Java aktuell

java-aktuell.eu

Ausgabe 01/2026

Java 25
Das aktuelle LTS-Release im Detail

Projekte
GitOps, OpenTelemetry, Maven

Neuronale Netze
Schlauer als das menschliche Gehirn?

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwicklerinnen und Entwickler

D: 7,95 EUR | CH: 11,99 CHF | A: 9,00 EUR | Benelux: 8,50 EUR

PROGRAMM ONLINE

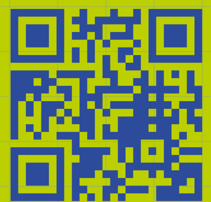
12. + 13. MÄRZ 2026

WWW.DEVLAND.EU

#DEVLAND

THE NEXT GENERATION OF DEVELOPMENT

EUROPA-PARK IN RUST



ARCHITECTURE AVENUE
AI VALLEY
CLOUD CLIFFS
DATA DOCKS
CAREER COAST

Liebe Leserinnen und Leser,

willkommen zur ersten Ausgabe des neuen Jahres, die auch gleichzeitig unsere JavaLand-Ausgabe ist! Wir hoffen, ihr hattet entspannte Feiertage und bisher einen guten Auftakt im neuen Jahr. Schön, dass ihr wieder da seid. Falls ihr dieses Heft gerade auf der JavaLand in den Händen haltet: Jatumba und ganz viel Erfolg und Spaß im Europa-Park!

Das Java-Tagebuch bringt euch wieder einmal schnell auf den aktuellen Stand, was kurz vor der Jahreswende so in der Java-Community passiert ist.

Zu Beginn werfen wir gemeinsam einen Blick auf das aktuelle Long-Term-Support-Release Java 25. Falk Sippach nimmt alle Neuheiten unter die Lupe und erörtert diese im Detail. Im darauffolgenden Artikel präsentiert Prof. Dr. Stefan Wagenpfeil einige Möglichkeiten, wie Teams mithilfe moderner Entwicklungsansätze effizienter und sicherer entwickeln können. Ab Seite 24 geht es um vereinfachtes und übersichtliches Abhängigkeitsmanagement. Dr. Jendrik Johannes zeigt, wie dies mit strukturiertem Vorgehen und modernem Toolset gelingen kann. Eine Möglichkeit, seine Java-Anwendung in die Cloud zu bringen, ist ein GitOps-Workflows. Thomas Michael zeigt uns in seinem Artikel nicht nur das finale Ergebnis, sondern erläutert Schritt für Schritt den Prozess dorthin. Industrieautomation mithilfe von Open Source? Na klar! Christofer Dutz zeigt ab Seite 38, wie dies umgesetzt werden kann.

Jörg Liedl präsentiert, wie Trunk-Based Development (TBD) und Pairing kombiniert werden, um Zusammenarbeit und Effizienz in Softwareentwicklungsteams zu verbessern. Im Anschluss geht es mit Sascha Selzer um Spring Boot und Container, die den Standard moderner Java-Microservices bilden. Doch Standard heißt nicht gleich schlank – der Autor zeigt, wie Spring-Boot-Anwendungen mit Dependency-Tuning, passendem JRE und minimalem Basis-Image optimiert werden können. Lang ist's mittlerweile her, dass wir Teil 1 der Neuheiten von Maven 4 in der Java aktuell 4/24 erschienen ist – doch nun hat das Warten ein Ende und die Fortsetzung ist endlich da! Matthias Bünger widmet sich ab Seite 58 den Major Changes: die Java-Version und Änderungen an der pom.xl. Memory Leaks treten auf, wenn nicht mehr benötigter Programmierspeicher trotzdem besetzt bleibt. Wie sie mithilfe von Open Telemetry diagnostiziert, behoben und von vorneherein verhindert werden können, zeigt Markus Meyer in seinem Artikel.

Ammoniak-Kälteanlagen sind klimafreundlich und für viele Industrien unverzichtbar, reagieren jedoch empfindlich auf kleinste Betriebsabweichungen. Da klassische Wartung hier oft zu spät greift, kann KI helfen, Störungen frühzeitig zu erkennen und Ausfälle zu verhindern. Ein solches Projekt haben Eldar Sultanow und Matthias Petrich begleitet und geben in ihrem Artikel spannende Einblicke. Zum Abschluss geht es in Dr. Sandra C. Signores Beitrag darum, wie neuronale Netze vom menschlichen Gehirn inspiriert sind und doch nur das erkennen, was ihnen zuvor beigebracht wurde. Der Artikel stellt die Frage, ob echte Intelligenz in der Maschine entsteht oder vielmehr in unserer Fähigkeit, zwischen Nachahmung und Verständnis zu unterscheiden.

Wir wünschen euch viel Spaß beim Lesen!

Lisa Damerow

Lisa Damerow
Redaktionsleitung Java aktuell

INHALT

10

Java 25:
Das aktuelle LTS-Release im Detail

44

Trunk-Based Development und Pairing
als ideale Kombination

3 Editorial

6 Java-Tagebuch

10 Java 25 – Die Neuerungen der LTS-Version
Falk Sippach

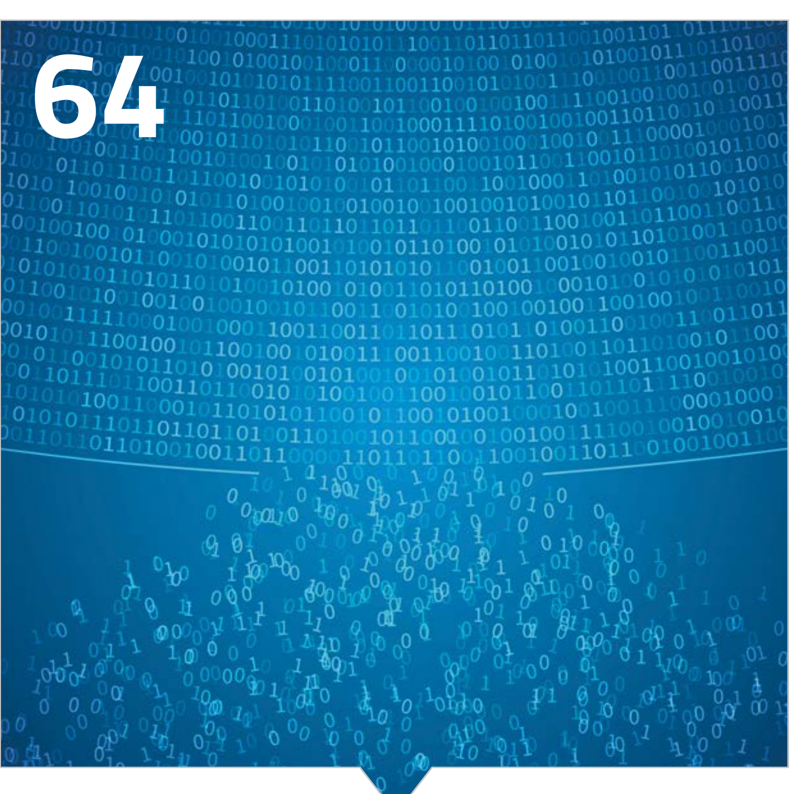
18 Software-Sicherheit im Java-Ökosystem
Prof. Dr.-Ing. Stefan Wagenpfeil

24 Unbeschwertes Dependency-Management für Java-Projekte
Dr. Jendrik Johannes

32 Java und GitOps
Thomas Michael

38 Java spricht mit Maschinen – Wie Open Source die Industrie wachrüttelt
Christofer Dutz

44 Trunk-Based Development und Pairing
Jörg Liedl



Memory Leaks mit OpenTelemetry diagnostizieren, beheben und vermeiden

Neuronale Netze: Wahre Intelligenz oder doch nur Nachahmung?

50 Klein, schnell, sicher: Wie man Spring Boot richtig in den Container packt
Sascha Selzer

58 Maven 4, Teil 2: Major Changes
Matthias Bünger

64 Diagnostik von Memory Leaks mit OpenTelemetry
Markus Meyer

70 Vom Sensor bis zum Dashboard: Echtzeit-Anomalieerkennung in Kälteanlagen
Eldar Sultanow & Matthias Petrich

76 Neuronale Netze in Java – Vom Gehirn inspiriert, vom Menschen gestaltet
Dr. Sandra C. Signore

82 Impressum/Inserenten

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

16. September 2025

Java 25

Zwei Jahre nach Java 21 ist es wieder so weit: Oracle hat heute JDK 25 als neue Long-Term Support (LTS) Version veröffentlicht – die Version für alle, die nicht jedes halbe Jahr ihre Produktionsumgebungen aktualisieren wollen (oder können). Das entsprechende Temurin-Release aus dem Eclipse-Adoptium-Projekt und die Versionen anderer Hersteller werden wie üblich in den nächsten Tagen folgen – alle basieren ja auf dem OpenJDK.

Java 25 führt viele Themen aus den letzten Releases fort, wie verbesserte Speicher- und Warmup-Effizienz sowie Concurrency.

Der Generational Mode des Shenandoah GC ist jetzt produktionsreif. Wir sind auch wieder ein bisschen Ahead-of-Time unterwegs: Mit JDK 24 wurde bereits AOT Class Loading & Linking (JEP 483) eingeführt, das Klassendatenstrukturen vorab lädt sowie verlinkt und als Load-and-Link-Cache speichert. Java 25 erweitert das AOT-Thema nun um AOT Method Profiling (JEP 514), das Profilinformationen aus Testläufen erzeugt, welche der Just In Time Compiler (JIT) beim Start für eine Beschleunigung des Warmups nutzen kann. Dazu gibt es noch die AOT Command-Line Ergonomics für ein möglichst einfach nutzbares Tooling, um die Profile aufzuzeichnen und zu nutzen.

Die Structured Concurrency API ist immer noch eine Preview (die fünfte), bietet aber bereits heute einen eleganten Weg, um hierarchische Task-Strukturen zu modellieren und Fehlerbehandlung zu vereinfachen. Zumindest für diejenigen, die zur Not ihren Code auch nochmal ändern würden, wenn es nach fünf Previews doch noch signifikante Änderungen gibt. Und einen neuen Rekord gibt es auch noch zu vermelden: Die Vector API bleibt für das 10. (in Worten: zehnte) Release hintereinander im Inkubator: Es gibt eine kleine API-Ergänzung, aber vor allem wird für den Zugriff auf native Funktionsbibliotheken jetzt auf die Foreign Function & Memory API gesetzt.

GraalVM für JDK 25 veröffentlicht

Wie gewohnt parallel zum Release des JDK hat Oracle die für das JDK 25 optimierte Version von GraalVM veröffentlicht.

Für die GraalVM war die „Ahead-of Time-Magie" ja von Anfang an ein zentrales Thema. Auch hier gibt es wieder Verbesserungen in der Ahead-of-Time Compilation für robustere und effizientere Native Images.

Die Java Vector API wird jetzt experimentell in effizienten Maschinencode übersetzt – für JIT und Native Images. Die Optimierung umfasst Operationen wie Load, Store, Arithmetik, Reduce, Compare sowie Blend und lässt sich mit dem Flag `-DOptimizeVectorAPI` de-/aktivieren.

Auf der Native Image-Seite gibt es gleich mehrere große Verbesserungen: Die „Whole-Program Sparse Conditional Constant Propagation" (WP-SCCP) ist jetzt standardmäßig aktiviert und reduziert die Größe nativer Binaries spürbar. Die Graal Neural Network (GraalNN) Modelle für Control-Split-Profile-Inferenz wurden verfeinert – mit zwei spezialisierten Varianten für die Level O2 (konservativ) und O3 (aggressiv). Das Ergebnis: 1 bis 3 % Performance-Verbesserung und über 1 % kleinere Binaries in O2. In O3 ist die GNN-basierte Inferenz standardmäßig aktiv.

Neu ist die erweiterte Unterstützung der Foreign Function & Memory (FFM) API in Native Image, einschließlich neuer Konfigurationssyntax, Arena.ofShared(), Tracing-Agent-Unterstützung und Verbesserungen für macOS- und Linux-AArch64. Die Vector API-Optimierungen sind ebenfalls in Native Images verfügbar – aktivierbar mit --add-modules jdk.incubator.vector und -H:+VectorAPISupport. Advanced Obfuscation (nur Oracle GraalVM) bietet experimentelle Symbol-Verschleierung zum Schutz vor Reverse Engineering. Und: Die Software Bill of Materials (SBOM) wird jetzt standardmäßig in Native Images eingebettet.

Auf der Sprachenseite: GraalJS aktiviert jetzt den ECMAScript 2025 Modus standardmäßig, GraalPy wurde auf Python 3.12.8 aktualisiert und unterstützt Arrow-Arrays zwischen Java, PyArrow und Pandas. GraalWasm implementiert WebAssembly SIMD (single instruction, multiple data) mit Hilfe der JDK Vector API für bessere Spitzen-Performance.

Und für die Polyglotten: Monocle (Software Fault Isolation) ist jetzt 4x schneller als der bisherige Ansatz – und macht Code-Ausführung in einer Sandbox deutlich performanter.

Die Liste ist noch deutlich länger, daher hier alles zum Nachlesen: *[1]*

24. September 2025

Quarkus 3.27

Auch das Quarkus-Team hat im September mit der Veröffentlichung von 3.27 eine neue LTS-Version bereitgestellt (die hier nicht ganz so langfristig angelegt ist, wie beim JDK, aber zumindest ein Jahr lang Bugfixes und Security-Updates bedeutet).

Feature-technisch ist 3.27 wie üblich die Summe der Releases seit dem letzten LTS Release 3.20 (also 3.21 bis 3.26, ganz neue Features wurden zeitgleich mit 3.28 veröffentlicht und tauchen dann im nächsten LTS Release auf).

Unter anderem wurde die Integration von Virtual Threads (Project Loom) auf SmallRye GraphQL ausgeweitet. Und es gibt einen Dev Assistant – Dein freundlicher Helfer in Quarkus' Dev UI. Vorsicht, Batterien (aka API Keys) sind natürlich nicht im Paket enthalten. Aber im Gegensatz zu manch anderem (Closed Source) Tool kann man auch den Zugriff auf ein lokales LLM konfigurieren. Ansonsten gibt es jede Menge kleinerer Verbesserungen für Persistenz, Security, Konfigurationsmanagement etc. *[2]*.

Mit dem inzwischen stetigen Strom an LTS Releases – neben der Microprofile-Basis, der engen GraalVM-Integration und seinen „Developer Productivity" Features – wird Quarkus zunehmend auch für klassische Enterprise-Organisationen interessant.

12. Oktober 2025

Java & AI: Liebe auf den zweiten Blick

Das Java AI-Ökosystem entwickelt sich prächtig: Java deckt heute große Teile der LLM- und Agentic-AI-Funktionalität ab, so dass niemand mehr für Prompting-, RAG- und Agent-Workloads ins Python-Ökosystem wechseln muss.

Besonders sichtbar sind Spring AI, LangChain4j und Jakarta EE. Das große Containerschiff unter den Java Frameworks bekommt mit „Jakarta Agentic Artificial Intelligence" ein neues Beiboot (um im Bild zu bleiben): Die neue Spezifikation soll erst mal nicht Teil der Plattform oder eines der Profile werden. An die Schaffung eines neuen „Jakarta EE Profile for AI" in der Zukunft wird allerdings schon gedacht. Ziele und Umfang des Projekts *[3]* sind an ein paar Stellen noch etwas kon- oder diffus formuliert. Es soll jedenfalls nicht das Rad neu erfunden werden (das war nie die Idee hinter „EE"), sondern es sollen eher mit dünnen Fassaden existierende Implementierungen unter anderem aus LangChain4j oder Spring AI „recycelt" werden.

Auch wenn es sich erstmal nur um eine freiwillig umzusetzende „Standalone API" handelt, wird es sich sicher trotzdem in die existierende Jakarta-Basis (CDI, JSON Binding, Security …) einfügen und somit dem aktuellen AI-Wildwuchs etwas von der Konsistenz und „Enterprisigkeit" (fand ich ein schönes Wort) der Java-Welt entgegensetzen können.

Die Basis wurde schon längst gelegt: Langchain4j, Java-Portierung von Langchain, ist ein von den großen Frameworks unabhängiger de-facto-Standard für den Umgang mit LLMs und wird stetig weiterentwickelt. Es greift zum Beispiel sehr schnell neue Modelle und Provider auf, und hat eine wachsende Agentic API, inspiriert vom Python-Vorbild, aber doch erkennbar Java-idiomatisch. Die Integration von Langchain4j ist nun keine neue Idee von Jakarta EE, sondern wird im SmallRye LLM Projekt bereits seit einiger Zeit erprobt. Das konkrete Projekt ist noch experimentell, aber die Small Rye Projekte sind Kern der MicroProfile-Implementierungen, also ist der Weg auch hier vorgezeichnet. Außerdem gibt es bereits „fertige" Integrationen von Langchain4j in Quarkus, das an dieser Stelle der Microprofile Spezifikation erst mal enteilt ist, und für Spring AI.

Womit wir beim nächsten „Enterprise Java" Framework wären: Spring ist schon länger in AI-Gewässern unterwegs, Version 1.1 von Spring AI ist gerade fertiggestellt worden. Natürlich mit Unterstützung für MCP Clients (über SSE / Streamable http, und – wer mag – auch mit STDIO für lokale Integration). Prompt Caches sind ein großes Ding, ebenso Support für den Thinking Mode und den „Reasoning Content" entsprechender Modelle, um ihnen schön beim „Denken" zuschauen zu können. Das alles sollte möglichst nahtlos in die gewohnte Spring „Developer Experience" integriert sein, um AI-Komponenten möglichst nahtlos in die Geschäftslogik von „Enterprise Applications" einbetten zu können. Alle Details hier zum Nachlesen *[4]*, und hier *[5]* findet sich die Ankündigung der „Spring AI Community".

Und bevor ich's vergesse (wobei es aber bestimmt noch ganz viel im Java AI-Bereich gibt, das ich noch gar nicht kenne oder wieder vergessen habe): Wer gerne komplexere Workflows für AI Agents entwirft und es entsprechend eher mit Langgraph als Langchain hält: Auch davon gibt es seit einiger Zeit eine (unabhängige) LangGraph4j-Portierung. Das Projekt macht einen aktiven Eindruck, und scheint genau der richtige Startpunkt zu sein, wenn man nicht das Java Ökosystem verlassen möchte, nur um „Agentic AI zu machen". Aber ich übernehme keine Gewähr, denn ausprobiert habe ich es noch nicht.

20. November 2025

Spring Framework 7.0 und Spring Boot 4.0 sind da

Und nochmal zu Spring: Die neue Generation ist komplett. Am 13. November wurde das Spring Framework 7.0 veröffentlicht, gefolgt von Spring Boot 4.0 am 20. November. Java 25 (LTS) ist im Fokus, aber die Minimum-Version bleibt Java 17 (LTS) für maximale Kompatibilität.

Weitere Abhängigkeiten wurden auf die neuesten Versionen angehoben: Jakarta EE 11 (mit Servlet 6.1, JPA 3.2, Bean Validation 3.1), Jackson 3.0 (mit deprecated Support für 2.x), Kotlin 2.2 und JUnit 6.0. Was gibt es auf der Feature-Seite zu vermelden?

Spring Framework 7 führt eine elegantere programmatische Bean-Registrierung über den neuen BeanRegistrar ein und integriert zentrale Resilience-Funktionen wie Retry und Concurrency-Limits direkt ins Core-Framework. Für Messaging kommt ein moderner JmsClient hinzu, und REST-Anwendungen profitieren von nativem API-Versioning sowie erweiterten HTTP-Client-Konfigurationen inklusive eines neuen RestTestClient.

Spring Boot 4 modularisiert die Codebasis vollständig und liefert dadurch deutlich kleinere, fokussiertere JARs für Microservice- und

Container-Deployments. Die neuen API-Versioning- und HTTP-Client-Features des Frameworks sind in Boot nahtlos integriert. Außerdem ist Spring Observability tiefer integriert, inklusive Micrometer- und OpenTelemetry-Verbesserungen, und die Native-Image-Unterstützung (GraalVM / AOT) wurde weiter verbessert. Das immer weiter integrierte Spring AI hatten wir schon oben thematisiert.

Ein interessanter Aspekt ist vielleicht noch die Annotation-basierte Null-Safety. Grundsätzlich gibt es die in Spring seit längerem. Mit der neuen Version wird aber durchgehend auf JSpecify gesetzt, ein herstellerübergreifendes Projekt, um ungültige Null-Referenzen auszumerzen.

25. November 2025

Der heilige Graal – kostenlos für alle

Oracle hat eine wichtige Lizenz-Änderung angekündigt: Die Oracle GraalVM ist ab sofort unter der neuen „GraalVM Free Terms and Conditions" (GFTC) verfügbar und kann damit kostenlos genutzt werden – auch produktiv. Die Lizenz erlaubt zudem die Weiterverteilung der unveränderten Binaries, sofern sie nicht gegen Entgelt erfolgt. Damit fallen viele der bisherigen kommerziellen Hürden weg, die GraalVM Enterprise lange unattraktiv gemacht haben.

Die komplette Ankündigung steht hier *[6]*. Und falls noch jemand außer mir sich den Kopf zerbricht, warum zum Henker sie dort nur alte Relases (17, 20) explizit und prominent erwähnen, die letzten LTS Releases 21 und 25 aber nicht (beziehungsweise nur in den Details als „subsequent releases"): Es scheint wohl so, dass es auf diese Weise zumindest für Juristen klar und eindeutig ist. Und wir erinnern uns, wer bei Oracle das Zepter schwingt. Aber ich will eigentlich nicht meckern – die neue Lizenz ist eine tolle Sache!

In dem Oracle Blogeintrag wird auch Project Galahad explizit genannt: Galahad soll den Graal übergeben beziehungsweise präziser: die Java JIT Compiler-Anteile der GraalVM langfristig in die OpenJDK-Codebasis integrieren. Das Projekt ist eigentlich schon 2023 gestartet, aber vielleicht nimmt es jetzt neue Fahrt auf.

2. Dezember 2025

Ausblick: JDK 26 im März 2026

JDK 25 LTS ist längst draußen, Zeit einen Blick auf Release 26 (non-LTS) zu werfen.

Bisher sind drei JEPs bestätigt: Die Entfernung der Java Applet API (was waren noch mal Applets?), HTTP/3 Support für die Client API (opt-in) und Performance-Verbesserungen für den G1 Garbage Collector durch reduzierte Synchronisation zwischen Application- und GC-Threads, was den Durchsatz spürbar erhöhen soll.

Interessanter wird es bei den wahrscheinlichen Kandidaten, insbesondere Preview- oder Incubator-Features aus JDK 25. Dazu gehören Structured Concurrency (aktuell ja in der fünften Preview), Primitive Types in Patterns, instanceof und switch (dritte Preview) sowie die PEM Encodings of Cryptographic Objects. Auch die Vector API wird vermutlich noch eine elfte Inkubator-Runde drehen und den Rekord weiter in die Höhe schrauben.

Spannend sind auch die neuen Language Features aus JDK 25, die möglicherweise finalisiert werden: Module Import Declarations, Compact Source Files mit Instance Main Methods und Flexible Constructor Bodies könnten den Sprung von Preview zu Standard schaffen.

Quellen

[1] *https://www.graalvm.org/release-notes/JDK_25/*
[2] *https://quarkus.io/blog/quarkus-3-27-released/*
[3] *https://jakarta.ee/specifications/agentic-ai*
[4] *https://spring.io/blog/2025/11/12/spring-ai-1-1-GA-released*
[5] *https://spring.io/blog/2025/10/07/spring-ai-community-announcement*
[6] *https://blogs.oracle.com/graal/graalvm-free-license*

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").

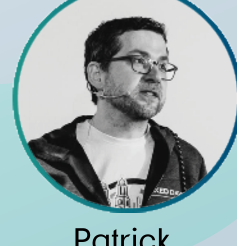
im EUROPA PARK in Rust

10. – 12. MÄRZ 2026

Die Java-Community-Konferenz

Keynote-Speaker

Venkat Subramaniam



Patrick Baumgartner

Bruno Borges

Ed Burns

Hanno Embregts

Milena Fluck

Miriam Greis

Ivar Grimstad

Sophie Küster

Josh Long

Thomas Much

Nicolai Parlog

Marc Philipp

Mario-Leander Reimer

Henning Schwentner

Marit van Dijk

Freut euch auf diese und viele weitere Speaker sowie jede Menge Wissensaustausch, Networking und noch mehr Highlights.

JAVALAND.EU

Präsentiert von:

Veranstalter:

Java 25 – Die Neuerungen der LTS-Version

Falk Sippach, embarc Software Consulting GmbH

Im September 2025 ist passend zur Jahreszahl das OpenJDK 25 erschienen. Für dieses Release werden die meisten Anbieter wieder Distributionen mit Long-Term-Support (LTS) anbieten. In diesem Artikel werfen wir einen Blick auf die Neuerungen. Einige der Features wurden abgeschlossen, andere sind weiterhin als Inkubator oder Preview dabei. Und es gibt auch neue Themen.

Für einen ersten Überblick ist wie immer die Projektseite des OpenJDK *[1]* ein guter Startpunkt – 18 JEPs (JDK Enhancement Proposals) sind dort diesmal gelistet:

- 470: PEM Encodings of Cryptographic Objects (Preview)
- 502: Stable Values (Preview)
- 503: Remove the 32-bit x86 Port
- 505: Structured Concurrency (Fifth Preview)
- 506: Scoped Values
- 507: Primitive Types in Patterns, instanceof, and switch (Third Preview)
- 508: Vector API (Tenth Incubator)
- 509: JFR CPU-Time Profiling (Experimental)
- 510: Key Derivation Function API
- 511: Module Import Declarations
- 512: Compact Source Files and Instance Main Methods
- 513: Flexible Constructor Bodies
- 514: Ahead-of-Time Command-Line Ergonomics
- 515: Ahead-of-Time Method Profiling
- 518: JFR Cooperative Sampling
- 519: Compact Object Headers
- 520: JFR Method Timing & Tracing
- 521: Generational Shenandoah

Das sind wieder etwas weniger als der Rekord von 24 JEPs im März 2025. Aber mit diesem Release setzt sich der Trend fort, Java durch gezielte Verbesserungen moderner, effizienter und leistungsfähiger zu machen. Neben Änderungen an der Sprache gibt es auch Optimierungen in der JVM, neue APIs sowie diverse produktivitätssteigernde Fortschritte für Entwicklerinnen und Entwickler.

Leichtere Entwicklung mit Java – für Einsteiger und Profis

Trotz seiner 30 Jahre zieht Java auch weiter viele Programmieranfänger an. Die JEPs 511 (Module Import Declarations) und 512 (Compact Source Files and Instance Main Methods) helfen aber nicht nur Anfängern, sie machen auch erfahrenen Entwicklern das Leben einfacher. Beide JEPs werden nun nach mehreren vorangegangenen Previews finalisiert.

Durch die Module Import Declarations lassen sich alle exportierten Packages eines Moduls auf einmal importieren. Das reduziert Boilerplate-Code bei den Importen und vereinfacht die Wiederverwendung modularer Bibliotheken (ohne dass der eigene Code selbst modularisiert sein muss). Das ist praktisch fürs Prototyping und für klare, kurze Quelltexte. In der Finalisierung hat sich nichts mehr geändert, aber beim letzten Preview gab es zwei Erweiterungen. Einerseits wurden Beschränkungen bei den transitiven Abhängigkeiten vom Modul java.se (eine Art Aggregator-Modul ohne eigene Packages/Klassen) zu `java.base` aufgehoben. Dadurch kann man nun mit dem Import dieses einen Moduls die gesamte API von Java SE importieren. Außerdem ist es jetzt möglich, dass sogenannte Type-Import-on-Demand-Deklarationen (zum Beispiel `import java.util.*`) vorherige Modul-Import-Deklarationen überdecken. Wenn beispielsweise die Module java.base und `java.sql` importiert werden, gäbe es eine Unklarheit beim Verwenden der Klasse Date. Die gibt es als `java.util.Date` und als `java.sql.Date`. Durch die On-demand-Deklaration `import java.util.*` wird in dem Fall `java.util.Date` verwendet.

Zum JEP 512 (Compact Source Files and Instance Main Methods) gab es bereits vier Previews und der Name hat sich immer mal wieder geändert. Es werden kompakte Ein-Datei-Programme mit einer Instanz-Main-Methode ermöglicht, sodass kleine Tools, Skripte oder Lernbeispiele ohne unnötige „Zeremonie" startklar sind. Das führt zu einem schnelleren Einstieg für Anfänger und zu weniger Hürden für erfahrene Entwickler beim Skizzieren, Ausprobieren und Automatisieren. In der Finalisierung gab es nochmal drei Änderungen:

- Die Klasse `IO` liegt nun in Package `java.lang` und wird damit immer automatisch importiert.
- Die statischen Methoden von IO werden nicht mehr implizit in Compact Source Files importiert, Aufrufe müssen qualifiziert werden (zum Beispiel `IO.println("Hello, world!")`) oder explizit statisch importiert werden.
- Die Implementierung von IO stützt sich jetzt auf `System.out/System.in` statt auf `java.io.Console`.

Unter Beibehaltung der bestehenden Java-Toolchain und nicht mit der Absicht, einen separaten Dialekt für Java einzuführen, lassen sich in Compact Source Files mit der vereinfachten main-Methode (Klassen-Deklaration entfällt, …) sehr simple, skriptartige Programme erstellen. Dazu kommt die leichtere Verwendung von Standardein- und -ausgabe durch die neuen statischen Methoden `print()`, `println()` und `readln()` der Klasse `java.io.IO`. *Listing 1* zeigt ein Beispiel.

```
// > java Main.java
void main() {
    IO.println("Hello, World!");
}
```

Listing 1: Instance Main Method

Flexible Konstruktor-Inhalte werden ebenfalls finalisiert

Dank JEP 513 (Flexible Constructor Bodies) dürfen in Konstruktoren Anweisungen nun bereits vor einem expliziten weiteren Konstruktoraufruf (`super()` oder `this()`) im sogenannten Prolog erscheinen. Die Anweisungen im Prolog unterliegen aber einigen Einschränkungen. Es ist insbesondere keine Verwendung von `this` erlaubt. Dazu zählen weder das Lesen von (noch nicht initialisierten) Feldern noch das Aufrufen von Methoden, die auf den internen Zustand zugreifen könnten. Nicht erlaubt sind zum Beispiel `this.hashCode()`,

`System.out.println(this)` und `var x = this.feld` (lesender Zugriff). Es gibt genau eine Ausnahme: einfache Zuweisungen auf eigene Felder ohne Initializer sind erlaubt (zum Beispiel `this.feld = …`). Diese Zuweisungen dürfen aber nicht innerhalb von Lambdas oder inneren Klassen stehen.

Im Prolog können jedoch Parameter validiert beziehungsweise transformiert werden (zum Beispiel das Aufsplitten des Namens in Vor- und Nachnamen). Außerdem kann im Konstruktor der Oberklasse dann auf bereits initialisierte Felder der Subklasse zugegriffen werden. *Listing 2* zeigt ein solches Beispiel, in dem nicht nur Felder validiert oder transformiert werden, sondern in dem auch in der Superklasse bereits auf die Werte von Feldern der Subklasse zugegriffen wird, obwohl der Subklassen-Konstruktor noch nicht beendet ist (Aufruf von `show()` im Konstruktor Person).

Das macht viele Konstruktoren natürlicher (zum Beispiel fail-fast-Validierung) und erhöht die Sicherheit, weil Felder eines Subtyps fertig initialisiert sein können, bevor Superklassen-Konstruktoren (gegebenenfalls via überschreibbare Methoden) darauf zugreifen. Insgesamt erhöhen sich sowohl die Lesbarkeit als auch die Performance, da unnötige Aufrufe bei der Validierung oder Weiterverarbeitung von Konstruktor-Parametern vermieden werden. Dieses Feature wird nun finalisiert, zum letzten Preview gab es keine Änderungen.

APIs für Virtual Threads

Für die in Java 21 finalisierten virtuellen Threads wurden zwei weitere APIs eingeführt: Structured Concurrency und Scoped Values. Sie ermöglichen eine effizientere, sichere und besser strukturierte Nebenläufigkeit.

Scoped Values werden nun mit den JEP 506 finalisiert. Ein `ScopedValue<T>` ist eine Alternative zu `ThreadLocal<T>`, die speziell für Virtual Threads optimiert wurde. Er ermöglicht es, unveränderbare Werte sicher über einen Code-Bereich hinweg zu propagieren, ohne dabei die Nachteile von `ThreadLocal` zu übernehmen. Diese sind bei Virtual Threads problematisch, da Speicherplatz potenziell nicht automatisch freigegeben wird. Da Virtual Threads sehr leichtgewichtig sind und tausende parallele Aufgaben ausgeführt

```
void main() {
     new Employee("Duke Java", 42, "C-7");
}

class Person {
  int age;
  String firstname, lastname;

  void show() {
     IO.println(this);
  }

  Person(String name, int age) {
     // Prolog
     String[] names = name.split("\\s" );
     this.firstname = names[0];
     this.lastname = names[1];
     if (age < 0) { // fail-fast im Prolog

        throw new IllegalArgumentException("age must be greater than or equal to zero");
     }
     this.age = age;
     super();
     // Epilog
     show(); // ruft ggf. überschriebenes show() in Subklasse auf!
  }

  @Override
  public String toString() {
     return String.format("Name=%s %s; Age=%d", firstname, lastname, age);
  }
}

class Employee extends Person {
  String officeId; // kein Initializer → im Prolog setzbar

  Employee(String name, int age, String officeId) {
     // Prolog
     this.officeId = Objects.requireNonNull(officeId); // Prolog: Feld der eigenen Klasse setzen
     super(name, age);  // erst jetzt Superkonstruktor
     // Epilog: weiterer Code erlaubt, Objekt ist konsistent
  }

  @Override
  public String toString() {
     return super.toString() + "; OfficeId=" + officeId;
  }
}
```

Listing 2: Flexible Constructor Bodies

```java
public class ScopedValueExample {
private static final ScopedValue<String> USER_ID = ScopedValue.newInstance();

     public static void main(String[] args) {
           ScopedValue.where(USER_ID, "User-123").run(() -> {
                 processRequest();
           });
     }

     static void processRequest() {
           System.out.println("Processing for user: " + USER_ID.get()); // Gibt "User-123" aus
     }
}
```

Listing 3: Scoped Values

```java
// --enable-preview beim Kompilieren und Starten
record Response(String user, int order) {}

Response handle() throws InterruptedException {
  try (var scope = StructuredTaskScope.<Object, Void>open()) {
     var user  = scope.fork(this::findUser);    // liefert String
     var order = scope.fork(this::fetchOrder); // liefert Integer

     scope.join(); // wartet auf alle; wirft eine Exception bei einem Fehler und cancelt die anderen

     return new Response((String) user.get(), (Integer) order.get());
  }
}

// Platzhalter:
String findUser()  { return "alice"; }
int     fetchOrder() { return 42; }
```

Listing 4: Structured Concurrency

werden können, würden `ThreadLocal`-Variablen schlecht skalieren. Denn jeder Thread würde seinen eigenen Wert speichern. Bei `ScopedValues` sind die Inhalte nur innerhalb eines bestimmten Bereichs gültig und verursachen dadurch keine Speicherprobleme. `ScopedValues` sind sicherer, performanter und expliziter in ihrer Lebensdauer. Durch ihre Unveränderbarkeit können keine ungewollten Nebenwirkungen auftreten. *Listing 3* zeigt ein Beispiel, bei dem `USER_ID` nur innerhalb des `run()`-Blocks gültig ist und dann auch automatisch aufgeräumt wird.

Zum letzten Preview gibt es in JEP 506 nur noch eine kleine Änderung: Die Methode `ScopedValue.orElse()` akzeptiert keine `null`-Argumente mehr.

Die Structured Concurrency (JEP 505) kommt ebenfalls aus dem Umfeld von Virtual Threads und sorgt dafür, dass nebenläufige Tasks innerhalb eines klar definierten Scopes gestartet und beendet werden. Das hilft einerseits bei der Fehlerbehandlung, denn beispielsweise, wenn eine Aufgabe fehlschlägt, werden alle anderen koordiniert abgebrochen. Andererseits hilft es bei der Lesbarkeit beziehungsweise Wartbarkeit, da so explizite Nebenläufigkeitsstrukturen sichtbar sind. Alternativ konnten Entwickler für diesen Zweck bisher die Parallel Streams, den ExecutorService oder reaktive Programmierung einsetzen. Alles sehr mächtige Ansätze, die aber gerade einfache Umsetzungen unnötig kompliziert und fehleranfällig machen.

Structured Concurrency behandelt Gruppen von zusammengehörigen Aufgaben als eine Arbeitseinheit, wodurch die Fehlerbehandlung sowie das Abbrechen der Aufgaben vereinfacht und die Zuverlässigkeit sowie die Beobachtbarkeit erhöht werden. *Listing 4* zeigt ein Beispiel. Beide Sub-Tasks laufen parallel (typischerweise als Virtual Threads) los. Wenn einer fehlschlägt, bricht der Scope ab und unterbricht den anderen Task automatisch, `join()` wirft dann eine `FailedException`. Waren beide erfolgreich, sind nach `join()` beide Ergebnisse verfügbar.

Zum letzten Preview hat sich nochmal einiges an der API verändert:

- statische Fabrikmethoden zum Öffnen eines Scopes (`StructuredTaskScope.open(...)`) statt öffentlicher Konstruktoren
- die Standardvariante ist open() ohne Parameter: Der Scope gilt als erfolgreich, wenn alle Sub-Tasks erfolgreich sind. Schlägt einer fehl, wird der Scope abgebrochen (Short-Circuit).
- Joiner-Konzept: Über `open(Joiner...)` werden alternative Abschluss-Policies (zum Beispiel „erstes erfolgreiches Ergebnis" oder „sammle alle Ergebnisse") definiert und `join()` liefert dann direkt das passende Ergebnis.
- `join()` wirft jetzt Exceptions selbst (zum Beispiel `FailedException`, `TimeoutException`) und liefert gegebenenfalls einen Rückgabewert. Das frühere Pattern `join().throwIfFailed()` entfällt.
- Es gibt eine optionale Konfiguration beim Öffnen (`open(joiner, cfg -> cfg.withTimeout(...).withThreadFactory(...))`) für Timeouts, `ThreadFactory` und Scope-Name – hilfreich für Monitoring und Tuning.

