu können.

Wissenschaft und Pseudowissenschaft

Setzen wir uns als Nächstes mit den Begriffen Wissenschaft und Pseudowissenschaft auseinander.

In der empirischen Wissenschaft geht es darum, Theorien aufzustellen und zu belegen. Ein:e Wissenschaftler:in stellt auf der Basis bestimmter Beobachtungen zunächst eine Theorie auf. Aus der Theorie lassen sich Hypothesen ableiten, die sich dann empirisch überprüfen lassen. Die Überprüfung einer Hypothese passiert im Rahmen eines Experiments. Ein wissenschaftliches Experiment muss dabei einige Kriterien erfüllen:

- das erwartete Ergebnis muss vorher formuliert sein
- es muss wiederholbar und durch andere überprüfbar sein

Ein entscheidendes Kriterium ist die Falsifizierbarkeit – eine widerlegte Hypothese muss daher immer zur Verwerfung oder Anpassung der Theorie führen. Ein:e seriöse:r Wissenschaftler:in versucht immer, die eigenen Hypothesen zu widerlegen. Daher ist auch der Diskurs zwischen Wissenschaftler:innen so wichtig, weil nur so die Theorien gehärtet werden können.

Sehen wir uns dagegen einmal den Begriff der Pseudowissenschaft an. Wir haben es mit einer pseudowissenschaftlichen Theorie zu tun, wenn sie zwar den Anspruch auf empirische Wissenschaftlichkeit erhebt, allerdings keine wissenschaftliche Begründung liefern kann. Die Differenzierung zwischen Pseudowissenschaft und Wissenschaft ist alles andere als trivial. Ein erfolgversprechenderer Ansatz zur Charakterisierung von Pseudowissenschaften, der sich in den letzten Jahren etabliert hat, stützt sich auf folgende Merkmale:

- Eine Pseudowissenschaft behauptet von sich selbst, eine Wissenschaft zu sein – das unterscheidet sie beispielsweise von den Parawissenschaften.
- Die pseudowissenschaftliche Argumentation ignoriert epistemische (erkenntnisbezogene) Standards.
- Die pseudowissenschaftliche Argumentation erweist sich als indifferent gegenüber der Wahrheit und entzieht sich so der Falsifizierbarkeit *[3]*.

Homöopathie als Beispiel für eine Pseudowissenschaft

Die medizinischen Behandlungsmethoden im 18. Jahrhundert waren extrem, es wurden Methoden wie Aderlass, Brechkuren oder Abführmittel eingesetzt, die den Kranken in der Regel mehr schadeten als nutzten. Samuel Hahnemann suchte nach sanfteren Therapiemethoden. Auf der Basis seines Chinin-Selbstversuchs entwickelte er seine Idee der homöopathischen Behandlung.

Die Homöopathie wird häufig als Paradebeispiel für eine Pseudowissenschaft herangezogen. Die beiden wesentlichen Prinzipien, die der Homöopathie zugrunde liegen sind das Ähnlichkeitsprinzip – das Prinzip „Similia similibus" – und das Potenzierungsprinzip. Das Ähnlichkeitsprinzip besagt, dass ein Krankheitssymptom durch ein Mittel behandelt wird, das bei einem gesunden Menschen eben genau diese Symptome hervorrufen würde. Das homöopathische Arzneimittel wird dabei durch Potenzierung hergestellt, das heißt, eine extreme Verdünnung der eigentlich wirksamen Ursubstanz. Je stärker das Mittel verdünnt ist, umso stärker soll nach dem Potenzierungsprinzip dessen Wirkung sein. Die auf diese Weise hergestellten homöopathischen Arzneimittel sind dabei oft so stark verdünnt, dass sie mit hoher Wahrscheinlichkeit kein Molekül der Ursubstanz mehr enthalten.

Story Points und Homöopathie

Setzen wir die Bausteine zusammen. Wir möchten untersuchen, inwieweit die Diskussionen um Story Points pseudowissenschaftlichen Charakter aufweisen, und wo wir Parallelen zu den Diskussionen um die Anwendung der Homöopathie finden können.