- Observability wurde erweitert: Thread-Dumps (JSON) zeigen die Task-/Sub-Task-Bäume inklusive Scope-Beziehungen.

Listing 5 zeigt einige Beispiele für alternative TaskScope-Strategien.

Aufgeschobene Initialisierung bei unveränderbaren Variablen

Der JEP 502 führt mit Stable Values Objekte ein, die genau einmal gesetzt werden und danach unveränderlich sind. Die JVM behandelt ihren Inhalt wie eine echte Konstante und kann dieselben Optimierungen (zum Beispiel Constant Folding) anwenden wie bei `final`-Feldern, sie bieten aber eine höhere Flexibilität in Bezug auf den Initialisierungszeitpunkt. Damit bekommt Java eine „Deferred Immutability", die Start- und Aufwärmzeiten verkürzt, ohne Thread-Sicherheitsrisiken einzugehen.

Die Ziele sind ein schnellerer Start (keine monolithische Initialisierung aller Komponenten mehr), Entkopplung von Erzeugung und Initialisierung eines unveränderlichen Werts ohne nennenswerte Performanceeinbußen, garantierte At-Most-Once-Initialisierung auch in hoch-parallelem Code sowie das Zugänglichmachen der Konstanten-Optimierungen für Anwendungscode (analog zu JDK-internen Code).

Stable Values sind eine reine Library-API, es wird kein neues Schlüsselwort geben. Außerdem gibt es keine Änderung an der `final`-Semantik, bestehender Code bleibt unberührt. Bisher müssen `final`-Felder eager (sofort) initialisiert werden – die Initialisierung eines Loggers oder eine Datenbank-Connection bremst so den Programmstart aus. Workarounds wie Lazy Holder, Double-Checked Locking oder `ConcurrentHashMap.computeIfAbsent` sind entweder eingeschränkt, fehleranfällig oder verhindern JIT-Optimierungen („Just in Time"). Stable Values schließen die Lücke zwischen strenger Immutability und flexibler Lazy-Initialization.

In *Listing 6* wird der Lambda-Ausdruck in `orElseSet` garantiert genau einmal ausgeführt, selbst wenn mehrere Virtual Threads gleichzeitig `logger()` aufrufen. Danach kann der JIT alle Zugriffe optimieren (zum Beispiel mit Constant Folding).

Die API von Stable Values zeigt *Tabelle 1*. Stable Values kombinieren Lazy und Fast Initialization. Risiken bestehen praktisch nur dort, wo

```
// Erstes erfolgreiches Ergebnis gewinnt
StructuredTaskScope.open(
                  Joiner.<T>anySuccessfulResultOrThrow(),
                  cfg -> cfg.withTimeout(java.time.Duration.ofSeconds(2)))

// Alle oder keiner
StructuredTaskScope.open(Joiner.<T>allSuccessfulOrThrow())

// Erfolgreiche und fehlgeschlagene Tasks unterscheiden
StructuredTaskScope.open(Joiner.<T>awaitAll())

// Früher Abbruch per Prädikat
StructuredTaskScope.open(
                  Joiner.<String>allUntil(st ->
                        st.state() == Subtask.State.SUCCESS && st.get().contains("OK")))
```

Listing 5: Structured Concurrency – alternative Strategien

```
class OrderController {
  private final StableValue<Logger> logger = StableValue.of();

  private Logger logger() {
     return logger.orElseSet(() -> Logger.create(OrderController.class));
  }
}
```

Listing 6: Deklaration und Initialisierung eines Stable Value

Methode	**Zweck**	Threadsafe?	JIT optimierbar?
StableValue.of()	Leere Hülle erzeugen	👍	👍
orElseSet (Supplier<T>)	Inhalt holen oder einmalig setzen	👍	👍
StableValue.supplier (Supplier<T>)	Kombiniert StableValue + Supplier	👍	👍
StableValue.list (int, IntFunction<T>)	Pool/Liste lazily erzeugen	👍	👍

Tabelle 1: API von Stable Values

Reflection weiterhin `final`-Felder ändern darf. Aber das ist bereits heute eine generelle JVM-Beschränkung und Stable Values stellen somit eine sichere Variante dar, diese Beschränkung zu umgehen. Sie passen hervorragend zu Virtual Threads (Loom) und Structured Concurrency (JEP 505). Theoretisch können nun Millionen Threads ohne teure SynchronisationsTricks auf „Deferred Constants" zugreifen.

Pattern Matching mit Primitive Type Patterns

Pattern Matching ist nun auch schon einige Jahre in der Entwicklung. Hier wurden immer wieder Teile abgeschlossen, zuletzt in Java 22 die Unnamed Variables & Patterns (JEP 456). Bei den nun im dritten Preview befindlichen „Primitive Types in Patterns, instanceof, and switch" (JEP 507) geht es um eine Erweiterung, sodass primitive Datentypen wie `int`, `byte` und `double` in allen Pattern-Kontexten (beim `instanceof` und im `switch`) verwendet werden dürfen. Entwickler haben dadurch weniger Limitierungen sowie Sonderfälle und können primitive und Referenzdatentypen auch im Kontext von Type Patterns oder als Komponenten in Record Patterns austauschbar verwenden. Seit dem letzten Preview gibt es keine Änderungen, die JDK-Entwickler wollen aber weiteres Feedback sammeln.

Beim Pattern Matching geht es darum, bestehende Strukturen mit Mustern abzugleichen, um komplizierte Fallunterscheidungen effizient und wartbar implementieren zu können. Ein Pattern ist dabei eine Kombination aus einem Prädikat, das auf die Zielstruktur passt, und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen und damit extrahiert. Die Intention des Pattern Matching ist die Destrukturierung von Datenobjekten, also das Aufspalten in die Bestandteile und das Zuweisen in einzelne Variablen zur weiteren Bearbeitung. Mit `instanceof` und `switch` können wir also überprüfen, ob ein Objekt von einem bestimmten Typ ist, und wenn ja, dieses Objekt einer Variable diesen Typs zuweisen und diese Variable in dem folgenden Programmpfad benutzen. Das funktionierte bisher aber nur mit Objekten und ließ sich nicht mit primitiven Datentypen kombinieren. Einzig im `switch` ließen sich bereits Variablen der primitiven Typen `byte`, `short`, `char` sowie `int` gegen Konstanten matchen und konnten sogar mit den neueren Type Patterns kombiniert werden *(siehe Listing 7)*.

```
int grade = 7;
String result = switch (grade) {
  case 1, 2 -> "very good or good";
  case 3, 4 -> "satisfactory or sufficient";
  case 5, 6 -> "poor or deficient";
  case Integer i -> "Undefined grade: " + i;
};
System.out.println(result);
```

Listing 7: Variablen mit primitiven Datentypen

JEP 507 verbessert nun die Typprüfung, Performance und Lesbarkeit in Java und macht Pattern Matching konsistenter, indem es primitive Typen direkt unterstützt. Dies reduziert überflüssiges Autoboxing und vereinfacht den Umgang mit primitiven Datentypen in switch-Anweisungen. Wie am Beispiel in *Listing 8* zu sehen ist, können Entwickler in Zukunft ganz einfach prüfen, ob ein ganzzahliger Wert in den Wertebereich eines byte passt.

```
private static String checkByte(int value) {
  if (value instanceof byte b) {
     return "byte b = " + b;
  } else {
     return "kein byte: " + value;
  }
}

System.out.println(checkByte(127)); // b = 127
System.out.println(checkByte(128)); // kein byte: 128
```

Listing 8: Prüfung auf primitiven Datentyp

Weitere Themen

Neben diesen prominenten und auch für viele Entwickler relevanten Neuerungen gibt es auch wieder einige kleine Änderungen. So wurden mit dem JEP 503 der Quellcode und Build-Support für den 32-Bit-x86-Port (nach Deprecation in JDK 24 via JEP 501) dauerhaft entfernt. Ziel ist es, neue Features nicht länger mit 32-Bit-Fallbacks bedienen zu müssen, alle 32-Bit-x86-Sonderpfade zu streichen und Build-/Test-Infrastruktur zu vereinfachen. Nicht betroffen sind 32-Bit-Support anderer Architekturen und frühere JDK-Releases. Der Pflegeaufwand stand in keinem Verhältnis zum Nutzen. Die Entfernung beschleunigt die Weiterentwicklung, zum Beispiel bei Loom, FFM API, Vector API, GC-Barrier-Expansion und so weiter.

Vector-API bleibt Inkubator

Die Vector-API ist der Langläufer der letzten Jahre. Sie ist nun schon das zehnte Mal als Inkubator enthalten und taucht seit Java 16 regelmäßig in den Releases auf. Es geht dabei um die Unterstützung der modernen Möglichkeiten von SIMD-Rechnerarchitekturen mit Vektorprozessoren. Single Instruction Multiple Data (SIMD) lässt viele Prozessoren gleichzeitig unterschiedliche Daten verarbeiten. Durch die Parallelisierung auf Hardware-Ebene verringert sich beim SIMD-Prinzip der Aufwand für rechenintensive Schleifen.

Der Grund für die lange Inkubationsphase der Vector-API wird in den Zielen des JEP 508 *[2]* erklärt:

Alignment with Project Valhalla — The long-term goal of the Vector API is to leverage Project Valhalla's enhancements to the Java object model. Primarily this will mean changing the Vector API's current value-based classes to be value classes so that programs can work with value objects, i.e., class instances that lack object identity.

Man wartet also auf die Reformen am Typsystem. Bei Java ist es aktuell zweigeteilt mit den primitiven und den Referenztypen (Klassen). Die primitiven Datentypen wurden ursprünglich zur Performanceoptimierung eingeführt, haben aber im Handling entscheidende Nachteile. Referenztypen sind auch nicht immer die beste Wahl, insbesondere was die Effizienz und den Speicherverbrauch angeht. Es braucht etwas dazwischen, was sich so schlank und performant wie primitive Datentypen verhält, aber auch die Vorzüge von selbst zu erstellenden Referenztypen in Form von Klassen kennt. Schon bald könnten daher aus dem Inkubator Projekt Valhalla die Value Types

(haben keine Identität) und Universal Generics (`List<int>`) ins JDK übernommen werden. Der JEP 401 (Value Classes and Objects) hat es leider wieder nicht geschafft, aber vielleicht sehen wir ihn ja dann im Jahr 2027. Dementsprechend werden wir die Vector-API wohl auch noch einige Releases als Inkubator- beziehungsweise dann hoffentlich bald als Preview-Feature wiedersehen.

Schneller Start und Warmup

Der JEP 514 (Ahead-of-Time Command-Line Ergonomics) wird das Erzeugen von Ahead-of-Time-Caches (AOT) vereinfachen. Diese können den Start von Java-Anwendungen deutlich beschleunigen. Für gängige Fälle soll jetzt ein einziger Schritt genügen, der Trainingslauf und Cache-Erstellung kombiniert. Ziel ist es, die bislang nötige Zwei-Phasen-Prozedur abzulösen – ohne an Ausdrucksstärke zu verlieren und ohne neue AOT-Optimierungen einzuführen. Heute braucht man zwei java-Aufrufe und bleibt mit einer temporären *.aotconf-Datei zurück; künftig entfällt dieser Ballast, was unter anderem Frameworks wie JRuby bei eigenen Trainingsläufen entgegenkommt. Für Spezialfälle bleiben die expliziten AOT-Modi und Konfigurationsoptionen aber weiterhin verfügbar.

Mit dem JEP 515 (Ahead-of-Time Method Profiling) wird es eine schnellere Warmup-Phase durch Profile aus dem AOT-Cache geben. Die JVM kann beim Start Methoden-Ausführungsprofile aus einem früheren Lauf sofort laden, sodass der JIT direkt die voraussichtlich relevanten Methoden kompiliert, statt zunächst Profile sammeln zu müssen. Die Profile werden in einem Training Run erzeugt und über den bestehenden AOT-Cache bereitgestellt. Dafür sind keine Codeänderungen und keine neuen Workflows nötig. Der Hintergrund ist, dass das Warmup heute viel Zeit kostet, weil die HotSpot VM erst während der Produktion herausfindet, welche Methoden relevant sind. Durch das Verlagern des Profilings in einen Training Run, erreicht die Anwendung in Produktion schneller ihre Spitzenleistung. Ein Webservice, der sonst erst nach einigen Minuten voll performant ist, kann mit vorab aufgezeichneten Profilen schon beim Start die kritischen Request-Pfade kompilieren und so sofort schneller unter Last reagieren.

Verbesserungen bei Observability und Profiling

Der JDK Flight Recorder wird mit dem experimentellen JEP 509 (JFR CPU-Time Profiling) so erweitert, dass er auf Linux den CPU-Zeit-Timer des Kernels nutzt und damit präzisere CPU-Profile erstellt – auch dann, wenn Java-Code gerade native Bibliotheken ausführt. Bisher stützte sich JFR (JDK Flight Recorder) vor allem auf einen „Execution Sampler“, der in festen Echtzeit-Intervallen (zum Beispiel alle 20 ms) Java-Stacks zieht, dabei aber Threads in nativem Code übersieht, Proben verpassen kann und nur einen Teil der Threads erfasst. Mit CPU-Zeit-Profiling bildet JFR die tatsächlich verbrauchten CPU-Zyklen ab und vermeidet unsichere, interne Schnittstellen, wie sie manche externen Tools verwenden. Ein Sortieralgorithmus verbringt zum Beispiel seine gesamte Laufzeit auf der CPU und taucht entsprechend stark im CPU-Profil auf, eine Methode, die meist auf Daten aus einem Socket wartet, beansprucht hingegen kaum CPU-Zeit und erscheint dadurch nur selten im Profil.

Durch den JEP 518 (JFR Cooperative Sampling) soll der JFR stabiler werden, indem er Java-Thread-Stacks nur noch an Safepoints traversiert und dabei den bekannten Safepoint-Bias so weit wie möglich reduziert. Bisher nahm der JFR in festen Intervallen (z. B. alle 20 ms) asynchron Samples auch außerhalb von Safepoints. Das erforderte Heuristiken zum Stack-Parsing, die ineffizient waren und im Fehlerfall sogar JVM-Crashes auslösen konnten (etwa bei gleichzeitigem Class-Unloading). Künftig wird auch die Profilerstellung der Wanduhrzeit (wall-clock time) weiter unterstützt, aber mit einem robusteren Verfahren, das Genauigkeit und Ausfallsicherheit besser ausbalanciert.

Der JEP 520 (JFR Method Timing & Tracing) erweitert den JDK Flight Recorder um eine Bytecode-Instrumentierung, die jeden ausgewählten Methodenaufruf exakt misst und mit Stacktrace aufzeichnet – im Gegensatz zu stichprobenbasierten Profilen. Methoden lassen sich ohne Codeänderungen zielgenau per Kommandozeile, Konfigurationsdatei, jcmd oder JMX auswählen. Nicht vorgesehen sind das Aufzeichnen von Argumenten/Feldwerten, das Tracen nicht-bytecodierter Methoden (zum Beispiel `native`, `abstract`) sowie das gleichzeitige Instrumentieren sehr vieler Methoden. In solchen Fällen bleibt Sampling der richtige Ansatz. Um zum Beispiel lange Startzeiten zu analysieren, können statische Initialisierer ausgewählter Klassen gezielt getraced werden. So wird sichtbar, welche Initialisierung sich auf später verschieben lässt oder wo ein jüngst eingespielter Fix tatsächlich Laufzeit spart.

Effizientere Garbage Collection und weniger Speicherverbrauch

Mit dem JEP 519 werden die Compact Object Headers finalisiert. Sie werden jedoch noch nicht zum Standard-Layout erklärt. Seit ihrer Einführung in JDK 24 (JEP 450) haben sie sich in Stabilität und Performance bewährt, unter anderem in hunderten Amazon-Services (teils auf JDK 21/17 zurückportiert). Experimente zeigen deutliche Vorteile: bis zu 22 % weniger Heap, 8 % weniger CPU-Zeit, 15 % weniger GCs (G1/Parallel) und ein hochparalleler JSON-Parser läuft 10 % schneller. Weniger Overhead pro Objekt verbessert die Speichernutzung und Cache-Lokalität, was spürbar Start-up und Durchsatz zugutekommt.

Seit einiger Zeit gibt es neben dem Standard Garbage Collector (G1) auch sogenannte Low Latency GCs wie ZGC und Shenandoah. Sie sind auf moderne Hardwarearchitekturen (Multi-Core Prozessoren und Terrabyte an RAM) ausgelegt und haben trotz großer Speichermengen kürzere GC-Pausen als Ziel. Mit dem JEP 521 (Generational Shenandoah) wird der generationale Modus des Shenandoah-GC (eingeführt als Experiment in JDK 24 via JEP 404) finalisiert. Der Standard bleibt unverändert: Per Default nutzt Shenandoah weiterhin eine Generation, kann aber auf Wunsch auch zwischen Old und Young Generation unterscheiden. Zum Aktivieren des generationalen Modus ist `-XX:+UnlockExperimentalVMOptions` jetzt nicht mehr nötig. Alle übrigen Optionen und Defaults bleiben gleich, bestehende Startskripte funktionieren weiter. Es gab noch zahlreiche Stabilitäts- und Performance-Verbesserungen sowie umfangreichen Tests (unter anderem DaCapo, SPECjbb2015, SPECjvm2008, Heapothesys). Und Anwender berichten von erfolgreichen Einsätzen unter Last.

Erweiterter Kryptographie-Support

Bereits in Java 24 gab es einige JEPs, die Implementierungen für Algorithmen für Schlüsselaustauschverfahren und digitale Signaturen eingeführt haben, um sicher vor zukünftigen Angriffen durch Quantencomputer zu sein. Mit dem JEP 470 (PEM Encodings of

Cryptographic Objects) wird nun eine einfache API eingeführt, um kryptografische Objekte — Schlüssel, Zertifikate und Sperrlisten — in das weit verbreitete PEM-Textformat (RFC 7468) zu kodieren und daraus wieder Objekte zu dekodieren. Sie unterstützt die Standardrepräsentationen PKCS#8 (Private Keys), X.509 (Public Keys, Zertifikate, CRLs) sowie PKCS#8 v2.0 (verschlüsselte Private Keys und asymmetrische Schlüssel). Bisher fehlte in Java eine komfortable PEM-API, Entwickler mussten Base64, Header/Footer-Parsing, Factory-Auswahl und Algorithmus-Erkennung selbst erledigen. Die neue Preview-API reduziert diesen Boilerplate deutlich.

Der JEP 510 finalisiert die als Preview in JDK 24 eingeführte API für Key Derivation Functions (KDFs), etwa HKDF (RFC 5869) und Argon2 (RFC 9106). Sie ermöglicht den Einsatz von KDFs in KEM/HPKE-Szenarien (z. B. ML-KEM, Hybrid Key Exchange in TLS 1.3), erlaubt PKCS#11-basierte Implementierungen und räumt auf, indem JDK-Komponenten wie TLS 1.3 und DHKEM auf die neue API statt auf interne HKDF-Logik umgestellt werden. Aus einem gemeinsamen Geheimnis (zum Beispiel ECDH-Shared-Secret) und einem Salt kann per HKDF deterministisch ein Satz Sitzungsschlüssel für Verschlüsselung und MAC abgeleitet werden. PBKDF1/2 wandern nicht in die neue API, sie bleiben wie bisher über SecretKeyFactory nutzbar.

Fazit und Ausblick

Nach dem Release ist bekanntlich vor dem Release. Bereits im März 2026 wird das OpenJDK 26 *[3]* erscheinen. Da werden wir einige der hier angesprochenen Preview-Themen erneut wiedersehen. Aber es sind auch schon neue Themen geplant, zum Beispiel wird der HttpClient dann HTTP/3 unterstützen, die Applet API wird endgültig entfernt und es wird in Zukunft nicht mehr möglich sein, per Reflection die final-Semantik zu umgehen, also über Hintertürchen als final markierte Felder nachträglich zu verändern.

Wer sich vorab über alle zukünftigen Themen informieren möchte, kann sich schon mal im JEP-Index unter „Draft and submitted JEPs" *[4]* umschauen. Ansonsten sind die Release Notes des OpenJDK 25 *[5]* und der Java Almanac *[6]* ein guter Startpunkt, um auch die vielen kleinen API-Änderungen nachverfolgen zu können.

Java ist und bleibt weiterhin sehr relevant. Auch nach 30 Jahren entwickelt sich die Sprache sowie die Plattform immer weiter und passt sich so den Herausforderungen beim Cloud-Computing und im KI-Umfeld an. Es gibt mittlerweile einige spannende Bibliotheken, die das Einbinden von Large Languages Models ermöglichen. An der dafür nötigen Performance wird es in naher Zukunft weitere Optimierungen geben. Mit den Value Objects steht zudem die nächste große Änderung vor der Tür. Im Oktober 2025 gab es seit langem mal wieder ein Early Access Release des Projekt Valhalla. Darin kann man jetzt nicht nur selbst Value Classes definieren, es wurden auch etwa 30 Referenzklassen wie Integer, Optional und LocalDate in Value Classes umgebaut. Das wird die Art, wie wir zukünftig Code schreiben, nochmal gründlich verändern. Und es geht dabei nicht nur um die Value Objects an sich, sondern auch um Null-Restricted Types (ähnliche Idee wie bei Kotlin: String! name) und Generics über primitive Datentypen (List<int>).

Referenzen

[1] *https://openjdk.java.net/projects/jdk/25/*
[2] *https://openjdk.org/jeps/508*
[3] *https://openjdk.java.net/projects/jdk/26/*
[4] *https://openjdk.org/jeps/0#Draft-and-submitted-JEPs*
[5] *https://jdk.java.net/25/release-notes*
[6] *https://javaalmanac.io/jdk/25/apidiff/24/*

Falk Sippach
falk@jug-da.de
https://twitter.com/sippsack

Falk Sippach ist bei der embarc Software Consulting GmbH als Softwarearchitekt, Berater und Trainer stets auf der Suche nach dem Funken Leidenschaft, den er bei seinen Teilnehmern, Kunden und Kollegen entfachen kann. Bereits seit über 20 Jahren unterstützt er in meist agilen Softwareentwicklungsprojekten im Java-Umfeld. Als aktiver Bestandteil der Community (Mitorganisator der JUG Darmstadt) teilt er zudem sein Wissen gern in Artikeln, Blog-Beiträgen sowie bei Vorträgen auf Konferenzen oder User Group Treffen und unterstützt bei der Organisation diverser Fachveranstaltungen. Falk twittert unter @sippsack.

Software-Sicherheit im Java-Ökosystem

Prof. Dr.-Ing. Stefan Wagenpfeil, SLW, PFH Göttingen

Sicherheitsvorfälle, wie beispielsweise die kritische Schwachstelle in Log4J (2021) oder auch der Lieferkettenangriff auf die SSH-Bibliothek (2024), rütteln die Softwarewelt mittlerweile regelmäßig wach. Die Häufigkeit solcher Vorfälle steigt kontinuierlich und deren Auswirkungen werden immer dramatischer. Innerhalb weniger Stunden müssen Teams unter Hochdruck nach Updates oder Patches suchen und – vor allem – auch häufig erst einmal herausfinden, ob sie betroffen sind oder nicht. Derartige Vorfälle führen uns immer wieder vor Augen, dass Software-Sicherheit nicht nur für den eigenen Code wichtig ist, sondern auch für die Komponenten, die wir täglich nahezu selbstverständlich und oftmals ungeprüft einbinden.

Insbesondere Java-Projekte basieren heute auf einer Vielzahl von Open-Source-Bibliotheken und Frameworks. Dies ist zwar einerseits charmant, erhöht andererseits aber sowohl die Komplexität als auch die Angriffsfläche. Gleichzeitig verschärfen sich regulatorische Anforderungen (zum Beispiel der EU Cyber Resilience Act oder die NIS2-Richtlinie) und zwingen uns, Sicherheit nicht mehr nur als technische, sondern zunehmend auch als rechtliche und organisatorische Verantwortung ernst zu nehmen.

In diesem Artikel wollen wir betrachten, wie Java-Teams mit modernen Ansätzen wie SBOM, strukturiertem Dependency-Management, automatisierten Security-Checks und einem besseren Verständnis der rechtlichen Rahmenbedingungen Software nachhaltiger und sicherer entwickeln und betreiben können.

Sicherheit als integraler Teil der Entwicklung

Beginnen wir mit einem wichtigen Grundprinzip moderner Software-Entwicklung: Sicherheit muss von Anfang an Teil der Architektur und des Entwicklungsprozesses sein. Die Zeiten, in denen man damit zufrieden war, kurz vor dem Release mal jemanden testen zu lassen und das dann als „Sicherheit" zu verkaufen, sind längst vorbei.

Das aktuelle Stichwort „**Security by Design**" hat dieses Vorgehen abgelöst und hilft, typische Angriffspunkte strukturiert und systematisch zu adressieren. Dies können zum Beispiel veraltete Bibliotheken oder unsichere Konfigurationen von Frameworks sein, aber auch unzureichende Validierungs- oder Authentifizierungsfunktionen in der Entwicklung. Von unseren Software-Architekten erwarten wir mittlerweile nicht nur die Konzeption nachhaltiger und langfristig nutzbarer Architekturen, sondern ebenso ein Verständnis von gültigem Recht und den regulatorischen Anforderungen an die zu erstellende Software. Unsere Projekt-Manager müssen nicht nur dafür sorgen, dass Projekte „in-time" und „in-budget" umgesetzt werden, sondern sie sollen außerdem gleichzeitig die Software-Qualität durch organisatorische Maßnahmen steigern und sichern. Viele Werkzeuge, Trends, Fachbegriffe und Methoden wollen auf ihre Tauglichkeit hin untersucht und gegebenenfalls integriert werden.

SBOM – Transparenz in der Lieferkette

Eine der wichtigsten Entwicklungen der letzten Jahre im Bereich Software-Sicherheit ist das Konzept der SBOM (Software Bill of Material) *[1]* – eine Stückliste aller Komponenten, die in einer Software enthalten sind. Im Java-Umfeld können wir hier glücklicherweise auf Maven *[2]* oder vergleichbare Werkzeuge zurückgreifen, die mit ihren Dependency-Managern zwar ursprünglich „nur" Konflikte in Abhängigkeiten genutzter Bibliotheken auflösen wollten, aber ganz nebenbei auch eine vollständige SBOM produzieren können.

Unternehmen, die auf derartige Mechanismen setzen, wissen im Normalfall auf Knopfdruck, welche Komponenten in einem Software-Produkt eingesetzt werden. Im Ernstfall kann somit schnell identifiziert werden, ob man von einer neu entdeckten Schwachstelle betroffen ist. Denn wer bei einem Security-Incident erst nach den verwendeten Komponenten und den entsprechenden Versionen suchen (oder gar reverse-engineeren) muss, verliert wertvolle Zeit. Formate wie CycloneDX *[3]* oder SPDX *[4]* haben sich mittlerweile etabliert und können automatisiert erzeugt werden. Auf diesem Weg kann eine permanent aktualisierte SBOM für jedes Java-Projekt bereits während des Build-Prozesses abgelegt und automatisiert bereitgestellt werden. Ganz nebenbei kommt man damit auch der Transparenz- und Dokumentationspflicht nach, die von diversen Gesetzen gefordert wird. SBOMs stellen einen wichtigen Startpunkt dar, um im Falle von Schwachstellen sehr schnell reagieren zu können.

Dependency-Hygiene

Die reine Kenntnis über die SBOM ist ein essenzieller erster Schritt, jedoch noch keine Lösung. Denn selbst, wenn wir die komplette SBOM einer Software dokumentieren können, bedeutet das nicht automatisch, dass unsere Software dadurch sicherer wird. Um das Grundproblem dabei zu verdeutlichen, sehen wir in *Abbildung 1* ein konkretes Beispiel einer SBOM in einem realistischen Projekt.

Selbst wenn wir in diesem Beispiel sehr genau wissen, welche Bibliotheken verwendet werden, und selbst wenn wir dann beispielsweise herausgefunden haben, dass eine unserer genutzten Bibliotheken kompromittiert wurde, ist das Problem damit noch längst nicht gelöst. Man kann eine Bibliothek normalerweise nicht „einfach mal so" ersetzen.

Wir alle haben schon mal eine Bibliothek von 1.4 auf 1.5 aktualisiert oder gar von 1.6 auf 2.0 und dadurch hunderte Compilerfehler im Projekt verursacht, die mühsam Stück für Stück herausgearbeitet

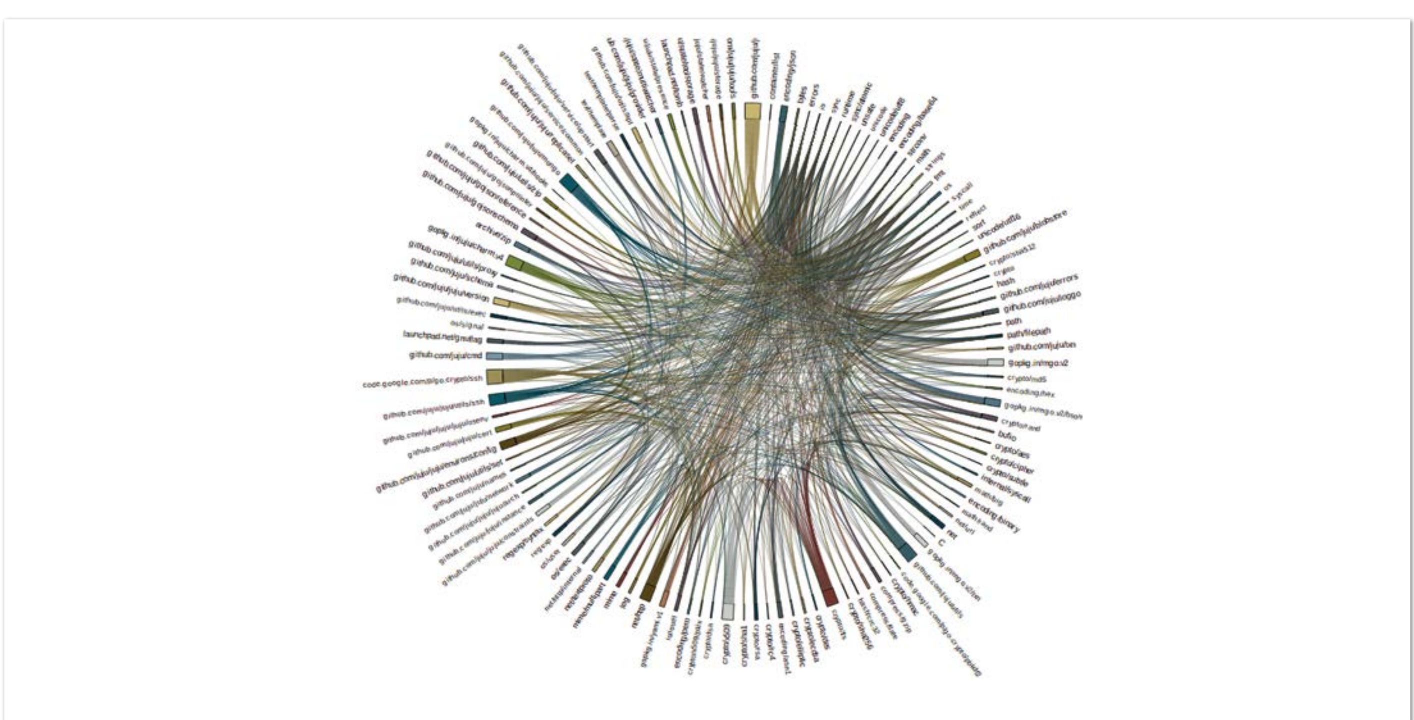


Abbildung 1 – Visualisierung eines Dependency-Graphen (Quelle: [5])

werden mussten. Anschließend muss ein kompletter Regressionstest folgen, der ebenfalls beliebig aufwändig werden kann. Was bei Minor-Versions oder Patches noch mit überschaubarem Aufwand funktioniert, kann im Ernstfall beliebig aufwändig werden! Ein wichtiger Schritt an dieser Stelle ist also die sogenannte **Dependency-Hygiene** – diese umfasst kontinuierliche Refactoring-Zyklen, die einerseits die benötigten Abhängigkeiten auf möglichst aktuelle Stände bringen und andererseits die Anzahl der Abhängigkeiten reduzieren. Derartiges Refactoring muss in der Projektmethodik verankert und auch entsprechend zeitlich und finanziell eingeplant sein.

Lieferketten-Validierung und Qualitätsstandards

Viele denken, wenn sie Open-Source-Komponenten einsetzen, ist die Software automatisch sicher, weil die Quellen ja offengelegt sind. Das ist leider schlichtweg ein Irrglaube, denn das reine Offenlegen bringt noch keine Sicherheit. Auch eine Open-Source-Lizenz bringt keine Sicherheit – im Gegenteil! Sie entbindet Sie lediglich von potenziellen Lizenzgebühren, schiebt Ihnen aber damit auch die komplette Verantwortung zu. Echte Sicherheit entsteht leider erst durch Qualitätsmaßnahmen auf Seiten des Entwicklerteams der Bibliothek. Denn nur, wenn der offene Quellcode tatsächlich geprüft, angriffssicher gemacht und kontinuierlich qualitätsgesichert wird, bringt Open-Source auch mehr Sicherheit in Ihr Projekt.

Viele Attacken der letzten Zeit haben sich gezielt auf Open-Source-Projekte gestützt, gerade weil der Quelltext offen verfügbar ist. Häufig hat man sich die Überlastung des Entwicklerteams zunutze gemacht, um Schadcode einzuschleusen, der dann „einfach durchgewunken“ wird. Im April 2024 wurde auf genau diesem Weg beispielsweise das Linux-Tool SSH kompromittiert, indem in die darin verwendete Komponente „XZ-tools“, die von nur einem einzigen, völlig überlasteten Entwickler verwaltet wurde, eine Backdoor eingebaut wurde *[6]*.

Möchte man also seine Lieferkette validieren und möglichst sichere Software-Komponenten einsetzen, auf die man sich auch unter Security-Aspekten „verlassen“ kann, braucht es nicht nur Transparenz und nachprüfbare Prozesse, sondern auch messbare Qualitätsmerkmale. Gute Ansätze hierzu sind beispielsweise die **OpenSSF Best Practices Badges** *[7]*, die dokumentieren, dass ein Projekt einen Katalog an Sicherheitskriterien erfüllt oder auch die **OpenSSF Scorecard** *[8]*, ein automatisches Bewertungssystem, das über 20 Sicherheitsmetriken prüft. Auf diesem Weg lassen sich Open-Source-Bibliotheken auch unter Sicherheitsaspekten auswählen.

Im Gegensatz zu Open Source gibt es natürlich auch Produkte oder kommerzielle Bibliotheken, die eingebunden werden können. Hierbei hat man dann häufig zwar Lizenzkosten zu bezahlen, gewinnt auf der anderen Seite aber vertragliche Sicherheit und Garantien, die der Hersteller zu erfüllen hat. ISO-Zertifizierungen (zum Beispiel die ISO 15408, 27001 oder 27034) oder auch dokumentierte BSI IT-Grundschutz-Audits schaffen hierbei zusätzlich Vertrauen und da die Hersteller kommerzieller Bibliotheken damit Geld verdienen wollen, sind solche Zertifizierungen in der Branche durchaus üblich. Unter Compliance-, Transparenz-, und Regulatorik-Aspekten ein sehr guter und wichtiger Schritt.

Rechtlicher Rahmen – Sicherheit wird Pflicht

Software-Sicherheit ist längst keine reine Entwickleraufgabe mehr. Weltweit erhöhen die Gesetzgeber den Druck auf die Hersteller, um sichere Softwareprodukte und -komponenten bereitzustellen und Sicherheitsvorfälle transparent zu machen. Einige Beispiele, die dies aktuell sehr eindrücklich verdeutlichen:

Der EU Cyber Resilience Act (CRA) *[9]* beispielsweise verpflichtet Hersteller von Software und digitalen Produkten, Sicherheitslücken über den gesamten Produktlebenszyklus hinweg zu überwachen und zu beheben – auch wenn Open Source enthalten ist. Insbesondere aus Haftungsgründen kann dies sehr schnell problematisch werden, denn Open-Source-Projekte selbst sind im Regelfall nicht haftbar, die Unternehmen, die Open Source in Produkte integrieren, jedoch schon! Der CRA markiert hierbei einen echten Paradigmenwechsel, denn Sicherheit wird nunmehr zur regulatorisch durchsetzbaren Anforderung.

Ein weiterer Baustein ist die **NIS2-Richtlinie** *[10]*, die sich insbesondere an Betreiber kritischer und wichtiger Infrastrukturen richtet und diese zu umfassenden Sicherheitsmaßnahmen, wie Risikoanalysen, technische Schutzmaßnahmen und obligatorische Meldungen verpflichtet. Jeder, der Software für diese Sektoren entwickelt, ist indirekt von NIS2 betroffen.

Natürlich spielt nach wie vor auch die **DSGVO** *[11]* eine wichtige Rolle, denn sie schreibt in Artikel 32 ausdrücklich vor, dass personenbezogene Daten mit geeigneten technischen und organisatorischen Maßnahmen zu schützen sind. Schwachstellen können nicht nur Reputationsschäden, sondern auch hohe Bußgelder zur Folge haben und unter die „technischen und organisatorischen Maßnahmen“ gehört mittlerweile eben auch das Thema SBOM-Validierung.

In Deutschland haben wir dann noch das **IT-Sicherheitsgesetz 2.0** (inklusive KRITIS) *[12]*, das beispielsweise den Einsatz von Systemen zur Angriffserkennung für bestimmte Branchen vorschreibt und die Rolle des BSI als zentrale Sicherheitsbehörde stärkt. Das **Produkthaftungsgesetz** *[13]* (derzeit auf EU-Ebene) definiert Software ausdrücklich als haftungsrelevantes Produkt. Wer also Software mit Sicherheitslücken ausliefert, die zu Schäden führen, muss mit Konsequenzen rechnen.

Ergänzende Vorschriften wie das **Telekommunikation-Telemedien-Datenschutz-Gesetz (TTDSG)** *[14]* für Anbieter von Web- und App-Diensten, der **Digital Operational Resilience Act (DORA)** *[15]* im Finanzwesen, oder auch die **UNECE R155** in der Automobilindustrie sind Beispiele für branchenspezifische Erweiterungen und besonders strenge Sicherheitsstandards.

Für Software-Entwickler und -Architekten bedeutet dies, dass Sicherheit nicht mehr „nice to have“ oder optional ist, sondern rechtlich relevant und bindend wird. Hersteller und Betreiber müssen Sicherheitsmaßnahmen nach dem Stand der Technik oder zumindest gemäß allgemein anerkannten Regeln der Technik umsetzen und auch nachweisen, was wiederum dazu führt, dass SBOM, Schwachstellenmanagement, Security-by-Design quasi als abgeleitete Anforderungen vieler Gesetze gesehen werden können. Hierbei ist anzumerken, dass der „Stand der Technik“ oder „allgemein anerkannte Regeln der Technik“ schwer greifbare Begriffe sind für deren Einordnung häufig IT-Sachverständige zu Rate gezogen werden sollten.

Sicherheitsarchitektur im Java-Kontext

Wenn wir nunmehr im Java-Kontext technische Sicherheitsmaßnahmen einziehen wollen, um resiliente Software zu entwickeln, brauchen wir eine solide Sicherheitsarchitektur, die Anwendungen nicht nur reaktiv, sondern proaktiv vor Angriffen schützt. Hierfür haben sich drei zentrale Sicherheitsprinzipien etabliert, die fast schon den Status von Design-Patterns erreicht haben:

1. Least Privilege – Rechtevergabe auf das Nötigste beschränken
   Ein häufiger Angriffsvektor sind überprivilegierte Komponenten. Das „Least Privilege"-Prinzip sorgt dafür, dass jedes Modul, jeder Service und jeder Benutzer nur die minimal erforderlichen Berechtigungen erhalten. Dies gilt sowohl für Benutzerrollen als auch für Datenbankzugriffe, Service-Accounts, API-Tokens und Infrastrukturrechte. Ein Java-Microservice, der beispielsweise nur lesenden Zugriff auf eine Datenbank benötigt, darf somit also keine Schreibrechte erhalten – auch nicht „zur Sicherheit" oder „für den Notfall".
2. Defense in Depth – Sicherheit in Schichten denken
   Layer-Architekturen sind schon seit jeher ein gutes und sinnvolles Prinzip. Moderne Java-Sicherheitsarchitekturen nutzen diese Architekturen, um mehrere Schutzebenen einzuziehen. Begonnen bei der Datenspeicherung, API-Schutz, über Authentifizierung, Autorisierung bis hin zu Input-Validierung, Präsentation und Administration/Betrieb – jede dieser Schichten sollte als Schutzschicht betrachtet werden und selbst wenn sie umgangen wird, greift dann die nächste. Die Java EE bietet hierfür bereits einen hervorragenden Blueprint, der konsequent umgesetzt werden sollte.
3. Fail Secure – Sicherheit im Fehlerfall gewährleisten
   Viele Sicherheitslücken entstehen in Ausnahmefällen. Ein robustes System fällt nicht unsicher, sondern standardmäßig sicher zurück. Wenn beispielsweise ein Authentifizierungsserver nicht erreichbar ist, darf der Service nicht automatisch in einen „Zugriff gewähren"-Modus zurückfallen, sondern sollte standardmäßig blockieren. Wenn ein System schon „kaputt geht", dann bitte so, dass der Schaden dadurch nicht noch schlimmer wird – klingt logisch, ist in der Praxis aber leider oft genug immer noch nicht der Fall.

Diese Sicherheitsprinzipien müssen sich am Ende auch konkret in Code und Konfiguration widerspiegeln. Java bietet dafür ein breites Ökosystem mit erprobten Frameworks und Tools an. Sehen wir uns an dieser Stelle einige Beispiele an:

- **Management Endpoints absichern**: Spring-Boot-Endpunkte wie /actuator oder /health dürfen nicht öffentlich erreichbar sein
- **Unsichere Defaults** (zum Beispiel offene Ports, schwache Passwörter) müssen angepasst werden
- Zur **Authentifizierung/Autorisierung** sollten etablierte Standards wie OAuth2, OpenID Connect oder SAML genutzt werden
- **Zentralisierte Identity Provider** vermeiden Credential-Sprawl (also die Verbreitung von Geheimnissen) und Inkonsistenzen
- **Multifaktor-Authentifizierung** für besonders kritische Bereiche
- Secrets nicht im Code speichern, nutzen Sie **Secret Stores** (zum Beispiel HashiCorp Vault, AWS Secrets Manager, Kubernetes Secrets)
- **TLS-Verschlüsselung** durchgängig vom client bis zum Backend
- Integrieren Sie **automatisierte Sicherheitsprüfungen** in die CI/CD-Pipeline
- **Architektur über den Code hinaus** denken durch API-Gateways, Zero-Trust-Ansätze, Observability (Logging, Metriken, Alarme)

Softwarequalität als Sicherheitsfaktor

Gute Softwarequalität ist nicht nur ein Garant für Wartbarkeit und Stabilität, sie ist auch eine entscheidende Grundlage für Sicherheit. Schwachstellen entstehen nicht immer durch spektakuläre Exploits, sondern oft auch durch schlecht strukturierten Code, unklare Verantwortlichkeiten oder fehlende Überprüfungen. Je höher die Codequalität, desto höher die Wahrscheinlichkeit, dass Sicherheitslücken entdeckt werden. Hierbei unterstützen nach wie vor statische und dynamische Codeanalysen.

Statische Analysen prüfen den Quellcode, bevor er ausgeführt wird. Tools wie SonarQube *[16]* erkennen dabei nicht nur klassische Qualitätsprobleme sondern auch sicherheitsrelevante Schwachstellen (zum Beispiel unsichere API-Aufrufe, unvalidierte Eingaben oder potenzielle Injection-Punkte). **Dynamische Analysen** wiederum testen das Laufzeitverhalten der Anwendung und können Sicherheitsprobleme sichtbar machen, die erst in der realen Nutzung auftreten (zum Beispiel Speicher- oder Transaktionsverhalten). Regelmäßige **Code Reviews** und **Security Audits** inklusive Pentests steigern die Wahrscheinlichkeit, dass unsichere Patterns oder riskante Entscheidungen frühzeitig erkannt und behoben werden (auch Quality-Gate genannt). Hierbei ist es sehr oft hilfreich, derartige **Quality-Gates** als wesentliche Elemente des Projekts vorzusehen und mit externer Unterstützung zu konzipieren und zu praktizieren, da bei Projekten jeder Art eine gewisse Betriebsblindheit auftreten kann.

Review-Checklisten mit Fokus auf Sicherheitsaspekte sind hierbei besonders effektiv und führen durch ihre kontinuierliche Nutzung auch zur Sensibilisierung des gesamten Teams. **Clean Code** und Architektur-Entscheidungen wirken sich ebenfalls positiv auf die Sicherheit aus, da gut strukturierter und modularer Code Sicherheitsprüfungen einfacher macht, die Komplexität reduziert und somit auch das Risiko unbeabsichtigter Nebenwirkungen senkt. Sicherheit wird dadurch nicht nachträglich „aufgepfropft", sondern ein natürlicher Teil guter Softwarequalität. Und das wiederum bringt uns schließlich zum sogenannten **DevSecOps**-Ansatz *[17]*, der Sicherheit als selbstverständlichen Bestandteil des Entwicklungsprozesses versteht. Ähnlich, wie wir durch DevOps schon den Bruch zwischen Entwicklung und Betrieb auflösen, können wir durch DevSecOps nun den klassischen Bruch zwischen Entwicklung, Betrieb und Security durch kontinuierliche Zusammenarbeit ersetzen.

Ausblick und Empfehlungen

Die IT-Branche befindet sich derzeit in einem der größten Umbrüche aller Zeiten mit umfassenden Herausforderungen insbesondere im Bereich Sicherheit. **KI-gestützte Systeme** können beispielsweise große Codebasen nach Anomalien durchsuchen, ungewöhnliche Abhängigkeiten identifizieren und Entwicklern als hervorragende Werkzeuge zur Hand gehen. Andererseits kann KI auch genutzt werden, um gezielte Angriffspunkte zu finden – und auszunutzen. Ein wichtiger Schlüssel, um diesen Umbruch zu bewältigen, ist das Thema **Sicherheitskultur**. Kein Tool ersetzt das Bewusstsein im Team. Schulungen, klare Prozesse, sichere Defaults und offene Kommunikation sind entscheidend, um Sicherheitsrisiken nicht nur technisch, sondern auch organisatorisch zu adressieren. Wer Sicherheit als Teamleistung versteht, kann Risiken nachhaltig reduzieren.

Speziell für Java-Teams ist daher ein ganzheitlicher Ansatz zu empfehlen: Sicherheitsanforderungen frühzeitig definieren, automatisierte Prüfungen in den Entwicklungsprozess integrieren, kontinuierlich lernen und klare Verantwortlichkeiten schaffen.

Das Ziel ist es nicht, ein perfektes Sicherheitsniveau zu erreichen – das ist nahezu unmöglich – sondern ein Sicherheitsnetz zu etablieren, das Angriffe erschwert und schnelle Reaktionen ermöglicht.

Quellen

[1] T. Stalnaker, N. Wintersgill, O. Chaparro, M. Di Penta, D. M. German, und D. Poshyvanyk, „BOMs Away! Inside the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems", in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon Portugal: ACM, Feb. 2024, S. 1–13. doi: 10.1145/3597503.3623347.

[2] „Maven Repository: Central". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://mvnrepository.com/repos/central*

[3] „CycloneDX Bill of Materials Standard | CycloneDX". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://cyclonedx.org/*

[4] R. Gandhi, M. Germonprez, und G. J. P. Link, „Open Data Standards for Open Source Software Risk Management Routines: An Examination of SPDX", in Proceedings of the 2018 ACM Conference on Supporting Groupwork, Sanibel Island Florida USA: ACM, Jan. 2018, S. 219–229. doi: 10.1145/3148330.3148333.

[5] D. Cheney, „Visualising dependencies | Dave Cheney". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://dave.cheney.net/2014/11/21/visualising-dependencies*

[6] „That was close.... Last week a crititcal security hole was detected and it reads almost like a thriller movie plot, when you dive into the details. Luckily almost unknown from the public, here's... | Prof. Dr.-Ing. Stefan Wagenpfeil". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://www.linkedin.com/posts/stefan-wagenpfeil_that-was-close-last-week-a-crititcal-activity-7181258411714453506-38l4*

[7] „Best Practices Badge – Open Source Security Foundation". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://openssf.org/projects/best-practices-badge/*

[8] N. Zahan, P. Kanakiya, B. Hambleton, S. Shohan, und L. Williams, „OpenSSF Scorecard: On the Path Toward Ecosystem-Wide Automated Security Metrics", IEEE Secur. Priv., Bd. 21, Nr. 6, S. 76–88, Nov. 2023, doi: 10.1109/MSEC.2023.3279773.

[9] Verordnung (EU) 2024/2847 des Europäischen Parlaments und des Rates vom 23. Oktober 2024 über horizontale Cybersicherheitsanforderungen für Produkte mit digitalen Elementen und zur Änderung der Verordnungen (EU) Nr. 168/2013 und (EU) 2019/1020 und der Richtlinie (EU) 2020/1828 (Cyberresilienz-Verordnung) (Text von Bedeutung für den EWR). 2024. Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *http://data.europa.eu/eli/reg/2024/2847/oj/deu*

[10] „EUR-Lex - 02022L2555-20221227 - EN - EUR-Lex". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://eur-lex.europa.eu/eli/dir/2022/2555/2022-12-27/eng*

[11] „Regulation - 2016/679 - EN - gdpr - EUR-Lex". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng*

[12] „Zweites Gesetz zur Erhöhung der Sicherheit informationstechnischer Systeme (IT-Sicherheitsgesetz 2.0)", Bundesamt für Sicherheit in der Informationstechnik. Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://www.bsi.bund.de/DE/Das-BSI/Auftrag/Gesetze-und-Verordnungen/IT-SiG/2-0/it_sig_2-0.html?nn=937202*

[13] „ProdHaftG - Gesetz über die Haftung für fehlerhafte Produkte". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://www.gesetze-im-internet.de/prodhaftg/BJNR021980989.html*

[14] A. Riechert und T. Wilmer, „Datenschutz und Schutz der Privatsphäre in der Telekommunikation", in TTDSG, in Berliner Kommentare. , Berlin: Erich Schmidt Verlag GmbH & Co. KG, 2022, S. 101–311. doi: 10.37307/b.978-3-503-20979-8.04.

[15] „Digital Operational Resilience Act (DORA)". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://www.esma.europa.eu/esmas-activities/digital-finance-and-innovation/digital-operational-resilience-act-dora*

[16] „Code Quality Tool & Secure Analysis with SonarQube". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://www.sonarsource.com/de/products/sonarqube/*

[17] „Was ist DevSecOps? | Sicherheit in DevOps integriert". Zugegriffen: 24. Oktober 2025. [Online]. Verfügbar unter: *https://www.redhat.com/de/topics/devops/what-is-devsecops*

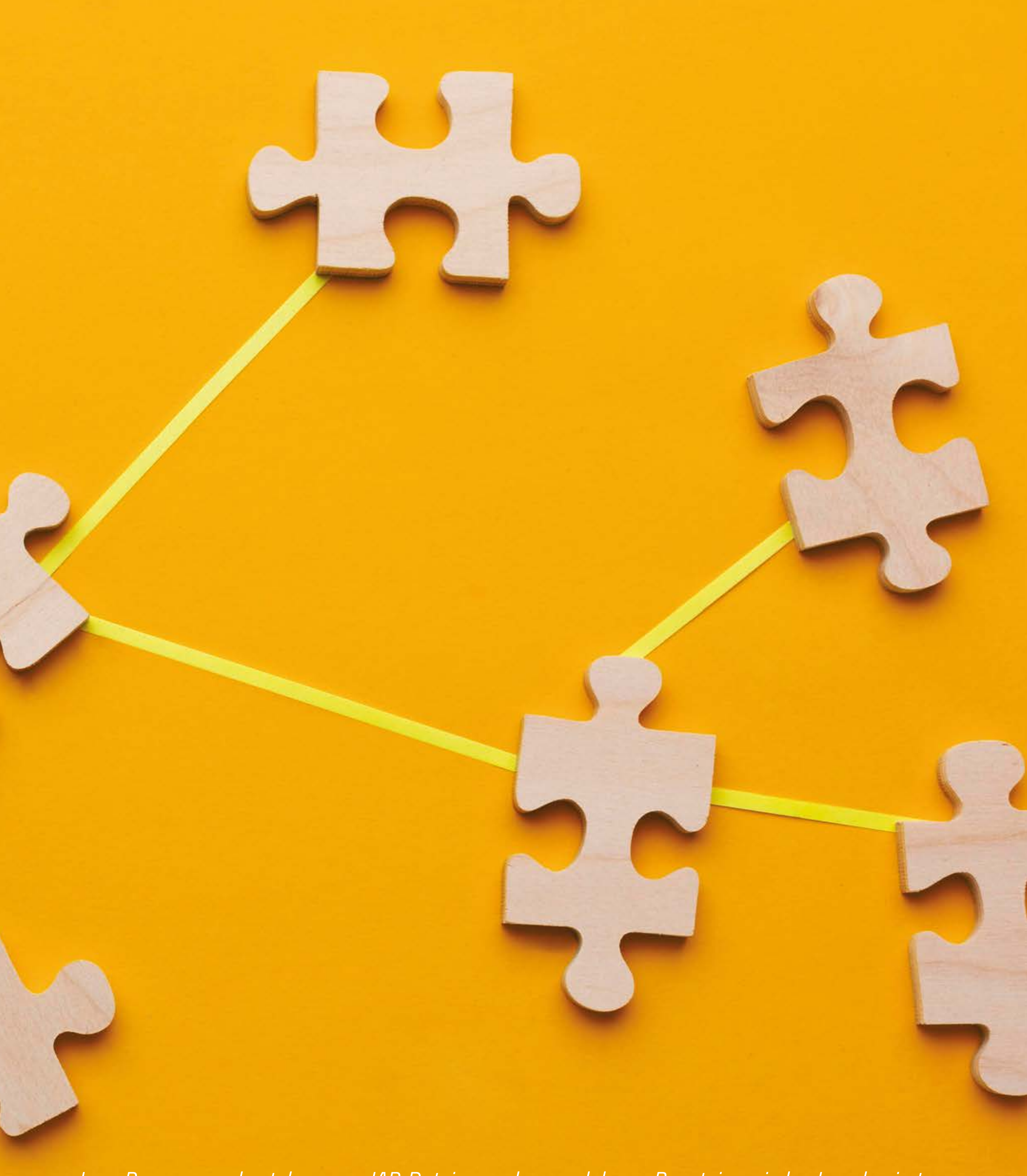
Prof. Dr.-Ing. Stefan Wagenpfeil

stefan.wagenpfeil@slw-it.de

Stefan Wagenpfeil ist Professor für Software-Engineering und IT-Management an der PFH Göttingen. Von 1998 bis 2022 arbeitete er als Java & JEE-Trainer und freiberuflicher Software- und Enterprise-Architekt für große Industrie-Unternehmen. Insbesondere die Themen Nachhaltigkeit von Software-Entwicklungen und die langfristige Sicherung von Investitionen stellen einen Schwerpunkt seiner Arbeit dar. Als Autor hat er beim Springer Verlag die Bücher „Moderne Software-Entwicklung mit Java und JEE", „Gamification Design", „Multimedia Information Retrieval" oder „Nachhaltiges Software Engineering" veröffentlicht. Seit 2022 arbeitet er als IT-Sachverständiger bei SWL – Streitz Liesegang Wagenpfeil & Partner. Außerdem ist er Fachbuchautor beim Springer Vieweg Verlag.

Unbeschwertes Dependency-Management für Java-Projekte

Dr. Jendrik Johannes



Java-Programme bestehen aus JAR-Dateien – den modularen Bausteinen jeder Java-basierten Software. In der Regel stammen die meisten JARs aus Open-Source-Bibliotheken, die sich leicht einbinden lassen: einfach kostenlos im öffentlichen Repository zugreifen! Doch im Laufe der Zeit bringen diese Abhängigkeiten versteckte Kosten mit sich. In langlebigen Projekten wird das Abhängigkeitsmanagement häufig zu einem verworrenen Durcheinander: mühsam, fehleranfällig und zeitaufwändig. Mit einem strukturierten Vorgehen und einem modernen Java-Toolset lässt sich dem wirksam entgegensteuern.

In diesem Artikel werfen wir einen Blick auf das *Java Recipe for Carefree Dependency Administration* – https://javarca.de (siehe Abbildung 1). Es bietet eine Leitlinie aus sechs Punkten, um ein Java-Projekt mit einem Abhängigkeitsmanagement ohne unnötige Komplexität aufzusetzen. Für jeden dieser sechs Punkte betrachten wir, welche Rolle er spielt, welches Problem er adressiert und wie sich dieses effektiv lösen lässt.

Punkt 1: Bewusste Wahl der Methode zur Definition von Abhängigkeiten

Wenn wir ein Open-Source-Modul verwenden wollen, müssen wir dies irgendwo definieren. Auch die Beziehungen zwischen den Modulen unserer eigenen Software müssen irgendwo spezifiziert werden. Im Java-Ökosystem existieren dafür mehrere Möglichkeiten. Daher steht am Anfang eine bewusste Entscheidung: *Wie und wo definieren wir Abhängigkeiten?*

Tun wir dies nicht, entstehen schnell Inkonsistenzen und Unübersichtlichkeit – und damit unnötige Komplexität. Die Definition von Abhängigkeiten spiegelt Teile der Softwarearchitektur im Code wider. Es ist wichtig, diese im Blick zu behalten und wenn nötig, bewusst anzupassen. Alle Entwicklerinnen und Entwickler sollten wissen, wie und wo Abhängigkeiten definiert werden. Und sie sollten sich für diese Definitionen genauso verantwortlich fühlen wie für den Code. Da Abhängigkeiten meistens nicht in Java, sondern in anderen Formaten definiert werden (zum Beispiel Kotlin-DSL oder XML), entsteht schnell ein gedanklicher Bruch. Es besteht die Gefahr, dass die Definition von Abhängigkeiten nur als Teil der Build-Konfiguration angesehen wird und nicht als Teil der Software selbst.

Um dem entgegenzuwirken, ist es wichtig, eine bewusste Entscheidung zu treffen: *In welchen Dateien werden Abhängigkeiten definiert, und in welcher Notation?* Ein historischer Aspekt führt dabei regelmäßig zu Missverständnissen: Die am häufigsten verwendeten Tools – Gradle [1] und Maven [2] – sind vor allem als Build-Tools bekannt. Sie sind allerdings beides: *Build-Tool* und Dependency-Management-Tool. (Früher gab es mit Ant [3] und Ivy [4] einmal die Idee, *Build-Tool* und *Dependency-Management-Tool* getrennt bereitzustellen.) Diese Unterscheidung ist vor allem unter dem Aspekt Code-Ownership wichtig: Während eine Build-Konfiguration von einzelnen Gradle- oder Maven-Expertinnen und -Experten gepflegt werden kann, ist das Verwalten von Abhängigkeiten Teil der Software und sollte von allen Entwicklerinnen und Entwickler mitgetragen werden. Dazu gehört, dass die Definitionen verstanden werden, was schwierig wird, wenn Build-Konfiguration und Abhängigkeitsdefinition in der gleichen Notation und den gleichen Dateien vermischt werden. Darum ist eine klare Trennung von Aspekten in mehrere Dateien das Grundthema des Rezepts.

Folgende Ansätze eignen sich, um Abhängigkeitsdefinition von Build-Konfiguration zu trennen:

1. Ein Gradle- oder Maven-Projekt hat in der Regel eine tool¬spezifische Datei pro Modul: *module/build.gradle* (siehe Abbildung 2) oder *module/pom.xml* (siehe Abbildung 3). Technisch erlauben es beide Tools, dort alle Aspekte der Abhängigkeitsverwaltung und der Build-Konfiguration zu definieren (und in vielen Projekten wird dies unvorteilhaft ausgenutzt). Die Lösung ist, diese Dateien rein als Abhängigkeitsdefinition zu betrachten. Die Dateien zeigen dann klar die Abhängigkeiten des entsprechenden Moduls. Entwicklerinnen und Entwickler können die Notation für Abhängigkeiten – *dependencies { … }* (siehe Abbildung 2) oder *<dependencies>…</dependencies>* (siehe Abbildung 3) – verinnerlichen, ohne dafür Gradle- oder Maven-Expertinnen oder -Experten zu sein.

Recipe for Carefree Dependency Management

1. Decide on a method and define Dependencies between Modules in dedicated files
Java Module System: .... <module-folder>/src/main/java/module-info.java
Gradle: .... <module-folder>/build.gradle.kts
Maven: .... <module-folder>/pom.xml

2. Version management in a dedicated file — Do not mix with 1.
Gradle: .... <bom-folder>/build.gradle.kts
Maven: .... <bom-folder>/pom.xml

3. Configure each build concern in a dedicated file — Do not mix with 1.
Gradle: .... <plugins-folder>/<build-concern>.gradle.kts (plugin composition)
Maven: .... <configs-folder>/<build-concern>/pom.xml (parent pom hierarchy)

4. Conflict management in a dedicated file — Do not mix with 1.
Gradle: .... <plugins-folder>/dependency-rules.gradle.kts
Maven: .... <configs-folder>/dependency-rules/pom.xml

5. Configure dependency analysis and make it a regular check
Gradle: .... <plugins-folder>/quality-check.gradle.kts
Maven: .... <configs-folder>/quality-check/pom.xml

6. Configure Renovate or Dependabot for automatic update PRs
Renovate: .... renovate.json
Dependabot: .... .github/dependabot.yml

Abbildung 1: Java Recipe for Carefree Dependency Administration (© Jendrik Johannes)

2. Java bietet mit *module-info.java*-Dateien eine eigene Notation für Abhängigkeiten als Teil des Java Module Systems (JPMS). Darüber werden Abhängigkeiten direkt in Java definiert *(siehe Abbildung 4)* und sind damit unabhängig vom Build-Tool. Nutzt man ein Plugin, das die Informationen aus *module-info.java*-Dateien an das Build-Tool weiterleitet, müssen diese nicht redundant in den Tool-spezifischen Dateien (*build.gradle* oder *pom.xml*) wiederholt werden *[5]*. Diese nicht-Java Dateien können dann fast leer sein oder sogar ganz weggelassen werden. Entwicklerinnen und Entwickler kümmern sich um Code und Abhängigkeiten eines Moduls direkt in Java und haben in der täglichen Arbeit nur noch wenig Berührungen mit toolspezifischer Notation *(siehe Abbildung 4)*.

Punkt 2: Versions-Management an dedizierter Stelle

Viele der definierten Abhängigkeiten (Punkt 1) zeigen auf Open-Source-Module. Diese Module liegen in mehreren Versionen und Varianten vor. Ein Dependency-Management-Tool sorgt dafür, dass für jedes Modul die passende JAR-Datei ausgewählt, heruntergeladen und gecacht wird.

Um diese Funktion zu erfüllen, braucht das Tool (Gradle oder Maven) Informationen. Die minimale Information ist jeweils eine Version pro direkt definierte Abhängigkeit. Diese sollte man, auch wenn technisch möglich, nicht direkt zusammen mit den Abhängigkeiten spezifizieren. Eine Abhängigkeit wie beispielsweise *Commons IO (siehe Abbildungen 2, 3, 4)* kann mehrmals auftauchen, wenn sie in mehreren Modulen der eigenen Software benötigt wird. Die verwendete Version sollte jedoch überall identisch sein und daher zentral gepflegt werden.

Die zentrale Verwaltung von Versionen gilt seit Langem als Best Practice. In diesem Punkt des Rezepts geht es daher vor allem darum, diese konsequent umzusetzen: also keine Versionen mit Abhängigkeitsdefinition (Punkt 1) zu vermischen.

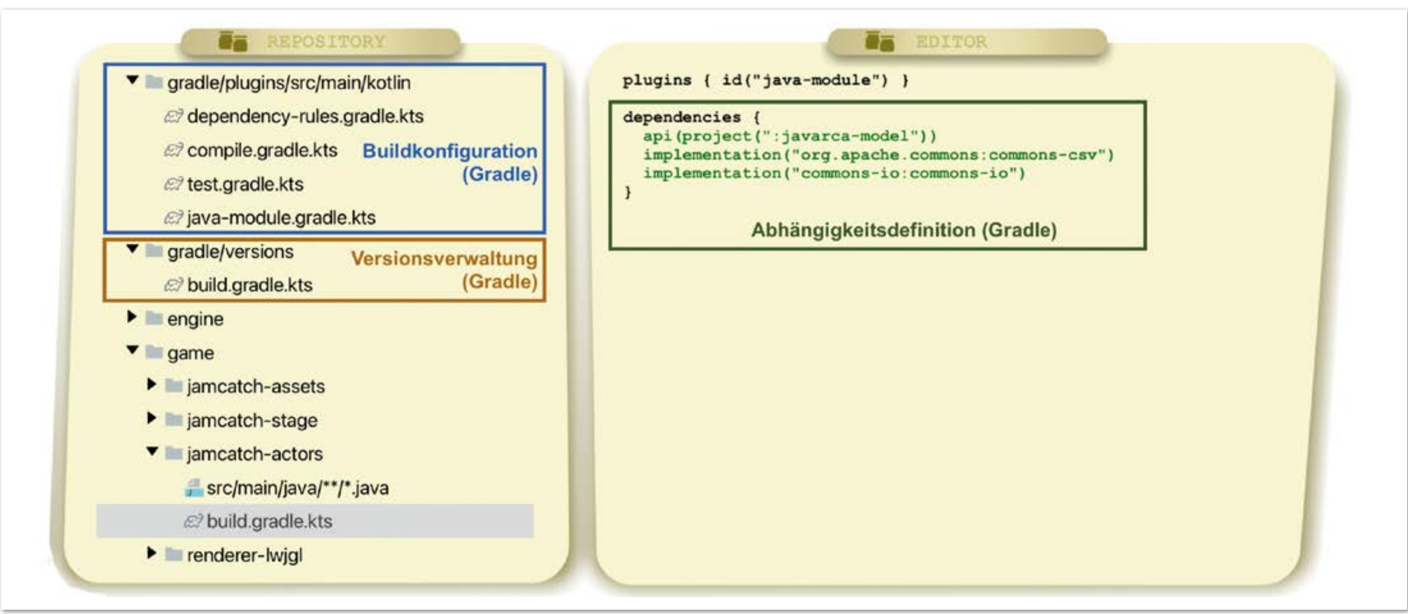


Abbildung 2: Projekt mit Gradle zur Abhängigkeitsdefinition und Gradle als Build-Tool (© Jendrik Johannes)

Abbildung 3: Projekt mit Maven zur Abhängigkeitsdefinition und Maven als Build-Tool (© Jendrik Johannes)

Abbildung 4: Projekt mit module-info.java Dateien zur Abhängigkeitsdefinition und Gradle als Build-Tool (© Jendrik Johannes)

Konkret kann man hier das von Maven eingeführte Konzept Bill-of-Material (BOM) [8] nutzen (siehe Listing 1). Dabei werden Versionen in einem eigens dafür vorgesehenen Block – *dependencyManagement* – in einer separaten Datei definiert. Gradle bietet das gleiche Konzept unter dem Namen Java Platform [9] an und stellt den *dependencies.constraints*-Block bereit (siehe Listing 2).

Punkt 3: Konfiguration aller Software-Build-Aspekte an dedizierter Stelle

Wie in Punkt 1 beschrieben, sollte die Abhängigkeitsdefinition getrennt von der übrigen Build-Konfiguration betrachtet werden. Eine gute Voraussetzung hierfür ist, die Build-Konfiguration insgesamt so zu organisieren, dass jeder Aspekt an einer klar definierten Stelle konfiguriert wird.

In vielen Projekten fehlt eine saubere Trennung einzelner Aspekte der Build-Konfiguration. Dies liegt vermutlich daran, dass die Tools hierfür keine Vorgabe machen und man auch alles ungeordnet in einer Datei konfigurieren kann. Dies führt zu unnötiger Komplexität durch schwer lesbare Konfigurationsdateien die mehrere hundert oder sogar tausend Zeilen lang sind.

Wie auch im Code, ist die Lösung eigentlich offensichtlich: Wir vermeiden unnötige Komplexität, indem wir die unabhängigen Konfigurationen in getrennte, sinnvoll benannte und sortierte Dateien packen. Dabei geht es sowohl um Themen, die das eigentliche Bauen der Software betreffen (Java-Compiler-Flags setzen, Test-Framework auswählen), als auch um Dinge, die im Umfeld der Abhängigkeitsverwaltung individuelle Konfiguration benötigen (siehe Punkte 4 und 5).

Hier glänzt Gradle mit seinem Kompositionsansatz: Jede Konfigurationsdatei ist technisch ein Plugin, auch Convention Plugin [8] genannt, und diese lassen sich beliebig kombinieren. So können einzelne Aspekte in Dateien wie *compile.gradle* und *test.gradle* definieren und dann zu einer *java-module.gradle* komponieren werden, die dann wiederum in allen Modulen verwendet wird (siehe Abbildungen 2 und 4). Maven macht diese Aufteilung in aktuellen Versionen schwerer. Eine Trennung in mehrere Dateien ist nur über Vererbung mittels *Parent POM* [9] möglich, und jede Datei kann dabei nur einen Parent besitzen. Mit einer Hierarchie von POM-Dateien lässt sich dieses Problem zwar umgehen (siehe Abbildung 3), jedoch bleibt die Flexibilität begrenzt. In Maven 4.1 wird mit den *POM Mixins* [10] ein neues Konzept eingeführt, das genau dieses Problem adressiert. Die

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-bom</artifactId>
      <version>2.0.17</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>commons-io</groupId>
      <artifactId>commons-io</artifactId>
      <version>2.16.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-csv</artifactId>
      <version>1.14.0</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Listing 1: Maven-BOM zur zentralen Versionsverwaltung – .mvn/versions/pom.xml in Abbildung 3

```
dependencies {
  api(platform("org.slf4j:slf4j-bom:2.0.17"))
}
dependencies.constraints {
  api("commons-io:commons-io:2.16.1")
  api("org.apache.commons:commons-csv:1.14.0")
}
```

Listing 2: Gradle Platform zur zentralen Versionsverwaltung – gradle/versions/build.gradle.kts in Abbildungen 2 und 4

Modul	Aufgabe in unserer Software	Mögliche Konflikte
Commons IO [11]	Einlesen von Dateien	—
Commons CSV [12]	Parsen von CSV-Dateien	Hat Commons IO als Abhängigkeit und es kann zu einem Konflikt mit der von uns selbst definierten Version kommen
SLF4J [13]	Logging	Aus mehreren Implementierungen muss eine ausgewählt werden: slf4j-simple (loggt direkt), slf4j-jdk14 (leitet an java.util.logging weiter)
LWJGL [14]	Rendern von Grafik	Stellt unterschiedlich JARs für unterschiedliche Betriebssysteme und Prozessor-architekturen bereit; Auswahl muss aufgrund von Zielplattform getroffen werden

Tabelle 1: Beispiele von Versions- und Variantenkonflikten in Open-Source-Modulen

```
jvmDependencyConflicts {
  conflictResolution {
     select("org.gradlex:slf4j-impl", "org.slf4j:slf4j-simple")
  }
  patch {
     module("org.lwjgl:lwjgl") {
        addTargetPlatformVariant("natives", "natives-linux", LINUX, X86_64)
        addTargetPlatformVariant("natives", "natives-macos", MACOS, X86_64)
        addTargetPlatformVariant("natives", "natives-windows", WINDOWS, X86_64)
     }
  }
}
```

Listing 3: Zentralisiertes Erweitern und Korrigieren von Metadaten in einem Gradle Convention Plugin – dependency-rules.gradle.kts in Abbildungen 2 und 4

Mixins ermöglichen dann eine flexible Komposition von Konfigurationsaspekten aus getrennten Dateien.

Punkt 4: Konfliktverwaltung an dedizierter Stelle

Das wichtigste Feature eines Dependency-Management-Tools ist, neben dem Herunterladen und Cachen von Artefakten, das Selektieren von Versionen und Varianten und gegebenenfalls das Erkennen und Lösen von Konflikten. Hierbei ist jedes Tool nur so gut wie die Daten, die es bekommt. Diese Daten werden aus den Metadaten, die den JARs beiliegen, gewonnen (**.pom*- und **.module*-Dateien). Manchmal sind diese jedoch unzureichend und man muss für sein Projekt, also den Kontext, in dem man existierende Module auf neue Art kombiniert, weitere Informationen bereitstellen.

Es gibt eine Vielzahl von Konflikten, die entstehen können, wenn man Open-Source-Module nutzt und kombiniert. *Tabelle 1* zeigt typische Fälle. Werden diese nicht gelöst oder gar nicht erst erkannt, führt dies meistens zu fehlerhaftem Verhalten.

Es gibt Fälle, in denen ein Konflikt nicht vollautomatisch gelöst werden kann, weil es keine eindeutig „richtige Entscheidung" gibt (so beispielsweise die Wahl einer Logger-Implementierung). In anderen Fällen fehlen Informationen in den Metadaten, um einen Konflikt überhaupt erst zu erkennen.

Ähnlich wie bei Versionen (Punkt 2) möchte man Konflikte einheitlich erkennen und lösen. Die zusätzliche Konfiguration, die dafür nötig ist, sollte daher ebenfalls zentral in einer Datei abgelegt werden, die dann für alle Module der eigenen Software gilt. Dies ist wichtig, um Inkonsistenten beim Testen zu vermeiden. Es ist zum Beispiel ungünstig, wenn die Unit-Tests eines isolierten Moduls andere Versionen verwenden als die, die später in Produktion laufen.

Gradle und Maven bieten in diesem Bereich unterschiedliche Lösungen an. Gradle verfügt grundsätzlich über das umfassendere Feature-Set in diesem Bereich und ist daher oft die bessere Wahl für Projekte mit hohen Anforderungen an das Abhängigkeitsmanagement. Unabhängig von der gewählten Lösung gilt: Die Konfiguration sollte so weit wie möglich zentralisiert werden. Das in Gradle am besten geeignete Konzept sind *Component Metadata Rules* *[15]*. Damit lassen sich fehlende Metadaten ergänzen und falsche Daten korrigieren *(siehe Listing 3)*. Da Gradle mit seinem Variant-Aware-Dependency-Management ein sehr ausdrucksstarkes Metadatenformat anbietet, kann man hiermit auch die Selektion von Jars nach Betriebssystem oder anderen Attribute steuern *[16]*.

In Maven gibt es kein Äquivalent dazu. Daher muss man dort mit *Dependency Exclusions* *[17]* arbeiten, um Abhängigkeiten zu entfernen und gegebenenfalls durch andere zu ersetzen. Bei dieser Methode ist es herausfordernder, die Konfliktverwaltung sauber zu isolieren.

```
<dependencyManagement>
  <dependencies>
     <dependency>
        <groupId>de.javarca</groupId>
        <artifactId>javarca-engine</artifactId>
        <exclusions>
           <exclusion>
              <groupId>org.slf4j</groupId>
              <artifactId>slf4j-jdk14</artifactId>
           </exclusion>
        </exclusions>
     </dependency>
  </dependencies>
</dependencyManagement>
```

Listing 4: Zentralisierte Exclusions in einer Parent-POM in Maven – dependency-rules/pom.xml in Abbildung 3

Man kann es relativ weit treiben, wenn man Exclusions in einem Dependency-Management-Block in einer Parent-POM definiert, statt direkt in Dependency-Blöcken *(siehe Listing 4)*. Eine weitergehende Variantenauswahl unterstützt Maven innerhalb eines einzelnen Builds nicht. Unterschiedliche Varianten können jedoch über *Build Profiles* [18] in getrennten Ausführungen selektiert werden.

Punkt 5: Abhängigkeitsanalyse als Teil des Build-Prozesses

Befolgt man die Punkte 1 bis 4 bekommt man ein übersichtliches Setup. Aber wie bei allen Dingen in der Softwareentwicklung, ist dies nur eine Momentaufnahme. Die Software entwickelt sich weiter und damit müssen auch Abhängigkeiten zwischen Modulen ständig überdacht werden. Ohne Toolunterstützung ist es schwer, die Übersicht zu behalten.

Arbeiten wir am Code eines Moduls unserer Software, um Features zu entwickeln oder Probleme zu fixen, denken wir dabei nicht immer an die Abhängigkeiten zu anderen Modulen. Dabei können über die Zeit Inkonsistenten entstehen. Vielleicht hat man Abhängigkeiten definiert, die man im Code gar nicht mehr verwendet. Oder man verwendet versehentlich Elemente aus Abhängigkeiten, die man nicht definiert hat, die aber dennoch für den Compiler sichtbar sind.