Um nachzuweisen, dass die beiden Gebiete pseudowissenschaftlichen Charakter haben, müssen wir folgende Merkmale nachweisen:

- Beide Gebiete erheben den Anspruch eines wissenschaftlichen Vorgehens oder den Anspruch, wissenschaftlich anerkannte Methoden korrekt anzuwenden.
- Die aufgestellten Thesen sind unplausibel.
- Die Verteidigung der eigenen Position erfolgt mit Scheinargumenten, die einer kritischen Untersuchung nicht standhalten und wissenschaftliche Erkenntnisse ignorieren.

Orientierung in komplexen Situationen

Gehen wir zunächst einmal einen Schritt zurück und überlegen, wie die beiden Ansätze entstanden sind. Sowohl die Homöopathie als auch der Story-Point-Ansatz sind aus einem Leidensdruck heraus entstanden.

Samuel Hahnemann wollte eine Alternative zu den rabiaten Methoden der im 18. Jahrhundert praktizierten Heilkunde schaffen. Er wollte Menschen helfen und eine sanftere Medizin entwickeln. Nun gibt es in der Medizin oft keine einfachen Lösungen – und noch viele weniger im 18. Jahrhundert, als die medizinische Forschung noch weit von der heutigen entfernt war. Hahnemann hat versucht, eine einfache, leicht fassbare Lösung für ein komplexes Problem zu finden – und bekanntermaßen neigt der Mensch dazu, auf der Suche nach Lösungen schon einmal zu früh abzubiegen und die Einfachste zu wählen.

Wie verhält es sich mit Story Points? Die Anhänger:innen der agilen Softwareentwicklung hatten die Schwächen der bis in die 80er Jahre vorherrschenden wasserfallartigen Softwareentwicklungsmethoden erkannt und mit dem agilen Manifest und der Methode des Extreme Programming einen Gegenentwurf geschaffen, der sich als ausgesprochen erfolgreich erwies. Ein Problem blieb aber: Die Softwareentwicklungsteams operierten nicht im luftleeren Raum und mussten belastbare Forecasts abgeben – woran bereits Generationen von Softwareentwickler:innen zuvor gescheitert waren. Softwareentwicklungsprojekte sind nicht ohne weiteres vorhersagbar, und der Einsatz von Story Points bot eine einfache Möglichkeit, diese Unsicherheit irgendwie in die Vorhersagen hineinzuarbeiten.

Die Absicht war in beiden Fällen nachvollziehbar und gut. Allerdings gilt wie so häufig: Gut gemeint bedeutet leider nicht, dass es gut funktioniert. In beiden Fällen haben wir es mit Methoden zu tun, die

sich ganz singulär aus dem Nichts heraus entwickelt haben, ohne sich auf Methoden zu stützen, die in verwandten Bereichen bereits erfolgreich zum Einsatz gekommen sind.

Schauen wir uns zunächst einmal die Homöopathie an: Hahnemanns Homöopathielehre ist zu einer Zeit entstanden, als die Medizin noch eine wilde Mischung aus Aberglauben, tradierten Methoden und wissenschaftlicher Forschung gewesen ist. Was Hahnemanns Methode aber auszeichnet: Sie berief sich auf keine andere Methode, die vorher ernsthaft verfolgt worden war, sondern entsprang im Wesentlichen einer einzigen Idee, die er dann mehr oder weniger gut belegt hatte und fantasievoll ausarbeitete. Bereits im 19. Jahrhundert wurde die Homöopathie scharf kritisiert und als Quacksalberei bezeichnet. Die beiden Prinzipien, das Ähnlichkeitsprinzip und das Potenzierungsprinzip, stehen im Widerspruch zu allen bisher erworbenen medizinischen, biologischen, chemischen und physikalischen Erkenntnissen. Anhänger der Homöopathie ignorieren also alle medizinischen und naturwissenschaftlichen Forschungsergebnisse und verfolgen ihr eigenes isoliertes Glaubenssystem weiter.

Und wie sieht es mit Story Points aus? Überlegen wir doch mal, warum es so schwer ist, Prognosen in der Softwareentwicklung zu erstellen: Bei einem Softwareprojekt handelt es sich um kein mathematisch berechenbares System, es arbeiten Menschen zusammen, die Probleme sind a priori nicht in allen Details bekannt – kurz: Wir haben es mit einem nicht vorhersagbaren System zu tun.