Es gibt Tools, die hier Abhilfe schaffen, indem sie den Code mit den deklarierten Abhängigkeiten abgleichen und so zum Beispiel ungenutzte Abhängigkeiten erkennen. Fügt man ein solches Tool in seinen Build-Prozess ein, stellt man sicher, dass Inkonsistenzen nicht versehentlich entstehen und über die Zeit immer größer werden.

Konkret gibt es für Gradle das Dependency-Analysis-Plugin [19], das solche Analysen bietet und sich reibungslos in Gradle-Buildprozesse einfügt. In Maven bringt das Maven-Dependency-Plugin direkt das *dependency:analyze* Goal [20] mit, das einige dieser Checks bereitstellt.

Punkt 6: Automatisierte Versions-Updates

Sobald wir ein Open-Source-Modul verwenden, entsteht ein Wartungsaufwand, der nicht vollständig unter unserer Kontrolle liegt. Immer, wenn eine neue Version veröffentlicht wird, sollten wir zumindest prüfen, ob ein Update in unserem Projekt sinnvoll ist.

Auf den ersten Blick wirkt das nicht notwendig: *Wenn ein Modul in meiner Software funktioniert, warum sollte ich es dann regelmäßig updaten?* Betrachtet man ein einzelnes Modul isoliert, ist der Hauptgrund wohl Security. Immer wieder werden Sicherheitslücken in Open-Source-Komponenten entdeckt und öffentlich gemacht. Fixes für solche Lücken sollten möglichst schnell eingespielt werden, um seine eigene Software nicht darüber angreifbar zu machen. Aber natürlich können Updates auch Verbesserungen, beispielsweise in der Performance, für die eigene Software bringen.

Ein weiterer Grund, warum man *alle* Modul-Versionen regelmäßig updaten sollte, ist die Beziehung von Open-Source-Modulen untereinander: Im Java-Ökosystem gibt es heute ein reichhaltiges Angebot von Modulen, die aufeinander aufbauen. Das führt dazu, dass mit dem Update eines Moduls oft auch Versionen anderer Module hochgezogen werden müssen, damit sie untereinander kompatibel bleiben – siehe als Beispiel Commons CSV und Commons IO *(siehe Tabelle 1)*. Macht man Updates nur, wenn es „unbedingt nötig" erscheint, kann man in Situationen geraten, in der man plötzlich „alles auf einmal" updaten muss.

Was die Toolunterstützung betrifft, unterscheidet sich dieser Punkt von den vorherigen: Wir müssen aktiv werden, selbst wenn wir unseren eigenen Code gar nicht weiterentwickeln. Ein gutes Tool-Setup sollte Teil unserer Code-Hosting-Platform und/oder CI-Pipeline sein und auch dann arbeiten, wenn an unserem Code keine Änderungen vorgenommen wurden. Stattdessen sollte es uns informieren, wenn ein Update für ein Open-Source-Module bereitsteht und das eigentliche Updaten so weit wie möglich automatisieren.

In den letzten Jahren haben sich einige Tools etabliert, die man zusätzlich zu den Dependency-Management-Tools in seiner Code-Hosting-Plattform und/oder CI-Pipeline nutzen kann. Am verbreitetsten sind Renovate [21] und Dependabot [22]. Beide informieren, wenn ein Update bereitsteht und erzeugen automatisch Pull-/Merge-Requests, um das Update durchzuführen. Je nachdem, welche manuellen Test-Schritte benötigt werden, kann man den Grad der Automatisierung konfigurieren. Wichtig ist, dass wenn manuelle Schritte nötig sind, diese in den Entwicklungsprozess integriert werden: Zum Beispiel, indem man wöchentlich alle offenen Requests prüft, um Updates auch tatsächlich durchzuführen.

Nutzt man eine Plattform wie GitHub, lassen sich beide Tools leicht über die zugehörige App einrichten – oft ohne zusätzliche Kosten. Der Open-Source-Kern von Renovate kann auch selbst gehostet und über beliebige CI-Systeme angesteuert werden. Sollte man diese Tools nicht nutzen wollen oder können, kann man auch Gradle oder Maven in seiner CI-Pipeline dazu nutzen, über neue Versionen zu informieren und auf der Basis ein eigenes Tooling aufsetzen.

Zusammenfassung

In diesem Artikel haben wir uns angesehen, warum jeder der sechs Punkte des *Java Recipe for Carefree dependency Administration* *(siehe Abbildung 1)* ein wichtiger Aspekt in der Verwaltung von Abhängigkeiten in Java-Projekten ist. Zu jedem Punkt haben wir aufgezeigt, wie er in einem Gradle- oder Maven-basiertem Setup umgesetzt werden kann. Mehr Details, inklusive vollständiger Beispielprojekte und Videos, sind auf der Website *https://javarca.de* zu finden.

Für neue Projekte empfiehlt es sich, das Rezept von Punkt 1 bis 6 durchzugehen und sich für jeden Punkt klarzumachen, wie und wo er sich im eigenen Projektsetup wiederfindet. Bei bestehenden Projekten können die Punkte in beliebiger Reihenfolge adressiert werden. So kann beispielsweise das Aktivieren einer Analyse (Punkt 5) helfen, sich eine Übersicht zu verschaffen, um anschließend gezielt weitere Punkte anzugehen. Auch das selektive Befolgen einzelner Punkte des Rezepts wird die Situation in einem Projekt, das unter einer unnötig komplexen Abhängigkeitsverwaltung leidet, verbessern.

Quellen

[1] Gradle Build Tool. *https://gradle.org*
[2] Apache Maven Build Tool. *https://maven.apache.org*
[3] Apache Ant Build Tool. *https://ant.apache.org/*
[4] Apache Ivy Dependency Manager. *https://ant.apache.org/ivy*
[5] GradleX Java Module Dependencies Plugin. *https://github.com/gradlex-org/java-module-dependencies*

[6] Maven Bill of Materials (BOM). *https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#bill-of-materials-bom-poms*
[7] Gradle Platform Projects. *https://docs.gradle.org/current/userguide/platforms.html*
[8] Gradle Convention Plugins. *https://docs.gradle.org/current/userguide/implementing_gradle_plugins_convention.html*
[9] Maven POM Inheritance. *https://maven.apache.org/guides/introduction/introduction-to-the-pom.html#project-inheritance*
[10] Maven Mixins. *https://maven.apache.org/guides/mini/guide-mixins.html*
[11] Apache Commons IO. *https://commons.apache.org/proper/commons-io*
[12] Apache Commons CSV. *https://commons.apache.org/proper/commons-csv*
[13] Simple Logging Facade for Java (SLF4J). *https://slf4j.org*
[14] Lightweight Java Game Library (LWJGL). *https://www.lwjgl.org*
[15] Gradle Component Metadata Rules. *https://docs.gradle.org/current/userguide/component_metadata_rules.html*
[16] Gradle Variant Selection and Attribute Matching. *https://docs.gradle.org/current/userguide/variant_aware_resolution.html*
[17] Maven Dependency Exclusions. *https://maven.apache.org/guides/introduction/introduction-to-optional-and-excludes-dependencies.html#dependency-exclusions*
[18] Maven Build Profiles. *https://maven.apache.org/guides/introduction/introduction-to-profiles.html*
[19] Dependency Analysis Gradle Plugin. *https://github.com/autonomousapps/dependency-analysis-gradle-plugin*
[20] Maven Dependency Plugin (dependency:analyze). *https://maven.apache.org/plugins/maven-dependency-plugin/analyze-mojo.html*
[21] Mend Renovate. *https://docs.renovatebot.com*
[22] GitHub Dependabot Version Updates. *https://docs.github.com/en/code-security/dependabot/dependabot-version-updates*

Dr. Jendrik Johannes
jendrik@onepiece.software

Dr. Jendrik Johannes beschäftigen die Themen Modularisierung und Wiederverwendung, zu denen er auch promoviert hat, schon lange. Er war einige Jahre im Gradle-Kernteam tätig. Als Trainer und Berater mit Schwerpunkt auf Buildautomatisierung hat er zuletzt viele Projekte gesehen, in denen das Abhängigkeitsmanagement unnötig komplex wurde. Dabei ist ihm die mangelhafte Aufarbeitung des Themas im Java-Ökosystem aufgefallen. Aus diesem Grund hat er die Website javarca.de ins Leben gerufen. Außerdem trägt er über das GradleX-Projekt dazu bei, das Open-Source-Tooling in diesem Bereich zu verbessern.

Java und GitOps
Thomas Michael, Cloudogu GmbH
Java™

Wir wollen mit unserer Java-Applikation in die Cloud, aber nicht mehr irgendwie. Der Weg dorthin unterscheidet sich stark von früher. Statt manuell zu deployen, setzen moderne Teams auf GitOps. In diesem Artikel möchte ich praxisnah zeigen, wie man eine Java-Anwendung mit einem echten GitOps-Workflow betreibt. Dabei präsentiere ich nicht nur das fertige Ergebnis, sondern leite Schritt für Schritt durch den Entstehungsprozess: vom ersten lokalen Kubernetes-Cluster über ein eigenes Docker-Image bis hin zu Staging-Umgebungen mit Kustomize und einem separaten Konfigurations-Repository.

Jede Entscheidung wird anhand eines realen Beispiels nachvollziehbar – und der Leser kann so den Ansatz direkt auf eigene Projekte übertragen. Ziel ist ein unkomplizierter, praxisorientierter Einstieg in GitOps mit Java, der zeigt, wie sich klassische Deployment-Prozesse in moderne, deklarative Workflows überführen lassen. Als konkretes Beispiel verwende ich die Spring-Boot-PetClinic [11], sodass der Leser die Schritte leicht auf sein eigenes Projekt adaptieren kann.

Warum per GitOps in die Cloud?

Reicht es nicht, wenn ein CI/CD-Server automatisch die Anwendung deployt? Oder zumindest die Anwendung baut und danach ein Admin die Anwendung deployt? Warum soll eine Applikation per GitOps in die Cloud?

GitOps ist ein Ansatz, Anwendungen und Infrastruktur vollständig über Git zu verwalten. Anstatt Konfigurationen manuell zu ändern oder per Skript auszuspielen, wird jede Änderung als Pull-Request in einem Git-Repository vorgenommen. Git wird damit zur einzigen, verbindlichen Quelle der Wahrheit – nicht nur für den Anwendungscode, sondern auch für die gesamte Betriebsumgebung.

GitOps basiert auf vier Prinzipien:

1. **Declarative:**
   Infrastruktur und Deployments werden deklarativ beschrieben – der gewünschte Endzustand steht im Vordergrund, nicht der Weg dorthin.
2. **Versioned and Immutable:**
   Der gewünschte Zustand wird vollständig versioniert und unveränderlich in Git abgelegt. Jede Änderung ist nachvollziehbar und jederzeit rücksetzbar.
3. **Pulled Automatically:**
   Das System holt sich Änderungen selbstständig aus Git. Deployments werden nicht manuell angestoßen, sondern automatisch ausgerollt.
4. **Continuously Reconciled:**
   Ein GitOps-Operator vergleicht dauerhaft den gewünschten mit dem tatsächlichen Zustand und korrigiert Abweichungen automatisch – Drift wird verhindert.

Kurz gesagt: GitOps macht Deployments reproduzierbar, sicher und automatisiert, indem Git zum zentralen Steuerungsmechanismus der gesamten Infrastruktur wird.

Weitere Vorteile sind:
- Kein externer Zugriff auf das Cluster erforderlich
- Keine Zugangsdaten auf einem CI-Server nötig
- Hohe Skalierbarkeit durch ein zentrales Repository für viele Anwendungen oder Umgebungen

Oder wie es Alexis Richardson einmal formulierte: *„The right way to do DevOps."*

Vorbereitungen

Was wird benötigt, um lokal den GitOps-Prozess aufzubauen?
- Ein GitOps-Operator
- Eine lokale Cloud für den GitOps-Operator
- Eine Docker-Registry, um Docker-Images in die lokale Cloud zu bekommen.
- Ein Docker-Image der Spring-Boot-PetClinic

Für das lokale Testen empfiehlt sich eine lokale Cloud. Die gängigsten Tools zum Erstellen sind dafür *Minikube* [1] und *k3d* [2]. Als GitOps-Operator wird *Flux CD* [3] oder *Argo CD* [4] bevorzugt. Aus persönlicher Erfahrung fokussiert sich dieser Artikel auf *k3d* mit *Argo CD*, aber jede andere Konstellation ist genauso möglich.

Um genau diese Komponenten zusammenzubringen, nutze ich das Tool GOP (*GitOps Playground*), mit dem sich die gewünschte Testumgebung in kurzer Zeit aufsetzen lässt. Das Open-Source-Tool bietet eine umfangreiche ReadMe [5] für unterschiedlichste Anwendungsfälle. Für uns reicht für den Anfang das Ausführen des Befehls aus *Listing 1*: Damit wird ein lokales Kubernetes über *k3d* mit einer

```
bash <(curl -s \
https://raw.githubusercontent.com/cloudogu/gitops-playground/main/scripts/init-cluster.sh) \
&& docker run --rm -t --pull=always -u $(id -u) \
-v ~/.config/k3d/kubeconfig-gitops-playground.yaml:/home/.kube/config \
--net=host \
ghcr.io/cloudogu/gitops-playground --yes --argocd --ingress-nginx --base-url=http://localhost --registry
```

Listing 1: Aufsetzen eines lokalen k3d Kubernetes mit Docke-Registry und Argo CD.

Docker-Registy und mit *Argo CD* als GitOps-Operator erzeugt.

Nach 2 bis 3 Minuten ist alles hochgefahren und einsatzbereit. Die Docker-Registry lauscht auf *http://localhost:30000*. Dort muss später das eigene Image der Spring-Boot-PetClinic hin. *Argo CD* lässt sich lokal über die URL *http://argocd.localhost* erreichen.

Erste Schritte mit GitOps

Argo CD benötigt ein Projekt und eine Applikation. In diesem Artikel werde ich die benötigen Ressourcen direkt über Kubernetes anlegen. Um die PetClinic nun per *GitOps* in *Kubernetes* zu deployen, muss nur die YAML-Datei aus *Abbildung 1* dem Kubernetes hinzugefügt werden. Dieses legt eine *Argo-CD*-Application an. Alternativ kann der Leser auch das UI von *Argo CD* nutzen. Bei der Projektauswahl in Zeile 10 wird das Projekt *argocd* wieder verwendet. Empfohlen ist ein eigenes Projekt, aber für den Einstieg ist es gut genug. Wenn der Leser dem Beispiel von oben gefolgt ist, sollte lokal die URL *http://argocd.localhost* erreichbar sein. Bei dieser Standard-Konfiguration kann sich der Leser über den Benutzernamen „admin" mit dem Passwort „admin" in das UI von *Argo CD* einloggen. Die initialen Passwörter sind auch in aus der ReadMe *[5]* hinterlegt.

Argo CD erstellt daraus eine Applikation und wenn alles gut läuft, sollte diese beim Leser genau wie in *Abbildung 2* aussehen. Dort sieht man auch, dass die PetClinic aus mehr als nur dem Jar-Package besteht. Zusätzlich ist eine Datenbank mit ihren Secrets erstellt worden.

Erstes Fazit

Mit nur einer YAML-Datei haben wir die PetClinic per *GitOps* in *Kubernetes* deployt.

Allerdings muss auch erwähnt werden:

- Dies ist nicht unser Git-Repository und schon gar nicht unser Docker-Image.
- Kubernetes-Manifeste waren auch schon vorbereitet.
- Wie sieht das Staging-Konzept aus?
- Kubernetes-Manifeste und Applikationscode sind im selben Repository.

Im nächsten Schritt werden wir einige der Punkte beheben.

Extended GitOps

Zuerst lösen wir die Herausforderung, dass wir eine fremde PetClinic bei uns laufen lassen. Die nächsten Schritte sind daher:

- Forken und Clonen der PetClinic
- Bauen eines eigenen Docker-Image
- Hochladen des Docker-Image in unsere Docker-Registry
- Erstellung einer neuen Argo-CD-Applikation

Die lokale Docker-Registry läuft unter *http://localhost:30000*. Nach dem Forken und Clonen der Spring-Boot-PetClinic kann im lokalen, geklonten Repository mit dem Maven-Befehl aus *Listing 2* direkt das Docker-Image gebaut und in die lokale Docker-Registry gepusht werden.

In den Kubernetes-Manifesten wird noch auf originale Docker-Image verwiesen. Dies muss durch unser eigenes ersetzt werden. Im Order k8s liegt die PetClinic.yml, dort muss das aktuelle Image dsyer/PetClinic durch localhost:30000/my-PetClinic ausgetauscht werden. Die aktuelle Argo-CD-Applikation synchronisiert sich noch mit der PetClinic von spring-projects. Dies kann entweder aus der vorhandenen Argo-CD-Applikation ausgetauscht werden oder man kann eine neue Argo-CD-Applikation mit unserem Repository angelegen. Dafür kann der Code aus *Abbildung 1* wiederverwendet werden, es muss nur Zeile 13 mit dem richtigen Repository angepasst

```
1  apiVersion: argoproj.io/v1alpha1
2  kind: Application
3  metadata:
4    name: petclinic-from-github
5    namespace: argocd
6  spec:
7    destination:
8      namespace: petclinic
9      server: https://kubernetes.default.svc
10   project: argocd
11   source:
12     path: k8s
13     repoURL: https://github.com/spring-projects/spring-petclinic
14     targetRevision: main
15   syncPolicy:
16     automated:
17       enabled: true
18       prune: true
19       selfHeal: true
20     syncOptions:
21       - CreateNamespace=true
22
```

Bild 1: PetClinic als Argo-CD-Application © Thomas Michael

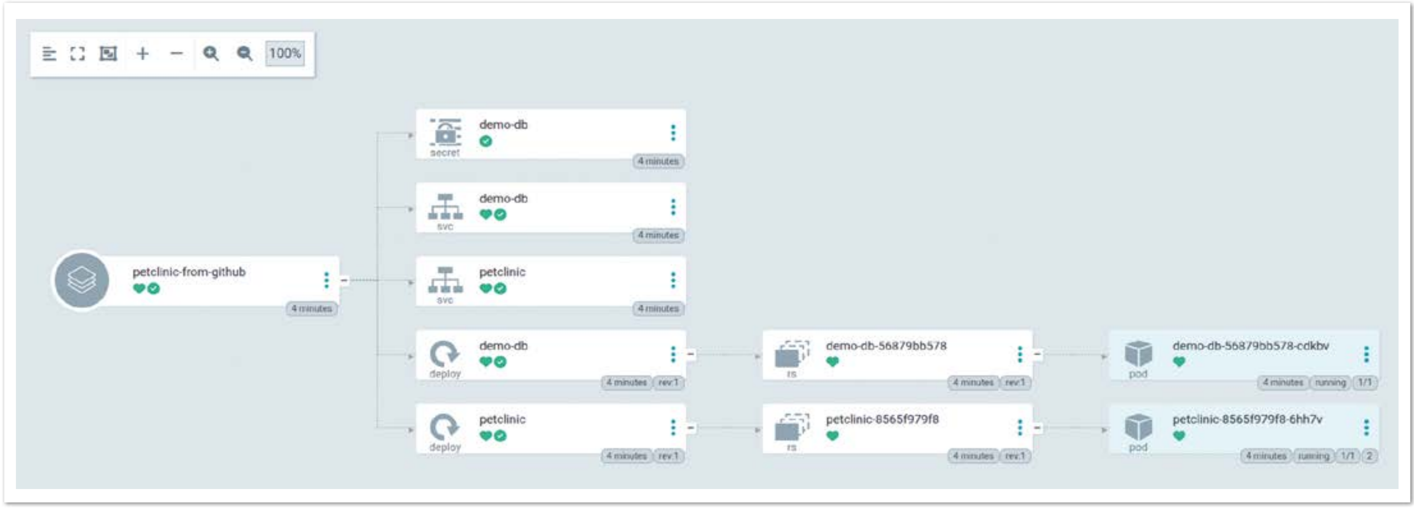


Bild 2: Screenshot der Ressourcen der PetClinic in Argo CD © Thomas Michael

```
mvn spring-boot:build-image \
  -Dspring-boot.build-image.imageName=localhost:30000/my-PetClinic:latest \
  -Dspring-boot.build.publish=true \
  -Dspring-boot.build-image.docker.publishRegistry.username=admin \
  -Dspring-boot.build-image.docker.publishRegistry.password=admin
```

Listing 2: Bauen des Docker-Image inklusive Push in die Registry per Maven

werden. Nicht vergessen zu pushen, da Argo CD sich mit eurem lokalen Repository synchronisiert!

Wenn dann Argo CD alles refresht und synchronisiert, dann sieht man in der Pod-Ansicht, dass das Image nun das lokale Image ist *(siehe Abbildung 3)*.

Fazit

Mit diesem Aufbau wird nun ein eigenes Repository mit einem eigenen Docker-Image in der lokalen Cloud deployt. Weiterhin offene Punkte sind:

- Kubernetes-Manifeste sind für genau eine Stage vorbereitet.
- Wie kann ein Staging-Konzept aussehen?
- Kubernetes-Manifeste und Applikationscode sind immer im selben Repository.

Um diese Herausforderungen zu adressieren, benötigen wir ein Manifest-Management-Tool. Zwei etablierte Lösungen stehen zur Auswahl: *Helm* [6] und *Kustomize* [7].

Helm arbeitet Template-basiert mit Values-Dateien pro Stage und ermöglicht Wiederverwendung durch Chart-Repositories. *Kustomize* nutzt Overlays, die auf eine Basis-Konfiguration aufbauen, und

```
cd k8s
# 1. Base erstellen
mkdir -p base overlays/{dev,staging,production}
# 2. Bestehende Manifeste nach base/ verschieben
mv *.yml base/
# 3. Base kustomization.yml erstellen
cat > base/kustomization.yml <<EOF
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- PetClinic.yml
- db.yml
# ... alle deine Manifeste
EOF

# 4. Dev Overlay erstellen
cat > overlays/dev/kustomization.yml <<EOF
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

namespace: PetClinic-dev

bases:
- ../../base

patchesStrategicMerge:
- patches.yml

commonLabels:
environment: dev
EOF
```

Listing 3: Erstellung der Ordner und YAML-Dateien für Kustomize

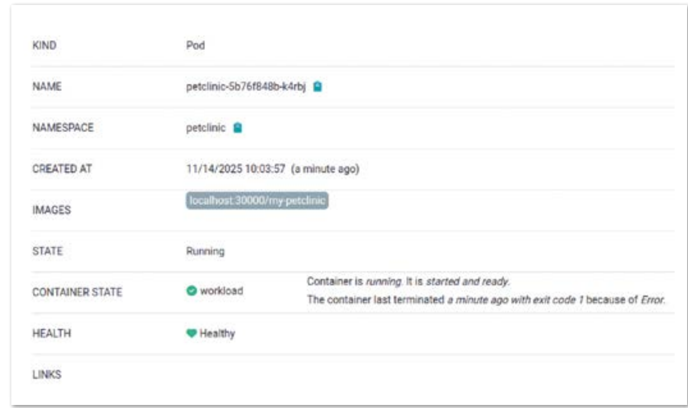


Bild 3: Detailansicht des PetClinic-Pod © Thomas Michael

kommt ohne Template-Sprache aus. Beide Tools lösen das Multi-Stage-Problem für Enterprise-Umgebungen.

Helm ist aufgrund seiner Verbreitung und Repository-Unterstützung jedoch die gängigere Wahl. Allerdings ist *Kustomize* in kubectl integriert und kommt ohne externe Template-Sprache aus. Das bedeutet, es ist weniger abstrakt, dafür explizit und das Beste: wir können die vorhanden Kubernetes-Manifeste wiederverwenden. Somit ist es genau richtig für den Einstieg.

Staging

Ein Staging-Konzept sorgt dafür, dass Änderungen nicht direkt im Live-Betrieb landen. Durch klar getrennte Umgebungen können neue Features und Konfigurationen realitätsnah getestet werden, bevor sie produktiv gehen. So steigt die Stabilität und Risiken werden deutlich reduziert.

Um dies im Rahmen von *GitOps* umzusetzen, nutzen wir ein Manifest-Management-Tool zur Verwaltung der Stages. Als Einstieg kommt *Kustomize* zum Einsatz. Die nächsten Schritte dafür sind:

- Erstellen einer Kustomize-Ordner-Struktur, bestehend aus base und overlays
- Erstellen einer Staging-Ordner Struktur im overlays-Ordner
- Migration der bestehenden Kubernetes-Manifeste in den base-Ordner
- Erstellung der verschiedenen Stages, dev, staging und production

Der Code in *Listing 3* legt direkt die notwendigen Ordner und die entsprechenden Dateien an, wenn man ihn im Root-Folder seiner PetClinic ausführt.

Die Dateien für die Stages „staging" und „production" sind analog anzulegen. Ein fertiges Beispiel findet sich in diesem Git-Repository [8] im Branch kustomize.

Für die 3 Stages werden auch drei Argo-CD-Applikationen benötigt.

Als Vorlage kann erneut *Abbildung 1* genutzt werden, mit dem Unterschied in *Listing 4* für die Nutzung der Dev-Stage. Für die anderen Stage muss der path von „k8s/overlays/dev" entsprechend auf „k8s/overlays/staging" und „k8s/overlays/poduction" angepasst werden.

```
source:
  path: k8s/overlays/dev
  repoURL: https://github.com/ThomasMichael1811/
spring-PetClinic
```

Listing 4: Code-Snippet einer Argo-CD-Application mit Anpassung für Kustomize

Nach dem Pushen des Codes und dem Anlegen der drei ArgoCD-Applikationen sollte die PetClinic in drei Stages laufen und Argo CD ähnlich wie in *Abbildung 4* aussehen.

Die Magie dahinter ist denkbar einfach: Kustomize nutzt die Original Kubernetes-Manifeste aus dem alten k8s-Ordner. Diese sind jetzt im k8s/base-Ordner und bilden die Basis für alle Stages.

Jede Stage hat im Order k8s/overlays/ ihren eigenen Unterordner, um eigene Anpassungen zu machen. Üblicherweise sind dies andere Werte für das Environment, eine andere Datenbankverbindung und andere Secrets. Weitergehende Informationen und Anwendungsbeispiele sind online bei kubectl *[9]* zu finden.

Fazit Staging

Mit der Einführung von *Kustomize* als Manifest-Management-Tool wurde das Staging-Problem gelöst. Jede Stage hat die Möglichkeit die Kubernetes-Manifeste bei Bedarf anzupassen und sogar um weitere Ressourcen zu erweitern. Falls zum Beispiel ein Mock-Server in einer Stage gebraucht wird, dann kann das Kubernetes-Manifest dafür einfach mit in dem Ordner abgelegt und in der kustomization.yaml der Stage bekannt gemacht werden.

Jede Änderung an den Kubernetes-Manifesten ist gleichzeitig eine Änderung am Applikations-Repository. Man stelle sich folgenden Ablauf vor: Ein Build-Server erstellt aus dem Applikationscode ein neues Docker-Image, vergibt einen Tag und pusht das Image in die Registry. Soll dieses neue Image anschließend in den verschiedenen Stages verwendet werden, muss im Kubernetes-Manifest der Image-Tag aktualisiert werden. Dadurch entsteht jedoch erneut ein Commit im selben Repository, wodurch der Build-Prozess wieder ausgelöst wird – und der Zyklus beginnt von vorn.

Jetzt ist es an der Zeit, ein eigenes Repository zu erstellen. Die nächsten Schritte sind:

- Die Kubernetes Manifeste in das neue Repository auslagern.
- Die Argo-CD-Applikationen entsprechend anpassen.

Die Umsetzung ist denkbar einfach und wurde als Beispiel auf GitHub *[10]* zur Verfügung gestellt. Es besteht nur aus den k8s-Ordern, die vorher im PetClinic-Repository waren. Für die Argo-CD-Applikation muss nur „spec.source.repoURL"auf das neue Repository geändert werden.

Fazit und Ausblick

Im letzten Schritt wurde ein neues Repository für die Kubernetes-Manifeste eingeführt. Damit ist der Code für die Applikation von der Konfiguration von Kubernetes getrennt. Dies entspricht dem Separate-Infineon-Positioniere-Pattern, auch Two-Repo-Pattern genannt. Für eine gute Repository-Struktur gibt es weitere Pattern, die alle ihre Vor- und Nachteile haben. Welches Pattern für das ei-

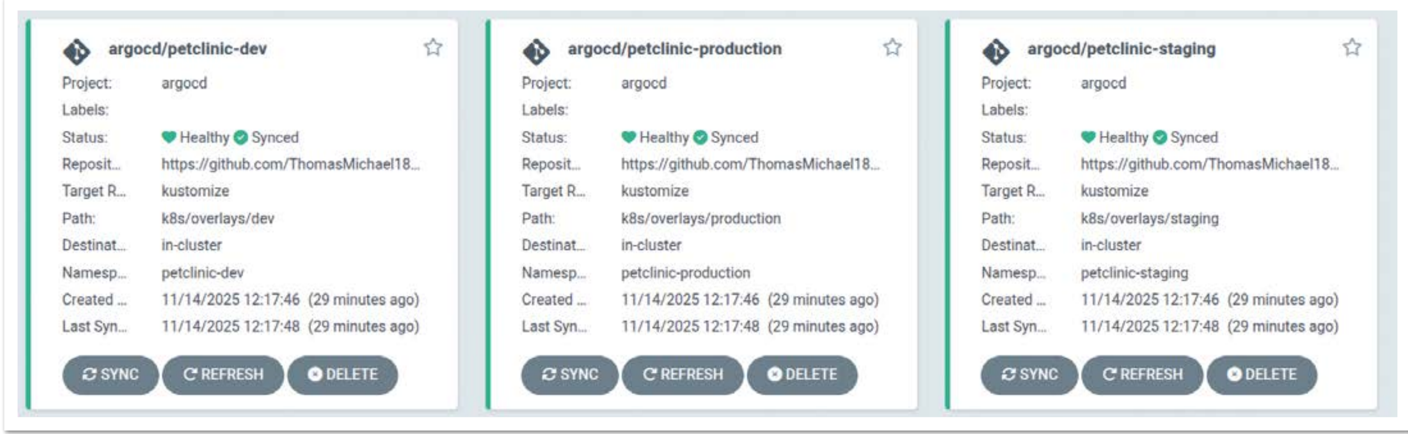


Abbildung 4: Screenshot von Argo CD mit der PetClinic in drei Stages ©Thomas Michael

```
bash <(curl -s \
https://raw.githubusercontent.com/cloudogu/gitops-playground/main/scripts/init-cluster.sh) \
&& docker run --rm -t --pull=always -u $(id -u) \
-v ~/.config/k3d/kubeconfig-gitops-playground.yaml:/home/.kube/config \
--net=host \
ghcr.io/cloudogu/gitops-playground --yes --argocd --ingress-nginx --base-url=http://localhost --registry --jenkins
--content-examples
```

Listing 5: Startparameter für ein komplettes GitOps-Beispiel mit der PetClinic.

gene Projekt das Richtige ist, kommt immer auf das Projekt an und kann somit variieren.

In diesem Artikel war mir wichtig, dass der Leser einen einfachen und praktischen Einstieg in das Thema GitOps mit Java bekommt. Ich hoffe, ich konnte den einen oder anderen für das Thema begeistern. Den Motivierten möchte ich zum Abschluss noch ein komplettes PetClinic mit Helm-Charts-Beispiel empfehlen: Das eingangs benutze Tool GOP *[5]*, das zum Aufsetzen der lokalen Kubernetes-Instanz mit Argo CD genutzt wurde, liefert auf Wunsch auch ein PetClinic-Beispiel mit. Dieses folgt dem Rendered-Manifest-Pattern, bei dem ein Build-Server (hier Jenkis) die Kubernetes-Manifeste erstellt und in einem Repository ablegt. Mit diesem Repository synchronisiert sich Argo CD. Der Entwickler kann für die Stages verschiedene Environments über values.yaml-Dateien erstellen und die ganze Helm-Chart-Magie wird vom Jenkins erledigt. Mit dem Befehl in *Listing 5* ist nach zirka 5 Minuten das komplette Beispiel aufgebaut und kann vom Leser durchstöbert und ausprobiert werden.

Quellen

[1] *https://minikube.sigs.k8s.io/*
[2] *https://k3d.io/*
[3] *https://fluxcd.io/*
[4] *https://argo-cd.readthedocs.io/*
[5] *https://github.com/cloudogu/gitops-playground/blob/main/README.md*
[6] *https://helm.sh/*
[7] *https://kustomize.io/*
[8] *https://github.com/ThomasMichael1811/spring-PetClinic/tree/kustomize*
[9] *https://kubectl.docs.kubernetes.io/guides/introduction/kustomize/*
[10] *https://github.com/ThomasMichael1811/PetClinic-deployment*
[11] *https://github.com/spring-projects/spring-PetClinic*

Thomas Michael
Cloudogu GmbH
X: @Kroneker_Delta

Thomas Michael arbeitet als Cloud Engineer bei der Cloudogu GmbH in Braunschweig und hat sich auf moderne Cloud-Infrastrukturen, Kubernetes, Helm-Charts und GitOps spezialisiert. Nach seinem Informatikstudium startete er mehrere Jahre in der Software-Entwicklung mit Java und Node.js, bevor ihn die Cloud komplett in ihren Bann zog.
Aktuell arbeitet er vor allem an der Entwicklung des GitOps Playground (GOP) in Groovy – einem Tool, das lokale GitOps-Testumgebungen schnell aufsetzt und Entwicklern den Einstieg in Kubernetes und Argo CD erleichtert. Sein Ziel ist es, komplexe Cloud-Prozesse verständlich, reproduzierbar und skalierbar zu machen.
Privat begeistert er sich für Klettern, Bouldern und Klavierspielen, wodurch er Ausgleich und Inspiration für seine beruflichen Projekte findet.

Java spricht mit Maschinen – Wie Open Source die Industrie wachrüttelt

Christofer Dutz, ToddySoft GmbH

In der klassischen IT-Welt gilt die Industrieautomation oft als fremder Planet. Doch mit Projekten wie Apache PLC4X, Apache TsFile und Apache IoTDB findet Java längst den Weg auf den Shopfloor und verändert damit die Art, wie Maschinen Daten liefern, speichern und verstehen.

Die unsichtbare Mauer zwischen IT und OT

Wenn Softwareentwickelnde an „Daten" denken, dann meist an APIs, Microservices oder Datenbanken, nicht aber an Produktionsmaschinen, Steuerungen oder Sensoren. In der Industrie hingegen sprechen Maschinen eine Vielzahl von Sprachen, die für viele IT-Teams fremd klingen: Protokolle wie S7, Modbus, EtherNet/IP oder in den letzten acht Jahren auch OPC-UA. HTTP und JSON sind hier eher Fremdwörter. Einzig MQTT ist mittlerweile in beiden Welten anzutreffen.

Über Jahrzehnte hinweg existierten diese beiden Welten – IT und OT (Operational Technology) – nebeneinander, durch eine unsichtbare, aber schier unüberwindbare Mauer getrennt. Doch diese Mauer beginnt zu bröckeln. Immer mehr Unternehmen wollen Produktionsdaten mit IT-Systemen verknüpfen, um Effizienz, Skalierbarkeit, Nachhaltigkeit und Transparenz zu steigern und dabei die Kosten zu senken.

Der Wille ist da, die Realität sieht jedoch, wie bereits erwähnt, oft anders aus: heterogene Systemlandschaften, inkompatible Protokolle, langsame Netzwerke, Lizenzkosten und Sicherheitsrisiken. Genau hier setzt eine kleine, aber wachsende Familie von Apache-Projekten an, die Java eine neue Bühne in der Industrie eröffnet.

Apache PLC4X – Java als Brückenbauer

Abbildung 1: Apache-PLC4X-Projekt-Logo (Quelle: Apache PLC4X Website [1])

Im Zentrum steht Apache PLC4X *[1]*, ein Open-Source-Projekt, das einen Traum vieler Ingenieur:innen erfüllt: eine einheitliche API, um mit industriellen Steuerungen zu kommunizieren, dabei aber unabhängig vom Hersteller oder Protokoll zu sein.

Was JDBC für relationale Datenbanken ist, will PLC4X für die Industrieautomation sein. Statt für jeden Typ SPS (Speicherprogrammierbare Steuerungen) ein neues Treiberpaket beschaffen und integrieren zu müssen, erlaubt PLC4X es, Datenpunkte („Tags") einheitlich zu lesen und zu schreiben. Unter der Haube spricht es die nativen Protokolle der Geräte. Bei Siemens etwa S7, ADS bei Beckhoff, Modbus bei Schneider Electric und EtherNet/IP bei Allen Bradley und vereinfacht damit die Anbindung erheblich.

OPC UA *[2]* kann man hier als komplementären Ansatz ansehen. Hier wird ebenfalls versucht, eine einheitliche Maschinenanbindung zu ermöglichen, hier allerdings indem alle Zielgeräte so angepasst sein müssen, dass sie das „neue" Protokoll unterstützen.

Während moderne, generische Standards wie OPC UA oder Sparkplug B *[3]* viele, teils auch durchaus nützliche Metadaten mit übertragen, arbeitet PLC4X so nahe wie möglich an der Hardware. Das Ergebnis: weniger Netzwerklast, geringere Latenzen und weniger Frust, wenn Steuerungen „grummelig" werden, weil sie von zu vielen Anfragen überfordert sind.

Und das Beste: PLC4X ist unter anderem in Java geschrieben. Zusätzlich bietet PLC4X auch passende Implementationen in C, Go, Python und vielen weiteren Sprachen – das „X" in PLC4X haben wir bewusst gewählt, um im Gegensatz zu PLC4J die Mehrsprachigkeit zu unterstreichen. Es bietet dieselbe solide Architektur, die Java-Entwickelnde schätzen. Starke Typisierung, klar strukturierte APIs, gute Testbarkeit.

Datenflut trifft Engpass – die Herausforderung der Industrie

Sobald die Kommunikation steht, folgt das nächste Problem: die schiere Datenmenge. Produktionslinien, Windparks oder Züge erzeugen bis zu Millionen Messwerte pro Minute oder gar pro Sekunde. Doch die wenigsten Systeme sind darauf ausgelegt, solche Datenmengen effizient zu speichern und zu übertragen.

Oft wird jedes Gerät von mehreren Clients parallel abgefragt: Dashboards, MES-Systeme sowie POCs (Proofs of Concept) der Data-Scientists. Teilweise wollen diese auch noch dieselben Werte. Das Ergebnis: überlastete Netzwerke, blockierte Steuerungen und ein gigantischer Berg an redundanten Daten, der irgendwie verarbeitet werden oder in die Cloud übertragen werden soll.

Doch Clouds sind nicht gratis und Internet-Verbindungen in der Praxis oft nicht leistungsfähig oder nicht stabil genug. Schon gar nicht im ländlichen Raum, auf Schiffen oder in Fahrzeugen.

Genau hier kommen zwei weitere Java-basierte Apache-Projekte ins Spiel: Apache TsFile und Apache IoTDB.

Apache TsFile – effiziente Zeitreihenspeicherung für die reale Welt

Abbildung 2: Apache-TsFile-Projekt-Logo (Quelle: Apache TsFile Website [4])

Apache TsFile *[4]* ist das Speicherformat von Apache IoTDB *[5]*, wurde aber als eigenständiges Projekt ausgegliedert, um es auch unabhängig vom Datenbankserver nutzen zu können. Es ist hochgradig auf Zeitreihendaten optimiert. Es bietet sich also an, damit genau jene Art von Daten zu speichern, die Maschinen, Sensoren und Steuerungen im Sekundentakt erzeugen.

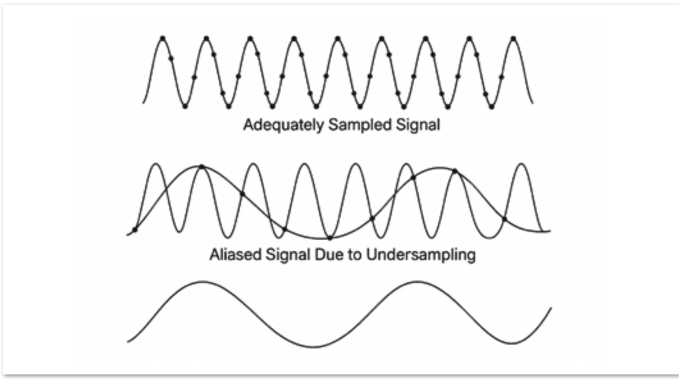


Abbildung 3: Das Sampling-Problem (Bild mit ChatGPT generiert)

Im Gegensatz zu klassischen Datenformaten, arbeitet TsFile mit hoher Kompression und zeitbasierter Aggregation, ohne dabei an Performanz einbüßen zu müssen. Verlustfreie Speicherung reduziert Datenvolumen um bis zu 90 %, leicht verlustbehaftete Varianten sogar auf 1 % der ursprünglichen Größe – ideal für Temperatur-, Druck- oder Vibrationsdaten. Millionen von Schreiboperationen pro Sekunde sind hier kein Problem.

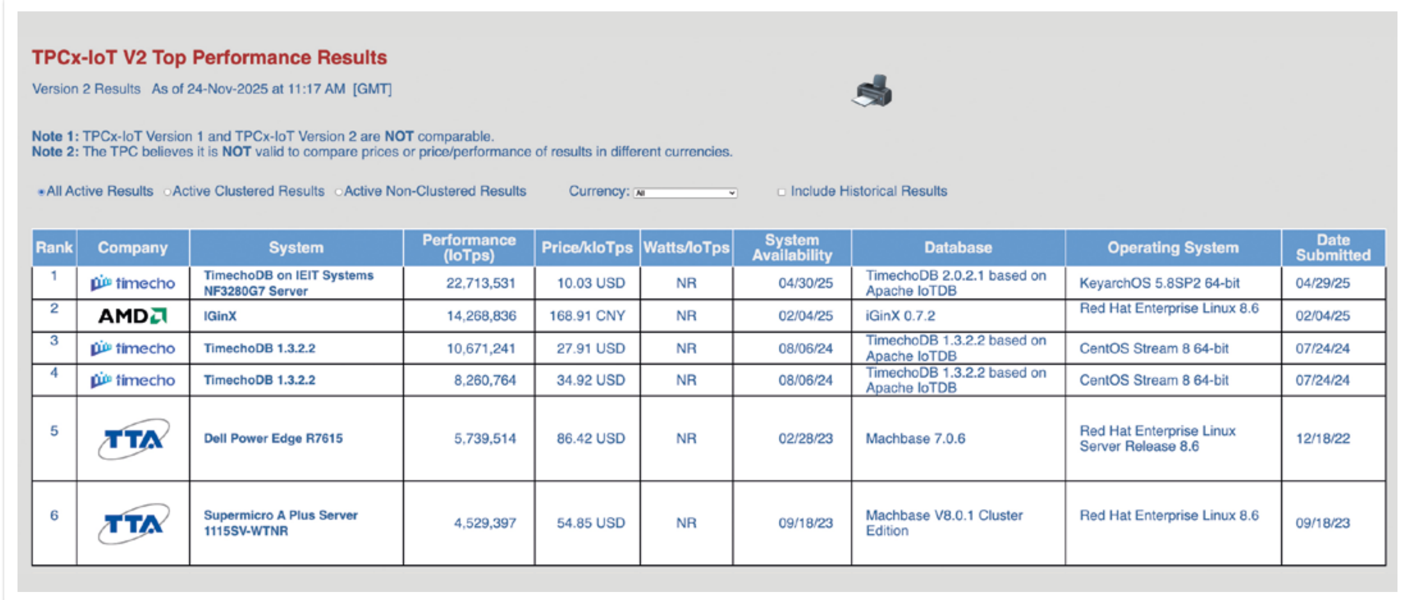
Ein besonderer Vorteil: TsFile lässt sich direkt auf Geräten, unabhängig von IoTDB, einsetzen. Bibliotheken in C und C++ ermöglichen es, dass SPS-Hersteller oder Embedded-Entwickelnde Daten schon lokal auf dem Gerät schreiben können.

Ein großes Problem, dass in der Industrie die Problematik verschärft, ist das viele IT-Lösungen durch Polling versuchen, den Zustand und die Signale innerhalb der Anlagen durch Sampling zu approximieren. Hierbei erschwert das „Sampling Problem" die Sache. Um nämlich ein Signal durch Sampling zu approximieren, ist mindestens die doppelte Frequenz des beobachteten Signals für das Sampling nötig *(siehe Abbildung 3)*. Wenn es also um hochfrequente Signale geht, ist es daher teils gar nicht möglich, diese mittels Sampling zu approximieren.

Ein aktives Schreiben des Signals auf dem Gerät erlaubt das Mitschreiben beliebig hochfrequenter Daten. Diese wären dann obendrein Zyklen-synchron. Beim Sampling vieler Daten kommt es nämlich zu einem weiteren Problem: Üblicherweise ist die Datenmenge, die pro Request übertragen werden kann, beschränkt. Ein System wird also einen Teil der Daten übertragen, dann seine Arbeit fortsetzen und dann den nächsten Block senden. Dieser Teil passt nicht immer zu 100 % zu den Werten aus dem ersten Teil, da die Anlage in der Zwischenzeit weitergearbeitet hat.

CLOUD	WRITE THROUGHPUT [ops]	STORAGE CONSUMPTION [GiB]	READ THROUGHPUT [ops]	READ LATENCY [ms]	MONTHLY COSTS [$]	OPERATIONS PER COST [ops/$]
AWS medium	8.840.193	2	86.894	1	178	488,17
AWS medium	6.720.972	33	56.790	1	178	319,04
AWS medium	3.636.312	2	11.497	3	178	64,59
AWS medium	1.987.341	2	8.446	6	180	46,97
AWS medium	1.572.058	70	518	193	180	2,88
AWS medium	529.220	3	2.197	45	180	12,22
AWS small			crashed		95	

Abbildung 5: BenchANT-Ergebnisse (Quelle: BenchANT Website)



TPCx-IoT V2 Top Performance Results

Version 2 Results As of 24-Nov-2025 at 11:17 AM [GMT]

Note 1: TPCx-IoT Version 1 and TPCx-IoT Version 2 are **NOT** comparable.
Note 2: The TPC believes it is **NOT** valid to compare prices or price/performance of results in different currencies.

All Active Results · Active Clustered Results · Active Non-Clustered Results · Currency: All · Include Historical Results

Rank	Company	System	Performance (IoTps)	Price/kIoTps	Watts/IoTps	System Availability	Database	Operating System	Date Submitted
1	timecho	TimechoDB on IEIT Systems NF3280G7 Server	22,713,531	10.03 USD	NR	04/30/25	TimechoDB 2.0.2.1 based on Apache IoTDB	KeyarchOS 5.8SP2 64-bit	04/29/25
2	AMD	IGinX	14,268,836	168.91 CNY	NR	02/04/25	iGinX 0.7.2	Red Hat Enterprise Linux 8.6	02/04/25
3	timecho	TimechoDB 1.3.2.2	10,671,241	27.91 USD	NR	08/06/24	TimechoDB 1.3.2.2 based on Apache IoTDB	CentOS Stream 8 64-bit	07/24/24
4	timecho	TimechoDB 1.3.2.2	8,260,764	34.92 USD	NR	08/06/24	TimechoDB 1.3.2.2 based on Apache IoTDB	CentOS Stream 8 64-bit	07/24/24
5	TTA	Dell Power Edge R7615	5,739,514	86.42 USD	NR	02/28/23	Machbase 7.0.6	Red Hat Enterprise Linux Server Release 8.6	12/18/22
6	TTA	Supermicro A Plus Server 1115SV-WTNR	4,529,397	54.85 USD	NR	09/18/23	Machbase V8.0.1 Cluster Edition	Red Hat Enterprise Linux 8.6	09/18/23

Abbildung 6: TPCxIOT-Ergebnisse (Quelle: TPC Website)

Ein weiterer Vorteil ist: Die Anlage wird während eines Produktionszyklus nicht mit unnötigen IO-Aufgaben abgelenkt. Daher ist es zum Beispiel möglich, nach Abschluss des Produktionszyklus alle zugehörigen Daten zu übertragen, da die Anlage sowieso gerade „Pause hat". Somit belastet die Datenerfassung nicht mehr den Betrieb der Anlage. Bei mobilen Einheiten ist es also auch möglich, die Daten erst zu übertragen, wenn eine Verbindung besteht.

Die Übertragung der hochkomprimierten Daten kann dann zum Beispiel via MQTT erfolgen.

Apache IoTDB – das zentrale Gedächtnis der Industrie

Abbildung 4: Apache-IoTDB-Projekt-Logo (Quelle: Apache IoTDB Website [5])

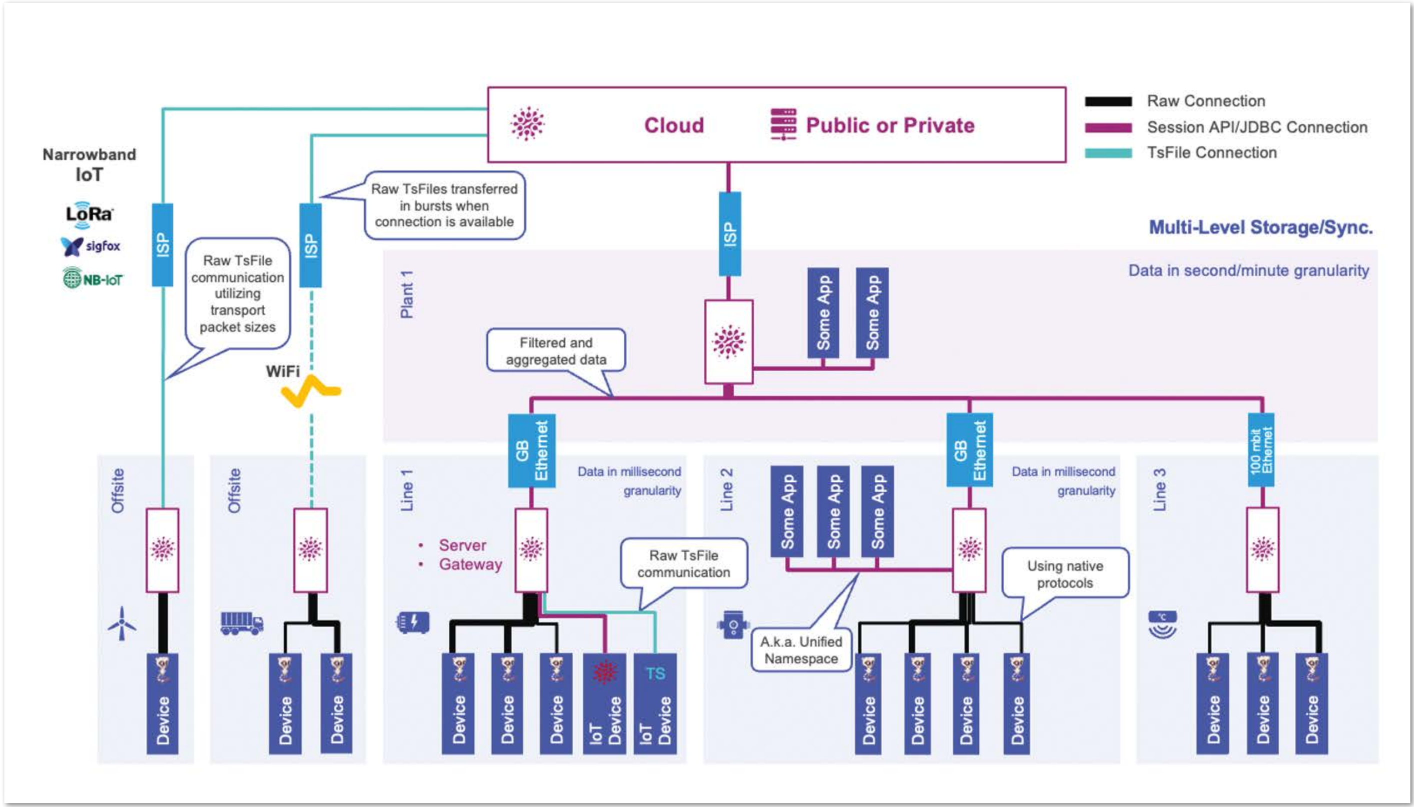


Abbildung 7: Big-Picture: Ideal-Lösung auf dem Shop-Floor (Quelle: Präsentation: „Magic industrial data acquisition with Apache PLC4X, TsFile and IoTDB" – Eclipse OCX 2024)

Während TsFile das Speichersubstrat bildet, ist Apache IoTDB die eigentliche Datenbank darüber. Dabei stellt IoTDB eher ein Time-Series Database Management System (TSDB) dar, das speziell für industrielle Anforderungen entwickelt wurde. Auch hier steckt wieder Java im Kern: IoTDB ist vollständig in Java implementiert und integriert sich nahtlos in moderne Softwarelandschaften. Es erlaubt Streaming-Abfragen, Aggregation, Kompression und Replikation. Auch IoTDB muss bezüglich Performanz nicht den Vergleich scheuen. In sämtlichen Benchmarks landet es hier auf dem Siegerpodest *(siehe Abbildung 5 [6] und 6 [7])*.

Von der Anlage bis zur Cloud

Setzt man PLC4X, TsFile und IoTDB zusammen, entsteht eine durchgängige, offene Datenpipeline. Vom Sensor über das Gateway bis zur Cloud. Auf den Geräten: TsFile-Bibliotheken in C/C++ sammeln Daten lokal, speichern sie komprimiert und übertragen sie bei Bedarf oder wenn eine Verbindung besteht. Auf den Gateways: PLC4X liest Daten aus Steuerungen über native Protokolle aus – effizient und ohne teure Zusatzhardware. In der Zentrale: IoTDB empfängt, aggregiert, repliziert und stellt die Daten über eine Java-API bereit *(siehe Abbildung 7)*.

Warum Java hier plötzlich glänzt

Lange Zeit galt Java in der Industrie als „zu groß" oder „zu träge". Doch diese Zeiten sind vorbei. Moderne JVMs, performante Garbage-Collector, modulare Laufzeitumgebungen und kompakte Build-Tools (GraalVM, JLink & Co.) haben den Weg frei gemacht. Dass gleich drei hochkarätige Apache-Projekte aus dem industriellen Umfeld in Java geschrieben sind, ist kein Zufall, sondern ein Zeichen dafür, dass sich IT und OT nicht mehr ausschließen müssen.

Fazit

Industrielle Datenakquise war lange ein Nischenthema, dominiert von proprietären Lösungen und teuren Gateways. Doch Open Source beginnt, auch hier die Regeln zu verändern. Mit Projekten wie Apache PLC4X, TsFile und IoTDB zeigt sich, dass Java nicht nur auf Servern und in der Cloud zuhause ist, sondern auch dort, wo Metall auf Elektronik trifft. Wer Java spricht, kann nun auch Maschinen verstehen und wer Maschinen versteht, kann die Industrie verändern.

Quellen

[1] Apache Software Foundation: Apache PLC4X Projekt Webseite: *https://plc4x.apache.org*

[2] Wikimedia: OPC UA Wikipedia Seite: *https://de.wikipedia.org/wiki/OPC_Unified_Architecture*

[3] Eclipse Foundation: Sparkplug B Project Seite: *https://sparkplug.eclipse.org/*

[4] Apache Software Foundation: Apache TsFile Projekt Webseite: *https://tsfile.apache.org*

[5] Apache Software Foundation: Apache IoTDB Projekt Webseite: *https://iotdb.apache.org*

[6] BenchAnt GmbH: Time Series: DevOps Benchmark Ranking *https://benchant.com/ranking/database-ranking* (Bitte bei „Workload Type" „DevOps Benchmark" auswählen)

[7] TPC Foundation: TPCxIoT Benchmark: *https://www.tpc.org/tpcx-iot/results/tpcxiot_perf_results5.asp?version=2*

Christofer Dutz

christofer.dutz@toddysoft.com

Christofer Dutz ist langjähriger Open-Source-Entwickler, Committer und PMC Member von Apache PLC4X, Apache TsFile, Apache IoTDB und vieler weiterer Open-Source Projekte sowie Geschäftsführer der neu gegründeten ToddySoft GmbH. Mit seinem Team entwickelt er schlüsselfertige Lösungen, die die IT- und OT-Welten mithilfe offener Open-Source-Lösungen wie PLC4X, TsFile und IoTDB verbinden.

Trunk-Based Development und Pairing

Jörg Liedl, SpringerNature AG

Ihr kennt das: Im Stand-up am Morgen sagt euer Kollege: „Ich habe hier einen Pull-Request gemacht, könnte diesen bitte jemand überprüfen?" Nach dem Meeting schaut ihr euch den Pull-Request an: 30 veränderte Files, alles in 3 Commits, sodass es schwierig ist, nachzuvollziehen, was warum passierte. Außerdem ist die Komplexität sehr groß und so richtig wohl fühlt ihr euch nicht, als ihr auf den Merge-Button klickt. So muss es aber nicht sein. Trunk-Based Development zusammen mit Pairing mit den Kolleg:innen kann hier Abhilfe schaffen.

Voraussetzungen für Trunk-Based Development

Um Trunk-Based Development einzusetzen, gibt es ein paar wichtige Voraussetzungen, die aber auch generell bei nicht Trunk-Based Development gegeben sein sollten:

- Eure Tests sollten bestenfalls alles abdecken, es sollte nicht nur Unittests geben, sondern auch Integrationstests. Test-Driven Development sollte kein Fremdwort für euch sein.
- CI/CD – Jeder Commit zum Main Branch sollte durch die Pipeline laufen und dann live gehen. Solltet ihr noch mit Releases arbeiten, könnt ihr ja mal überprüfen, ob das geändert werden kann. Ihr könnt zwar auch mit Releases auf Trunk-Based Development umstellen, allerdings habt ihr den Vorteil von vielen schnellen Releases nicht. Das Ziel sollte sein, möglichst oft und schnell zu releasen, um Fehler schneller zu entdecken. Dazu später mehr.
- Ebenso gehe ich davon aus, dass ihr an kleinen Microservices arbeitet und nicht an Monolithen, bei denen das Durchlaufen aller Tests 20 Minuten dauert. Sollte dies der Fall sein, wäre es eine gute Idee, diesen in kleine Microservices zu teilen, aber dazu gab es schon viele Beiträge in älteren Ausgaben, weswegen ich darauf nicht eingehen werde.

Disclaimer: Dieser Artikel bezieht sich auf den normalen Developer-Alltag in Firmen, in denen niemand ums Leben kommt, sollte mal ein Bug passieren. Also wenn ihr an Raketen, Flugzeugen, medizinischen Geräten oder an sicherheitsrelevanten Codes in Automobilen arbeitet, genießt diesen Artikel bitte mit Vorsicht. Ebenso, wenn ihr in einer Bank oder an der Börse arbeitet und euer Fehler eventuell einen Börsencrash nach sich ziehen könnte – außer, ihr wollt die Welt brennen sehen.

Pairing

Damit euer Code nicht ungesehen auf das Production-System geht, würdet ihr im Normalfall einen Pull-Request eröffnen. Da wir dies aber im Trunk-Based Development vermeiden wollen, gilt es eine, für euch eventuell neue, Technik anzuwenden, nämlich Pairing.

Idealerweise sitzt ihr zusammen im Büro nebeneinander. Beim Pairing unterscheidet man zwischen Driver und Navigator. Der Driver ist die Person, die am Keyboard sitzt und der Navigator ist die Person, die das Geschehen von außen betrachtet und Tipps gibt, wie die Änderung aussehen könnte. Sollte es zwischen den Personen ein großes Ungleichgewicht in der Erfahrung geben, sollte die Person am Keyboard die Person mit weniger Erfahrung sein. Um Erschöpfung vorzubeugen, ist es aber sowieso ratsam, alle 20 bis 40 Minuten zu wechseln.

Bleiben wir aber realistisch: Die meisten von uns arbeiten seit der Pandemie 3 bis 5 Tage von zuhause aus. Und selbst wenn ihr einmal im Büro seid, müssen eure Kolleg:innen, mit denen ihr pairen wollt, auch noch genau an denselben Tagen im Büro sein, wie ihr. Das wird, zumindest aus meiner Erfahrung, oft nicht der Fall sein. Doch in der heutigen Zeit, mit schnellen Internetverbindungen, gibt es ein paar Tools, die uns hier unterstützen.

Je nachdem, welches Programm ihr in eurer täglichen Arbeit für Video-Calls benutzt, könnt ihr dieses einfach weiterhin verwenden, um euren Bildschirm zu teilen. Ich persönlich benutze sehr gerne *Pop* [2], da man dort auch noch die Möglichkeit hat, mittels eines Stifts auf dem Bildschirm des anderen digital zu malen und etwas zu markieren. Dies ist oft sehr hilfreich, um zu zeigen, wo die andere Person klicken soll, beziehungsweise worüber man gerade redet. Bei allen anderen Möglichkeiten des Screen-Sharings wie *Slack, Teams, Google Meet, Around* usw. habe ich nie eine Möglichkeit gefunden, bei einem anderen auf dem Bildschirm zu zeichnen. Noch sind die Pro-Lizenzen bei Pop kostenlos, aber dies wird sich sicherlich noch ändern.

Wenn ihr in eurer Firma *JetBrains IntelliJ* in der Ultimate-Edition verwendet, habt ihr automatisch die Lizenz für *Code With Me* [3]. Hier kann man als Driver die Session starten, als Navigator bekommt man eine abgespeckte *IntelliJ*-Ansicht und sieht jederzeit, was die andere Person macht, und kann auch selbst den Code verändern. Gerade in internationalen Teams wie bei uns in der Firma, wo manche Leute eine deutsche, andere eine englische und wieder andere eine portugiesische Tastenbelegung haben, ist das Tool ein Riesenvorteil: Jede Person hat noch immer die eigene Tastenbelegung und sogar die Shortcuts können wie gewohnt verwendet werden. Auch wenn manche Leute am Mac arbeiten und andere auf Linux, ist dies ein Vorteil von *Code with Me*. Ich habe in der Vergangenheit auch an-

Definition von Trunk-Based Development

Eine Source-Control für die Quellcodeverwaltung (wie zum Beispiel Git), bei dem Developer:innen gemeinsam an einem einzigen Branch namens „Trunk" (zum Beispiel Main) arbeiten, ohne dem Druck, andere langlebige Entwicklungsbranches zu erstellen, indem es dokumentierte Techniken einsetzt. So vermeiden sie Merge-Probleme, unterbrechen den Build nicht und leben glücklich bis ans Ende ihrer Tage [1].

dere Tools probiert, werde aus Platzgründen jetzt aber nicht darauf eingehen. Wenn ihr also sowieso schon für die *IntelliJ* Ultimate-Edition bezahlt, könnt ihr *Code With Me* gratis ausprobieren.

Wenn ihr viel gemeinsam am Code arbeitet, solltet ihr, wie bereits erwähnt, alle 20 bis 40 Minuten wechseln. Damit ihr dafür nicht einen langlebigen Branch öffnen müsst, oder euch gegenseitig die Patches schicken müsst, gibt es das Open-Source-Tool *mob.sh* [4]. Dies erstellt einen sogenannten mob-Branch.

Wir starten einen neuen mob-Branch namens TEST. Wie ihr seht, wird ein neuer Branch unter mob/main-Test veröffentlicht *(siehe Listing 1)*.

Wenn ihr jetzt mit eurer Arbeit fertig seid und ihr im Pairing tauschen wollt, genügt ein *mob next* und die nächste Person ist an der Reihe *(siehe Listing 2)*.

Mit *mob done* beendet ihr jetzt den kurzlebigen Branch *(siehe Listing 3)*.

Jetzt könnt ihr nach erfolgreichem Durchlaufen aller Tests das Ganze wieder zum Main-Branch committen und pushen.

Wenn ihr noch mehr zum Thema Remote Pairing und auch zu Mob Programming (also 1 Driver und viele Navigatoren) erfahren wollt, empfehle ich euch die Seite „Remote Mob Programming" [5]. Hier gibt es auch ein empfehlenswertes kostenloses E-Book zum Download.

Veröffentlicht oft und kleinteilig

Wie schon oben in den Voraussetzungen erwähnt, sollte euer Ziel sein, oft und schnell den Code live zu veröffentlichen. Die Testabdeckung ist hier das A und O, um jederzeit Code veröffentlichen zu können, ohne Angst haben zu müssen, dass euch das ganze System um die Ohren fliegt.

Mit Open-Source-Tools wie *JaCoCo* [6] oder *SonarQube* [7] könnt ihr überprüfen, wie es um eure Codeabdeckung steht.

Gehen wir davon aus, dass ihr ein Refactoring machen wollt und euer Code auch durch Tests abgedeckt ist. Wenn ihr also aus langem Code mehrere Funktionen extrahiert, hindert euch nichts daran, dies sofort zu pushen, solange die Tests noch immer alle durchlaufen. Wenn sowohl Driver und Navigator mit dem Refactoring zufrieden sind, dann pusht es direkt auf den Main-Branch.

Zugegeben, wenn man mit Trunk-Based Development anfängt, fühlt es sich anfangs falsch an und kostet einiges an Überwindung, nicht wie gewohnt alle Änderungen, die man machen will, in einem Pull-Request zu veröffentlichen. Stattdessen pusht man jetzt kleine Commits direkt in den Main-Branch. Dadurch habt ihr den Vorteil, dass ihr keine Merge-Konflikte mit anderen Kolleg:innen haben werdet. Diese wiederum haben ebenso sofort den neuesten Code und können ihrerseits arbeiten, ohne später euren Code kompliziert mergen zu müssen. Eine Win-win-Situation.

Der Code wird, eine ordentlich CI/CD-Pipeline vorausgesetzt, direkt veröffentlicht. Wenn ihr mit zwei Monitoren arbeitet, schadet es nicht, *Sentry*, *Grafana* oder euer Tool der Wahl für Fehlererkennung am zweiten Monitor zu öffnen, um sofortige Fehler auf dem Server zu bemerken. Auch wenn die Tests zuvor alle durchliefen, kann es trotzdem passieren, dass bestimmte Teile nur schlecht getestet waren. Der Vorteil bei kleinen Commits und des darauffolgenden Deployments ist, dass ihr einen Fehler auf dem Server leichter einem Commit zuordnen könnt, als wenn ihr eine Pull-Request mit 20 Dateiänderungen durchsuchen müsst. Fehler werden dadurch kleinteiliger und schneller lokalisierbar.

```
$ mob start -b TEST
  git fetch origin --prune
  git merge FETCH_HEAD --ff-only
> starting new session from origin/main
  git checkout -B mob/main-TEST origin/main
  git commit --no-verify --allow-empty -m mob start [ci-skip] [ci skip] [skip ci]
  git push --no-verify --set-upstream origin mob/main-TEST:mob/main-TEST
> you are on wip branch 'mob/main-TEST' (base branch 'main')
08ae487 2 seconds ago <joergi>
> It's now 00:03. Happy collaborating! :)
```

Listing 1: Start eines mob-Branches

```
$ touch foo.txt
$ mob next
  git add --all
  git commit --message mob next [ci-skip] [ci skip] [skip ci]

  lastFile:foo.txt --no-verify
     foo.txt | 0
     1 file changed, 0 insertions(+), 0 deletions(-)
     204b568ce2b2efd0ab98fc82345eb04328ffddb1
  git push --no-verify origin mob/main-TEST
```

Listing 2: Mit mob next wird getauscht.

```
$ mob done
  git fetch origin --prune
  git push --no-verify origin mob/main-TEST
  git checkout main
  git merge origin/main --ff-only
  git merge --squash --ff mob/main-TEST
  git branch -D mob/main-TEST
  git push --no-verify origin --delete mob/main-TEST
     foo.txt    | 0
     1 file changed, 0 insertions(+), 0 deletions(-)
👉 To finish, use

  git commit ...
```

Listing 3: Den mob-Branch mit mob done beenden.

Neue Features wollen getestet werden

Nicht nur beim Refactoring, sondern auch beim Erstellen neuer Features solltet ihr Pairing und Trunk-Based Development einsetzen. Da neue Features im Normalfall noch auf der Plattform getestet und abgenommen werden müssen und alle neuen Codeänderungen sofort live sind, klingt dies erst einmal nach einem Widerspruch.

Abhilfe schaffen euch hier sogenannte Feature-Flags wie *FF4J* [8] oder *Togglz* [9]. Das Prinzip von Feature-Flags ist ganz einfach: Ihr programmiert ein neues Feature und setzt es „hinter ein Feature-Flag". Dies bedeutet, dass das neue Feature nur zu sehen ist, wenn das entsprechende Feature-Flag aktiv ist. Ist es deaktiviert, sieht die Seite noch genau so aus, wie vor der Programmierung des Features. Ihr müsst dies ebenso in euren Tests abbilden: ein Test, wenn der Toggle an ist und einer, wenn er aus ist.

Jetzt könnt ihr auf eurem Staging-System die Feature-Flag anschalten, sodass auf dem geschützten System euer neues Feature getestet und abgenommen werden kann. Auf dem Produktionssystem ist diese ausgeschaltet und das System sieht aus wie zuvor. Ist euer neues Feature dann abgenommen, könnt ihr das Ganze ohne ein neues Deployment einfach live schalten. Nach einer kurzen Testphase könnt ihr den Code für den Feature-Flag wieder entfernen.

Manche Feature-Flags eignen sich auch dafür, dass nur ein kleiner Prozentsatz eurer Browseranfragen das neue Feature ausspielt. So könnt ihr mittels Tracking erst einmal beobachten, ob das Verhalten der User:innen und der Website den Erwartungen entspricht und nach und nach die Prozentzahl erhöhen, bis 100 % der Besucher:innen das neue Feature sehen. Ebenso lassen sich mit Feature-Flags, falls diese das unterstützen, A/B-Tests ausrollen und neue Features nur 50 % der Besucher:innen zeigen.

Was tun bei wirklich komplexem Code?

Gerade bei komplexem Code und Änderungen an der Businesslogik ist es von Vorteil, wenn ihr das Ganze in kleine Schritte unterteilt. Statt normalem Pairing zu zweit könnt ihr auch jederzeit noch mehr Kolleg:innen zum Mob Programming hinzuholen. Mehr Augen sehen mehr Fehler und gerade bei Änderungen an Businesslogik ist es oft ratsam, wenn es Knowledge-Sharing gibt, statt dass nur wenige das Wissen für sich behalten.

Wenn ihr euch unsicher seid, oder keine Kolleg:innen zum Pairen zur Verfügung stehen, bleibt euch natürlich auch weiterhin die Möglichkeit, einen Pull-Request zu öffnen. Und dies solltet ihr dann auch machen.

Generell gilt als Best Practice: Wenn der Code nicht mindestens von einer weiteren Person gelesen und auch verstanden wurde, sollten die Änderungen nicht zum Main-Branch gepusht werden.

Quellen

[1] *https://trunkbaseddevelopment.com*
[2] *https://pop.com*
[3] *https://www.jetbrains.com/code-with-me*
[4] *https://mob.sh*
[5] *https://www.remotemobprogramming.org*
[6] *https://www.jacoco.org/jacoco*
[7] *https://www.sonarsource.com/products/sonarqube*
[8] *https://ff4j.github.io*
[9] *https://www.togglz.org*

Jörg Liedl
SpringerNature AG
joerg.liedl@rocket-design.de

Jörg Liedl arbeitet als Senior Softwareentwickler bei dem Wissenschaftsverlag SpringerNature AG. Das Java-Ökosystem ist seine Heimat, in der mittlerweile aber Kotlin das gute alte Java komplett verdrängt hat. Er bevorzugt es, Sachen zu automatisieren, um den eigenen Arbeitsalltag angenehmer zu gestalten. Privat beschäftigt er sich derzeit viel mit dem Fediverse.

Java aktuell

NEWSLETTER

Anmeldung

Der Java aktuell Newsletter informiert dich alle vier Wochen über das aktuelle Geschehen in der Java-Welt.

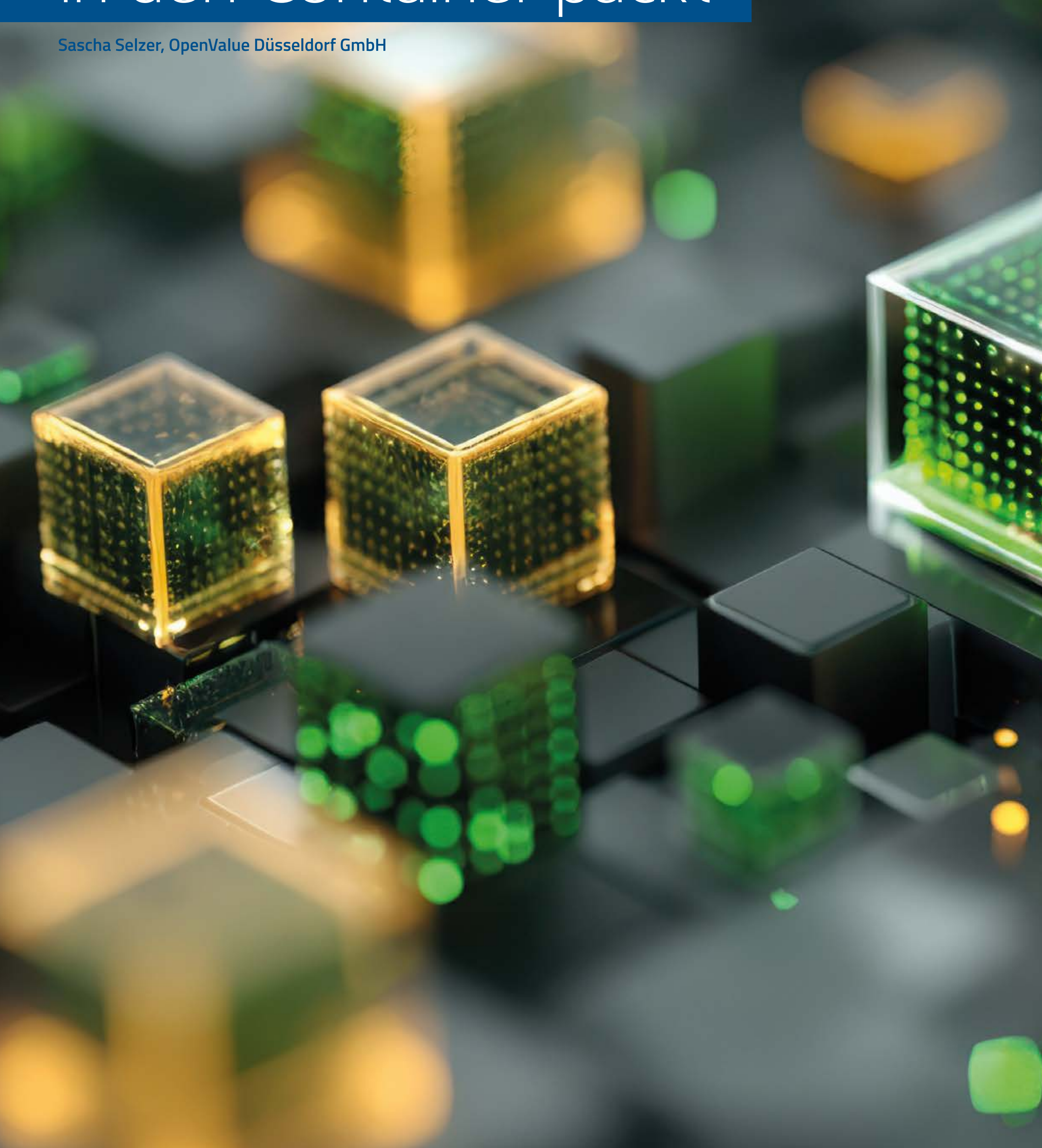
Abonniere ihn kostenfrei und bleibe immer auf dem Laufenden!