Nun gibt es aber ebenso in vielen anderen Zusammenhängen solche unvorhersagbaren Systeme. Beispiele dafür sind die Meteorologie, die Finanzwissenschaften oder die Kernphysik. Auch hier gibt es die Anforderung, einen weiteren Verlauf zu prognostizieren. In allen diesen Bereichen greift man beim Forecasting auf stochastische Methoden zurück, die keine deterministischen Aussagen über den weiteren Verlauf, sondern immer nur Wahrscheinlichkeitsaussagen liefern. Nirgendwo anders als in der agilen Softwareentwicklung werden willkürliche Zahlen als Vorhersage der weiteren Entwicklung aus einer – nicht einmal korrekten – Fibonacci-Folge auf den Tisch geworfen.

Anspruch auf Wissenschaftlichkeit

Ein wichtiges Merkmal von Pseudowissenschaften besteht darin, dass sie den Anspruch auf Wissenschaftlichkeit erheben. Das bedeutet, dass die pseudowissenschaftliche Theorie das Vokabular und das Auftreten seriöser Wissenschaften nachahmt, ohne aber ihren Standards tatsächlich gerecht zu werden.

Homöopathie wird an verschiedenen Hochschulen gelehrt, es gibt diverse homöopathische Gesellschaften und Vereinigungen mit wissenschaftlichem Anspruch. Ansonsten betreiben Vertreter der Homöopathie oft eine Art wissenschaftlichen Cargo-Kult, in dem sie ihren Artikeln einen wissenschaftlichen Anstrich geben. Es werden Studien durchgeführt und zitiert, die die Wirkung homöopathischer Behandlungen belegen sollen, bei genauerem Hinsehen allerdings nicht den sonst in der Medizin geforderten Standards entsprechen. Die meisten Homöopath:innen halten sich selbst für ernstzunehmende Wissenschaftler:innen.

Wie verhält es sich denn mit dem ganzen Story-Point- und Velocity-Ansatz? Wird hier der Anspruch auf Wissenschaftlichkeit erhoben? Sicherlich nicht so extrem wie in der Homöopathie. Tatsächlich betonen die Befürworter:innen des Story-Point-Ansatzes immer wieder, dass es sich nur um Schätzungen handele und erheben keinen Anspruch auf Objektivität. Verräterisch ist hier allerdings die Verwendung mathematischer Begrifflichkeiten, wie zum Beispiel der Fibonacci-Folge, oder verschiedener Konzepte aus dem statistischen Kontext. Allein die Verwendung des mathematischen Vokabulars gibt der Thematik einen wissenschaftlichen Anstrich – der Philosoph Philipp Hübl spricht hier von sogenannten Prestige-Markern *[4]*. Es werden immer wieder mathematische Zusammenhänge behauptet und verwendet, die im Kern aber nie überprüft werden, was zu einer wissenschaftlich erscheinenden Scheindiskussion führt.

Homöopathie-Geschwurbel und Story-Point-Pseudomathematik

Die der Homöopathie zugrundeliegenden Ideen sind ausgesprochen unplausibel. Zum einen ist es vollkommen unklar, warum das Ähnlichkeitsprinzip gelten sollte, wissenschaftliche Erklärungen für ein solches Prinzip gibt es nicht. Als Begründung wird aufgeführt, dass die Verabreichung eines entsprechenden Mittels die Abwehrkräfte des Körpers gegen die Erkrankung aktivieren soll. Das zweite Prinzip der Potenzierung geht davon aus, dass ein homöopathisches Mittel umso stärker wirkt, je stärker es potenziert, also verdünnt, ist. Die Homöopathie liefert als Erklärung die „Imprint-Theorie", nach der der Lösung durch die Potenzierungsschritte die Wirkung der Ursubstanz eingeprägt und verstärkt wird, sodass im Endergebnis kein Molekül der Ursubstanz mehr erforderlich ist. Sogar eine sogenannte „verallgemeinerte Quantentheorie" wird herangezogen, um die beiden Prinzipien zu begründen – dabei wird von einer doppelten Quantenverschränkung gesprochen, die die Wirkung der bis zur Nichtexistenz verdünnten Ursubstanz auf die zu behandelnde Krankheit erklären soll.