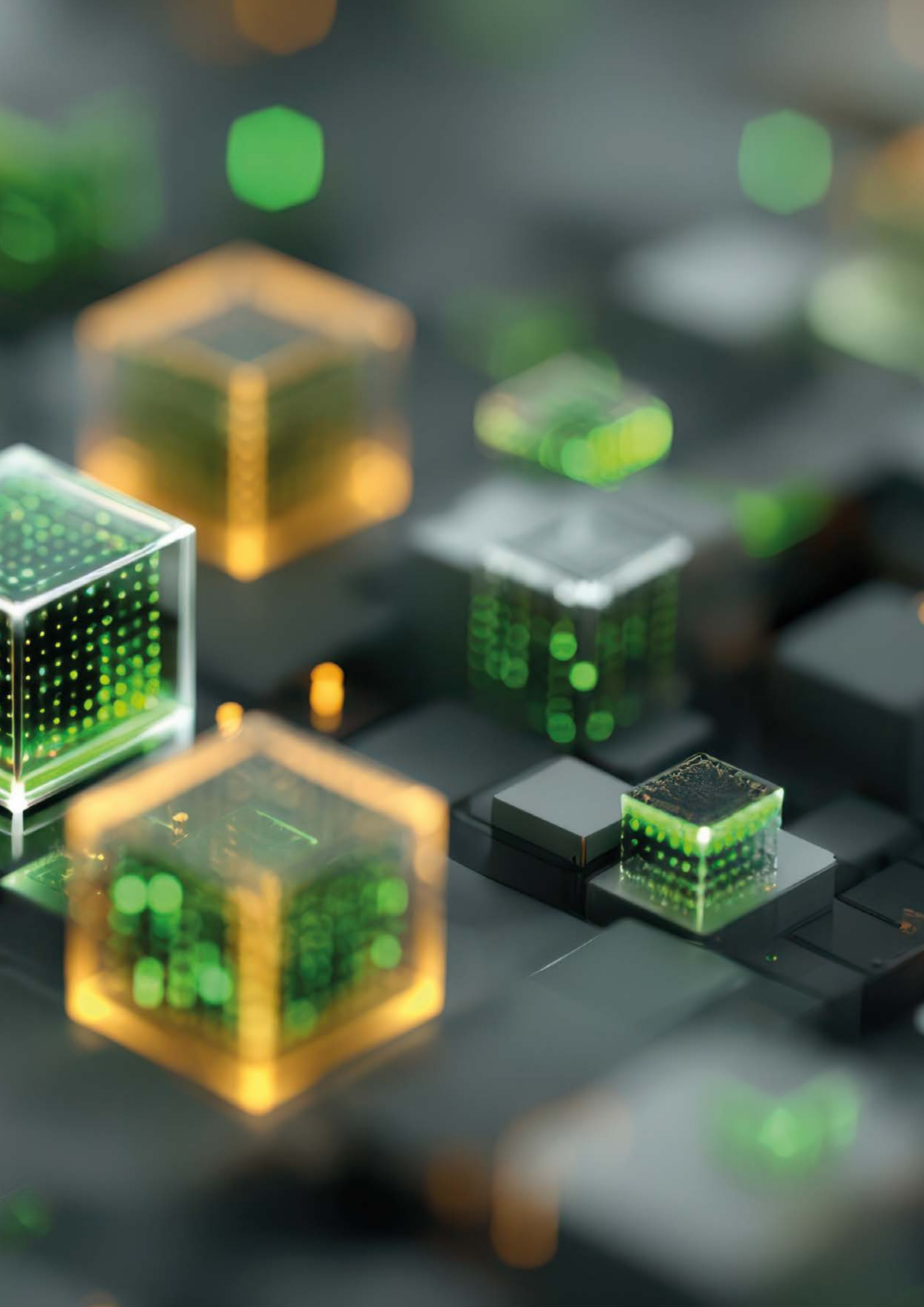
https://meine.doag.org/newsletter/groupSet.ja-nl/

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

Klein, schnell, sicher: Wie man Spring Boot richtig in den Container packt

Sascha Selzer, OpenValue Düsseldorf GmbH





Spring Boot und Container gelten als De-Facto-Standard moderner Java-Microservices – doch Standard heißt nicht automatisch schlank. Dieser Artikel zeigt, wie Sie Ihre Spring-Boot-Anwendung von Ballast befreien: mit gezieltem Dependency-Tuning, einer maßgeschneiderten JRE und einem minimalen Basis-Image.

Spring Boot und Container-Technologien wie Docker oder Podman bilden heute den De-facto-Standardstack für die Entwicklung und den Betrieb moderner Microservices in der Java-Welt. Kaum ein Projekt kommt noch ohne diese Kombination aus — sie ermöglicht schnelle Entwicklungszyklen, einfache Skalierung und eine hohe Portabilität über verschiedene Umgebungen hinweg.

Trotz dieser weiten Verbreitung wird ein entscheidender Aspekt oft übersehen: Best Practices im Containerumfeld. Allzu häufig entstehen Images, die weit mehr enthalten, als für die eigentliche Anwendung nötig ist – und damit unnötig groß, langsam und potenziell unsicherer werden. Eine typische Spring-Boot-Anwendung bringt es schnell auf 400 bis 500 Megabyte – ein Gewicht, das in Zeiten von Cloud-Native-Deployments und schnellen Rollouts teuer bezahlt wird.

Dieser Artikel zeigt praxisnah, wie sich Spring-Boot-Anwendungen gezielt verschlanken lassen – ohne auf Komfort oder Funktionalität zu verzichten. Von der Wahl des richtigen Basis-Images über Layer-Optimierung bis hin zur Erstellung einer eigenen angepassten JRE: Wir zeigen, wie Sie Ihre Container kleiner, sicherer und schneller machen.

Ein einfacher Service mit typischem Technologie-Stack

Um die im Folgenden beschriebenen Probleme und Lösungsansätze greifbar zu machen, begleitet uns eine kleine Beispielanwendung durch diesen Artikel. Die Anwendung bietet ein schlichtes REST-Interface, mit dem sich Filmtitel anlegen, lesen, aktualisieren und löschen lassen. Im Kern basiert sie auf Spring Boot Web, das eine HTTP-API bereitstellt, während PostgreSQL als relationale Datenbank dient. Der Datenzugriff erfolgt über Spring Data JPA, und das Datenbankschema wird mithilfe von Liquibase versioniert und automatisch verwaltet. Bei jeder Interaktion — etwa dem Anlegen oder Löschen eines Films — wird zusätzlich ein Event über Apache Kafka publiziert. *Listing 1* zeigt einen Auszug aus der pom.xml mit den Abhängigkeiten.

Der gesamte Quellcode ist in GitHub *[1]* zu finden. Auch wenn die Anwendung bewusst einfach gehalten ist, bildet sie doch einen typischen Software-Stack ab, wie er in vielen produktiven Microservice-Architekturen vorgefunden wird. Deshalb eignet sie sich, um typische Herausforderungen und Optimierungsmöglichkeiten beim Container-Build aufzuzeigen.

Wird die Anwendung nach gängiger Empfehlung mit einem klassischen Dockerfile gebaut – so, wie es etwa in der offiziellen Spring-Boot-Dokumentation *[2]* beschrieben ist – entsteht ein Container-Image mit einer Größe von rund 413 MB. Als Basis dient dabei das Image **eclipse-temurin:25-jre-noble**. Bereits dieser einfache Ansatz offenbart typische Schwächen: Eine Analyse mit Syft *(siehe Listing 2)* zeigt zahlreiche Pakete, die für den Betrieb der Anwendung gar nicht notwendig sind – darunter apt, bash, grep oder gzip. Insgesamt sind 108 Debian-Pakete in dem Image installiert. Diese Komponenten erhöhen nicht nur die Größe des Images, sondern stellen potenzielle Angriffsflächen und Fehlerquellen dar.

Im weiteren Verlauf dieses Artikels werden wir anhand dieser Anwendung verschiedene Strategien zur Verbesserung betrachten:

```
<dependencies>
   <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
   <dependency>
        <groupId>org.liquibase</groupId>
        <artifactId>liquibase-core</artifactId>
   </dependency>
   <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
   </dependency>
   <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
   </dependency>
   <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
   </dependency>
   <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
   </dependency>
</dependencies>
```

Listing 1: pom.xml der Beispielanwendung

```
> docker build -t app -f src/main/docker/Dockerfile .
…
> syft app
apt                          2.8.3
…
bash                         5.2.21-2ubuntu4
…
grep                         3.11-4build1
deb
gzip                         1.12-1ubuntu3.1
openjdk                      25+36-LTS
sed                          4.9-2build1
tar                          1.35+dfsg-3build1
…
```

Listing 2: Auflisten der installierten Pakete im Image (Auszug)

- das Entfernen ungenutzter Bibliotheken aus dem Laufzeit-Image,
- das Bauen einer eigenen, minimalen JRE,
- sowie das Erstellen oder Finden eines maßgeschneiderten Basis-Images, das nur die unbedingt notwendigen Pakete enthält.

Ziel ist es, den Container schlanker, sicherer und effizienter zu machen – ohne dabei auf die gewohnte Leistungsfähigkeit und Flexibilität von Spring Boot zu verzichten.

Wie finde ich ungenutzte Bibliotheken in der Laufzeit?

Einer der größten Vorteile von Spring Boot ist sein riesiges Ökosystem: Mit nur wenigen Zeilen in der **pom.xml** zieht man sich dank der zahlreichen Starter genau die Bibliotheken in der passenden Version, die man für eine bestimmte Funktionalität benötigt. Ein Beispiel ist der **spring-boot-starter-data-jpa**, der automatisch alle notwendigen JPA- und Hibernate-Bibliotheken mitbringt – inklusive Abhängigkeiten, die häufig gebraucht werden, aber nicht zwingend in jedem Projekt zum Einsatz kommen.

Dieses Konzept macht die Entwicklung einfach, führt aber auch dazu, dass häufig mehr Bibliotheken im finalen Artefakt landen, als tatsächlich benötigt werden. Der Grund: Spring Boot kann rein statisch nicht wissen, welche Teile einer Bibliothek zur Laufzeit tatsächlich verwendet werden. Das hängt vollständig vom Anwendungscode und den verwendeten Features ab. So entstehen leicht unsichtbare Überhänge – JARs, die zwar ausgeliefert, aber nie geladen werden.

Um genau das sichtbar zu machen, hilft das Projekt loosejar *[3]*. Obwohl es schon einige Jahre alt ist, erfüllt es seinen Zweck nach wie vor zuverlässig und funktioniert auch mit neueren Java Versionen. Loosejar wird als Java-Agent in die Anwendung eingebunden und zeichnet während der Laufzeit auf, welche Klassen aus welchen JARs tatsächlich geladen wurden. Beim Beenden der Anwendung erzeugt der Agent einen übersichtlichen Bericht, der zeigt, welche Bibliotheken verwendet wurden – und welche ungenutzt geblieben sind. Loosejar kann nicht Uber-Jars auswerten, wie Spring Boot Anwendungen normalerweise verpackt werden. Deswegen muss vor der Ausführung das Uber-Jar entpackt werden. Glücklicherweise bietet Spring Boot in neueren Versionen mit dem **spring-boot-jar-mode-tools** jar einen einfachen Weg an, das zu erreichen. Das Ganze kann wie in *Listing 3* dann ausgeführt werden.

Nach dem Testlauf liefert loosejar eine Auswertung, anhand derer man in der pom.xml gezielt aufräumen kann. Ein Auszug dieser Auswertung ist in *Tabelle 1* zu sehen.

JAR	Utilization	Loaded Classes	Total Classes
spring-boot-autoconfigure-3.5.7.jar	31,06%	460	1481
log4j-api-2.24.3.jar	1,41%	3	213
tomcat-embed-core-10.1.48.jar	20,83%	320	1536
…			
commons-text-1.13.0.jar	0,00%	0	165

Tabelle 1: Auszug der geladenen Klassen und zugehörigen Bibliotheken (test.csv)

Häufig lassen sich ganze Bibliotheken in der Auswertung finden, die über Exclusions dann aus den Spring-Startern entfernt werden können, ohne dass die Anwendung an Funktionalität verliert. *Listing 4* zeigt so eine Exclusion in der **pom.xml**.

```
<dependency>
   <groupId>org.liquibase</groupId>
   <artifactId>liquibase-core</artifactId>
   <exclusions>
        <exclusion>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-text</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-collections4</artifactId>
        </exclusion>
        </exclusions>
</dependency>
```

Listing 4: Exclusions von nicht genutzten Bibliotheken in der pom.xml

Mit dieser Methode konnten in der Beispielanwendung insgesamt 13 Bibliotheken identifiziert werden, die im Betrieb nie geladen wurden. Nach deren Entfernung reduziert sich das Gesamtvolumen der

```
> ./mvnw package
> java -Djarmode=tools -jar target/movie-service-0.0.1-SNAPSHOT.jar extract --destination extracted
> curl -Lo extracted/loosejar-1.2.0.jar https://github.com/kyrill007/loosejar/releases/download/v1.2.0/loosejar-1.2.0.jar
> java -jar -javaagent:./extracted/loosejar-1.2.0.jar -Dloosejar.outputFile=test.csv -Dloosejar.format=csv extract-
ed/movie-service-0.0.1-SNAPSHOT.jar
```

Listing 3: Aufruf von der Anwendung mit loosejar

Anwendung um rund 13 MB. Das mag auf den ersten Blick nicht viel erscheinen, ist aber ein wichtiger erster Schritt in Richtung schlanker, wartungsärmerer Container – und schafft die Basis für die nächsten Optimierungen.

Vom Aufräumen zum Optimieren

Das Entfernen ungenutzter Bibliotheken ist ein wichtiger erster Schritt, um das eigene Anwendungspaket zu verschlanken. Doch damit sind wir noch längst nicht am Ziel: Auch wenn der Java-Code nun nur noch die tatsächlich benötigten Abhängigkeiten enthält, bleibt die zugrunde liegende Laufzeitumgebung oft deutlich größer als nötig.

Gerade bei Spring-Boot-Anwendungen wird häufig ein komplettes JRE-Image mitgeliefert – inklusive vieler Module, die im konkreten Einsatz gar keine Rolle spielen. Hier liegt weiteres Optimierungspotenzial verborgen: eine **eigene, minimalistische JRE**, die exakt auf die Anforderungen der Anwendung zugeschnitten ist.

Im nächsten Abschnitt sehen wir uns an, wie sich mit Bordmitteln des JDKs und etwas Know-how eine schlanke Laufzeitumgebung erstellen lässt, die sowohl Platz spart als auch Sicherheitsrisiken reduziert.

Eine eigene JRE bauen

Seit Java 9 ist das JDK in Module aufgeteilt *[4]*. Diese Modularisierung macht es möglich, eine maßgeschneiderte Java Runtime Environment (JRE) zu erstellen, die nur die Module enthält, die eine Anwendung tatsächlich benötigt. Für diese Aufgabe stellt das JDK selbst zwei Werkzeuge bereit: **jdeps** zur Analyse der Modulabhängigkeiten und **jlink** zum Erzeugen einer kompakten Laufzeitumgebung.

Viele bereits verfügbare JREs wie zum Beispiel Temurin werden über diesen Weg gebaut. Das in unserem Beispiel verwendete Basis-Image eclipse-temurin:25-jre-noble enthält standardmäßig 48 JDK-Module *(siehe Listing 5)*, von denen ein Großteil für eine typische Spring-Boot-Anwendung überflüssig ist, etwa **jdk.graal.compiler** oder jdk.graal.compiler.management. Das führt dazu, dass allein das JRE in diesem Image rund 200 MB groß ist – ein guter Ansatzpunkt für Optimierungen.

```
> docker run --rm -it eclipse-temurin:25-jre-noble
java --list-modules

java.base@25
java.compiler@25
java.datatransfer@25
java.desktop@25
java.instrument@25
java.logging@25
java.management@25
java.management.rmi@25
java.naming@25
java.net.http@25
java.prefs@25
java.rmi@25
java.scripting@25
java.se@25
java.security.jgss@25
java.security.sasl@25
java.smartcardio@25
java.sql@25
java.sql.rowset@25
java.transaction.xa@25
java.xml@25
java.xml.crypto@25
jdk.accessibility@25
jdk.charsets@25
jdk.crypto.cryptoki@25
jdk.dynalink@25
jdk.graal.compiler@25
jdk.graal.compiler.management@25
jdk.httpserver@25
jdk.incubator.vector@25
jdk.internal.vm.ci@25
jdk.jdwp.agent@25
jdk.jfr@25
jdk.jsobject@25
jdk.localedata@25
jdk.management@25
jdk.management.agent@25
jdk.management.jfr@25
jdk.naming.dns@25
jdk.naming.rmi@25
jdk.net@25
jdk.nio.mapmode@25
jdk.sctp@25
jdk.security.auth@25
jdk.security.jgss@25
jdk.unsupported@25
jdk.xml.dom@25
jdk.zipfs@25
```

Listing 5: Auflistung der Module in einer üblichen JRE

Welche Module werden tatsächlich benötigt?

Um herauszufinden, welche Module die eigene Anwendung wirklich verwendet, kommt **jdeps** zum Einsatz – ein Analysewerkzeug, das die Abhängigkeiten von Klassen und Paketen bis hin zu den zugrunde liegenden Java-Modulen auflöst.
Ein typischer Aufruf ist in *Listing 6* beschrieben.

```
> jdeps --ignore-missing-deps -q \
   --recursive \
  --print-module-deps \
  --multi-release 25 \
  --class-path 'extracted/lib/*' \
extracted/application/movie-service-*.jar > deps.info
```

Listing 6: Aufruf von jdeps

Erklärung der Parameter:

- **--ignore-missing-deps:** Ignoriert Abhängigkeiten, die nicht aufgelöst werden können (z. B. optionale Klassen oder native Bibliotheken).
- **-q:** Unterdrückt unnötige Ausgaben, damit das Ergebnis übersichtlicher bleibt.
- **--recursive:** Analysiert nicht nur die angegebenen JARs, sondern auch deren transitive Abhängigkeiten.
- **--print-module-deps:** Gibt die tatsächlich benötigten Module in kommagetrennter Form aus – perfekt für die Weiterverwendung in Skripten.
- **--multi-release 25:** Berücksichtigt, dass es sich um eine Multi-Release-JAR handelt, und nutzt dabei die Klassenversionen für Java 25.
- **--class-path:** Gibt den Pfad zu allen verwendeten Bibliotheken an, damit jdeps vollständige Abhängigkeitsinformationen erhält.

Listing 7 zeigt das Ergebnis.

In unserem Beispiel werden damit 14 Module identifiziert, die die

```
java.base,java.compiler,java.desktop,java.instrument,java.management,java.net.http,java.prefs,java.rmi,java.
scripting,java.security.jgss,java.security.sasl,java.sql.rowset,jdk.jfr,jdk.unsupported
```

Listing 7: Die benötigten JDK-Module für die Anwendung

Anwendung tatsächlich nutzt – deutlich weniger als die 48 Module der Standard-JRE.

Eine minimale JRE mit jlink erzeugen

Mit diesem Wissen lässt sich nun eine maßgeschneiderte Laufzeitumgebung erzeugen. Dazu wird das Werkzeug **jlink** verwendet, das aus einem bestehenden JDK die benötigten Module extrahiert und zu einer eigenen, lauffähigen JRE zusammensetzt *(siehe Listing 8)*.

```
> jlink \
     --add-modules $(cat deps.info) \
     --strip-debug \
     --compress zip-9 \
     --no-header-files \
     --no-man-pages \
     --output /myjre
```

Listing 8: Erstellung einer eigenen JRE mit jlink

Erläuterung der Optionen:

- **--add-modules:** Gibt an, welche Module eingebunden werden sollen – hier aus einer Datei, die das Ergebnis von jdeps enthält.
- **--strip-debug:** Entfernt Debug-Informationen, um die Größe weiter zu reduzieren.
- **--compress zip-9:** Komprimiert alle Dateien mit dem höchsten ZIP-Kompressionsgrad (Stufe 9).
- **--no-header-files:** Entfernt C-Header-Dateien, die für unsere Anwendung nicht relevant sind.
- **--no-man-pages:** Spart zusätzlichen Speicherplatz, indem Dokumentationsseiten ausgelassen werden.
- **--output:** Gibt das Zielverzeichnis an, in dem die neue JRE erzeugt wird.

Das Ergebnis ist eine maßgeschneiderte Laufzeitumgebung mit genau den Modulen, die die Anwendung wirklich benötigt. In unserem Beispiel reduziert sich die Größe der JRE damit von rund 200 MB auf nur 65 MB – eine Ersparnis von etwa 140 MB.

Diese kompakte JRE kann nun in das finale Container-Image integriert werden – beispielsweise auf Basis eines schlanken Images wie ubuntu:24.04, das ohnehin die Grundlage des Temurin-Images bildet.

Im nächsten Schritt betrachten wir, wie man ein passendes Basis-Image für eine solche angepasste JRE auswählt oder selbst erstellt, um das volle Potenzial schlanker Container auszuschöpfen.

Ein schlankes Basis-Image für Java

Nachdem wir unsere JRE bereits auf das Nötigste reduziert haben, stellt sich die nächste Frage: Wie schlank kann eigentlich das Basis-Image selbst werden?

Das bisher verwendete Ubuntu-Image bringt von Haus aus etwa 100 MB auf die Waage – und enthält zahlreiche Komponenten, die für den Betrieb einer Java-Anwendung nicht erforderlich sind: einen Paketmanager, Shell-Werkzeuge, eine Bash, Systemtools und vieles mehr. Für viele Microservice-Szenarien ist das schlicht überdimensioniert.

Um herauszufinden, was Java tatsächlich zum Starten benötigt, kann man mit dem Tool **ldd** untersuchen, welche Shared Libraries eine Executable zur Laufzeit einbindet *(siehe Listing 9)*.

```
> ldd $(which java)
  linux-vdso.so.1 (0x0000ffff8ff2b000)
  libjli.so => /user/java/jdk17/bin/../lib/libjli.so
  libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6
  libdl.so.2 => /lib/aarch64-linux-gnu/libdl.so.2
  libpthread.so.0 => /lib/aarch64-linux-gnu/libpthread.so.0
  /lib/ld-linux-aarch64.so.1
```

Listing 9: Analysieren der Laufzeitabhängigkeiten der Java-Anwendung

Die Ausgabe zeigt: Die JVM benötigt im Wesentlichen nur die libc (C-Standardbibliothek) sowie einige Basiskomponenten des Betriebssystems, um lauffähig zu sein. Alles andere – Paketmanager, Shells, Debug-Tools – ist für eine produktive Laufzeitumgebung überflüssig.

Vorgefertigte Minimal-Images

Eine einfache Möglichkeit, auf ein schlankes Basis-Image zu setzen, ist die Verwendung von Chainguard-Images *[5]*. Diese Container-Images sind speziell darauf ausgelegt, klein, sicher und frei von unnötigen Abhängigkeiten zu sein.

Für unsere Zwecke bietet sich etwa das Image **cc-dynamic** an – es bringt nur die glibc mit, also genau das, was Java benötigt.

Chainguard-Images bieten darüber hinaus weitere Vorteile:

- Sie werden standardmäßig mit einem Software Bill of Materials (SBOM) ausgeliefert.
- Sie sind kryptografisch signiert und damit besser überprüfbar.
- Sie folgen einem Zero-CVE-Prinzip, bei dem bekannte Sicherheitslücken aktiv ausgeschlossen werden.

Ein möglicher Nachteil: Viele dieser Images sind nicht vollständig frei verfügbar oder unterliegen Nutzungsbeschränkungen, was in manchen Unternehmensumgebungen eine Hürde darstellen kann.

Ein eigenes Basis-Image mit Chisel bauen

Wer die volle Kontrolle behalten möchte, kann ein eigenes, minimales Basis-Image aufbauen – beispielsweise auf Ubuntu-Basis. Hier kommt das Projekt **Chisel** *[6]* ins Spiel.

Chisel trennt Debian-Pakete in logische Komponenten, sogenannte Slices. Ein typisches deb-Paket enthält neben den eigentlichen Bibliotheken auch ausführbare Dateien, Dokumentation und Lizenzinformationen. Normalerweise werden all diese Komponenten gemeinsam installiert, inklusive ihrer transitiven Abhängigkeiten – selbst dann, wenn sie gar nicht gebraucht werden.

Mit Chisel lassen sich dagegen gezielt nur einzelne Slices installieren, etwa nur die reinen Libraries eines Pakets *(siehe Abbildung 1)*. So vermeiden wir unnötige Abhängigkeiten. Für unser Beispiel genügt es etwa, nur das Slice **libc6_libs** zu installieren, da wir ausschließlich die libc-Bibliotheken benötigen.

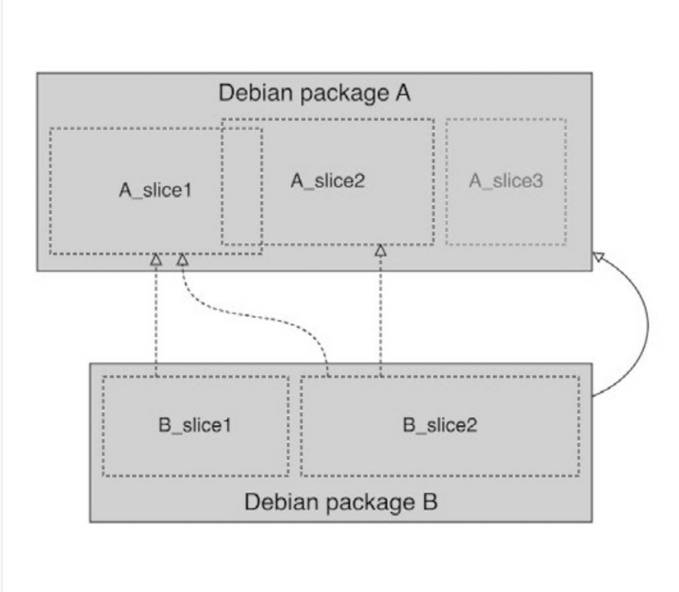


Abbildung 1: Beschreibung von Slices in Paketen (© Ubuntu/ https://documentation.ubuntu.com/chisel/en/latest/explanation/slices/)

Ein minimales Root-Filesystem lässt sich mit folgendem Befehl erstellen *(siehe Listing 10)*.

```
> mkdir /staging-rootfs \
    && chisel cut --release "ubuntu-24.04" --root /myroot \
        base-files_base \
        base-files_release-info \
        base-files_chisel \
        ca-certificates_data \
        libc6_libs \
        tzdata_zoneinfo
```

Listing 10: Erstellung eines minimalen Root Filesystems mit Chisel

Was bedeuten die Slices hier?

- **base-files_*** stellt grundlegende Systemdateien und -ordner bereit.
- **ca-certificates_data** bringt vertrauenswürdige Zertifikate mit – häufig nötig für HTTPS-Verbindungen.
- **libc6_libs** liefert die C-Laufzeitbibliotheken, die Java benötigt.
- **tzdata_zoneinfo** sorgt für aktuelle Zeitzoneninformationen.

Das Ergebnis ist ein komplettes, lauffähiges Root-Filesystem mit allem, was eine Java-Anwendung benötigt – nicht mehr und nicht weniger.

Die finale Größe beträgt gerade einmal 4,7 MB. Damit ist dieses Image mehr als 20-mal kleiner als das ursprüngliche Ubuntu-Basisimage – und bereit für die Integration unserer maßgeschneiderten JRE.

Fazit und Ausblick

Spring Boot und Container-Technologien haben sich längst als De-facto-Standard für moderne Microservice-Architekturen etabliert. Doch Standardisierung bedeutet nicht automatisch Effizienz. Wie wir gesehen haben, lohnt es sich, genauer hinzuschauen – und bewusst zu hinterfragen, was eine Anwendung tatsächlich benötigt, um zuverlässig, sicher und performant zu laufen.

Unsere Beispielanwendung hat gezeigt, wie schnell ein scheinbar harmloser Container auf mehrere hundert Megabyte anwächst – und wie viel Potenzial in gezielten Optimierungen steckt. Schon das Entfernen ungenutzter Bibliotheken kann die Grundlage für ein schlankeres und wartungsfreundlicheres System schaffen. Der größte Effekt entsteht jedoch durch die Reduktion der Laufzeitumgebung: Eine maßgeschneiderte JRE spart Speicherplatz, beschleunigt Builds und reduziert Sicherheitsrisiken.

Abgerundet wird das Gesamtbild durch ein minimales Basis-Image, das nur die notwendigen Systemkomponenten enthält. Das Ergebnis ist beeindruckend: Aus einem ursprünglich rund 400 MB großen Image wird ein voll funktionsfähiger Container von unter 140 MB – bei identischer Funktionalität. Das finale Dockerfile kann hier *[7]* eingesehen werden und kombiniert alle vorher beschriebenen Verbesserungen.

Größe ist nicht alles

Trotz aller Vorteile darf man eines nicht vergessen: Klein ist nicht immer gleich besser.

Gerade in komplexen CI/CD-Umgebungen und bei häufigen Deployments spielt auch die Cachbarkeit von Images eine große Rolle. Ein Basis-Image, das von mehreren Services geteilt wird, kann – obwohl es etwas größer ist – in Summe schnellere Build-Zeiten und geringere Netzwerkbelastung bringen.

Der Schlüssel liegt also im bewussten Abwägen:

- Schlanke, spezialisierte Images eignen sich ideal für hochoptimierte Produktionsumgebungen oder sicherheitskritische Deployments.
- Etwas größere, aber geteilte Images punkten bei Entwicklungsgeschwindigkeit, Wiederverwendung und Wartbarkeit.

Die Kunst besteht darin, beides zu kombinieren: Sicherheit und Effizienz auf der einen Seite, Wiederverwendbarkeit und Stabilität auf der anderen.

Am Ende geht es nicht nur um ein paar Megabyte weniger – sondern um Transparenz, Kontrolle und Qualität in der eigenen Build- und Laufzeitumgebung. Der hier gezeigte Weg – von der Laufzeitanalyse über den Bau einer eigenen JRE bis hin zum maßgeschneiderten

Basis-Image – kann als Basis dienen, den eigenen Stack zu analysieren und zu optimieren.

Quellen

[1] *https://github.com/tommy1199/spring-boot-container-optimization-demo*

[2] *https://docs.spring.io/spring-boot/reference/packaging/container-images/dockerfiles.html*

[3] *https://github.com/kyrill007/loosejar*

[4] *https://openjdk.org/jeps/261*

[5] *https://images.chainguard.dev*

[6] *https://github.com/canonical/chisel*

[7] *https://github.com/tommy1199/spring-boot-container-optimization-demo/blob/feature/chisel/src/main/docker/Dockerfile*

Sascha Selzer

sascha.selzer@openvalue.de

Sascha Selzer arbeitet seit Januar 2024 als Principal Engineer bei der OpenValue Düsseldorf GmbH und verfügt über umfassende Erfahrung in der Entwicklung mit JVM-basierten Sprachen sowie in der Softwarearchitektur. Weiterhin interessiert er sich für Themenbereiche wie Cloud-Monitoring und -Tracing, Microservices und DevOps und alles Rund um Container-Technologien.

MITMACHEN UND AUTORIN ODER AUTOR WERDEN!

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

Nimm Kontakt zu uns aufund sende deinen Artikelvorschlag an:
redaktion@ijug.eu
Wir freuen uns, von dir zu hören!

Weitere Informationen

Maven 4, Teil 2: Major Changes

Matthias Bünger

Im zweiten Teil der Artikelreihe (Teil 1 erschien in Java aktuell Ausgabe 04/24) über neue Features von Maven 4 werfen wir einen Blick auf die Major Changes – namentlich die Java-Version und Änderungen an der pom.xml.

Respektiere deine Nutzer – die Herausforderung von Maven 4

Bei jeder geplanten Änderung einer Software muss bedacht werden, welche Auswirkungen sie auf Nutzer älterer Versionen hat. Wie viele Nutzer sind betroffen? Wie wechselwillig sind die Nutzer? Sind die Nutzer in der Lage zu wechseln und wenn ja, wie schnell können sie migrieren? Maven ist eines der meistgenutzten Build-Tools in der Java-Softwareentwicklung und wird von einer sehr heterogenen Nutzerbasis, die sowohl Privatpersonen als auch großen Firmen umfasst, eingesetzt.

Das Maven-Team legt daher großen Wert auf Abwärtskompatibilität, um Nutzern das Umsteigen auf neue Versionen zu erleichtern. Nur selten kommt es zu „Breaking Changes", die den Nutzern zwingen, Änderungen vorzunehmen. Maven 4 ist eine der wenigen Version, bei der es zu solchen Änderungen kommt.

Java-Version

Maven ist dafür bekannt, die minimal notwendige Java-Version nur sehr langsam anzuheben. So wurde beispielsweise erst 2023 mit der Version 3.9.0 Java 8 als minimal benötige Java-Version vorausgesetzt. Zur zeitlichen Einordnung: im gleichen Jahr wurde Java 21 veröffentlicht. Im Jahr 2024 wurde entschieden, die minimal notwendige Java-Version für Maven 4 auf Java 17 anzuheben, also auf die (zu diesem Zeitpunkt) vorletzte Java-Version zu der Unternehmen LTS anbieten.

An dieser Stelle sei betont, dass Java 17 die minimal benötigte Java-Version zur Ausführung von Maven 4 ist. Das bedeutet nicht, dass alle mit Maven 4 gebauten Applikationen mindestens gegen Java 17 kompilieren müssen, denn das gewünschte Java-Release einer zu bauenden Applikation wird weiterhin im Maven Compiler Plugin *[1]* konfiguriert. Mit dem Maven Toolchain Plugin *[2]* können zudem individuelle JDKs für den Build genutzt werden.

Mit den neuen Sprachfeatures einer neuer Java-Version ergeben sich viele Möglichkeiten für die Entwickler, den Quellcode wartbarer zu gestalten. Es erlaubt zudem, Projektabhängigkeiten zu aktualisieren, die Java 8 bereits hinter sich gelassen hat.

Die pom.xml – mehr als nur eine Datei

Mehrfach im Jahr wird von Personen der Wunsch/der Vorschlag geäußert „mal eben" Änderungen an der Struktur der „pom.xml" vorzunehmen. Doch Änderungen an Mavens Kernstück vorzunehmen, ist nicht „mal eben" gemacht – zumindest nicht in Maven 3. Bevor ich darauf eingehe, dass es in Maven 4 tatsächlich Änderungen gibt (und welche es sind), ist es unerlässlich zu verstehen, wie bedeut-

sam diese XML-Datei ist.

Die pom.xml enthält sämtliche Informationen, die für den Build der Anwendung relevant sind: Aufbau des Projekts (Module), Plugins, Konfigurationswerte und benötige Abhängigkeiten („Dependencies"). Die Struktur der pom.xml *(Listing 1 zeigt ein Beispiel)*, basierend auf der Maven-Modellversion 4.0.0, hat sich seit Maven 2 aus dem Jahr 2005 nicht verändert. Einige dieser Informationen, zum Beispiel über Abhängigkeiten, werden aber nicht nur für den eigenen Build benötigt, sondern auch wenn das Artefakt, beispielsweise eine Bibliothek, in anderen Projekten verwendet wird.

Darüber hinaus gibt es bei Maven eine einzigartige Besonderheit, denn Maven ist heutzutage nicht mehr ein alleinstehendes Werkzeug: Maven ist zu einem Ökosystem geworden. Von Maven wurde das Maven Central Repository, das zentrale Artefakt-Repository für Java-Bibliotheken, erfunden. Ohne das „Maven Central Repository" und den, in der pom.xml definierten, Informationen über benötigte Abhängigkeiten, würden Entwickler vielleicht immer noch die benötigte Bibliotheken manuell von deren Webseiten herunterladen. Auch andere Werkzeuge, insbesondere Build-Tools (zum Beispiel Gradle) sowie Entwicklungsumgebungen (zum Beispiel Eclipse oder IntelliJ), nutzen das „Maven Central" und sind somit auf Mavens pom.xml angewiesen.

Diese immense Bedeutsamkeit von Maven und der pom.xml für das gesamte Ökosystem muss jederzeit vom Maven-Team berücksichtigt werden. Deshalb sollte nicht beliebig die Struktur der pom.xml (manchmal auch „POM-Schema" genannt) geändert werden, da dies weitreichende Auswirkungen hat: Alte, bereits im „Maven Central" veröffentliche Artefakte können nicht nachträglich das neue Format unterstützen, Werkzeuge müssen mit verschiedenen Modellen umgehen können, Nutzer müssen ihre Softwareprojekte anpassen und so weiter.

Das Maven-Team steckte also in einem Dilemma. Ohne eine Lösung für diese Situation, wären Verbesserungen, von denen das Team ausgeht, dass sie eine Änderung des POM-Schema erfordern, unmöglich und Maven könnte sich kaum weiterentwickeln. Hervé Boutemy, heutiger PMC-Chairman von Maven, fasste die Situation 2021 treffend zusammen *[3]*:

„With the Maven build schema preserved in amber, we can't evolve

```
<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>my.example</groupId>
  <artifactId>POMExample</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>
  <modules>
     <module>api</module>
     <module>service</module>
  </modules>
  <properties>
     <maven.compiler.release>21</maven.compiler.release>
  </properties>
  <dependencies>
     <dependency>
        <groupId>first.dependency</groupId>
        <artifactId>dep-name</artifactId>
        <version>1.2.3</version>
     </dependency>
  </dependencies>
  <build>
     <plugins>
        <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-enforcer-plugin</artifactId>
           <version>3.6.2</version>
           <executions>
              <execution>
                 <id>enforce-versions</id>
                 <goals>
                    <goal>enforce</goal>
                 </goals>
                 <configuration>
                    <rules>
                       <requireMavenVersion>
                          <version>3.9.11</version>
                       </requireMavenVersion>
                    </rules>
                 </configuration>
              </execution>
           </executions>
        </plugin>
     </plugins>
  </build>
</project>
```

Listing 1: Beispiel einer pom.xml-Datei in Model Version 4.0.0

much: we'll stay forever with Maven 3 minor releases, unable to implement improvements that we imagine will require seriously updating the POM schema[...]"

Die Lösung: Einführung einer Consumer-POM

Die Lösung für die festgefahrene Situation liegt in der Einführung der so genannten „Consumer-POM". Diese Variante der pom.xml ist – der Name lässt es erkennen – darauf ausgelegt, lesend von (ein Artefakt) „konsumierenden" Projekten und Werkzeugen genutzt zu werden. Die für den Build eines Projektes ausgelegte pom.xml wird dementsprechend „Build-POM" genannt.

Die Build-POM wird weiterhin im Source-Code-Repository der Anwendung gespeichert. Sie enthält sämtliche Informationen, die für den Build benötigt werden, zum Beispiel Plugins. Während des Builds wird, basierend auf der Build-POM, das effektive Projektmodell („enriched POM" oder auch „effective POM" genannt) im Arbeitsspeicher erzeugt, wobei unter anderem sämtliche transitiven Abhängigkeiten ermittelt und Variablen aufgelöst werden. Dieser Schritt ist nicht neu und wird auch in Maven 3 durchgeführt. In Maven 4 wird nun zusätzlich am Ende des Builds die Consumer-POM erzeugt und im Dateisystem abgelegt. Dabei wird sichergestellt, dass die Consumer-POM immer in der Modellversion 4.0.0 erstellt wird – der Version, die vom gesamten Ökosystem genutzt wird (vgl. vorherige Kapitel). *Abbildung 1* veranschaulicht den beschriebenen Prozess. In der Folge erlaubt diese Vorgehensweise nun das Format der Build-POM zu ändern. Dies passiert mit der Einführung der Modellversion 4.1.0 in Maven 4.

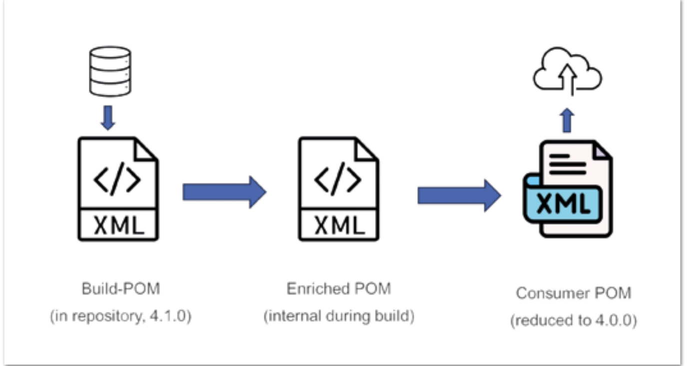


Abbildung 1: Vereinfachte Darstellung der Consumer-POM Generierung in Maven 4

Neue Modellversion 4.1.0

Die neue Modellversion 4.1.0 ist nur für die Build-POM verfügbar. Um den Wechsel auf Maven 4 zu erleichtern,s wird die Nutzung der Modellversion 4.0.0 weiterhin unterstützt. Die Version 4.1.0 ist optional und wird nur für die Nutzung einiger weniger, neuer Features benötigt. In diesem Artikel wird für die Build-POM immer Modellversion 4.1.0 verwendet. Bevor wir einen Blick auf neue Features werfen, vergleichen wir in *Tabelle 1* die Inhalte der Build- und Consumer-POM.

Es ist wenig überraschend, dass die Consumer-POM keine Build-Informationen enthält, da diese von nutzenden Projekten nicht benötigt werden. Sämtliche anderen Informationen hingegen sind enthalten. Für Artefakte des Typs „pom" gibt es keine Consumer-POM,

	Build-POM	Consumer-POM
Modell (POM) Version	4.1.0	4.0.0
Abhängigkeiten ▪ Vollständige Parent-/Subprojekt-definition ▪ Fremdbibliotheken („3rd party")	 ✗ ✓	 ✗ ✓
Properties	✓	✗
Konfiguration von Plugins	✓	✗
Profile	✓	✓
Informationen zum ▪ Projekt (Organisation, Lizenzen, etc.) ▪ SCM, Issue-Management, etc.	 ✓ ✓	 ✓ ✓
Veröffentlichung im Remote Repository (zum Beispiel „Maven Central")	✓	✓

Tabelle 1: Inhalte der Build- und Consumer-POM

da dieser Typ explizit und nur auf Builds ausgelegt ist. Ins Auge fällt, dass selbst in der Build-POM nicht mehr vollständige Definitionen des Parent- oder Subprojekts enthalten sein müssen. Beide POM-Dateien werden in einem Remote-Repository veröffentlicht. Nur so können Nutzer prüfen, ob das angebotene Artefakt auch tatsächlich und unverfälscht mit dieser Build-Defintion erzeugt wurde (Reproducible Builds).

Änderungen mit der Modellversion 4.1.0

Im ersten Teil der Artikelserie wurde das verbesserte Versionshandling von Maven 4 beschrieben. Dank diesen internen Verbesserungen und Bugfixes erkennt Maven nun auch die Version von projekteigenen Eltern- beziehungsweise Unterprojekten. Letztere werden in der Modelversion 4.0.0 bekanntlich „Module" genannt. Da der Begriff im Java-Umfeld, nicht erst seit der Einführung des Java Platform Module Systems (JPMS) in Java 9 immer wieder zu Verwirrung führte, werden Subprojekte ab Maven 4 als solche ausgezeichnet. Dies spiegelt sich im neuen Element „subproject" (und dem umschließenden Element „subprojects") wider, welches neu mit Modellversion 4.1.0 eingeführt wird. Die bisher bekannten Elemente „modules" und „module" können weiterhin genutzt werden, sind aber als deprecated markiert.

Zur Veranschaulichung dieser beiden Änderungen sind sie in *Listing 2* (für Modelversion 4.0.0) und *Listing 3* (Modelversion 4.1.0) gegenübergestellt. Im Beispiel ist ein Projekt mit zwei Subprojekten A und B gegeben, wobei das Subprojekt B das projekteeigene Subprojekt A als Abhängigkeit einbindet. Der obere Teil jedes Codebeispiels ist jeweils das Elternprojekt, während der untere Teil die pom.xml des Subprojektes B zeigt.

Wer genau hinsieht, dem fällt im Elternprojekt von *Listing 3* zudem auf, dass dort im Wurzelelement das neue Attribut „root" angegeben ist. Das Attribut ist vom Typ „boolean" und definiert mit dem Wert „true", dass ein Maven-Projekt das oberste Elternprojekt einer Projekthierarchie ist. Somit wird das Verzeichnis, in dem diese pom.xml liegt, automatisch als Wurzelverzeichnis der Projekthierarchie deklariert, wodurch Pfadangaben eindeutig aufgelöst werden können. Falls man bei der Migration auf Maven 4 noch nicht auf die neue XML-Struktur wechseln möchte, erzielt man den gleichen

```
<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>
     <groupId>test.bukama.maven</groupId>
     <artifactId>Maven4Demo</artifactId>
     <version>0.0.1-SNAPSHOT</version>
     <packaging>pom</packaging>
     <name>Maven 4 Demo</name>

     <modules>
           <module>ModuleA</module>
           <module>ModuleB</module>
     </modules>
</project>

<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>test.bukama.maven</groupId>
        <artifactId>Maven4Demo</artifactId>
        <version>0.0.1-SNAPSHOT</version>
     </parent>
     <artifactId>ModuleB</artifactId>

     <dependencies>
           <dependency>
              <groupId>test.bukama.maven</groupId>
              <artifactId>ModuleA</artifactId>
              <version>${project.version}</version>
        </dependency>
     </dependencies>
</project>
```

Listing 2: Beispiel für Modelversion 4.0.0

```
<project xmlns="http://maven.apache.org/POM/4.1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.1.0 http://maven.apache.org/xsd/maven-4.1.0.xsd"
root="true">
     <modelVersion>4.1.0</modelVersion>
     <groupId>test.bukama.maven</groupId>
     <artifactId>Maven4Demo</artifactId>
     <version>0.0.1-SNAPSHOT</version>
     <packaging>pom</packaging>
     <name>Maven 4 Demo</name>

     <subprojects>
           <subproject>ModuleA</subproject>
           <subproject>ModuleB</subproject>
     </subprojects>
</project>

<project xmlns="http://maven.apache.org/POM/4.1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.1.0 http://maven.apache.org/xsd/maven-4.1.0.xsd">
     <modelVersion>4.1.0</modelVersion>
     <parent>
        <groupId>test.bukama.maven</groupId>
        <artifactId>Maven4Demo</artifactId>
     </parent>
     <artifactId>ModuleB</artifactId>

     <dependencies>
           <dependency>
            <groupId>test.bukama.maven</groupId>
              <artifactId>ModuleA</artifactId>
        </dependency>
     </dependencies>
</project>
```

Listing 3: Beispiel für Modelversion 4.1.0

```
[WARNING] Unable to find the root directory. Cre-
ate a .mvn directory in the root directory or add the
root="true" attribute on the root project's model to
identify it.
```

Listing 4: Beispiel einer Warnung

```
mvnup apply --model-version 4.1.0 -all
```

Listing 5: Automatisierte Migration eines Projektes auf Modelverison 4.1.0

Effekt auch durch das Erstellen eines „.mvn"-Ordners im Projektverzeichnis. Das „.mvn"-Verzeichnis kann in Maven 4 weiterhin zur Definition von Konfigurationen genutzt werden. Seine Existenz hat in Maven 4 jedoch die beschriebene, zusätzliche Bedeutung. Die Definition des obersten Elternprojektes sollte immer erfolgen. Aus diesem Grund wird eine Warnung *(siehe Listing 4)* angezeigt, wenn ein solches nicht definiert ist.

Migration auf Maven 4

Es wurden zwar noch nicht alle Neuerungen von Maven 4 vorgestellt, aber da wir in diesem Artikel die neue Modelversion betrachtet haben, möchte ich zum Abschluss dieses Teils darauf eingehen, wie man sich auf den Wechsel zu Maven 4 vorbereiten und ihn dann tatsächlich auch durchführen sollte.

Für einen vermutlich problemlosen Umstieg auf Maven 4 sollte zunächst sichergestellt sein, dass die eigenen Projekte fehlerfrei auf der letzten Maven 3.x.x Version (3.9.11 – November 2025, Versionsübersicht siehe *[4]*) unter Nutzung aktueller Pluginversionen *[5]* gebaut werden. Achtet hierbei unbedingt auf Warnungen im Buildlog! Im Hinblick auf Maven 4, gibt Maven 3.9.x bereits zu vielen Konfigurationsfehlern Warnungen aus, die Maven 3.9.x zwar noch toleriert, aber in Maven 4 zum Fehlschlag führen.

Anschließend sollte die Build-Umgebung ertüchtigt werden, (mindestens) Java 17 zu unterstützen, da Maven 4 Java 17 voraussetzt. Diese beiden Schritte können bereits vor der Veröffentlichung von Maven 4 durchgeführt werden. Ziel sollte sein, dass eigene Builds bereits auf Java 17 und Maven 3.9.x erfolgreich durchgeführt werden können. Ist dies der Fall, dann besteht der Wechsel auf Maven 4 tatsächlich nur aus der Installation und Nutzung von Maven 4. Im Normalfall sind keine weiteren Änderungen notwendig, da Maven 4 kompatibel mit Maven 3 und den Maven 3 kompatiblen Plugins ist. Letztes kann nur für die offiziellen Maven-Plugins gewährleistet werden. Über Plugins der Community kann keine Aussage getroffen werden. Eine Migration der pom-Dateien auf Modelversion 4.1.0 ist nur notwendig, wenn man die neuen Features nutzen möchte, die diese voraussetzen (zum Beispiel Verzicht auf Versionsangaben bei projekteigenen Abhängigkeiten).

Zur Unterstützung bei der Migration beinhaltet Maven 4 das „Maven Upgrade"-Tool *[6]*. Mit diesem können das eigene Projekt, Plugins und deren Konfiguration geprüft sowie aktualisiert werden. Dabei unterstützt es zwei Varianten: In der ersten wird sichergestellt, dass das Projekt weiterhin mit Maven 3/Modelversion 4.0.0 genutzt werden kann. Die zweite Variante hebt die genutzte Modelversion auf 4.1.0, wodurch es zwingend mit Maven 4 gebaut werden muss.

Für eine automatisierte Migration eines Projektes auf Modelversion 4.1.0 reicht ein Kommandozeilenbefehl *(siehe Listing 5)*.

Für weitere Funktionen des Werkzeugs sei auf die Dokumentation *[6]* verwiesen.

Fazit

In diesem Artikel wurden mit der Einführung einer Consumer-POM, der Modelversion 4.1.0 und der angehobenen Mindestanforderung bezüglich der Java-Version die drei großen Änderungen von Maven 4 beschrieben. Darüber hinaus wurde gezeigt, wie der Umstieg auf Maven 4 vorbereitet und durchgeführt werden kann. Im nächsten Teil der Serie werden wir einen Blick auf weitere Features von Maven 4 werfen und sehen, wie diese die Nutzung von Maven weiter verbessern.

Quellen

[1] *https://maven.apache.org/plugins/maven-compiler-plugin/*

[2] *https://maven.apache.org/plugins/maven-toolchains-plugin/*

[3] Hervé Boutemy, Javaadvent 2021, *https://www.javaadvent.com/2021/12/from-maven-3-to-maven-5.html, zuletzt aufgerufen 01.11.2025*

[4] *https://maven.apache.org/docs/history.html*

[5] *https://maven.apache.org/plugins/index.html*

[6] *https://maven.apache.org/tools/mvnup.html*

Matthias Bünger

mbuenger@apache.org

Matthias Bünger ist Softwarearchitekt und Teamleiter mit einer Vorliebe für Java und Testen. In seiner Freizeit unterstützt er Open-Source und ist Teammitglied von Apache Maven.

Diagnostik von Memory Leaks mit OpenTelemetry

Markus Meyer

Der Memory Leak dürfte wohl allen Vertretern der Software-Entwicklung, die auf stabile Systeme setzen, ein Dorn im Auge sein. Im Gegensatz zu den offensichtlichen Crashes, korrumpieren Memory Leaks den Systemspeicher schleichend; ihre Symptomatik beginnt mit nervigen Leistungseinbußen und erhöhter Latenzzeit, bevor sie schließlich den OOM (Out-of-Memory) triggern und ganze Geschäftsabläufe in den Untergang führen. Damit ist die Diagnose von Memory Leaks gerade in Microservice-Architekturen, in denen eine einzige Anfrage Dutzende von Services durchlaufen kann, nicht nur richtig und wichtig; sie ist auch für erfahrene Entwickler die reinste Detektivarbeit.

Hier kommt OpenTelemetry *[1]* ins Spiel. Als herstellerunabhängiges Set von Tools, APIs und SDKs hat sich OpenTelemetry (OTel) schnell zum De-Facto-Standard für die Erfassung von Telemetriedaten – sprich: Traces, Metriken, Protokolle – etabliert. Die Daten, die OTel liefert, durchleuchten das Dunkel verteilter Systeme und helfen uns bei der Lokalisierung von Memory Leaks.

Weil wir uns genau das anschauen wollen, wird sich dieser Artikel vor einem gewissen investigativen Charakter nicht retten können. Wir untersuchen die Anatomie von Memory Leaks und schauen uns dann einen exemplarischen Produktionsvorfall an, der so oder so ähnlich vorgefallen ist. Dabei werden wir lernen, wie wir die Funktionen von OpenTelemetry in Bezug auf Traces, Metriken und Protokolle nutzen können, um Memory Leaks systematisch zu schließen.

Memory Leaks verstehen

Im Wesentlichen tritt ein Memory Leak auf, wenn Programmspeicher, der nicht mehr benötigt wird, besetzt bleibt. Die allmähliche Anhäufung von ungenutztem Speicher wird das Betriebssystem in seinen Handlungsspielräumen limitieren, blockieren und in manchen Fällen sogar aussperren.

Die üblichen Verursacher für Memory Leaks sind:

- **Nicht freigegebene Ressourcen:** Wenn Datenbankverbindungen, File-Handles, Netzwerk-Sockets oder Stream-Reader nach ihrer Verwendung nicht geschlossen werden.
- **Unbegrenzte Caches:** Caching-Mechanismen, die ad Infinitum wachsen und neue Einträge hinzufügen, ohne alte zu entfernen.
- **Zirkuläre Referenzen:** In Programmiersprachen mit Garbage Collection, wie zum Beispiel Java, können Objekte in selbstreferenziellen Bezügen feststecken. Der Reference-Counter bleibt somit größer als 0, was bedeutet: Der Garbage Collector skipt die Objekte.
- **Vergessene Event-Listener:** Event-Listener werden vergessen und halten Subscriptions am Leben, obwohl sie befreit werden können.

Abbildung 1: Visualisierungen einer OpenTelemetry-Benutzeroberfläche

- **Unsachgemäße Fehlerbehandlung:** Fehler, die dazu führen, dass eine Funktion vorzeitig beendet wird, ohne zugewiesene Ressourcen freizugeben – häufig am fehlenden „finally" zu erkennen.

Die Tücke von Memory Leaks liegt in ihrer typisch vagen und schwer nachvollziehbaren Erscheinungsform. Sie bemerken vielleicht eine leichte Latenz oder einen erhöhten Ressourcenverbrauch, mehr aber auch nicht. Nach und nach schwindet der freie Speicher. Der Garbage Collector (GC) zeigt erhöhte Aktivität. Dann „Stop-the-World"-Pausen, noch mehr Latenzen, schließlich bremst der Dienst komplett aus, wirft OOM-Fehler und crasht.

Um Memory Leaks zu schließen, reicht Vanilla-Debugging oft nicht aus. Dashboards, die nur „von-oben"-Metriken wie CPU-Auslastung oder PIDs beobachten, reichen ebenfalls nicht aus. Was wir brauchen, sind granulare, kontextbezogene Daten, die mit bestimmten Operationen oder Codesegmenten verknüpft sind und uns Strategien des fachkompetenten Debuggings aufzeigen. Wir brauchen Daten, die den Ressourcenverbrauch mit bestimmten Aktivitäten innerhalb unserer Anwendung in Korrelation setzen und nicht nur sagen, was passiert (der Speicher wächst), sondern auch, warum (welcher Teil der Anwendung, welche Anfrage oder welcher Codepfad den Speicher belegt).

Unser mehr oder weniger hypothetisches Memory Leak-Szenario

Um die Leistungsfähigkeit von OpenTelemetry zu veranschaulichen, stellen wir uns ein alltägliches, aber frustrierendes Szenario vor.

Der Vorfall: Im hypothetischen Software-Unternehmen Excellence Ltd., einem schnell wachsenden Technologieunternehmen, ist der `UserAuthService`, der für die Authentifizierung von Benutzern und die Verwaltung von Sessions zuständig ist, in den letzten Tagen recht langsam geworden. Nutzer klagen über Anmeldefehler und Time-Outs. Das Ops-Team einigte sich darauf, dass der typische Rush-Hour Traffic das Problem sei. Dann eskalierte die Situation: Die `UserAuthService`-Instanz stürzte wiederholt ab und zeigte in den Kubernetes-Protokollen OOMKilled-Fehler an. Es wurde beobachtet, dass der RSS-Speicher (Resident Set Size) stetig anstieg, und zwar weit über den erwarteten Ausgangswert hinaus, selbst in Zeiten mit geringem Datenverkehr. Der Dienst musste mehrfach neu gestartet werden.

Wir forschten im Dashboard für den `UserAuthService`. Die „hohe Speichernutzung" konnten wir sehen. Die Neustarts konnten wir auch sehen. Doch die Metriken waren zu oberflächlich, um die Ursache nach dem Anstieg des Speicherverbrauches zu finden.

Also brachten wir OpenTelemetry ins Spiel. *Abbildung 1* zeigt die visualisierte OpenTelementry-Benutzeroberfläche, die die Fehlersuche erleichtert und tiefere Einblicke in die Anwendung ermöglicht.

Die Diagnose

Mit der Implementierung von OpenTelemetry erhielten wir Zugriff auf drei neue Tools: Traces, Metriken und Protokolle. Gehen wir sie im Einzelnen durch.

Traces verfolgen den Weg einer einzelnen Anfrage, während die Anfrage sich durch Dienste und Komponenten bewegt. Jedes Segment dieser Reise wird durch einen „Span" dargestellt, der Details wie den Namen der Operation, die Dauer nebst weiteren Attribute enthält.

Für die Diagnose von Memory Leaks sind Traces recht praktisch:

- Mit Traces können wir sehen, wann Boiler-Plate ausufert: Datenbankabfragen oder API-Calls, die zu lange dauern und auf Ressourcenkonkurrenz hinweisen.
- Wir können Traces mit Custom Attributes erweitern: Zum Beispiel `db.rows_affected`, `cache.items_added` oder `request.body.size`
- Wir können Traces kombiniert plotten und mithilfe unserer fachkompetenten Retina die Situationen im Gesamten betrachten *(Abbildung 2 zeigt vier kombinierte Traces)*.

Metriken liefern numerische Datenpunkte im Zeitverlauf, die sich zur Quantifizierung des System- und Anwendungszustandes eignen. Es handelt sich um hochgranulare und benutzerdefinierte Metriken auf Anwendungsebene, die es uns leichter machen, Memory Leaks zu erkennen.

Zu den wichtigsten OTel-Metriken für die Speicherdiagnose gehören:

- **Metriken auf Systemebene**, die uns eine Makrosicht über den Speicherbedarf der Anwendung liefern, wie `system.memory.usage` (`total`, `heap`, `non-heap`), `system.cpu.utilization`, oder `process.runtime.jvm.memory.heap_usage`.
- **cache_size_bytes** oder **cache_item_count**: Wenn ein Cache undicht ist, zeigen diese Metriken ein unbegrenztes Wachstum an.
- **active_connections_count**: Für Dienste, die externe Verbindungen verwalten.
- **concurrent_sessions_count**: Für unser UserAuthService-Szenario kann diese Metrik die Anzahl der aktiven User-Sessions verfolgen.

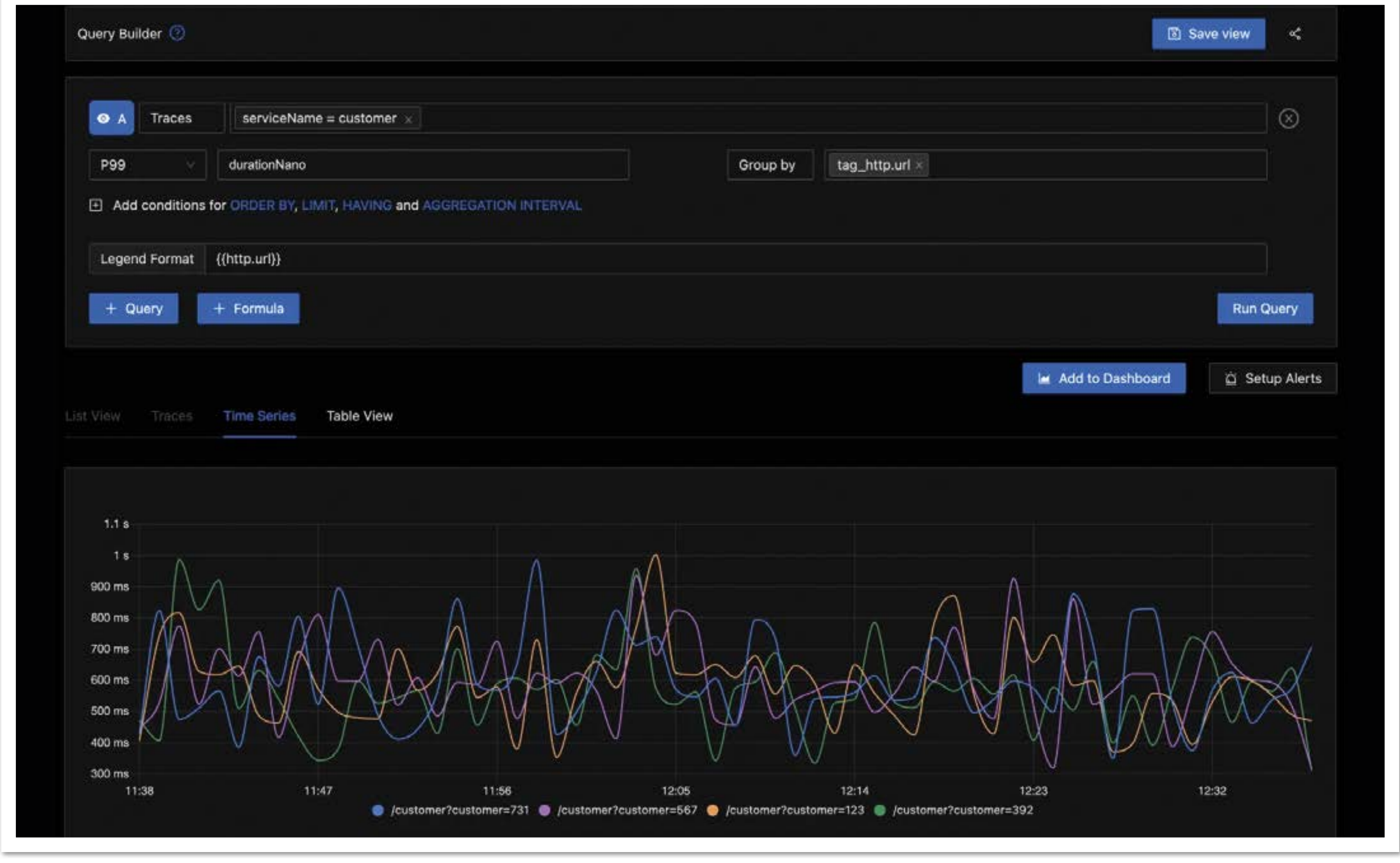


Abbildung 2: Anstatt Logfiles nachträglich zu plotten, sollten kritische Metriken proaktiv getraced werden. Mit kombinierten Plots – wie hier dargestellt - werden Korrelationen zwischen den Metriken sofort sichtbar.

- **allocated_object_count**: Zur Verfolgung bestimmter Objektzuweisungen, die zu einem Leak beitragen könnten.
- **thread_count**: Ein ständig steigender Zähler könnte auf undichte Routinen oder Threads hinweisen, die den Speicher blockieren.

Diese Metriken liefern spezifische, quantitative Hinweise darauf, wo in unserer Anwendungslogik sich Speicher ansammelt. Ein stetiger, unerklärlicher Anstieg einer dieser benutzerdefinierten Metriken ist ein deutlicher Hinweis auf ein Memory Leak.

Protokolle: OTel's Zwischenbilanzen

Während Traces die Pfade zeigen und Metriken die Zahlen, liefern Protokolle den Kontext. Ähnlich wie Logfiles beschreiben Protokolle detailliert, was zu bestimmten Zeitpunkten passiert. OpenTelemetry nutzt ausdrucksstarkes Logging, weil nämlich auch Trace-IDs und Span-IDs geloggt werden.

Wir können also:

- Logs mit Traces korrelieren
- Protokolle nach einer bestimmten `trace_id` oder `span_id` filtern
- Nach Operationen suchen, die Ressourcen geholt oder freigegeben haben (zum Beispiel „DB-Verbindung geöffnet", „Datei X geschlossen", „Session erstellt")
- Fehlerbedingungen aufzeigen, die eine korrekte Freigabe der Ressource verhindert, zum Beispiel wenn Sie nur „created"-Ereignisse ohne entsprechende „closed"- oder „destroyed"-Ereignisse für eine bestimmte Ressource sehen.

Implementierung des OTel-Instrumentariums

Um OpenTelemetry zu nutzen, müssen wir unseren Code instrumentieren. Wir unterscheiden zwischen automatischer Instrumentierung und manueller Instrumentierung.

Automatische Instrumentierung: Viele OTel-SDKs bieten Auto-Instrumentierungs-Agenten (zum Beispiel Java-Agent, Python opentelemetry-instrumentation packages), die gängige Bibliotheken und Frameworks (HTTP-Clients/Server, Datenbanktreiber) ohne Codeänderungen automatisch instrumentieren können. Falls es schnell gehen muss, oder nur die rudimentären Features von OTel benötigt werden, ist automatische Instrumentierung der bevorzugte Weg.

Manuelle Instrumentierung: Für die Diagnose von Memory Leaks ist oft eine manuelle Instrumentierung erforderlich. Wir fügen OTel-API-Aufrufe direkt in unseren Anwendungscode ein, um Spans zu customizen, Attribute zu loggen und Metriken auszugeben. Die manuelle Instrumentierung ermöglicht es uns, Codepfade, die sich der Memory Leaks verdächtig machen, genau zu instrumentieren. Zum Beispiel ist das Einbinden von Funktionen, die Ressourcen zuweisen, mit `span.SetAttribute("resource.allocated", true)` und `span.SetAttribute("resource.id", resourceID)` ein einfacher, aber wirkungsvoller Schritt.

Aus Erfahrung wird empfohlen, die manuelle Instrumentierung bei Verbindungspools zu beginnen: Hier Instrumentieren wir das Öffnen und Schließen von Datenbankverbindungen, Message-Queue-Verbindungen oder HTTP-Client-Verbindungen. Außerdem:

- **Caches:** Damit verfolgen wir die Größe, die Anzahl der Elemente und die Auslagerungsraten von In-Memory-Caches.
- **File-I/O:** Damit überwachen wir das Öffnen und Schließen von File-Handles.
- **Session-Management:** Entscheidend für unser `UserAuthService`-Beispiel, um das Anlegen und Löschen von Sessions zu verfolgen.
- **Datenstrukturen:** Wenn benutzerdefinierte, potenziell große Datenstrukturen (zum Beispiel Maps, Listen) im Einsatz sind, können Metriken hinzugefügt werden, um deren Größe im Laufe der Zeit zu verfolgen.
- **Garbage Collection Metriken:** OTel bietet eine Standardinstrumentierung für GC-Aktivitäten, einschließlich Pausenzeiten, Heap-Größe und Anzahl der GC-Zyklen (eine hohe GC-Aktivität, die OOM-Fehlern vorausgeht, ist ein starkes Signal).

Zurück zur Fallstudie

Erinnern wir uns daran, dass Memory Leaks unseren `UserAuthService` mehrmals zum Neustart zwangen.

Nach dem Absturz des `UserAuthService` platzierten wir eine auf OTel basierende Observability-Plattform. Muster fielen auf: Traces, die den `UserAuthService` involvierten, waren deutlich länger als üblich, selbst bei einfachen Authentifizierungsanfragen. Bei einer genaueren Untersuchung der `UserAuth`-Traces konnte festgestellt werden, dass die Operationen innerhalb des `UserAuthService` selbst, insbesondere die Operationen im Zusammenhang mit `userSessionLogin` und `sessionVerification`, ungewöhnlich lange dauerten. Noch aufschlussreicher war das Attribut `session.map.size`, das mit den Zeitspannen innerhalb dieser Operationen verknüpft war und das bei aufeinanderfolgenden Anfragen, die von derselben Serviceinstanz ausgingen, stetig zunahm. Der `UserAuthService` konnte damit als direkte Quelle identifiziert werden.

Leakage lokalisieren mit Metriken

Im nächsten Schritt korrelierten wir die Erkenntnisse aus den Traces mit OpenTelemetry-Metriken des `UserAuthService`. Während die Gesamtmetrik `process.runtime.jvm.memory.heap_usage` vor dem OOM einen klaren Aufwärtstrend zeigte, lieferte unsere benutzerdefinierte Metrik `userauth_service.active_sessions_map_size` den entscheidenden Hinweis. Die Metrik wurde entwickelt, um die Anzahl der Einträge in der In-Memory-Map zu erfassen, die zum Speichern aktiver User-Sessions verwendet wird.

Das Diagramm für `userauth_service.active_sessions_map_size` zeigte einen unaufhaltsamen, unbegrenzten Anstieg ab dem Zeitpunkt, an dem der Dienst gestartet wurde, und blieb auch in Zeiten geringer Anmeldeaktivitäten oder Abmeldungen von Benutzern konstant hoch. Der Zusammenhang zwischen Speicherwachstum und Trace-Anomalien wurde damit offensichtlich: Der Dienst fügte Sessions hinzu, entfernte sie aber nicht.

Diving Deep mit Protokollen: Aufdecken der Grundursache

Nachdem das Session-Management und die wachsende `active_sessions_map_size` als Ursache des Problems erkannt wurden, untersuchten wir die Logs. Mit einer Filterung der `trace_id` für einige der problematischen Authentifizierungsanfragen (problematisch, weil abnormal langsam), fielen lückenhafte Log-Entries ins Auge. *Listing 1* zeigt einen solchen lückenhaften Log-Entry.
Aus dem Log-Entry im *Listing 1* können wir sehen, dass Sessions hinzugefügt wurden, doch wir sehen nicht, dass Sessions entfernt wurden. Weitere Untersuchungen der `SessionManager`-Codes brachten das Problem ans Licht. Die Funktion `SessionManager.addSession` fügte Sessions zwar korrekt zu einer `ConcurrentHashMap` hinzu, allerdings wurde die Funktion `SessionManager.removeSession`, die beim Abmelden des Benutzers oder beim Ablauf der Session aufgerufen werden sollte, unter einer bestimmten Fehlerbedingung während der Token-Validierung übersprungen. Wenn das Aktualisierungs-Token eines Benutzers ungültig war, beendete die Anwendung die Anfrage, entfernte aber nicht explizit die veraltete Session aus der speicherinternen Zuordnung. Das Session-Objekt war zwar für den Benutzer nicht mehr gültig, wurde aber immer noch von der Map referenziert. Die Garbage Collection konnte hier nicht aufräumen.

Der Bugfix

Der Bugfix war einfach: Die Logik des `SessionManagers` wurde so geändert, dass `removeSession` bedingungslos aufgerufen wird, wenn eine Session ungültig wird oder abläuft, unabhängig davon, ob der Abmeldevorgang Exceptions geworfen oder nicht geworfen hat. Konkret bedeutet das ein `finally`-Block.

Nach der Implementierung des Fixes überwachten wir die Metriken genau. Die Metrik `userauth_service.active_sessions_map_size` stabilisierte sich sofort und nahm dann allmählich ab. Die Metrik `process.runtime.jvm.memory.heap_usage` kehrte zu ihrem gesunden Ausgangswert zurück. Die Sorgen um die `UserAuthService`-Instanzen waren damit vom Tisch.

Über Memory Leaks hinaus: Proaktives Speichermanagement mit OTel

Der Vorfall hat unsere Herangehensweise an das Debugging von reaktiv auf proaktiv umgestellt. Dies harmoniert auch mit OTel: Wir haben Thresholds definiert, die erstens: Grenzen ziehen zwischen normalen und abnormalen Ressourceneinsatz, und zweitens: Grenzen ziehen zwischen Spans von normaler und abnormaler Länge. Die Thresholds wurden um ein Alerting-System erweitert, das uns warnt, sobald Threshholds überstiegen werden. Dadurch herrscht Klarheit über:

- **Plötzliche Speicherspitzen**: Auch wenn es sich nicht immer um ein Leck handelt, können plötzliche, unerklärliche Speicherspitzen auf eine ineffiziente Verarbeitung großer Nutzdaten oder eine Ressourcenüberlastung hinweisen.

```
{"timestamp": "...", "level": "INFO", "message": "SessionManager: Neue Sessions für user_id=12345 hinzugefügt",
"session_id": "abc-123", "trace_id": "...", "span_id": "..."}
```

Listing 1: Einzelnes Item aus den gefilterten Logs

Abbildung 3: Aggregierte Daten „bottom-up“, dargestellt in Dashboards, die beliebig anpassbar sind.

- **Hohe GC-Aktivität**: Bei verwalteten Programmiersprachen eine zunehmende Häufigkeit oder Dauer von Garbage Collection-Pausen.
- **OOM-Ereignisse**: Kritische Alarme für Out-of-Memory-Fehler, die eine sofortige Reaktion auf den Vorfall auslösen.

Unsere Systemlandschaft ist damit wesentlich besser zu managen und beschleunigt das Debugging signifikant.

Continous Observing und Power-Profiling

OTel-Daten können in fortgeschrittene Tools zur Erstellung von Leistungsprofilen integriert werden. Zwei Beispiele:

1. In Kombination mit dem Java Flight Recorder (JFR) oder anderen JVM-Profiling-Tools können Objektzuweisungen, Garbage Collection-Verhalten und Thread-Aktivitäten noch detaillierter überwacht werden.
2. Kontinuierliches Profiling in Staging- oder Vorproduktionsumgebungen kann Memory Leaks erkennen, noch bevor sie auf Produktion deployed werden.

Aufbau einer Kultur der Beobachtbarkeit

Das Ziel besteht darin, Entwickler dazu zu ermutigen, von Anfang an über Instrumentierung nachzudenken, und nicht erst im Nachhinein. In dem wir Traces, Metriken und Protokolle in den Entwicklungsprozess einfließen lassen, veranlassen wir Entwickler zu einem „shift-left“. Fehlersuche gelingt damit schneller, Teams managen ihre Probleme von selbst, und am Ende des Tages unterstützt eine Kultur des Verantwortungsbewusstseins das Unternehmen als Ganzes.

Schlussfolgerung: Verbesserte Fehlersuche mit OpenTelemetry

Memory Leaks sind eine Chaostheorie für sich. Im zutiefst undurchschaubaren Dschungel unserer Software heraus nehmen sie Gestalt an und ziehen mit katastrophalen Zerwürfnissen über unsere Systemlandschaft her. Eine naive Überwachung wird solcherart Entwicklungen nicht verhindern – unsere Fallstudie von Excellence Ltd. hat dies gezeigt.

Mit OTel nutzen wir eine Trigonometrie der Tatsachen, die das Chaos durchschaut: (1) Traces zur Verfolgung von Anforderungsflüssen, (2) Metriken zur Quantifizierung des Ressourcenverbrauches und (3) Protokolle zur Bereitstellung von detailliertem Kontext. Dieser Ansatz verwandelt reaktive Brandbekämpfung in proaktive Problemlösung und ermöglicht es uns, Memory Leaks zu diagnostizieren, zu beheben, und – noch besser – von vornherein zu verhindern.

Referenzen

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

Markus Meyer

messerblatt@protonmail.com

Markus Meyer ist als IT-Berater und Software-Entwickler auf eigene Faust unterwegs. Eigentlich auf Machine Learning geeicht, springt er in alle Krisenherde, die IT-Skills benötigen.

Vom Sensor bis zum Dashboard: Echtzeit-Anomalieerkennung in Kälteanlagen

Eldar Sultanow, Universität Potsdam, Matthias Petrich, Kältetechnik Dresen + Bremen GmbH

Ammoniak-Kälteanlagen sind ein unsichtbares Rückgrat vieler Industrien: von der Lebensmittelproduktion über die Pharmaindustrie bis hin zu energieintensiven Fertigungsprozessen und den großen Logistikzentren, die Warenströme rund um die Uhr auf konstant niedrigen Temperaturen halten müssen. Ihr großer Vorteil besteht darin, dass Ammoniak ein natürliches Kältemittel ist, das effizient arbeitet und weder die Ozonschicht angreift noch das Klima belastet. Damit sind diese Anlagen eine nachhaltige und zukunftssichere Alternative zu fluorierten Kältemitteln, die immer stärker reguliert werden.

Doch die Technik hat eine Kehrseite: Ammoniak-Anlagen sind komplex und sensibel. Schon kleine Abweichungen bei Temperatur, Druck, Leitfähigkeit oder pH-Wert können Ablagerungen, Korrosion oder sogar gesundheitsgefährdende Legionellen verursachen. Klassische Wartungszyklen und Grenzwertsteuerungen stoßen hier an Grenzen, besonders in Logistikzentren, wo jede Störung der Kühlung sofort Lieferketten und Waren im Millionenwert gefährdet. Genau hier setzt Künstliche Intelligenz an.