Auch wenn natürlich niemand die Verwendung von Story Points und Velocity in Scrum mit so halbseidenen Theorien wie einer verallgemeinerten Quantentheorie begründen würde, glänzt leider auch diese Praxis nicht durch Plausibilität. Dazu schauen wir uns einmal die zugrundeliegenden Annahmen an. Wie bereits festgestellt, fließen in eine Story-Point-Schätzung verschiedene Bewertungen hinsichtlich des Umfangs, der Komplexität und des Risikos ein. Dabei ist vollkommen undefiniert, um was für Größen es sich genau handelt. Wie wird der Umfang gemessen? Als Zeit? Wie die Komplexität, wie das Risiko? Ebenfalls undefiniert ist, wie diese Parameter einfließen, ob additiv, multiplikativ oder noch ganz anders. Das sind aber Fragen, die man zunächst einmal beantworten müsste, bevor überhaupt klar ist, was herauskommt, wenn man den Gesamtaufwand zweier in Story Points geschätzter Items betrachtet. Die Annahme, dass der Gesamtaufwand von zwei Items mit vom Aufwand drei und fünf Story Points identisch ist zum Aufwand eines Items von acht Story Points, ist daher zunächst einmal aus der Luft gegriffen und angesichts der Unklarheiten sicherlich nicht universell richtig. Die weiter oben aufgeführte Annahme von Mike Cohn, dass sich der Aufwand in etwas linear zur Story-Point-Schätzung verhält, ist damit ebenso wenig haltbar, empirische Belege dafür liegen nicht vor, vielmehr handelt es sich hier um Annahmen.

Es gibt noch mehr Unplausibilitäten in diesem Zusammenhang: So werden Bugs in der Regel nicht geschätzt, die Schätzung von Bugs wird von vielen Story-Point-Befürworter:innen als Antipattern be-

trachtet. Das führt uns aber zu der absurden Situation, dass sich Bugs im Sprintumfang nicht widerspiegeln. Wir können also beliebig viele Bugs im Sprint Planning mit aufnehmen, scheinbar ohne Einfluss auf den Sprintumfang.

Auch die für den Sprint-Forecast oft eingesetzte Fortschreibung der Mittelwerte erweist sich als nicht haltbar. Nach meinen Erfahrungen schwankt die Velocity von Sprint zu Sprint sehr stark, das heißt, wir werden bei der Verwendung des Velocity-Mittelwertes doch nur in etwa der Hälfte aller Fälle den Sprint auch erfolgreich abschließen – mit großen Ausreißern nach unten oder oben, siehe *Abbildung 1*. Daher ist auch die Nutzung des Mittelwertes als Planwert für den Sprintumfang höchst fragwürdig.

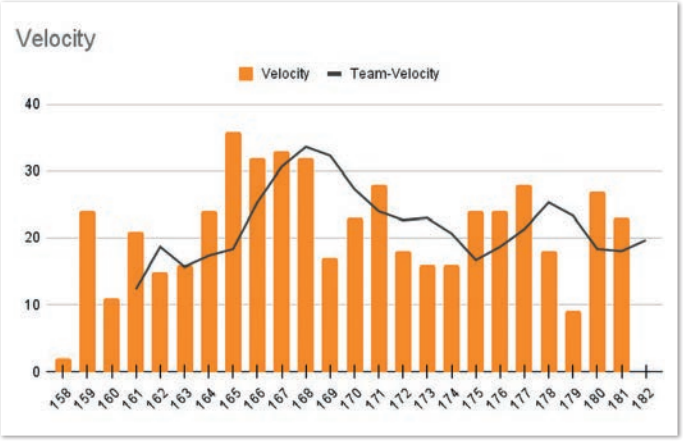


Abbildung 1: Die Velocity schwankt sehr stark, ihr Mittelwert ist daher für den Sprint Forecast ungeeignet (© Susanne Pfeiffer)

Vergleichbarkeit verweigern

Eine Vermeidungsstrategie, die wir sowohl in der Homöopathie als auch bei den Befürworter:innen von Story Points und Velocity finden, besteht darin, sich Vergleichen jedweder Form zu verweigern.

In der homöopathischen Literatur finden sich beispielsweise Aussagen, dass sich hoch individualisierte Therapien, wie sie in der Homöopathie angewandt werden, nicht mit einer Medikation aus der konventionellen Pharmazie vergleichen lassen oder, dass randomisierte Kontrollstudien von homöopathischen Behandlungsmethoden eine Verschwendung von Zeit, Geld und Energie seien. Ein führender Homöopath stellt das Prinzip randomisierter Doppelblindstudien, die als Goldstandard in der medizinischen Forschung gelten, grundsätzlich in Frage. Wir sehen hier also eine ausgeprägte Verweigerungshaltung, wenn es darum geht, homöopathische Mittel gegen die der Schulmedizin einer objektiven Prüfung zu unterziehen *[2]*.

Auch bei den Vertreter:innen von Story-Point-Schätzungen finden wir eine ähnliche Haltung, was die Aufstellung von Vergleichen angeht. Die Umrechnung zwischen Story Points und realen Stunden wird abgelehnt, sodass für den Stakeholder die mit der Umsetzung eines Features verbundenen Kosten nicht ersichtlich sind. Teamübergreifende Vergleiche werden abgelehnt, ebenso wie die Bewertung der Team-Performance über die Zeit auf der Basis der Team-Velocity. Das Argument, das gegen solche Vergleiche ins Feld geführt wird, ist immer die Ablehnung von äußerem Leistungsdruck auf das Team. Dass sich ein solcher Leistungsdruck nur negativ auf ein Team auswirken kann, ist unbestritten – ob objektivierte Messungen unbedingt Hand in Hand gehen mit einem ungesunden Leistungsdruck, ist allerdings fraglich. Prateek Singh liefert hier ein überzeugendes Gegenbeispiel, wie bei Ultimate Software Flussmetriken als objektive Leading Indicators verwendet und transparent über alle Teams kommuniziert werden, um eine frühzeitige Identifikation von Engpässen zu ermöglichen *[5]*. Ob Druck ausgeübt wird, ist weniger eine Frage der verwendeten Kennzahlen, sondern viel mehr eine Frage der Unternehmenskultur.

Cherry Picking

Sehen wir uns an, wie Anhänger:innen von Homöopathie mit Kritik umgehen. Hier finden wir verschiedene Ausweichstrategien, um der offenen Auseinandersetzung mit kritischen Argumenten zu entgehen. In der Homöopathie finden wir hier oft die Strategie des Cherry Picking. Dabei werden einzelne Fälle aufgeführt, in denen schwere Erkrankungen nach einer homöopathischen Behandlung geheilt waren, und in einer – wissenschaftlich nicht haltbaren – Verallgemeinerung dafür verwendet wurden, die Wirksamkeit der homöopathischen Therapiemethode für derartige Fälle zu belegen. Die vielen Fälle, in denen eine homöopathische Behandlung in vergleichbaren Fällen erfolglos war, werden dabei ignoriert.

Und wie sieht es bei den Story Points aus? Die übliche Praxis ist, dass das Scrum-Team Mittelwerte der Velocity als Planwert für den nächsten Sprint verwendet. Liegt dann die Menge an tatsächlich geschaffter Arbeit darunter oder darüber, so wird das meist als Ausnahme betrachtet. Tatsächlich stellt die Ausnahme aber die Regel dar – nur in wenigen Fällen werden wir tatsächlich auf diesem Mittelwert landen. Der Statistiker Sam L. Savage meint dazu: „Plans based on Average fail on Average" *[6]*. Bemerkenswert ist, dass die Nichterfüllung oder Übererfüllung des anvisierten, auf der Team-Velocity basierenden Sprintumfangs immer wieder Gegenstand in der Sprint-Retrospektive ist und so unnötigerweise Aufmerksamkeit bindet, die besser in tatsächliche Probleme investiert wäre.

Umgang mit Kritik

Schauen wir uns einmal an, wie die Anhänger:innen der Homöopathie oder die der Velocity-basierten Sprint Plannings mit Kritik und Gegenargumenten umgehen. Auch hier können wir Gemeinsamkeiten erkennen.

Die Homöopathie setzt beispielsweise auf Immunisierungsstrategien, um Kritik entgegenzuwirken. So beschreibt der Homöopath George Vithoulkas eine Forschungsmethode, nach der eine homöopathische Behandlung als erfolgreich anzusehen ist, unabhängig von Dauer, Verlauf und Erfolg der homöopathischen Behandlungsmethode. Damit entzieht er sich jeglicher kritischen Bewertung. Generell ist zu beobachten, dass Forschungsergebnisse und Erkenntnisse, die dem homöopathischen Glaubenssystem widersprechen, ignoriert werden *[3]*.

Auch die Anwendung von Story Points scheint unter ihren Befürworter:innen über jeder Kritik zu stehen. Versucht man, mit rationalen Argumenten die Anwendung von Story-Point-Schätzungen im Sprint Planning zu kritisieren, so stößt man nach meinen Erfahrungen in der Regel vor allem auf zwei Gegenargumente:

- „Du hast es nicht verstanden!" – Leider findet sich tatsächlich eine Menge Dogmatismus, der so gar nicht zu einem agilen, offenen Mindset, wie es in der agilen Blase propagiert wird, zu passen scheint. Wer Story Points in Personenstunden umrechnen möchte, hat es nicht verstanden. Wer Bugs schätzen möchte, um sie im Sprint Planning als Aufwand zu berücksichtigen, hat es nicht verstanden. Ich denke, die Begründung muss anders sein: Versuche mal, deinen Story-Point-Ansatz einer komplett Fachfremden zu erläutern und die logischen und mathematischen Widersprüche aufzulösen. Wenn dir das nicht gelingt, dann solltest du dir überlegen, ob dein Ansatz nicht einen Haken hat. Das Planen eines Sprints sollte kein hochkomplexes Thema darstellen, dass sich einem Außenstehenden nicht relativ einfach, widerspruchsfrei und nachvollziehbar erklären lässt.
- „Ach – für die Beurteilung, ob ein Sprintplan realistisch erscheint, betrachte ich gar nicht die Story Points, ich schätze das eher aus dem Bauch heraus. " – Das habe ich mehrfach gehört, nachdem ich in meinen Entwicklungsteams anhand der Vergangenheitswerte dargestellt habe, dass die Story Points eigentlich keine zuverlässige Größe für irgendwelche Forecasts darstellen.

Wie wir zusammenkommen

Auch wenn die mathematischen Grundlagen fragwürdig sind, heißt das nicht, dass Story-Point-Schätzungen wertlos wären. Die Widersprüche fangen eigentlich erst so richtig an, wenn man anfängt, mit den Story-Point-Schätzungen zu rechnen.

Story Points geben den Entwickler:innen ein Mittel an die Hand, Schätzungen abzugeben, ohne unter dem Druck zu stehen, objektive Zahlen abliefern zu müssen, für die sie anschließend zur Rechenschaft gezogen werden können. Das gibt allen viel Sicherheit im Schätzprozess und sorgt dadurch für eine offene und unvoreingenommene Diskussion. Die Diskussionen, die im Refinement auf der Basis der Schätzungen entstehen, sind in der Regel sehr wertvoll und liefern oft Umsetzungsideen oder Hinweise auf Aufwandstreiber. Insofern ist der Schätzprozess im Refinement ein gutes Mittel, Entwickler:innen abzuholen und eine Kommunikation in Gang zu bringen.

Einer der wichtigsten Kritikpunkte ist die Verwendung der Team-Velocity für das Forecasting im Sprint Planning. Ein Gegenentwurf für das Sprint Planning aus der Kanban-Welt besteht in einer flussorientierten und probabilistischen Betrachtungsweise. Das bedeutet:

- Flussmetriken wie Durchsatz und Durchlaufzeiten statt Velocity und Story Points
- Service Level Expectations (SLEs) statt Mittelwerte

Der Durchsatz ist die Anzahl der Items, die innerhalb eines bestimmten Zeitraums, also beispielsweise eines Sprints, fertiggestellt werden. Eine SLE ist die Anzahl der Items, die das Team mit einer bestimmten Wahrscheinlichkeit innerhalb eines bestimmten Zeitraums abschließen kann. Einzelheiten dazu gibt es in Daniel Vacantis *Actionable Agile Metrics for Predicability* [7].

Um den Sprintumfang festzulegen, können wir beispielsweise zunächst die Anzahl der Items ermitteln, die das Entwicklungsteam mit einer Wahrscheinlichkeit von 85 % innerhalb des Sprints abschließen kann. Danach sind für die Festlegung des Sprintumfangs die Items auszuwählen, die im Rahmen des Sprints umgesetzt werden sollen.

Dies ist der Punkt, an dem wir die Story-Point-Schätzungen sinnvoll im Sprint einsetzen können – ohne esoterische Pseudomathematik. Unsere SLE sagt uns ja nur, wie viele Items wir im Sprint aufnehmen wollen, aber noch nicht, welche denn genau.

Für genau diese Auswahl können Schätzungen eine hilfreiche Orientierung geben. Denn natürlich ist es ein Unterschied, ob wir fünf 1-Story-Point-Items in den Sprint ziehen, oder fünf 13-Story-Point-Items. Die Story-Point-Schätzungen können hier also bei der Auswahl helfen. So könnte das Entwicklungsteam beispielsweise ein oder zwei größere Items aufnehmen, die auf das Sprintziel einzahlen und weitere kleinere Items, um eine Wertmaximierung des Sprints zu erreichen. Auf diese Weise lassen sich die statistischen Erkenntnisse aus der Vergangenheit mit dem Bauchgefühl der Entwickler:innen zu einer Synthese vereinigen, um einen zuverlässig leistbaren Sprintumfang zu finden, mit dem sich das Entwicklungsteam wohlfühlt und der hohe Erfolgsaussichten verspricht.

Quellen

[1] Jeffries, Ron (2019): Story Points Revisited. *https://ronjeffries.com/articles/019-01ff/story-points/Index.html*

[2] Cohn, Mike (2022): Don't Equate Story Points to Hours. *https://www.mountaingoatsoftware.com/blog/dont-equate-story-points-to-hours*

[3] Mukerji, Nikil und Ernst, Edzard (2022): Why homoeopathy is pseudoscience. Synthese 200, 394 (2022). *https://doi.org/10.1007/s11229-022-03882-w*

[4] Hübl, Philipp (2023): Vorlesung "Bullshit-Resistenz. Einführung ins kritische Denken" an der Universität der Künste Berlin (UDK) im Sommersemester 2023, 9. „Pseudowissenschaft", *https://www.youtube.com/watch?v=gOEGKm5IAE4*

[5] Vacanti, Daniel (2017): When Will It Be Done? ActionableAgile Press

[6] Savage, Sam (2012): The Flaw of Averages. John Wiley & Sons, Inc.

[7] Vacanti, Daniel (2015): Actionable Agile Metrics for Predicability. ActionableAgile Press

Susanne Pfeiffer

s.pfeiffer@qvest-digital.com

Susanne Pfeiffer arbeitet als Product Owner bei der Qvest Digital AG. Seit vielen Jahren arbeitet sie mit agilen Entwicklungsteams und setzt auf schlanke Prozesse, um eine maximale Wertschöpfung in der Softwareentwicklung zu erreichen.

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
42 JUG Frankfurt

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.

DOAG e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. DOAG e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Björn Bröhl. DOAG 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 © Designed by catalyststuff
https://freepik.com
S. 10 + 11: Bild © Designed by vectorpocket
https://freepik.com
S. 14: Bild © Sloth McSloth
https://stock.adobe.com
S. 16 + 17: Bild © ant
https://stock.adobe.com
S. 26 + 27: Bild © IBEX.Media
https://stock.adobe.com
S. 34 + 35: Bild © semisatch
https://stock.adobe.com
S. 42 + 43: Bild © Designed by upklyak
https://freepik.com
S. 50: Bild © Open Elements
https://open-elements.com
S. 56 + 57: Bild © Designed by stories
https://freepik.com
S. 64 + 65: Bild © Designed by macrovector
https://freepik.com
S. 70 + 71: Bild © PCH.Vector
https://stock.adobe.com
S. 76 + 77: Bild © Designed by freepik
https://freepik.com
S. 80 + 81: Bild © Designed by macrovector
https://freepik.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

Alle Rechte vorbehalten. Jegliche Vervielfältigung oder Weiterverbreitung in jedem Medium als Ganzes oder in Teilen bedarf der schriftlichen Zustimmung des Verlags.

Die Informationen und Angaben in dieser Publikation wurden nach bestem Wissen und Gewissen recherchiert. Die Nutzung dieser Informationen und Angaben geschieht allein auf eigene Verantwortung. Eine Haftung für die Richtigkeit der Informationen und Angaben, insbesondere für die Anwendbarkeit im Einzelfall, wird nicht übernommen. Meinungen stellen die Ansichten der jeweiligen Autoren dar und geben nicht notwendigerweise die Ansicht der Herausgeber wieder.

Inserentenverzeichnis

DOAG e. V. U 4, S. 63
iJUG e.V. S. 53, S. 59, S. 69, S. 75
JavaLand GmbH U 2
JUG Saxony e. V. S. 9
Techniker Krankenkasse S. 23

APEX connect

by DOAG

22. - 24.04.2024

VAN DER VALK AIRPORTHOTEL DÜSSELDORF

apex.doag.org