Warum KI für Kälteanlagen?

Ammoniak gilt als eines der nachhaltigsten Kältemittel: Es arbeitet hoch effizient, belastet nicht die Ozonschicht oder das Klima und macht Kälteanlagen zu einer zukunftssicheren Alternative zu F-Gasen (fluorierte Treibhausgase). Allerdings erfordert der Betrieb höchste Präzision: Schon kleine Abweichungen bei Temperatur, Leitfähigkeit oder pH-Wert können Ablagerungen, Korrosion oder sogar Hygienerisiken verursachen.

Besonders kritisch ist die Abschlämmung, also der regelmäßige Austausch von Kreislaufwasser. Durch die ständige Verdunstung im Kühlkreislauf steigt die Konzentration von Salzen und Schmutzpartikeln kontinuierlich an. **Läuft die Abschlämmung zu selten**, drohen Kalkausfällungen, Salzablagerungen und Korrosionsschäden. Gleichzeitig begünstigen organische Stoffe und hohe Temperaturen Hygienerisiken, nämlich die Bildung von Biofilmen – eine ideale Brutstätte für Mikroorganismen wie Legionellen, die über Aerosole schwere Erkrankungen verursachen können. **Läuft die Abschlämmung zu oft**, explodieren Wasser-, Abwasser- und Chemikalienkosten. Sobald die Abschlämmung über das notwendige Maß hinaus betrieben wird, schlägt sie unmittelbar auf die Betriebskosten durch. Jeder unnötig abgeleitete Kubikmeter Wasser enthält wertvolles Frischwasser und kostspielige Chemikalien, die im Abwasser verloren gehen. Die Folgen sind ein massiv erhöhter Verbrauch an Frischwasser, steigende Abwassergebühren und zusätzliche Kosten für die Nachdosierung von Bioziden und Korrosionsschutzmitteln.

Abbildung 1 zeigt eine Ammoniak-Kälteanlage im Norden Deutschlands.

Klassische Grenzwertsteuerungen in modernen Anlagen greifen zu kurz, denn sie reagieren nur auf einzelne Parameter und übersehen das Zusammenspiel vieler Faktoren. Leitfähigkeitssensoren steuern dort die Abschlämmung folgendermaßen: Überschreitet die Salzkonzentration einen Grenzwert, öffnet ein Ventil und führt Wasser ab, bis die Werte wieder im Sollbereich liegen. Warum greift das zu kurz? Da weitere Faktoren wie der pH-Wert eine entscheidende Rolle spielen: Ist er zu niedrig, wird das Wasser aggressiv und fördert Korrosion. Ist er zu hoch, entstehen Kalkablagerungen, die den Wärmeübergang verschlechtern. Zudem wirkt Biozid nur im richtigen pH-Bereich zuverlässig gegen Biofilme und Keime.

Hier entfaltet KI ihre Stärke: KI-gestützte Systeme erkennen Muster und Abweichungen in Echtzeit, die für das menschliche Auge kaum sichtbar sind. Statt nur auf feste Grenzwerte zu reagieren, analysieren sie kontinuierlich Temperatur, Leitfähigkeit, pH-Wert und weitere Sensordaten im Zusammenspiel. So lassen sich Anomalien frühzeitig identifizieren, bevor sie zu Schäden, Produktionsausfällen oder hygienischen Risiken führen können.

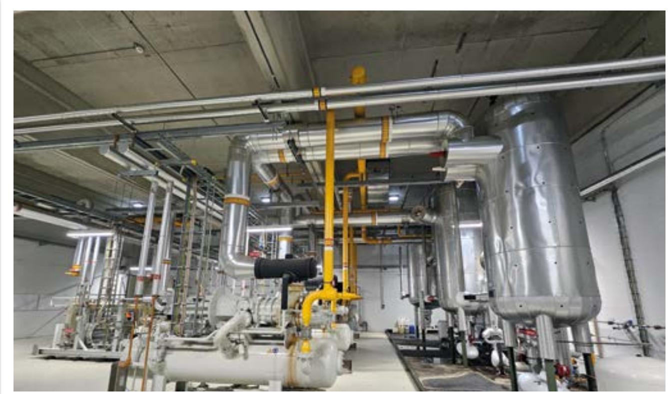
Unser Anomalie-Erkennungsmechanismus analysiert kontinuierlich die Sensorwerte der Kälteanlage und erkennt sofort, wenn die Abschlämmung übermäßig läuft. Anstatt den Fehler erst bei der nächsten Abrechnung oder Wartung zu bemerken, wird der Betreiber direkt benachrichtigt. Das stoppt Kosten und Chemikalienverluste unmittelbar, bevor sich der Schaden summiert. Das Ergebnis ist ein intelligentes, selbstlernendes System, das Kälteanlagen effizienter, sicherer und nachhaltiger macht, weil der intelligent gesteuerte Abschlämmprozess Wasser und Chemikalien sparsamer einsetzt, Betriebsrisiken minimiert, Kosten reduziert und die Umwelt entlastet.

Nachhaltige Kältetechnik trifft moderne Datenarchitektur

Gemeinsam mit der Kältetechnik Dresen + Bremen GmbH haben wir ein System entwickelt, das den Betrieb industrieller Ammoniak-Kälteanlagen durch Echtzeit-Datenanalyse revolutioniert. Die Idee: Statt reaktiv zu handeln, überwacht das System diese industriellen Ammoniak-Kälteanlagen in Echtzeit, prüft Sensorwerte nicht nur punktuell, sondern überwacht und analysiert sie mittels KI kontinuierlich.

Die Vorteile liegen auf der Hand:

- Betreiber erkennen Abweichungen sofort und können eingreifen.
- Der Betrieb wird effizienter, sicherer und hygienischer.
- Wasser- und Chemikalienverbrauch sinken, was Kosten spart und die Umwelt entlastet.



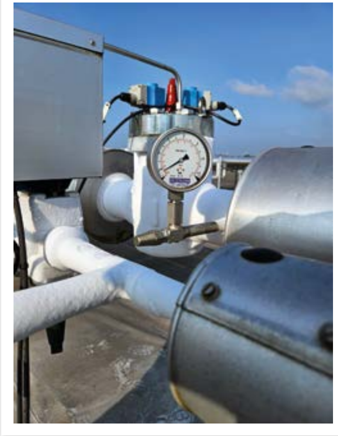


Abbildung 1: Kälteanlage in einem norddeutschen Logistikzentrum

Ein Beispiel macht die Wirkung deutlich: Wird zu viel Wasser abgeschlämmt, explodieren die Betriebskosten schnell. Jeder unnötig abgeführte Kubikmeter enthält nicht nur Frischwasser, sondern auch teuer dosierte Chemikalien. In großen Anlagen summiert sich das leicht zu fünf- oder sechsstelligen Beträgen pro Jahr. Neben den ökonomischen Belastungen entsteht auch eine ökologische Schieflage durch Ressourcenverschwendung und Umweltbelastung. Der Anomaly Detector erkennt Fehlsteuerungen in Echtzeit und verhindert wirtschaftliche wie ökologische Schäden.

Unsere Datenarchitektur überträgt die Sensorwerte per MQTT an die Event-Streaming-Plattform *Apache Kafka*, die die Daten

- zwecks Prüfung an einen Anomaly-Detector und
- zwecks Speicherung an Elasticsearch

übermittelt. *Kibana* visualisiert alle Daten in einem Dashboard in Echtzeit, wie zum Beispiel die pH-Wertzeitreihe mitsamt Anomalien. *Abbildung 2* zeigt, dass um etwa 4:00 Uhr eine Anomalie aufgetreten ist. Wir erinnern uns: Die pH-Wertmessung ist wichtig zur Überwachung der Sicherheit und Integrität der gesamten Kälteanlage.

Vom Sensor bis zum Dashboard

Die Steuerung einer industriellen Ammoniak-Kälteanlage basiert auf einer Vielzahl von Sensoren und Aktoren. *Abbildung 3* stellt die Gesamtarchitektur grafisch dar, die im Folgenden erklärt wird. Temperatursensoren, Druck- und Füllstandsensoren, Leitfähigkeits- sowie Durchfluss- und Strömungssensoren erfassen kontinuierlich den Anlagenzustand. Pumpen, Ventilatoren und Magnetventile setzen diese Signale in konkrete Aktionen um, gesteuert durch die SPS (Speicherprogrammierbare Steuerung). Über das S7-Protokoll können die in der SPS gespeicherten Daten direkt ausgelesen werden (1a/2).

Node-RED dient als Brücke zwischen der SPS und dem MQTT-Broker *Eclipse Mosquitto*. Es liest die Daten über S7 aus und überträgt sie im MQTT-Format an den Broker (3). Für Tests ohne reale Anlage steht ein Data Simulator bereit, der künstlich Sensordaten erzeugt und ebenfalls über MQTT einspeist (1b).

Vom MQTT-Broker fließen die Daten in die Kafka-Umgebung, die modular aus mehreren Containern besteht. Ein Kafka-Setup-Container konfiguriert das System beim Start, das Kafka UI erlaubt die komfortable Überwachung, und Kafka Connect sorgt für die Anbindung an weitere Systeme (4).

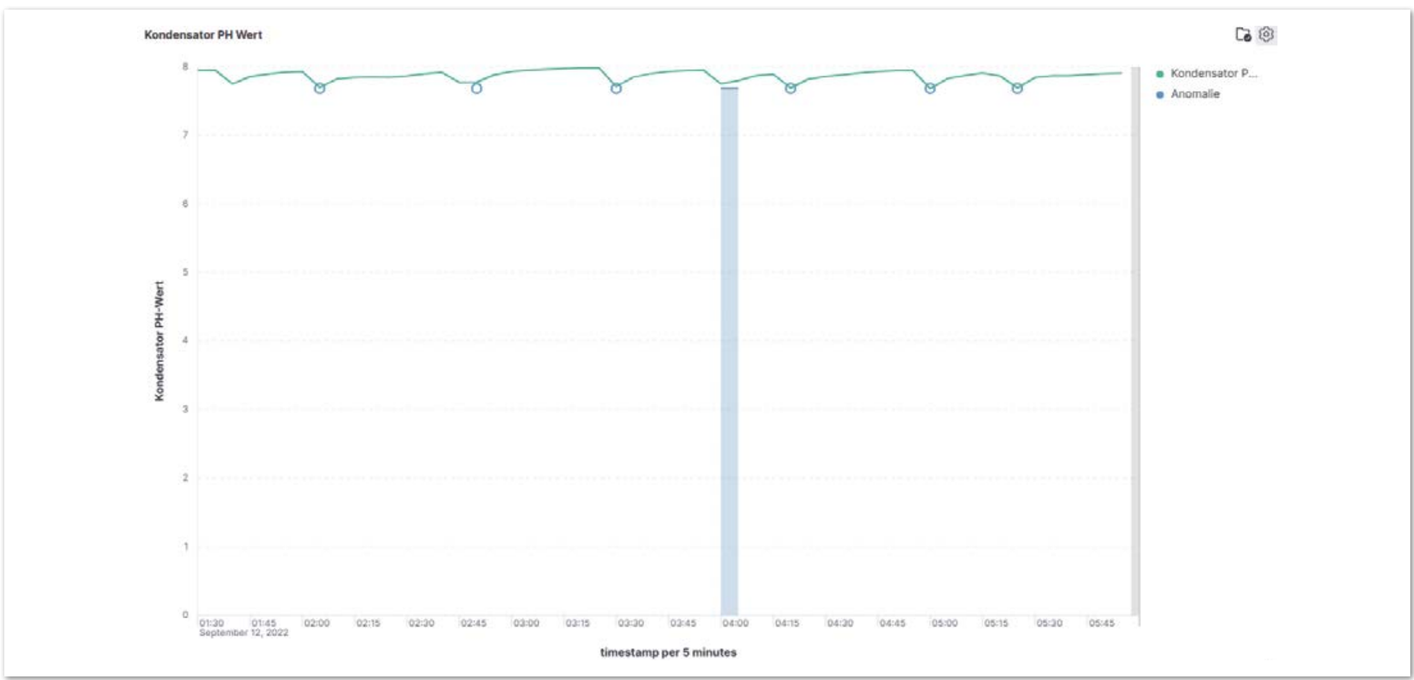


Abbildung 2: Kibana-Dashboard visualisiert den kontinuierlich gemessenen pH-Wert und auftretende Anomalien.

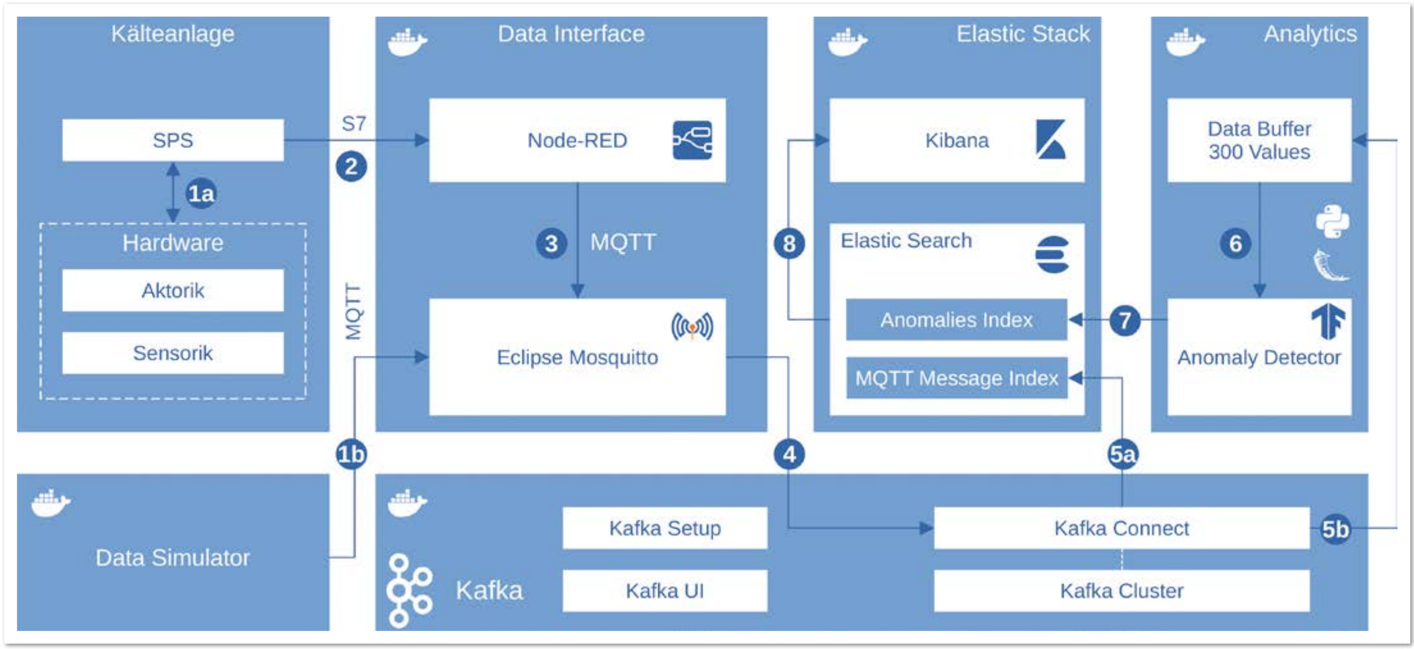


Abbildung 3: Architektur des Gesamtsystems zur Erkennung von Geräteanomalien

Sobald neue Daten eintreffen, werden sie parallel an zwei Ziele verteilt: Einerseits in den Elastic Stack, wo sie im MQTT Message Index gespeichert werden (5a). Andererseits in den Analytics-Container (5b), der jeweils 300 Werte sammelt und gebündelt auf Anomalien prüft (6). Erkennt der Algorithmus Abweichungen, schreibt er die Ergebnisse in den Anomalies Index von *Elastic Search* (7).

Für die Nutzer entsteht so ein klares Gesamtbild: *Kibana* visualisiert die Zeitreihen und markiert Anomalien sofort sichtbar (8). Betreiber sehen auf einen Blick, ob die Anlage im grünen Bereich läuft oder ob Handlungsbedarf besteht.

Zusammenfassung

KI-gestützte Anomalieerkennung macht Ammoniak-Kälteanlagen zu einem doppelten Gewinn: Sie steigert die Betriebssicherheit und senkt gleichzeitig Kosten sowie Ressourcenverbrauch. Betreiber erkennen Abweichungen sofort, reagieren gezielt und vermeiden so Schäden, Ausfälle und unnötigen Verbrauch von Wasser und Chemikalien.

Die Verbindung von nachhaltiger Kältetechnik mit moderner Datenarchitektur zeigt, wie Digitalisierung ganz konkret in der Praxis wirkt: Sie macht komplexe Systeme transparent, effizient und klimafreundlich. Das Projekt verdeutlicht, dass KI nicht nur ein theoretisches Zukunftsthema ist, sondern heute schon messbaren Mehrwert in industriellen Prozessen liefert, und zwar von mehr Sicherheit über geringere Kosten bis hin zum aktiven Beitrag zum Klimaschutz.

Eldar Sultanow
Universität Potsdam
eldar.sultanow@wi.uni-potsdam.de

Dr. Eldar Sultanow ist IT-Stratege bei einer der weltweit führenden Beratungsgesellschaften für digitale Transformation, Lehrperson an der Universität Potsdam und Gutachter für das Deutsche Zentrum für Luft- und Raumfahrt (DLR). Der promovierte Wirtschaftsinformatiker blickt auf eine mehr als zwanzigjährige Erfahrung in der Softwareentwicklung zurück: von der Programmierung bis hin zum KI-Design.

Matthias Petrich
Kältetechnik Dresen + Bremen GmbH
mape@ktdb.de

Matthias Petrich ist Software-Engineer bei Kältetechnik Dresen + Bremen GmbH, einem führenden Spezialisten für industrielle Kälte- und Klimatechnik. Er entwickelt datengetriebene Lösungen zur Prozessüberwachung und bringt moderne Softwarearchitekturen mit klassischer Anlagentechnik zusammen. Sein Schwerpunkt liegt auf der Integration von IoT-Sensorik und Streaming-Plattformen.

CLOUD NATIVE FESTIVAL

im Heide Park Soltau

CloudLand
www.cloudland.org

19. – 22. MAI 2026

Aufregende Atmosphäre eines Freizeitparks!

Ein Treffen mit den Hyperscalern AWS, Microsoft Azure & Google Cloud!

Neueste Trends & Innovationen rund um Cloud Native!

Spannende Vorträge, interaktive Workshops & kreatives Networking!

#CLOUDLAND2026

Neuronale Netze in Java – Vom Gehirn inspiriert, vom Menschen gestaltet

Dr. Sandra C. Signore

Neuronale Netze sind mehr als nur ein technisches Konzept. Sie sind der Versuch, das Denken selbst zu verstehen. Inspiriert vom menschlichen Gehirn, von Synapsen, Impulsen und der Fähigkeit, zu lernen, bilden sie heute die Grundlage moderner Künstlicher Intelligenz. Doch während Maschinen immer besser darin werden, Muster zu erkennen, bleibt die Frage: Erkennen sie wirklich oder reagieren sie nur?

Als Neurologin fasziniert mich, wie eng die Grenzen zwischen Biologie und Algorithmus verlaufen und wie schnell wir vergessen, dass das, was wir „Lernen" nennen, in der Maschine etwas völlig anderes ist. Ein neuronales Netz erkennt, weil wir ihm beigebracht haben, was es sehen soll. Es entscheidet, weil wir die Grenzen dafür definiert haben. Vielleicht zeigt sich darin die eigentliche Intelligenz nicht in der Maschine, sondern in uns selbst, nämlich in der Fähigkeit, zu reflektieren, wo Nachahmung aufhört und Verständnis beginnt.

Vom Gehirn inspiriert – Wie neuronale Netze denken

Ein biologisches Neuron empfängt über Dendriten elektrische Signale, summiert sie und sendet beim Überschreiten eines Schwellenwertes ein Signal weiter. Dieses Prinzip ist erstaunlich einfach – und zugleich die Grundlage allen Lernens.

Künstliche Neuronen funktionieren ähnlich, nur in mathematischer Form *[1]*. Jedes „Neuron" in einem künstlichen Netz empfängt Eingabewerte, multipliziert sie mit Gewichten und gibt das Ergebnis über eine Aktivierungsfunktion weiter. Ein einfaches Modell könnte so aussehen wie in *Listing 1* gezeigt. Dieses vereinfachte Modell zeigt, wie Eingabewerte mit Gewichten multipliziert, aufsummiert und durch eine Aktivierungsfunktion weitergegeben werden.

```
output = activation(w1*x1 + w2*x2 + b)
```

Listing 1: Grundprinzip eines künstlichen Neurons

Was im Gehirn synaptische Plastizität ist – die Fähigkeit, Verbindungen zu stärken oder zu schwächen –, entspricht in der KI dem Anpassen von Gewichten durch das sogenannte Backpropagation-Verfahren *[3]*.

Schon in den 1950er-Jahren entwickelte Frank Rosenblatt mit dem Perceptron die erste lernfähige Maschine *[2]* – ein Meilenstein, der zeigte, dass sich biologische Prinzipien auch algorithmisch abbilden lassen.

Hier zeigt sich die Parallele: Sowohl biologische als auch künstliche Netze lernen aus Erfahrung, doch auf völlig unterschiedliche Weise. Während das Gehirn chemisch-elektrisch arbeitet, basiert das neuronale Netz auf Gradienten und Optimierung. Das Ziel ist dasselbe – Fehler zu minimieren –, doch der Weg dorthin bleibt mechanisch.

Gerade als Neurologin erkennt man darin die Grenze: Maschinen imitieren Strukturen, nicht Bewusstsein. Und doch offenbart diese Nachahmung etwas Faszinierendes über uns selbst – wie Denken überhaupt möglich ist.

Zwischenfazit – Die Grenze der Nachahmung

So sehr die Architektur künstlicher neuronaler Netze an die Biologie erinnert, so unterschiedlich sind die zugrunde liegenden Prinzipien.

Ein biologisches Gehirn arbeitet hochgradig nichtlinear, kontextabhängig und mit einer enormen Fähigkeit zur Selbstorganisation. Künstliche Netze hingegen beruhen auf festen Architekturen und mathematisch definierten Funktionen.

Der Mensch vergisst oft, dass sein Gehirn nicht nur Daten verarbeitet, sondern auch Bedeutung konstruiert. Neuronale Netze hingegen lernen Korrelationen, aber kein Bewusstsein. Diese Unterscheidung ist entscheidend, vor allem in der Medizin, in der Fehlinterpretationen gravierende Folgen haben können.

Das bedeutet: Die Faszination für maschinelles Lernen darf nicht zu einer Romantisierung führen. Wenn Maschinen Muster erkennen, ist das beeindruckend; wenn Menschen daraus Bedeutung ziehen, ist das Kultur. Beides zusammen gedacht, eröffnet jedoch ein ungeheures Potenzial: eine Symbiose aus biologischer Intuition und technischer Präzision.

Vom Menschen gestaltet – Neuronale Netze in Java

Die Java-Welt hat in den letzten Jahren enorme Fortschritte gemacht, um neuronale Netze praxistauglich zu machen. Besonders zwei Frameworks sind hier relevant:

- **Deep Java Library (DJL):** Eine moderne Open-Source-Bibliothek, die Deep-Learning-Modelle auf Basis von TensorFlow, PyTorch oder MXNet unterstützt *[4]*.
- **Deeplearning4j (DL4J):** Eines der ersten umfassenden Java-Frameworks für neuronale Netze, geeignet für verteiltes Training und Big-Data-Umgebungen *[5]*.

Beide ermöglichen es, neuronale Netze direkt in Java zu erstellen oder vortrainierte Modelle einzubinden – ohne auf Python zurückgreifen zu müssen.

Das Beispiel in *Listing 2* demonstriert, wie sich ein vortrainiertes neuronales Netz in Java laden und zur Inferenz verwenden lässt, wie etwa für Klassifikation oder Anomalieerkennung in medizinischen Daten.

```
Criteria<float[][], float[]> criteria = Criteria.builder()
            .setTypes(float[][].class, float[].class)
            .optModelUrls("models/simple-network.zip")
            .build();

try (ZooModel<float[][], float[]> model = ModelZoo.loadModel(criteria)) {
     Predictor<float[][], float[]> predictor = model.newPredictor();
     float[] prediction = predictor.predict(inputData);
}
```

Listing 2: Laden eines vortrainierten Modells mit der Deep Java Library (DJL)

Der Vorteil von Java liegt hier in der Stabilität, Typensicherheit und Skalierbarkeit. KI-Modelle lassen sich so in bestehende Unternehmenssysteme integrieren, auditieren und langfristig pflegen – ein Aspekt, der in hochregulierten Märkten wie Pharma, Biotech und Healthcare entscheidend ist.

Neuronale Netze sind also nicht nur ein Forschungswerkzeug, sondern werden durch Java zu einem verlässlichen Bestandteil produktiver Softwarelandschaften.

Zwischen Logik und Intuition – Grenzen des maschinellen Lernens

Maschinen erkennen Muster, das heißt, sie lernen, aber sie verstehen nicht. Ihre Stärke liegt in der Präzision, nicht in der Bedeutung. Ein neuronales Netz kann ein Tumormuster erkennen, aber es weiß nichts über Angst, Hoffnung oder Kontext.

Das unterscheidet menschliches Denken von maschineller Verarbeitung: Menschen interpretieren, abstrahieren, verbinden. Maschinen optimieren Gewichte.

In der Praxis bedeutet das: Neuronale Netze brauchen Rahmenbedingungen – definierte Daten, klare Zielgrößen und vor allem menschliche Kontrolle *[6]*.

Hier kommt wieder das Konzept des *Human-in-the-Loop* ins Spiel: Systeme, in denen der Mensch Feedback gibt, korrigiert und die Maschine dadurch in eine ethisch vertretbare Richtung lenkt *[7]*.

Für Entwicklerinnen und Entwickler bedeutet das: Verantwortung. Wer ein neuronales Netz baut, gestaltet indirekt ein System, das Entscheidungen vorbereitet. Und diese Verantwortung lässt sich nicht an Algorithmen delegieren – sie bleibt beim Menschen, der die Maschine geschaffen hat.

Praxisbeispiel – KI in der Neurologie

Gerade in der Neurologie zeigen neuronale Netze, wie stark sich Technologie und Medizin gegenseitig inspirieren können. Ein Beispiel ist die automatisierte Auswertung von MRT-Bildern: Künstliche Netze erkennen mit beeindruckender Präzision Läsionen, Mikroinfarkte oder Tumorstrukturen, die selbst geübten Augen entgehen könnten *[8]*.

Solche Modelle werden in der Regel mit zehntausenden annotierten Datensätzen trainiert und lernen, subtile Muster zu identifizieren, die auf pathologische Veränderungen hinweisen. Java-basierte Frameworks wie DL4J ermöglichen, diese Modelle in klinische Systeme zu integrieren – etwa für die Vorselektion von Fällen, die anschließend ärztlich überprüft werden.

Doch der entscheidende Punkt ist: Das neuronale Netz liefert keine Diagnose, sondern eine Hypothese. Die Bewertung, Einordnung und letztliche Entscheidung bleiben menschlich. So entsteht ein Zusammenspiel zwischen ärztlicher Intuition und maschineller Präzision: ein Prozess, in dem Technik nicht ersetzt, sondern erweitert.

Gerade hier zeigt sich, wie wichtig Transparenz ist: Ein erklärbares Modell kann Ärztinnen und Ärzten helfen, Vertrauen in die Technologie zu entwickeln – nicht, weil sie alles versteht, sondern weil sie nachvollziehbar bleibt.

Vertrauen und Verantwortung – Die neue Allianz

Technologie ist nur so verlässlich wie die Menschen, die sie gestalten.
In der Medizin bedeutet das, dass neuronale Netze nicht nur technisch korrekt, sondern ethisch verantwortungsvoll eingesetzt werden müssen. Es braucht Standards, die Nachvollziehbarkeit, Fairness und Datensicherheit garantieren.

Die Einführung solcher Systeme verändert auch die Arzt-Patienten-Beziehung: Wenn Maschinen Muster erkennen, aber Menschen entscheiden, verschiebt sich die Verantwortung – hin zu einer geteilten Intelligenz. Das Ziel ist nicht, Entscheidungen zu automatisieren, sondern sie zu erweitern. Ein vertrauenswürdiges System ist dabei nicht das, das keine Fehler macht, sondern eines, dessen Fehler verstanden werden können.

Ausblick – Zwischen Biologie, Code und Bewusstsein

Die Zukunft neuronaler Netze liegt in der Verbindung von technischer Exaktheit und menschlicher Reflexion. Neue Ansätze wie Explainable AI (XAI) und hybride Modelle, die Wissen mit Lernen verbinden, könnten künftig Systeme hervorbringen, die nicht nur effizient, sondern auch verstehbar sind.

Für die Java-Welt bedeutet das: KI-Modelle werden nicht mehr isoliert existieren, sondern in komplexe, transparente Software-Ökosysteme eingebettet. Gerade im Gesundheitswesen eröffnet sich hier eine Chance – für Anwendungen, die nicht nur Daten auswerten, sondern helfen, Menschen besser zu verstehen.

Darüber hinaus zeichnet sich eine neue Generation von neuronalen Netzen ab, die sich stärker an der Biologie orientieren, sogenannte neuromorphe Systeme. Diese Hardwarearchitekturen versuchen, die Struktur und Dynamik des Gehirns nachzubilden, um Energieeffizienz und Verarbeitungsgeschwindigkeit zu vereinen. Wo klassische Netze große Rechenzentren benötigen, könnten neuromorphe Chips eines Tages in mobilen Geräten laufen – etwa in medizini-

schen Wearables, die Vitaldaten interpretieren, bevor der Mensch sie überhaupt wahrnimmt.

Auch Edge-KI gewinnt an Bedeutung: Anstatt Daten zentral zu verarbeiten, geschieht das Lernen direkt am Ort der Entstehung, wie bei Sensoren, Implantaten oder mobilen Geräten. Gerade im medizinischen Umfeld ist das entscheidend: Patientendaten bleiben lokal, die Systeme reagieren in Echtzeit und gleichzeitig wird die Privatsphäre gewahrt. Für Entwicklerinnen und Entwickler bedeutet das eine neue Herausforderung – neuronale Netze müssen künftig nicht nur intelligent, sondern auch verantwortungsvoll integriert sein.

Vielleicht liegt genau darin die nächste Stufe der KI: Nicht in immer tieferen Netzen oder größeren Datensätzen, sondern in der Fähigkeit, Technologie so zu gestalten, dass sie Vertrauen verdient. Das bedeutet auch, dass Programmiererinnen und Ärztinnen, Ingenieure und Neurowissenschaftler enger zusammenarbeiten – um Maschinen zu schaffen, die nicht nur rechnen, sondern begreifen, was auf dem Spiel steht.

Vielleicht wird der größte Fortschritt in der KI nicht die Maschine sein, die denkt, sondern der Mensch, der durch sie lernt, sein eigenes Denken neu zu sehen.

Fazit – Technologie mit Bewusstsein

Am Ende bleibt die Erkenntnis: Neuronale Netze sind keine Nachbildung des Denkens, sondern eine Hommage an seine Struktur. Sie erinnern uns daran, dass Lernen – ob biologisch oder maschinell – immer ein Prozess des Suchens ist.

Java, mit seiner Stabilität und Klarheit, zeigt, dass sich technologische Kraft und menschliche Verantwortung nicht ausschließen müssen. Es erlaubt uns, KI-Systeme zu entwickeln, die nachvollziehbar, transparent und sicher sind, also Werkzeuge, die uns dienen, statt uns zu ersetzen.

Vielleicht ist das die wahre Aufgabe künstlicher Intelligenz: nicht, Bewusstsein zu imitieren, sondern uns den Spiegel vorzuhalten, damit wir verstehen, was es heißt, wirklich bewusst zu sein.

Vielleicht liegt die wahre Romantik der künstlichen Intelligenz nicht in der Maschine selbst, sondern in der Sehnsucht des Menschen, sich in ihr wiederzuerkennen.

Quellen

[1] McCulloch, W.S. & Pitts, W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. The Bulletin of Mathematical Biophysics, 5, 115–133.
[2] Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. Psychological Review, 65(6), 386–408.
[3] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
[4] Deep Java Library (DJL). *https://djl.ai*
[5] Deeplearning4j (DL4J). *https://deeplearning4j.org*
[6] Doshi-Velez, F. & Kim, B. (2017). Towards a Rigorous Science of Interpretable Machine Learning. arXiv:1702.08608.
[7] Rahwan, I. (2018). Society-in-the-Loop: Programming the Algorithmic Social Contract. Ethics and Information Technology, 20(1), 5–14.
[8] Lundervold, A.S. & Lundervold, A. (2019). An Overview of Deep Learning in Medical Imaging Focusing on MRI. Zeitschrift für Medizinische Physik, 29(2), 102–127.

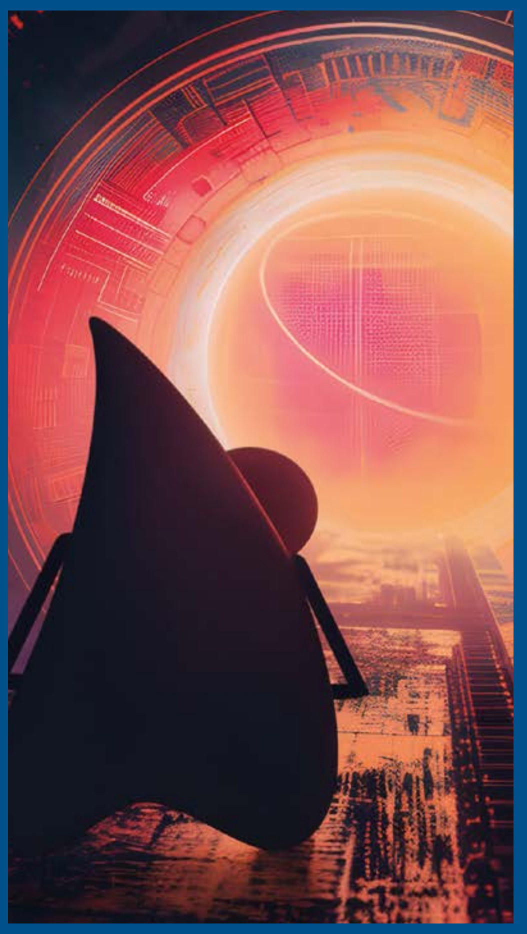
Dr. Sandra C. Signore
s.signore@online.de

Dr. Sandra C. Signore ist Neurologin und freiberufliche Beraterin für Medical Affairs und Marketing in der Healthcare-Branche. Sie arbeitet an der Schnittstelle von Medizin, Ethik und Technologie und forscht an Projekten zu Künstlicher Intelligenz. Ihr besonderes Interesse gilt dem Zusammenspiel von neuronalen Netzen, maschinellem Lernen und menschlicher Intuition. Sie setzt sich für eine verantwortungsvolle, erklärbare und empathisch gestaltete KI ein, die Wissenschaft und Menschlichkeit verbindet.

JAHRESABO

2026

Java aktuell



ALLES RUND UM JAVA

für Neueinsteigende und erfahrene Developer

FÜR 29,00 €

bestellen

ERSCHEINUNGSWEISE

4 x jährlich

JAVA-AKTUELL.EU

Mehr Informationen zum Magazin und Abo

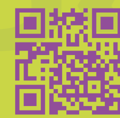
EVENT CALENDAR

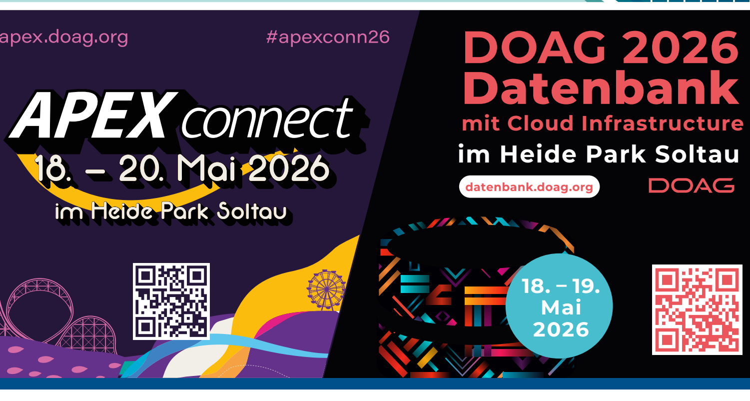
12. – 13. MÄRZ 2026

EUROPA-PARK IN RUST

#DEVLAND

WWW.DEVLAND.ORG





SAVE THE DATE
19. – 22. MAI

#CLOUDLAND2026

Impressum

Java aktuell wird von der Interessensgemeinschaft aus DOAG e.V., JavaLand GmbH und DevLand GmbH (Tempelhofer Weg 64, 12347 Berlin, java-aktuell.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. Als herstellerunabhängiger Verein bietet die DOAG Software-Anwenderinnen und -Anwendern sowie Entwicklerinnen und Entwicklern eine Plattform für den Wissens- und Erfahrungsaustausch.

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@java-aktuell.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 © Bendix
https://stock.adobe.com
S. 10: devenorr
https://123RF.com
S. 18 + 19: Bild © igor.nazlo
https://stock.adobe.com
S. 24 + 25: Bild © Designed by freepil
https://freepik.com
S. 32: Bild © Designed by freepil
https://freepik.com
S. 38 + 39: Bild © mila103
https://stock.adobe.com
S. 44 + 45: Bild © Maureen
https://stock.adobe.com
S. 50 + 51: Bild © Oulaphone
https://stock.adobe.com
S. 58 + 59: Bild © Designed by freepil
https://freepik.com
S. 64 + 65: Bild © magicmary
https://stock.adobe.com
S. 70 + 71: Bild © MAY
https://stock.adobe.com
S. 76 + 77: Bild © Alchemy
https://stock.adobe.com

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org
Mediadaten und Preise:
www.doag.org/go/mediadaten

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

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

DevLand GmbH	U2
DOAG e. V.	S. 75, S. 82, U 3
Java aktuell	S. 49, S. 57, S. 81
JavaLand GmbH	S. 9, U 4

APEX connect

18. - 20. Mai 2026

DOAG 2026 Datenbank

mit Cloud Infrastructure

18. - 19. Mai 2026

Heide Park Soltau

Die Java-
Community-Konferenz

Präsentiert von:

Veranstalter: