Java aktuell

Jakarta EE
Moderne Berechtigungssteuerung

Transformer-Architekturen
Was steckt hinter ChatGPT, DeepSeek und Co.?

Metaversum
Digitale Transformation im öffentlichen Sektor

Ausgabe 02/2025

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

JAVALAND

2025 VERPASST?

ALLE ON-DEMAND-ANGEBOTE IM TICKETSHOP

JETZT ON-DEMAND-TICKET BUCHEN UND VORTRAGSAUFZEICHNUNGEN ANSCHAUEN!

#JAVALAND JAVALAND.EU

Präsentiert von:

heise medien DOAG

Veranstalter:

Liebe Leserinnen und Leser,

freut euch in dieser Ausgabe auf eine bunte Mischung an Themen. Wir starten wie gewohnt mit dem Java-Tagebuch und einem neuen Teil der goldenen Regeln. Auch die unbekannten Kostbarkeiten sind wieder einmal mit dabei, diesmal mit dem Thema Source- und Bytecode-Versionen.

Im ersten Fachartikel dieser Ausgabe geht es um die bekannten Faustregeln des Programmierens DRY, YAGNI und SRP. Christian Kosmowski beleuchtet diese Prinzipien näher und zeugt auf, was sie beinhalten und wie sie korrekt angewendet werden können. Im Anschluss bringt uns Jelmen Guhlke im ersten Teil seiner Artikelreihe die Verwaltung von Berechtigungen in Jakarta EE näher, die ein elementarer Bestandteil moderner Unternehmensanwendungen ist. Im Anschluss tauchen wir mit Frederik Pietzko und Frank Steidinger in die Welt der Microfrontends ein. Im Artikel werden grundlegende Konzepte, Implementierungsansätze sowie Vorteile und Herausforderungen untersucht, um eine hohe Effizienz und Autonomie zu erreichen. Thomas Iffland wirft ab Seite 36 einen Blick hinter die Kulissen von Apache Kafka und deckt die Blackbox der zugrundeliegenden Kommunikation zwischen Clients und Brokern auf. Das Autorentrio Stephan Rauh, Alexander Pahn und Julian Schmidt zeigt uns im ersten Teil ihrer Artikelreihe die Neuerungen rund um Angular. Sie erläutern die Hintergründe und das Potenzial dieser Änderungen.

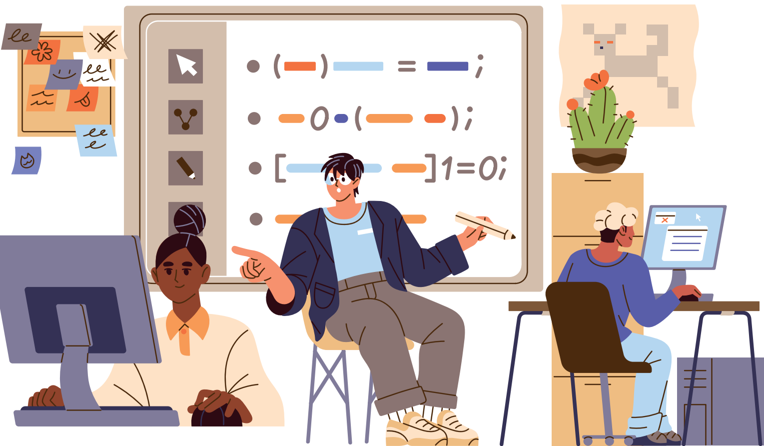
Im Anschluss folgen gleich zwei Artikel zu faszinierenden technologischen Neuerungen der Zukunft: der Transformer-Architektur und dem Metaversum. Die Transformer-Architektur liegt vielen Large Language Models wie beispielsweise ChatGPT oder DeepSeek zugrunde. Doch wie genau funktioniert diese Architektur eigentlich und was zeichnet sie aus? Diese und weitere Fragen erläutert Adrian Füller in seinem Artikel ab Seite 50. Danach geht es um das Metaversum. In ihrem Artikel beschreiben Malte Teichmann, Georg Ritterbusch, Sarah Victoria Mohr, Collin Croome und Eldar Sultanow mögliche Perspektiven des Metaversum für den öffentlichen Sektor anhand der Services der Agentur für Arbeit.

Zum Abschluss dieser Ausgabe erklärt Alexander Domene einige Gründe dafür, warum Menschen häufig ablehnend auf Veränderungen reagieren und wie man mithilfe von Architecture Decision Records Vorteile und Erklärungen von Architekturentscheidungen deutlich machen kann.

Wir wünschen euch viel Spaß beim Lesen!

Lisa Damerow
Redaktionsleitung Java aktuell

INHALT



Softwareentwicklungsprinzipien korrekt anwenden

Microfrontends: Motivation, Implementierung, Herausforderungen und Vorteile

3 Editorial

6 Java-Tagebuch
Andreas Badelt

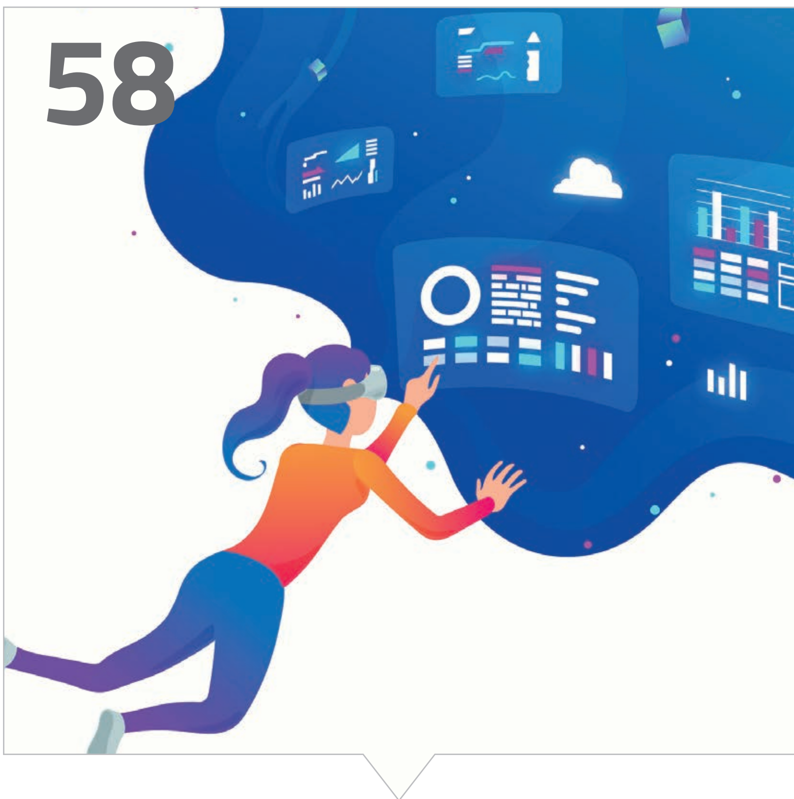
10 Die goldenen Regeln: Teil 6
Andreas Monschau

14 Unbekannte Kostbarkeiten des JDK – Heute: Source- und Byte-Code-Versionen
Bernd Müller

18 Dry Hard – Jetzt erst recht!
Christian Kosmowski

24 Moderne Berechtigungssteuerung in Jakarta EE
Jelmen Guhlke

30 Nahtlose Microfrontend-Integration für autonome Teams
Frank Steidinger, Frederik Pietzko



Transformer-Architekturen: Was steckt hinter ChatGPT, DeepSeek und Co.?

Metaversum: Digitalisierung im öffentlichen Sektor

36 Apache Kafka intern:
Wie man das Kafka-Wire-Protokoll spricht
Thomas Iffland

42 Angular – The Silent Revolution
Stephan Rauh, Alexander Pahn, Julian Schmidt

50 Die Kunst hinter ChatGPT und DeepSeek:
Geheimnisse der Transformer-Architektur
Adrian Füller

58 Metaversum: Ein neuer Ansatz zur digitalen
Transformation im öffentlichen Sektor
Malte Teichmann, Georg Ritterbusch, Sarah Victoria Mohr, Collin Croome, Eldar Sultanow

64 Endlich was bewegen!
Alexander Domene

70 Impressum/Inserenten

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

25. November 2024

Mark Scavenge

Mark Scavenge ist kein amerikanischer Schauspieler, sondern der Titel für einen neuen Garbage-Collection-Ansatz beschrieben in einem wissenschaftlichen Paper: „Mark-Scavenge: Waiting for Trash to Take Itself Out". Müll, der sich selbst rausbringt, das klingt toll. Die Publikation entstammt einem Forschungsprojekt der Universität Uppsala in Zusammenarbeit mit Oracle und basiert auf einer Kombination der beiden bislang typischen Vorgehensweisen: „scavenging" („durchstöbern"), bei der ausgehend von einem „Root Set" von Objekten die erreichbaren Objekte umgezogen („relocated") werden, um Speicherbereiche freizumachen; und „mark-evacuate", bei der in einer ersten Phase erreichbare Objekte nur markiert werden, um sie dann im zweiten Schritt umzuziehen. Die „scavenging"-Algorithmen werden insbesondere für die „Young Generation" verwendet (Serial/Parallel GC, G1), müssen dazu aber exklusiv laufen (das sogenannte „Stop-the-world").

Der „mark-evacuate"-Ansatz benötigt in der Regel weniger Speicher und verrichtet weniger Arbeit in der alten Generation, führt aber dafür zu viel unnötiger Arbeit gerade bei jungen Objekten: Die vorgeschlagene Kombination basiert auf der Beobachtung, dass junge Objekte zum Zeitpunkt der tatsächlichen „Relocation" mit hoher Wahrscheinlichkeit schon nicht mehr erreichbar sind. Durch eine verzögerte „Relocation" (erst bei dringendem Bedarf oder im drauffolgenden GC-Zyklus) könne damit gerade bei jungen Objekten ein großer Teil dieser nötigen Relocations vermieden werden. Der neue, auf Grundlage des ZGC prototypisch implementierte Algorithmus soll sich damit für eine effizientere parallele Garbage Collection gerade bei großen Heaps und großem Speicherdruck eignen. Das Paper der Forscher ist frei verfügbar *[1]*; auf inside.java gibt es eine Zusammenfassung *[2]*.

7. Dezember 2024

Jakarta EE 11 (Teil-)Release

Gestern wurde Jakarta EE 11 einstimmig vom Specification Committee freigegeben. Ganz Jakarta? Nein, aber immerhin das Core Profile, also die kleinste Untermenge der Einzelspezifikationen als minimale Basis für „Cloud Native Runtimes". Die volle Jakarta-EE-Plattform-Spezifikation und das dazwischen liegende Web Profile sollen voraussichtlich noch im ersten Quartal 2025 folgen. Der Grund für die Verzögerung war beziehungsweise ist, dass das Projekt die Modernisierung und Restrukturierung der Test Compatibility Kits (TCKs) angeht, womit unter anderem ein Umstieg auf Maven als Build-Tool (vom altehrwürdigen *Ant*) und *Arquillian* (anstelle des JT Harness) einhergeht *[3]*. Nach den Namespace-Anpassungen von „javax" zu „jakarta" in EE 9 und den Vereinfachungen in EE 10 baut Jakarta EE hier noch einmal ordentlich technische Schulden ab.

14. Dezember 2024

Ambassadors mit EE 11 Folien

Zum neuen Release haben Reza Rahman und andere Jakarta EE Ambassadors schon mal Präsentationsfolien vorbereitet, die insbesondere über die Features von EE 11 informieren (die sich ja auch außerhalb der Core-Profile-Spezifikationen nicht mehr großartig ändern werden), aber auch ein wenig über die Historie berichten und weitere Ausblicke geben. Das „Slide Deck" und seine Inhalte kann und soll auch von anderen für eigene Vorträge genutzt werden, egal ob ganz oder teilweise. Die Ambassadors sind nach eigener Aussage „always happy to help". Den Link zum Deck hat Reza zusammen mit einem YouTube-Link zu einem „Beispielvortrag" im E-Mail-Verteiler der Ambassadors geteilt *[4]*. Wer sich für das Video interessiert, aber kein Spanisch spricht – keine Angst, Reza auch nicht ☺; einfach ein bisschen vorspulen, der eigentliche Vortrag ist auf Englisch.

16. Dezember 2024

Java damals, heute, morgen

James Gosling, der „Vater" von Java, und auch der schon lange als „Java Language Architect" tätige Brian Goetz haben viel zu erzählen. Von beiden habe ich gerade jeweils einstündige Vortragsaufzeichnungen aus den letzten Monaten entdeckt, die den Bogen von Javas Vergangenheit in die Zukunft schlagen: Zunächst James Gosling über historische Merkwürdigkeiten und anhaltende Ärgernisse in Java (der englische Titel – „Historical Oddities & Persistent Itches" –

ist natürlich besser) [5]. Und Brian Goetz über „Valhalla – Java's Epic Refactor" (episch ist es in der Tat, aufgrund der tiefgreifenden Änderungen, aber auch der entsprechend langen Laufzeit [inzwischen über 10 Jahre]) [6].

20. Januar 2025

Eine Interessengruppe für die Zukunft von Jakarta EE

Es gibt zwar noch nichts Neues vom Jakarta-EE-Plattform- beziehungsweise Web-Profile-Release, aber wir können ja in der Zwischenzeit mal darauf schauen, was parallel zu Release 11 beziehungsweise darüber hinaus so passiert. Eine Entwicklung der letzten Monate ist die Gründung einer „Interest Group Jakarta EE – future directions". Das ist ein organisatorisch und prozessual in der Eclipse Foundation verankerter, aber trotzdem eher loser Verbund von Interessierten mit einer gemeinsamen Mailingliste und regelmäßigen Online-Meetings; die Gruppe soll die Innovations-Aktivitäten von Jakarta EE bündeln und deutlich verstärken [7]. Erste Meetings hat es auch bereits gegeben, in denen es insbesondere um die Themen Jakarta EE 12, das „KI-Potential" von/für Jakarta EE und höhere Akzeptanz bei Entwicklerinnen und Entwicklern ging.

Wer über die Zukunft von Jakarta EE mitdiskutieren möchte, ist in diesen Meetings und der Mailingliste herzlich willkommen. Die Termine sind im Jakarta EE Community Calendar zu finden [8].

Apropos Jakarta EE 12: Ein zentrales Thema wird die Migration von EJBs zu CDI sein. Wobei es mahnende Stimmen gibt, nicht zu offensiv davon zu reden, EJBs durch CDI zu ersetzen, da dies als „EJBs werden mit EE 12 abgeschafft" interpretiert werden könnte; es soll erst einmal CDI so weit aufgerüstet werden, dass es eine moderne Alternative für *alle* wesentlichen EJB-Funktionalitäten darstellt. Ein generelles Problem ist aber (nicht so neu) der Zeitmangel bei den Beteiligten, die neben all ihren anderen Tätigkeiten in der Regel kaum dazu kommen, sich um das inhaltliche Vorantreiben dieser und anderer Themen zu kümmern.

Ein weiteres Diskussionsthema ist die Unterstützung von „AI", also insbesondere die Nutzung von Large Language Models (LLMs). Es wird überlegt, ob bestehende APIs wie *LangChain4J* direkt übernommen oder angepasst werden sollten, um eine nahtlose Integration zu gewährleisten. *MicroProfile* hat sich des Themas aber auch schon angenommen und es besteht wohl Einigkeit darin, statt neue Baustellen zu öffnen, lieber gemeinsam mit MicroProfile und den beteiligten Herstellern an CDI-Erweiterungen in LangChain4J zu arbeiten. Zumal es für eine wirkliche Standardisierung, die Jakarta EE als Aufgabe sehen würde, in der schnelllebigen ML-Welt noch viel zu früh ist. Dazu gibt es auch ein experimentelles Projekt auf smallrye-Basis [9], das wohl demnächst ein Teil des LangChain4j-Projekts werden soll.

Weitere wichtige Themen sind unter anderem die Aufnahme von Java SE 21 und SE 25 Features (inklusive Records), modernisiertes „cloud native" Messaging, der Einsatz der Configuration Spec über die gesamte Plattform und das Abkündigen alter Anforderungen (zum Beispiel der Application Client Container). Dieses Google Doc dient unter anderem der „Future Directions"-Interessengruppe zur Sammlung der Ideen und des Feedbacks für Jakarta EE 12 [10].

25. Januar 2025

Wo backe ich einen Apfelkuchen und starte eine Java-Applikation?

„Wenn du einen Apfelkuchen von Grund auf herstellen möchtest, musst du zuerst das Universum erfinden", dieser Satz des Astrophysikers Carl Sagan dient als Ansatz für ein Erklärungsvideo zum JVM Start-Up, das den Prozess deutlich detaillierter beschreibt, als er den meisten von uns bekannt ist. Ok, das Universum wird in den 20 Minuten auch nicht erfunden; trotzdem hätte das Video, das Teil der „Stack Walker"-Playlist des offiziellen Java-YouTube-Kanals ist, gerne auch schon viele Jahre früher rauskommen können. Ich spare mir hier den Versuch einer Zusammenfassung, aber empfehle einen „Besuch" der Playlist

11. Februar 2024

JDK 24: First Release Candidate

Der erste Release-Kandidat von JDK 24 ist da – mit einer extrem langen Liste an Features [13], siehe auch letzte Tagebuch-Ausgabe. Dem pünktlichen Release Mitte März sollte nichts im Wege stehen. Picken wir uns nur noch mal einen Aspekt heraus: Wie schon berichtet, wird JEP 491 Hilfe bringen, was das „pinning" von virtuellen Threads angeht (Situationen, in denen ein virtueller Thread nicht von seinem „Carrier Thread" entfernt werden kann, um einen anderen virtuellen Thread darauf laufen zu lassen; das ist zum Beispiel bislang bei „synchronized"-Methoden und Statements der Fall). Es werden damit die meisten Probleme behoben, aber nicht alle – die verbleibenden sind in „Future Work" [14] gelistet. Wie das Pinning im Detail „funktioniert" beziehungsweise gelöst wird, und was es sonst noch für Skalierungsprobleme bei virtuellen Threads gibt, beschreibt Nicolai Parlog sehr anschaulich in einem kurzen Video [15].

19. Februar 2024

Java und AI

Es tut sich einiges im Java-Ökosystem in Bezug auf AI und ML. „Natürlich", könnte man sagen, weil es einfach zu groß ist und zu viele Investments daran hängen, um eine derart disruptive Technologie einfach zu ignorieren, aber es dauert halt immer etwas.

LangChain4j hat gerade eine Beta-Version von Release 1.0 herausgebracht, wird also langsam auch für die Nicht-„Early Adopter" interessant. *Spring AI* ist mit dem heutigen „1.0.0 M6"-Meilenstein auch kurz vor der „Produktionsreife". Ebenfalls seit zirka eineinhalb Jahren wurde *Jlama* (mit nur einem „l") als „pure Java Inference Engine" entwickelt [16]. Und *MicroProfile* möchte, wie schon erwähnt, eine Integration mit *LangChain4J* auf CDI-Basis angehen. Eine konkrete Integration mit *LangChain4J* hat *Quarkus* bereits, und Quarkus implementiert ja wiederum die *MicroProfile*-Spezifikation; die sogenannte *Quarkus LangChain4J Extension* bietet nun auch eine Integration des Model Context Protocol (MCP) an, das zum Ziel hat, die Anreicherung des Kontexts für LLMs zu standardisieren (also im Wesentlichen die Integration lokaler und entfernter Datenquellen sowie Tools – also mögliche Funktionsaufrufe, die einem LLM angeboten werden).

26. Februar 2025

JDK 25?

Nach dem richtig großen Release 24 ist vom nächsten bislang nicht viel zu hören beziehungsweise lesen. Es gibt einen neuen JEP 502: Stable Values (als Preview) zur Definition einer API für den Zugriff auf Objekte mit „immutable data" (die dann von der JVM als Konstanten behandelt werden). Ob dieser JEP-Teil von Release 25 werden soll, ist aber noch nicht festgelegt. Ansonsten gibt es eine Reihe weiterer (teilweise aber schon lange) offener JEPs. Mal sehen, was insbesondere aus den vier großen Projekten *Valhalla*, *Loom*, *Amber* und *Panama* dazukommt *[17]*.

Quellen

[1] *https://dl.acm.org/doi/10.1145/3689791*
[2] *https://inside.java/2024/11/22/mark-scavenge-gc*
[3] *https://projects.eclipse.org/projects/ee4j.jakartaee-tck/developer*
[4] *https://groups.google.com/g/jakartaee-ambassadors/c/pFt-qcx0gVM/m/Ic357OSiAAAJ*
[5] *https://www.youtube.com/watch?v=zg8xM0xxFa8*
[6] *https://www.youtube.com/watch?v=Dhn-JgZaBWo*
[7] *https://www.eclipse.org/lists/jakarta.ee-community/msg03295.html*
[8] *https://calendar.google.com/calendar/u/0/embed?src=eclipse-foundation.org_3281qms6riu4kdf354jn5idon0@group.calendar.google.com*
[9] *https://github.com/smallrye/smallrye-llm*
[10] *https://docs.google.com/document/d/1U2qEqF9K969t5b3YuX4cwex5LJPvF3bt1w27cdKNpDM*
[11] *https://www.youtube.com/playlist?list=PLX8CzqL3ArzVpnvuuVxEtMAazHDLhdrgv*
[12] *https://www.youtube.com/watch?v=ED1oc7gn5uY*
[13] *https://openjdk.org/projects/jdk/24*
[14] *https://openjdk.org/jeps/491*
[15] *https://www.youtube.com/watch?v=QDk1c0ifoNo*
[16] *https://github.com/tjake/Jlama*
[17] *https://openjdk.org/jeps/0*

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").

KI

KINAVIGATOR.EU

KI Navigator

BERLIN

DER NAVIGATOR ZUR ANWENDUNG VON KI

→ 2. + 3. JUNI 2025
MOA BERLIN

JETZT AUCH IN BERLIN:
DIE KONFERENZ, DIE ORIENTIERUNG
BEI DER ANWENDUNG VON KI BIETET.

Die goldenen Regeln: Teil 6

Andreas Monschau, Haeger Consulting

Stetig bin ich auf der abenteuerlichen Suche nach ihnen – den goldenen Regeln, mit denen man Neulingen, Junioren oder Quereinsteigern den Start möglichst vermiesen kann. Softwareentwicklung ist hart und unfair. So soll es auch bleiben. Du hast gelitten, alle sollen leiden.

Diese Regeln entspringen nicht meiner bunten Fantasie, sie finden sich noch in vielen Projekten, Unternehmen und Organisationen wieder – mit dieser Sammlung möchte ich zum einen erheitern, und zum anderen mahnen. Mahnen, diese Regeln nicht mehr anzuwenden, und Menschen, die darunter leiden, ermutigen, sich zu wehren. Hast du weitere Beispiele für Regeln und möchtest sie mit mir und anderen teilen? Dann kontaktiere mich gerne!

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

Hin und wieder spreche ich auf Meetups und Konferenzen über diese Regeln. Schaut gerne mal in den JUG-Kalender der DOAG, falls ihr Lust habt, den zugehörigen Talk zu sehen.

Regel 6: Fördere Entwicklung vs. Testing

Stell' dir doch mal vor: Dein Neuling kommt in dein Projekt, dein Unternehmen, deine Organisation. Wie wir bereits gelernt haben bist du nicht erreichbar, die Dokumentation ist unbrauchbar, und die Kommunikation läuft verdammt schlecht. Eigentlich ist alles dazu verdammt, vor die Wand zu fahren. Aber wir können noch eins draufsetzen.

In vielen Unternehmen gibt es einen Konflikt. Oftmals gärt er im Untergrund, aber häufig wird er offen ausgetragen. Es geht um Vorurteile, festgefahrene Meinungen oder auch pure (gegenseitige) Abneigung, und alles lässt sich auf zwei Bereiche herunterbrechen.
In Zeiten der „Agilität" (dazu kommen wir übrigens im nächsten Teil dieser Reihe) mischen sie sich eigentlich durch, aber die klassische Denkweise besteht immer noch in vielen Köpfen fort. Dämmert es

dir, worum es geht? Wie wäre es mit einem kleinen Quiz? Kennst du Jeopardy? Ich liefere dir eine Antwort, und du musst die passende Frage stellen. Hast du Lust? Dann ist hier die Antwort, in diesem Fall eine Definition:

„Der Prozess innerhalb des Softwareentwicklungslebenszyklus, der die Qualität einer Komponente oder eines Systems und der zugehörigen Arbeitsergebnisse bewertet."

Die Frage lautet hier natürlich „Was ist Testen?" und dies ist die Antwort, zumindest gemäß der Definition aus dem ISTQB-Glossar *[1]*. Und sei ehrlich – du hattest die Jeopardy-Musik im Ohr…

Wir sind also im Bereich „Entwickler" versus „Tester" unterwegs, und beidseitig liegen hier Vorurteile, unterschiedliche Meinungen und Blickwinkel auf die Softwareentwicklung vor - zumindest ist es häufig so, und das sollte man befüttern.

Wie kann man das machen? Ganz einfach, man begrüßt die Junior-Entwickler beziehungsweise Junior-Tester ganz standesgemäß – das geht sogar in „agilen" Projekten.

Damit ist diese Regel eine sehr gute Ergänzung von Regel #1, in der man für ein unerfreuliches Willkommen sorgt. Wie sieht diese Ergänzung nun aus?

„Hallo, neuer Tester"

Welche Nettigkeit kann man denn nun als Entwickler einem neuen Junior-Tester an den Kopf werfen? Da hätte ich drei Beispiele (und sie stammen leider auch wie immer aus dem echten Leben).

„Schön, dass du da bist. Ich muss meine Software nicht selber testen, dafür bist du ja jetzt da". Sowas hört man gerne, am besten muss man das abfällig betonen und abschätzend rüberbringen. Der nächste Ausspruch muss aber unbedingt direkt hinterherkommen: „Ihr Tester seid ja eh nur gescheiterte Entwickler." Wäre das nicht so, wären sie ja Entwickler, und keine Tester, ganz klar. Auf jeden Fall können diese beiden Aussprüche schon mal sehr gut dafür sorgen, einen neuen Tester zu demotivieren. Falls ein Tester dann irgendwann doch mit einem Bugreport oder einer Fehlerwirkung um die Ecke kommt, kann man immer noch sagen „Du testest das falsch, bei mir läuft es doch lokal, schau her!" Wer das als Entwickler noch nie gesagt hat, möge den ersten Stein werfen.

Es geht auch andersrum.

„Hallo, neuer Entwickler!"

Nun kann man auch durchaus als Tester neuen Entwickler-Kollegen die eine oder andere Nettigkeit beibringen. Gut ist schon immer zur Begrüßung der Hinweis „Ach noch so einer, der nur Mist programmiert, der nicht funktioniert". Kann man übrigens nicht nur zu den Neulingen sagen, sondern auch zu denjenigen, die schon länger dabei sind. Immer. Wichtig ist es auch, darauf hinzuweisen, dass man den Prozess (gemäß ISTQB natürlich) mehr liebt als die Software, und daher alles zu 110 % danach ausrichtet, sodass das Verfassen eines Testabschlussberichts zu einer 5-tägigen Testphase schon mal drei Wochen dauern kann. Zu guter Letzt argumentiert man in seiner Funktion als Testmanager gegenüber der Obrigkeit, dass man die Entwickler in den harten Phasen ebenfalls zum (unter Umständen manuellen) Testing heranziehen muss – denn man habe ja zu wenig Personal und destruktive Testfälle können sogar Entwickler ausführen. Ob sie wollen oder nicht.

Du siehst, man muss sich nicht mögen, um schlecht miteinander arbeiten zu können. Um es daher Neulingen beim Einstieg (und darüber hinaus!) möglichst schwer zu machen, förderst den Grabenkampf zwischen Entwicklern und Testern. Die Möglichkeiten sind quasi unendlich, und sie enden vielleicht wirklich erst dann, wenn disziplinarische Maßnahmen getroffen werden müssen…

Auch hiermit kannst du wunderbar dein Unternehmen, dein Projekt oder deine Organisation an die Wand fahren.

In der nächsten Ausgabe der Java aktuell drehen wir uns im Kreis – denn es wird iterativ, inkrementell, in einem Wort: agil!

Referenzen

[1] *https://glossary.istqb.org/de_DE/term/testen-2*

Andreas Monschau
Haeger Consulting
amonschau@haeger-consulting

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

DEINE VORTEILE

25 % Rabatt
auf JavaLand-Tickets

Java aktuell
Jahres-Abonnement

Java Community Process
Mitgliedschaft

JETZT MITGLIED WERDEN!

Ab 15 Euro im Jahr

www.ijug.eu

iJUG Verbund

Unbekannte Kostbarkeiten des JDK – Heute: Source- und Byte-Code-Versionen

Bernd Müller, Ostfalia

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

Unsere heutige Ausgabe der unbekannten Kostbarkeiten widmet sich den beiden Enums `SourceVersion` und `ClassFileFormatVersion`. Für das Tagesgeschäft eines Java-Entwicklers eher weniger interessant, sind sie doch eine nützliche Unterstützung, wenn man – was beim Autor häufiger vorkommt – sich mal wieder nicht an die Major-Versionen des Byte-Codes oder versionsabhängige Syntaxänderungen erinnern kann.

Byte-Code-Versionen

Der Autor wird häufig von Studenten gefragt, was es wohl mit der Fehlermeldung

```
java.lang.UnsupportedClassVersionError: Main has been compiled by a more recent version of the Java Runtime (class file version <n>), this version of the Java Runtime only recognizes class file versions up to <k>
```

auf sich habe. Der erfahrene Java-Entwickler weiß, dass eine JVM sich geflissentlich weigert, eine neuere Byte-Code-Version zu verarbeiten als die, für die sie gemacht ist. Eine Möglichkeit, die oben genannten Byte-Code-Versionen <n> und <k> einer Java-Version zuzuordnen, ist etwa die Übersicht der Java-Releases mit jeweiliger Byte-Code-Versionsnummer des Java Version *Almanac [1]*, Wikipedia oder die Verwendung des LLMs des Vertrauens.

Sportlicher – und für Java-Enthusiasten befriedigender – ist es, Java selbst beziehungsweise das JDK als Lösungshilfe zu verwenden. Alle benötigten Informationen sind in dem Enum `ClassFileFormatVersion` im Package `java.lang.reflect` vorhanden und können abgefragt werden. Wir verwenden hierzu `JShell`, da das API sehr schlank ist und das Schreiben und Ausführen eines Programms mit einer IDE deutlich aufwendiger wäre. *Listing 1* zeigt eine `Jshell`-Sitzung, die mit den eingefügten Kommentaren hoffentlich selbsterklärend ist. Bitte gehen Sie die einzelnen Zeilen des Listings durch.

```
// letzte Class-File-Version dieser JVM
jshell> ClassFileFormatVersion.latest()
$183 ==> RELEASE_23

// jede Version ist als Enum 'RELEASE_N' definiert
jshell> ClassFileFormatVersion.RELEASE_23
$184 ==> RELEASE_23

// kann auch über einen String erzeugt werden
jshell> ClassFileFormatVersion.valueOf("RELEASE_12")
$185 ==> RELEASE_12

// und die Major-Version als Integer
jshell> ClassFileFormatVersion.RELEASE_23.major()
$186 ==> 67

// Suche Release für Major-Version, hier 61
jshell> Arrays.stream(ClassFileFormatVersion.values())
           .filter(v -> v.major() == 61).findFirst()
$187 ==> Optional[RELEASE_17]
```

Listing 1

Quell-Code-Versionen

IDEs zeigen uns verlässlich Syntaxfehler an und empfehlen uns Code-Verbesserungen. Nichtsdestotrotz fragt sich der Autor manchmal: „War das schon immer so?", oder, „Seit wann ist das so?" Auch für diese tiefergehenden Fragen des Lebens gibt es ein API: das Enum `SourceVersion` im Package `javax.lang.model`. Dieses Enum modelliert die einzelnen Versionen identisch zum Enum `ClassFileFormatVersion`. Entstehungsgeschichtlich ist es jedoch so, dass `ClassFileFormatVersion` die Enums von `SourceVersion` übernommen hat, da `SourceVersion` seit Java 6, `ClassFileFormatVersion` aber erst seit Java 20 existiert.

Bevor wir das API auch hier mit Beispielen in der JShell erläutern wollen, müssen wir uns aber zunächst mit Teilen der Java-Syntax beschäftigen, nämlich Bezeichner, Namen und Schlüsselwörter, da diese sich im API widerspiegeln.

Die Java-Sprachbeschreibung *[2]* nutzt hierzu die folgenden Abschnitte: 3.8 Identifiers, 3.9 Keywords und 6.2 Names and Identifiers, wobei aber auch noch andere Abschnitte zumindest teilweise weitere Informationen beisteuern. Da sich die genannten Abschnitte über mehrere Seiten erstrecken, wollen wir hier nur eine erste Näherung zusammengefasst wiedergeben. Ein Bezeichner besteht aus ASCII-Zeichenfolgen, die auch als Unicode geschrieben werden dürfen und darf kein Schlüsselwort, `true`, `false` oder `null` darstellen. Bei den Schlüsselwörtern gibt es 51 reservierte und 17 kontextuelle Schlüsselwörter (Stand Java 23). Reservierte Schlüsselwörter dürfen nicht als Bezeichner verwendet werden. Kontextuelle Schlüsselwörter dürfen in bestimmten Kontexten als Schlüsselwörter verwendet werden, in anderen Kontexten nicht. Beispiele für reservierte Schlüsselwörter sind etwa `class`, `for` und `new`. Beispiele für kontextuelle Schlüsselwörter sind etwa `module`, `record` und `sealed`. Namen existieren als einfache und als qualifizierte Namen, wobei qualifizierte Namen die Namensbestandteile (Bezeichner) durch einen Punkt trennen.

Bitte, lieber Leser, nageln Sie uns nicht auf diese simplifizierenden Definitionen fest. Sie erlauben es uns aber, dass wir uns näher mit dem API des Enums `SourceVersion` beschäftigen können, was wir wiederum mit der JShell tun wollen und in *Listing 2* dargestellt haben. Wir beginnen mit Schlüsselwörtern, die nicht von Anfang an in Java enthalten waren, sondern später eingeführt wurden: `strictfp` in 1.2, `assert` in 1.4 und `enum` in 5. Dann versuchen wir noch den Unterschied zwischen reservierten und kontextuellen Schlüsselwörtern herauszuarbeiten und widmen uns zuletzt dem Unterstrich. Da sich die Semantik der Methoden, insbesondere von `isIdentifier()`, nicht sofort erschließt, zeigt *Abbildung 1* das JavaDoc der verwendeten Methoden.

Bei den Recherchen zu diesem Artikel haben wir uns auch den Quellcode des Enums `SourceVersion` angeschaut. Die Dokumentation der einzelnen Enum-Werte ist in *Listing 3* dargestellt. Es ist die kom-

```
jshell> SourceVersion.isKeyword("class")
$183 ==> true

// ein Schlüsselwort ist (leider) auch ein Bezeichner
jshell> SourceVersion.isIdentifier("class")
$184 ==> true

jshell> SourceVersion.isKeyword("strictfp", SourceVersion.RELEASE_1)
$185 ==> false

// In Java 1.2 eingeführt für exakte Gleitkommaarithmetik
// mit JEP 306 [3] obsolet
jshell> SourceVersion.isKeyword("strictfp", SourceVersion.RELEASE_2)
$186 ==> true

jshell> SourceVersion.isKeyword("assert", SourceVersion.RELEASE_3)
$187 ==> false

// In Java 1.4 als Schlüsselwort eingeführt
jshell> SourceVersion.isKeyword("assert", SourceVersion.RELEASE_4)
$188 ==> true

jshell> SourceVersion.isKeyword("enum", SourceVersion.RELEASE_4)
$189 ==> false

// In Java 5 als Schlüsselwort eingeführt
jshell> SourceVersion.isKeyword("enum", SourceVersion.RELEASE_5)
$190 ==> true

// Identifier, reservierte und kontextuelle Schlüsselwörter
jshell> SourceVersion.isIdentifier("sealed")
$191 ==> true

// gilt nicht für kontextuelle Schlüsselwörter
jshell> SourceVersion.isKeyword("sealed")
$192 ==> false

jshell> SourceVersion.isName("_", SourceVersion.RELEASE_8)
$193 ==> true

// Unterstrich mit JEP 213 [4], Java 9, kein Bezeichner mehr
jshell> SourceVersion.isName("_", SourceVersion.RELEASE_9)
$194 ==> false

jshell> SourceVersion.isKeyword("_", SourceVersion.RELEASE_8)
$195 ==> false

jshell> SourceVersion.isKeyword("_", SourceVersion.RELEASE_9)
$196 ==> true

jshell> SourceVersion.isIdentifier("System.out")
$197 ==> false

// Unterschied zwischen Namen und Bezeichnern
jshell> SourceVersion.isName("System.out")
$198 ==> true
```

Listing 2

Modifier and Type	Method	Description
static boolean	isIdentifier(CharSequence name)	Returns whether or not name is a syntactically valid identifier (simple name) or keyword in the latest source version.
static boolean	isKeyword(CharSequence s)	Returns whether or not s is a keyword, boolean literal, or null literal in the latest source version.
static boolean	isKeyword(CharSequence s, SourceVersion version)	Returns whether or not s is a keyword, boolean literal, or null literal in the given source version.
static boolean	isName(CharSequence name)	Returns whether or not name is a syntactically valid qualified name in the latest source version.
static boolean	isName(CharSequence name, SourceVersion version)	Returns whether or not name is a syntactically valid qualified name in the given source version.

Abbildung 1

```
/*
 * Summary of language evolution
 * 1.1: nested classes
 * 1.2: strictfp
 * 1.3: no changes
 * 1.4: assert
 * 1.5: annotations, generics, autoboxing, var-args...
 * 1.6: no changes
 * 1.7: diamond syntax, try-with-resources, etc.
 * 1.8: lambda expressions and default methods
 *   9: modules, small cleanups to 1.7 and 1.8 changes
 *  10: local-variable type inference (var)
 *  11: local-variable syntax for lambda parameters
 *  12: no changes (switch expressions in preview)
 *  13: no changes (text blocks in preview; switch expressions in
 *      second preview)
 *  14: switch expressions (pattern matching and records in
 *      preview; text blocks in second preview)
 *  15: text blocks (sealed classes in preview; records and pattern
 *      matching in second preview)
 *  16: records and pattern matching (sealed classes in second preview)
 *  17: sealed classes, floating-point always strict (pattern
 *      matching for switch in preview)
 *  18: no changes (pattern matching for switch in second preview)
 *  19: no changes (pattern matching for switch in third preview,
 *      record patterns in preview)
 *  20: no changes (pattern matching for switch in fourth preview,
 *      record patterns in second preview)
 *  21: pattern matching for switch and record patterns (string
 *      templates in preview, unnamed patterns and variables in
 *      preview, unnamed classes and instance main methods in preview)
 *  22: unnamed variables & patterns (statements before super(...)
 *      in preview, string templates in second preview, implicitly
 *      declared classes and instance main methods in second preview)
 *  23: no changes (primitive Types in Patterns, instanceof, and
 *      switch in preview, module Import Declarations in preview,
 *      implicitly declared classes and instance main in third
 *      preview, flexible constructor bodies in second preview)
 *  24: tbd
 */
```

Listing 3

pakteste Darstellung der syntaktischen Entwicklung von Java, die wir kennen, sodass wir sie dem Leser nicht vorenthalten wollen.

Zusammenfassung

Das JDK besitzt mit den Enums `SourceVersion` und `ClassFileFormatVersion` zwei Informationsquellen, die Entwickler für verschiedene Fragestellungen zur Entwicklungsgeschichte von Java auf Quell- und Byte-Code-Ebene nutzen können. Für den Entwickler-Alltag nicht von zentralem Interesse, da typischerweise durch IDE-Funktionalitäten abgedeckt, ist es aber eventuell doch hilfreich, diese Enums zu kennen, da sie insbesondere Diskussionen bezüglich der Einführungszeitpunkte verschiedener Sprach-Features sowie verschiedener Major-Versionen schnell beenden können.

Quellen

[1] JDK Releases (Java Version Almanac), *https://javaalmanac.io/jdk/*

[2] The Java Language Specification – Java SE 23 Edition, *https://docs.oracle.com/javase/specs/jls/se23/jls23.pdf*

[3] JEP 306: Restore Always-Strict Floating-Point Semantics

[4] JEP 219: Milling Project Coin

Bernd Müller

Ostfalia

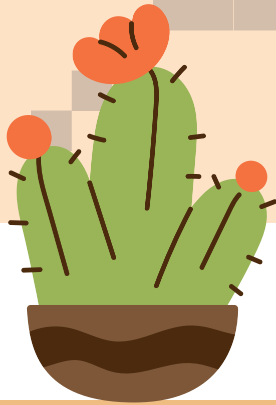
bernd.mueller@ostfalia.de

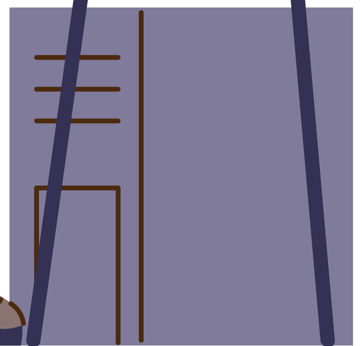
Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Autor mehrerer Bücher zu den Themen JSF und JPA sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

Dry Hard – Jetzt erst recht!

Christian Kosmowski, Continentale Versicherungsverbund

]1=0;





DRY, YAGNI, SRP sind schnell als einfache Faustregeln auswendig gelernt und weitergegeben. Doch werden sie den fundamentalen Konzepten, die dahinterstecken, auch gerecht? Dieser Artikel zeigt am Beispiel von YAGNI, DRY und SRP, wann die griffigen Akronyme zu Fehlinterpretationen führen und wie sie korrekt angewendet werden.

Software-Entwicklungsprinzipien sind eine sinnvolle Möglichkeit, häufig wiederkehrende Problemstellungen zu erfassen und immer auf die gleiche Art zu lösen. Das hilft, die Wiederholung von bestimmten Fehlertypen zu vermeiden und die Software-Qualität zu steigern.

Abkürzungen helfen uns, diese Prinzipien zusammenzufassen und besser zugänglich zu machen. Mit jeder Abkürzung geht aber auch eine Vereinfachung einher, die dazu führen kann, dass der eigentlich gemeinte komplexere Zusammenhang verloren geht. Das passiert auch häufig mit Programmierprinzipien.

YAGNI – You ain't gonna need it

Manche Software-Entwickelnde haben mit YAGNI oder „Du wirst es nicht brauchen" eine bequeme Ausrede gefunden, nicht über die Mittagspause hinaus oder gar an morgen denken zu müssen.

YAGNI hat seinen Ursprung in dem Ende der 90er Jahre hauptsächlich von Kent Beck beschriebenen Programmiermethode „Extreme Programming", in der das Prinzip als Regel so formuliert wurde: „Never add functionality early" *[1]*.

Unstrittig ist dabei, dass keine Funktionalitäten implementiert werden sollen, die nicht beauftragt wurden. Wenn die Anforderung an ein System besteht, dass eine Moped-Versicherung angelegt werden soll, darf man als Software-Entwickler:in nicht eigenständig die Funktionalität für die Versicherung von KFZ gleich mit implementieren. Auch dann nicht, wenn man glaubt zu wissen, dass diese Funktionalität in absehbarer Zeit benötigt wird. Diese Regel ist vor allem deshalb wichtig, da Vorhersagen über zukünftig vermutlich benötigte Funktionalitäten sehr häufig nicht zutreffen. Es wurden dann viele Ressourcen für den Einbau und den Ausbau einer Funktionalität verbraucht, die nie benötigt wurde.

Wie verhält es sich aber mit der grundsätzlichen Vorhersage, dass das System in Zukunft mehr als nur Moped-Versicherungen unterstützen soll? Einerseits wäre es nach YAGNI nicht erlaubt, in der initialen Implementierung diese Möglichkeit in Betracht zu ziehen und das System entsprechend flexibel auszulegen. Andererseits würde damit gegen das Open/Closed-Prinzip verstoßen werden, das besagt, dass eine Software offen für Erweiterung, aber geschlossen für Veränderung sein soll.

Robert C. Martin schreibt dazu im Buch „Clean Architecture": „There is wisdom in this message, […] On the other hand, when you discover that you truly do need an architectural boundary where none exists, the costs and risks can be very high to add such a boundary." *[2]*

Wenn also mit ausreichend hoher Sicherheit vorhergesagt werden kann, dass eine solche Architekturgrenze (also beispielsweise eine Verallgemeinerung durch Interfaces) mit hoher Wahrscheinlichkeit benötigt wird, sollte in Betracht gezogen werden, diese im Vorhinein einzubauen. Entscheidend ist dabei die Frage, ob ihre spätere Einführung deutlich höhere Kosten verursachen würde, als diese Grenze zum jetzigen Zeitpunkt einzubauen.

Martin Fowler, der zusammen mit Kent Beck an dem Projekt gearbeitet hat, in dem Extreme Programming zum ersten Mal angewendet wurde, löst das Dilemma zwischen YAGNI und Open/Closed so auf: „YAGNI only applies to capabilities built into the software to support a presumptive feature, it does not apply to effort to make the software easier to modify." *[3]*

YAGNI eignet sich also nicht als universelle Rechtfertigung, nicht in die Zukunft denken zu müssen. Im Gegenteil: Es macht besonders im Spannungsverhältnis zum Open/Closed-Prinzip eine bewusste Entscheidung über Architekturgrenzen notwendig. Welches Prinzip schwerer wiegt, ist eine Einzelfallentscheidung, die vom Entwicklungsteam bewusst getroffen werden sollte.

Neben der Frage der vorzeitigen Implementierung, ist die Strukturierung des Codes entscheidend. Hier kommt das Single-Responsibility-Principle (SRP) ins Spiel.

SRP – Single-Responsibility-Principle

Das Single-Responsibility-Principle – also das „eine Verantwortlichkeit"-Prinzip – besagt, dass es nur einen Grund geben soll, um eine Klasse zu ändern. Häufig rezitiert als „jede Klasse soll nur eine Sache tun", bietet das Prinzip viel Spielraum für intensive Diskussionen im Team.

Denn wie ließe sich diese „eine Sache" am besten greifen? Befasst sich die Klasse „EmailClient" aus *Listing 1* mit genau einer Sache, weil sie sich nur um E-Mails kümmert oder umfasst sie mehrere Sachen, weil sie benutzt wird, um generell E-Mails zu versenden und zusätzlich Willkommens-E-Mails an Benutzer verschickt?
Die genaue Formulierung: „a given method/class/component should have a single reason to change" *[4]*. Also es soll nur „einen Grund

```
public class EmailClient {
  // [...]
  public void sendEmail(String to, String subject,
String body) {
      connectToSmtpServer();
      authenticateUser();
      composeEmail(to, subject, body);
      sendEmailContent();
      closeConnection();
  }
  public void sendWelcomeEmail(User user) {
      sendEmail(user.email, "Willkommen", "Herzlich
willkommen!");
  }
  // [...]
}
```

Listing 1: Klasse EmailClient, eine Sache?

zur Änderung geben" lässt dennoch Raum für Interpretationen. „Die Anforderungen an den E-Mail-Versand haben sich geändert" könnte ein Grund sein, der hier alle Eventualitäten abdeckt.

Robert C. Martin präzisiert die Formulierung später so: „The SRP says to separate the code that different actors depend on" [2]. Code, von dem verschiedene Akteure abhängen, soll also getrennt werden – oder positiv formuliert: Nur der Code, der zu einem Akteur gehört, soll in einem Modul zusammengefasst werden dürfen. Ein Akteur kann hierbei ein einzelner Stakeholder oder auch ein Personenkreis sein, der eine bestimmte Rolle innehat. Wichtig ist hierbei, dass im Fall von mehreren Personen einheitliche Aussagen über die Anforderungen an die Software getroffen werden. Besteht die Gefahr, dass unterschiedliche Aussagen getroffen werden, handelt es sich um unterschiedliche Akteure und der Code muss getrennt werden.

Wenn diese Aussage auf *Listing 1* angewendet wird, sind mindestens zwei Akteure auszumachen. Die System-Administration könnte beispielsweise Änderungen an der Verbindungsmethode bewirken, die zur Verbindung mit dem E-Mail-Server verwendet wird, während das Marketing-Team den Willkommenstext ändern möchte. Die Methode `sendWelcomeEmail` müsste also in eine separate Klasse ausgelagert werden. Sind alle weiteren Schritte, die zum Versand der E-Mail notwendig sind, ausschließlich durch die System-Administration beeinflussbar, können diese gemeinsam in einer Klasse verbleiben.

Der Merksatz zu SRP sollte also nicht mehr heißen: „Eine Klasse soll eine Verantwortlichkeit haben", sondern: „Es soll nur eine verantwortliche Person/einen verantwortlichen Personenkreis geben, der Änderung an der Komponente bewirken kann". Dieses Prinzip ist auch bei der folgenden Betrachtung des DRY-Prinzips nützlich.

Eng verwandt mit dem SRP ist das DRY-Prinzip („Don't Repeat Yourself"), das sich mit der Vermeidung redundanter Informationen beschäftigt, jedoch oft falsch interpretiert wird.

DRY – Don't Repeat Yourself

Als besonders populäre Abkürzung bei den Programmierprinzipien kommt „DRY" eine ganz besondere Rolle zu; kann man es sich doch besonders gut merken, da es für sich genommen wieder ein sinnvolles Wort ergibt. Professionelle Software-Entwickler:innen wollen, dass Quellcode sauber und trocken, also „dry" ist, oder etwa nicht?

„Wiederhole dich nicht", wird oft als „Quellcode soll nicht wiederholt werden" interpretiert, was eine häufige Fehleinschätzung ist, die in vielen Fällen mehr Probleme verursacht, als sie löst.

Anhand eines Beispiels lässt sich die Herausforderung mit dupliziertem Code besser verstehen. *Listing 2* zeigt die Implementierung eines Einkaufskorbs in einem Webshop, in dem ein Warenkorb maximal drei Artikel beinhalten kann. *Listing 3* zeigt die Implementierung einer Waren-Sendung desselben Webshops. Einer Warensendung dürfen maximal drei Artikel hinzugefügt werden. Der Code beider Klassen ist identisch, auch die IDE hat die Code-Duplikation bemerkt und weist auf die Duplikation hin.
Laut der zuvor genannten Interpretation „Code darf sich nicht wiederholen" verstößt dieser Code gegen das DRY-Prinzip. Der Code müsste ausgelagert und von beiden Stellen aus aufgerufen werden. Doch was passiert, wenn sich die Anforderungen ändern? Im Warenkorb dürfen nur noch zwei Artikel sein, da sich die Lieferbarkeit der Produkte verschlechtert hat. In einer Warensendung dürfen nun vier Artikel hinzugefügt werden, weil größere Versandkartons angeschafft wurden.

Der vorher zusammengefasste Code müsste angepasst werden. Im besten Fall wird der Code wieder in zwei Klassen getrennt, im schlechtesten Fall wird eine Fallunterscheidung eingeführt. Der Grund ist: Der Code aus *Listing 2* und *Listing 3* ist zufällig gleich, aber er repräsentiert unterschiedliches Wissen. Das Wissen darum, wie viele Artikel im Warenkorb sein dürfen und das Wissen darum, wie viele Artikel pro Warensendung versendet werden können.
In diesem Artikel bezeichnet „Wissen" die fachlichen Anforderungen und Regeln, die im Code umgesetzt werden. Dies beinhaltet sowohl die offensichtliche Logik als auch unausgesprochene Annahmen. Der gleiche Code kann unterschiedliches Wissen repräsentieren, abhängig vom jeweiligen Kontext. DRY zielt darauf ab, dieses Wissen im System eindeutig und konsistent darzustellen.

```
public class ShoppingCart {

  private List<Item> items = new ArrayList<>();

  public void addItem(Item item) {
        if (items.size() >= 3) {
              throw new TooManyItemsException();
        }
        this.items.add(item);
  }
}
```

Listing 2: Klasse ShoppingCart

```
public class Shipment {

  private List<Item> items = new ArrayList<>();

  public void addItem(Item item) {
        if (items.size() >= 3) {
              throw new TooManyItemsException();
        }
        this.items.add(item);
  }
}
```

Listing 3: Klasse Shipment

Als die Abkürzung „DRY" zum ersten Mal im Buch „The Pragmatic Programmer" erwähnt wurde, war die Definition wie folgt: „Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [5]. Es ging also bei der Ursprungsidee nicht um die Vermeidung von dupliziertem Code, sondern um die *Vermeidung von dupliziertem Wissen*.

Wenn dasselbe Wissen mehrfach innerhalb eines Systems repräsentiert ist, führt dies dazu, dass bei einer Änderung des Wissens mehrere Stellen berücksichtigt werden müssen. Ist dagegen unterschiedliches Wissen an derselben Stelle repräsentiert, besteht bei

einer Änderung die Gefahr, dass ein im System hinterlegtes Wissen verändert wird, das nicht verändert werden sollte.

Eine große Herausforderung in Sachen DRY besteht darin, das Bewusstsein dafür zu schärfen, dass sowohl IDEs als auch Tools zur statischen Code-Analyse nicht zuverlässig in der Lage sind, herauszufinden, ob es sich um dasselbe Wissen handelt. Eine Warnung über duplizierten Code darf und sollte daher bewusst ignoriert werden, wenn es sich um unterschiedliches Wissen handelt.

Wie lässt sich in der Praxis nun beurteilen, ob das DRY-Prinzip in einem konkreten Fall auf einen bestimmten Code angewendet werden sollte oder nicht? Hierzu soll folgende Abfolge von Überlegungen als Hilfestellung dienen:

Handelt es sich um dasselbe Wissen?

Kann diese Frage klar mit „Ja" beantwortet werden, sollte das DRY-Prinzip angewendet und der Code zusammengefasst werden. Einen Ausnahmefall stellen Software-Tests dar. In Test-Klassen könnte ein Entwicklungs-Team beispielsweise entscheiden, dass es wichtiger sei, dass alle Informationen in einem Test sofort sichtbar sind und so das Lokalitätsprinzip über das DRY-Prinzip stellen. Kann diese Frage klar mit „Nein" beantwortet werden, darf das DRY-Prinzip nicht angewendet werden. Ist die Antwort auf die Frage zunächst unklar, kann die Fragestellung aus dem SRP-Prinzip genutzt werden.

Welche Akteure können Änderungen an den Code-Stellen in Auftrag geben?

Wenn anhand des Codes nicht klar beantwortet werden kann, ob es sich um dasselbe Wissen handelt oder nicht, sollte der Akteur oder die Akteure, die fachlich für die fraglichen Code-Stellen verantwortlich sind, für eine Klärung mit einbezogen werden. Ob durch mehrere unterschiedliche Akteure dasselbe Wissen im Code eingebracht werden kann, hängt stark von der Software-Architektur und damit von der Organisationsstruktur des Unternehmens ab.

DRY und Quellcode-Kommentare

Ein großer Anteil an Quellcode-Kommentaren verstößt gegen das DRY-Prinzip. Auch für Kommentare gilt: Sie sollen das im System vorhandene Wissen nicht duplizieren. JavaDoc-Kommentare, die ursprünglich als Dokumentationsmöglichkeit für öffentliche APIs gedacht waren, duplizieren Wissen im System. Methodensignaturen, Parameter und Rückgabewerte werden wiederholt, der Zweck von Klassen und Methoden, der sich auch aus deren Benennung erschließen sollte, wird erneut beschrieben.

Quellcode und zugehörige Kommentare laufen allzu oft nach der ersten Änderung auseinander. Es ist daher ratsam, mit diesem bewussten Verstoß gegen das DRY-Prinzip sehr sparsam umzugehen und ihn nur dort einzusetzen, wo Entwickler:innen, die die Software verwenden sollen, keinen Zugang zum Quellcode haben.

Fazit

YAGNI, SRP und DRY sind unerlässliche Werkzeuge, aber nur, wenn man sie richtig einsetzt und die zugrunde liegenden Konzepte versteht. Das bedeutet: Bei YAGNI das Open/Closed-Prinzip im Blick zu behalten, bei SRP die verantwortlichen Akteure zu kennen und bei DRY zwischen Code- und Wissensduplizierung unterscheiden zu können. Mit dem richtigen Verständnis dieser Prinzipien ist es einfach möglich, besser lesbaren und wartbaren Code zu schreiben.

Quellen

[1] Kent Wells (1999): *http://www.extremeprogramming.org/rules/early.html* abgerufen am 27.01.2025

[2] Robert C. Martin (2017): Clean Architecture: A Craftsman's Guide to Software Structure and Design Prentice Hall, Ort.

[3] Martin Fowler (2015): *https://martinfowler.com/bliki/Yagni.html* abgerufen am 14.01.2025

[4] Robert C. Martin (2018): Clean Code: A Handbook of Agile Software Craftsmanship

[5] Andrew Hunt, Dave Thomas (1999): The Pragmatic Programmer. From Journeyman to Master

Christian Kosmowski
Continentale Versicherungsverbund
ksmwsk@gmail.com

Christian Kosmowski ist ein erfahrener Softwareentwickler und -architekt mit über 15 Jahren Praxiserfahrung und spezialisiert auf Java und Kotlin. In seiner aktuellen Rolle als Technical Coach im Continentale Versicherungsverbund vermittelt er sein Wissen in zahlreichen Projekten. Seine breite Erfahrung, von der Entwicklung bis zur Architektur großer Systeme – gepaart mit seiner Begeisterung für Software-Qualität –, macht ihn zu einem interessanten Gesprächspartner und vielseitigen Experten.

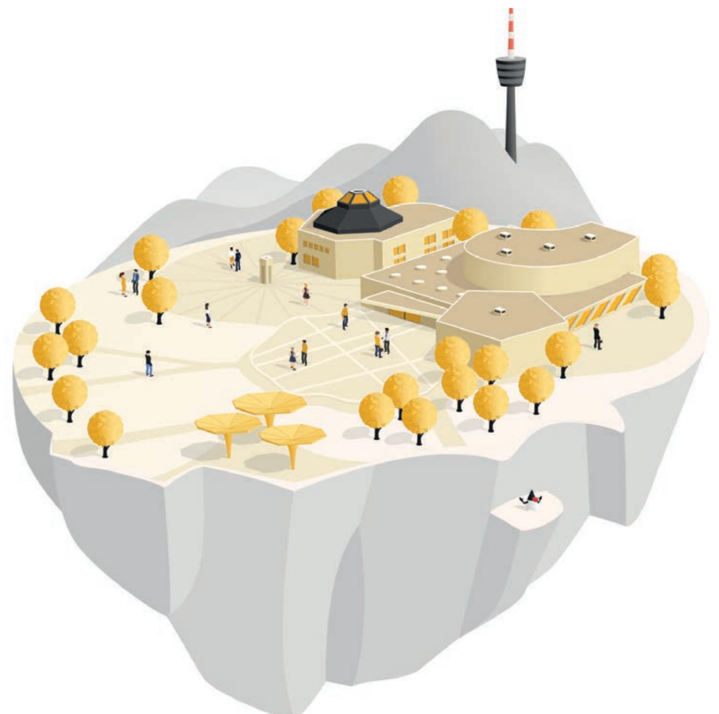
Java Forum Stuttgart 2025

10. Juli 2025 | Liederhalle Stuttgart

Ob IT-Experte oder Jungprofi - das ‚JFS', das Java Forum Stuttgart bringt die gesamte Community zusammen. Erleben Sie 49 Fachvorträge in 7 parallelen Tracks, von aktuellen Trends bis hin zu praxisnahen Best Practices.

Lernen, Netzwerken, Weiterkommen – sichern Sie sich Ihren Platz!

Jetzt anmelden:
https://www.java-forum-stuttgart.de



JAVAFORUM 2025
stuttgart

Moderne Berechtigungs-steuerung in Jakarta EE

Jelmen Guhlke, Frachtwerk GmbH

Die Verwaltung von Berechtigungen ist ein zentraler Bestandteil moderner Unternehmensanwendungen. Mit steigender Komplexität der Anwendungen und wachsender Nutzendenbasis wird es immer wichtiger, eine flexible, sichere und skalierbare Handhabung von Berechtigungen zu etablieren. In dieser dreiteiligen Artikelserie werden verschiedene Ansätze beleuchtet, wie Berechtigungen in Jakarta-EE-Anwendungen effizient implementiert und verwaltet werden können – von der Nutzung der Bordmittel von Jakarta EE bis hin zu hochverfügbaren, ausgelagerten Systemen.

Die In-House-Lösung: Berechtigungen in Eigenregie

Freude! Ein neues Projekt startet, nach konzeptioneller Arbeit wird am Datenmodell geschraubt und möglichst schnell entsteht ein erster Prototyp. Soweit sieht alles gut aus, neue Funktionen und Schnittstellen werden ergänzt und neue Nutzende werden testweise auf das System gelassen. Etwas Ausprobieren hier, etwas Testen dort, bis der ersten Person auffällt: „Darf ich das überhaupt machen?" „Alles halb so wild", sagt die Projektleitung, „Wir haben nur eine Hand voll Rechte und Rollen, das kann schnell eingebaut werden." Zwei Jahre und zahlreiche Releases später steht das Team vor einem *Big Ball of Mud* *[1]* aus verworrenen und abhängigen Berechtigungen, und es kommen Fragen auf wie: „Wieso darf User XYZ diese Schnittelle abrufen?" oder „Wieso sehe ich diesen Knopf nicht?"

Woher kommen Berechtigungen?

Berechtigungen haben in unserer Gesellschaft eine bedeutsame Rolle. Fast täglich kommen Menschen in verschiedenen Situationen mit Berechtigungen in Berührung – sei es mit der Berechtigung, ein Auto zu fahren, eine Tür zu öffnen oder auf eine bestimmte Webseite zuzugreifen. Dabei kommt die Begrifflichkeit aus einem nichttechnischen Kontext: wird das Wort Berechtigung in seine Einzelteile zerlegt, wird schnell klar, dass ein Zustand beschrieben wird, der ein bestimmtes Recht auf „etwas" richten lässt *[2][3]*. Allgemeiner formuliert: Eine Berechtigung beschreibt den Zustand, dass ein Subjekt durch den Erhalt eines Rechtes legitimiert wird, „etwas" zu tun. Im Folgenden soll ein Blick auf die Umsetzung dieser Definition in einer Jakarta EE-Applikation geworfen werden.

Zugriffssteuerung in Jakarta EE

Das Thema Berechtigungen wird im Bereich der Softwareentwicklung oftmals aus dem Blinkwinkel des Datenschutzes und der Sicherheit gesehen. Daneben gibt es aber durchaus andere Bereiche, die mittelbar durch das Thema beeinflusst werden. Zum Beispiel UX- (*User Experience*) und UI- (*User Interface*) Entscheidungen, wie die Platzierung von Inhalten und Steuerungselementen oder der Umfang von ausgespielten Informationen, werden indirekt durch das Querschnittsthema Berechtigungen beeinflusst.

Historisch gesehen hat das Thema innerhalb der Jakarta EE (vormals Java EE) Nutzendenschaft keinen guten Stand. Hierbei lässt sich die Schuld nicht nur bei externen Parteien suchen. Die Entwicklung der Sicherheitsdefinitionen war von Beginn an undurchsichtig. Ein klar erkennbarer roter Faden hat gefehlt, was dazu führte, dass die sicherheitsbezogenen APIs zum Teil verteilt oder sogar dupliziert in mehreren Spezifikationen des Jakarta(Java)-EE-Stacks beschrieben waren (vgl. Kapitel 1 aus *[4]*). Einfache Use-Cases benötigten sehr viele Zeilen Code.

Dazu kam, dass es – trotz der eigentlich angestrebten Produkt-Unabhängigkeit – viele produktbezogene Eigenheiten innerhalb einzelner Applikationsservern gab. Mit dem JSR-375 *[5]* wurde das Problem adressiert. Eine einheitliche Java *EE Security API Specification* sollte zentrale Funktionen im Bereich der Autorisierung, Authentifizierung und allgemeinen Sicherheit einer Enterprise-Applikation vorgeben. Obwohl der JSR aus dem Jahr 2017 ist, findet sich sowohl in der für Jakarta EE 10 genutzten Version 3 der *Security-Spezifikation* *[6]*, als auch in der geplanten Version 4 für das anstehende Jakarta-EE-11-Release *[7]*, folgendes Zitat (bezugnehmend auf den `SecurityContext`):

Various specifications in Jakarta EE provide similar or even identical methods to those provided be [sic] the SecurityContext. It is the intention of this specification to eventually supersede those methods and provide a cross-specification, platform alternative.

Wie ist der aktuelle Stand?

Mit dem ersten Major-Release nach der Namespace-Anpassung von Java EE zu Jakarta EE wurde mit der Version 10 das in diesem Magazin schon oft thematisierte *core*-Profil eingeführt. Mit einem Blick auf die enthaltenen Spezifikationen wird schnell deutlich, dass sich *Jakarta Security* nicht in dem *core*-Profil wiederfindet *[8]*. Dies liegt zum einen an dem erklärten Ziel des Profils, für den Einsatz in der Cloud möglichst schlank zu sein, zum anderen gibt es durch die zuvor beschriebene zerfahrene Entwicklung im Bereich der Sicherheits-APIs von Jakarta EE innerhalb der *Jakarta-RESTful-Web-Services*-Spezifikation eigene, durch die Praxis etablierte Mechanismen für den Sicherheitsaspekt im *REST*-Kontext.

Im Zuge dieser Artikelserie soll ein Blick auf die dedizierten Spezifikationen im Bereich der „Sicherheit" gewagt werden. Aus diesem Grund wird die gesamte Jakarta-EE-10-Plattform betrachtet. Konkret sollten hier drei Spezifikationen hervorgehoben werden:

- Jakarta Authentication *[9]*
- Jakarta Authorization *[10]*
- und die schon angesprochene Jakarta Security *[6]*.

Die beiden Spezifikationen *Authentication* und *Authorization* definieren hierbei allgemeinere „low-level" Service Provider Interfaces (SPI). Im Zuge einer Applikationsentwicklung ist ein direkter Kontakt mit den beiden Spezifikationen nicht zwingend notwendig. Vielmehr dienen die Definitionen aus *Jakarta Security* der direkten Nutzung im Applikationscode. An einigen Stellen kann die Spezifikation als niedrigschwelliger Kleber für die darunter liegenden *Authentication*- und *Authorization*-Spezifikationen verstanden werden. Hierbei hat Jakarta Security einen starken Fokus auf webbasierte Anwendungen.

Authentication versus Authorization

Um bei der Entwicklung nicht die beiden oft in einem Satz fallenden Begriffe *Authentication* und *Authorization* zu verwechseln, ergibt es Sinn, diese klar voneinander zu differenzieren:

Authentication: Im Zuge der Authentifizierung wird geprüft und sichergestellt, dass die andere Partei wirklich die ist, die sie vorgibt zu sein. In der Regel wird dazu ein „Etwas" abgefragt, das nur die andere Partei *wissen kann* – zum Beispiel ein Passwort, Secret Key oder eine PIN *(vgl. S. 71 aus [4])*.

Authorization: Im Zuge der Autorisierung wird geprüft, ob das zuvor authentifizierte Subjekt die *Berechtigung* hat, auf eine Ressource zuzugreifen oder eine bestimmte Aktion auszuführen.

Der Schwerpunkt dieser Serie soll auf den Berechtigungen – der *Authorization* – liegen. Allen an der Authentifizierung interessierten Menschen sei der Einstieg mit Jakarta im Web *[11]* ans Herz gelegt. Insbesondere die beiden Interfaces `IdentityStore` und `HttpAuthenticationMechanism` bieten hierfür sehr flexible und einfache Möglichkeiten.

Modelle der Zugriffssteuerung

Im Laufe der Jahre haben sich in der IT und zum Teil auch in der analogen Welt zahlreiche Modelle zur Handhabung von Zugriffsberechtigungen entwickelt. Hierbei muss oft zwischen dem Aufwand der Umsetzung und der Vielfalt an Einsatzoptionen abgewogen werden.

Auf der einen Seite gibt es das aus dem Militär bekannte *Mandatory Access Control* (MAC) *Model*, bei dem Ressourcen in starre Schubladen wie „ohne Restriktionen", „vertraulich" und „streng vertraulich" geclustert werden. Personen und Organisationen bekommen eine feste Freigabestufe, anhand derer geprüft werden kann, ob ein Zugriff gewährt wird oder nicht. Auf der anderen Seite gibt es moderne Modelle, wie die *Risk Adaptive Access Control [12]*, die in Echtzeit verschiedenste Parameter von Nutzungsverhalten, über Standort und Gewohnheiten analysieren und oftmals mittels KI (künstliche Intelligenz) bewerten, um so dynamische Entscheidungen treffen zu können, ob Zugriff gewährt wird oder nicht.

Für Anwendungsentwicklungen liegt die Wahl des Zugriffmodells oftmals in der Mitte der beiden Extreme. Die Entscheidung, welches Modell am Ende genutzt werden sollte, hängt wie fast immer von den konkreten Anforderungen und Gegebenheiten ab.

Als eines der bekannteren Modelle hat sich die *Role-Based Access Control* (RBAC) in der IT etabliert. Dies liegt maßgeblich an der aus der realen Welt leicht nachvollziehbaren Übertragbarkeit. Bei RBAC werden als zentrale Einheiten Rollen (*Roles*) definiert. Diese bündeln in der Regel aus der fachlichen Domäne abgeleitete Rechte, die ermöglichen, bestimmte Daten einsehen zu können oder manipulieren zu dürfen. So kann beispielsweise eine IT-Administratorin eines Unternehmens auf die Konfiguration des E-Mail-Servers zugreifen, jedoch bleiben ihr Einblicke in Kundendaten verwehrt. Auf der anderen Seite darf ihr Kollege in der Kundenbetreuung Informationen zu einem Kunden einsehen, die Konfiguration des Mail-Servers bleibt für ihn aber verschlossen. So werden auf der einen Seite Ressourcen und Aktionen mit einer Rolle als Bedingung in Verbindung gebracht.

Auf der anderen Seite wird ein Subjekt (Mensch oder Maschine) einer oder mehreren Rollen zugeordnet. Versucht nun das Subjekt, auf die gesicherte Ressource zuzugreifen, wird zuerst geprüft, ob die als notwendig definierte Rolle dem Subjekt zugeordnet wurde. Ist dies der Fall, wird der Zugriff gewährt.

Das `Principal`-Interface

Jakarta Security bietet als Grundgerüst für die Autorisierung ebenfalls die *Role-Based Access Control* an. Hierbei spielt das `Principal`-Interface aus dem Java SE `java.security` Package eine zentrale Rolle. Nach erfolgreicher Authentifizierung wird eine Instanz der konkreten Implementierung `CallerPrincipal` initialisiert. Dieses repräsentiert den aktuell – oftmals bezogen auf den aktuellen HTTP-Request – authentifizierten Client. Der `CallerPrincipal` wird mit einem Set aus zugehörigen Rollen im Zuge der Authentifizierung über das Security-Modul propagiert. Die `CallerPrincipal`-Klasse ist simpel gehalten und implementiert die eine aus dem `Principal` stammende Methode `public String getName()`. *Jakarta Security* erlaubt es aber auch, eigene Sub-Klassen von `CallerPrincipal` zu nutzen. Diese können somit zusätzliche applikationsspezifische Attribute enthalten *(siehe dazu Listing 1)*.

```
package org.openknowlegdehub.jakarta.security.config;
import jakarta.security.enterprise.CallerPrincipal;

  public class UserPrincipal extends CallerPrincipal {
  private final long departmentId;

  public UserPrincipal(String name, long departmentId) {
     super(name);
     this.departmentId = departmentId;
  }

  public long getDepartmentId() {
     return departmentId;
  }
}
```

Listing 1: Eigene CallerPrincipal-Klasse

Deklarative Steuerung

Um in Jakarta-Applikationen bestimmte Ressourcen oder Methoden abzusichern, kann in der Entwicklung auf die deklarative Konfiguration zurückgegriffen werden. Hierbei werden entweder mittels Angaben im XML-Format oder per Annotationen dem Applikationsserver mitgeteilt, welche Bedingungen erfüllt sein müssen, um auf einen bestimmten Teil des Codes zugreifen zu dürfen.

Angaben in der web.xml

In dem Deployment-Descriptor web.xml können mittels der Angabe von `<security-constraint>`-Elementen solche Regeln definiert werden. In *Listing 2* ist dies von Zeile 10 bis 18 zu sehen. Hierbei wird als `<web-resource-collection>` festgelegt, welche Ressource betroffen ist. In diesem Fall betrifft dies alle Web-Endpunkte, die mit `/api/` beginnen. In dem `<auth-constraint>`-Element wird dann die eigentliche Regel definiert. Im Beispiel aus *Listing 2* muss der aktuell authentifizierte `Principal` der Rolle `user` angehören. Ist dies nicht der Fall, würde der Request mit einem entsprechenden Fehlercode abgelehnt werden. Unterhalb der Bedingung werden alle für die Applikation notwendigen Rollen angegeben. Im dem konkreten Listing gibt es nur die Rolle `user`.

```
<security-constraint>
  <web-resource-collection>
     <web-resource-name>Protected Ressource</web-resource-name>
     <url-pattern>/api/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
     <role-name>user</role-name>
  </auth-constraint>
</security-constraint>

<security-role>
  <role-name>user</role-name>
</security-role>
```

Listing 2: Eine web.xml mit Sicherheitsregeln

Angaben mittels Annotationen

Die Konfiguration der Anwendung über XML-Dateien ist schon seit geraumer Zeit nicht die Lieblingsbeschäftigung einiger Kolleg:innen. Zumal Jakarta EE (Java EE) nicht zuletzt dadurch ein etwas eingestaubtes Image bekommen hat. Schon seit einigen Jahren wird versucht, neben der XML-Konfiguration eine zusätzliche codenahe Möglichkeit der Verwaltung für die Applikationsentwicklungen anzubieten. Mit Hilfe der Nutzung von Annotationen können Metainformation eng mit dem eigentlichen Code verbunden werden. Dies gilt gleichermaßen für die Autorisierungsverwaltung innerhalb einer Jakarta-Anwendung.

Die aus der *Jakarta-Annotation-Spezifikation* [13] stammenden Annotationen

- @RunAs
- @RolesAllowed
- @PermitAll
- @DenyAll
- @DeclareRoles

können allesamt genutzt werden, um die Zugriffe innerhalb der Anwendung zu steuern. @DeclareRoles wird verwendet, um die für die Applikation gegebenen Rollen zu definieren *(siehe Listing 2)*. Diese können verteilt über mehrere Klassen oder an einer zentralen Stelle angegeben werden. Hier bietet sich zum Beispiel die `jakarta.ws.rs.core.Application` an – sofern Jakarta *RESTful Web Services* in der Anwendung genutzt werden. *Listing 3* zeigt dies beispielhaft für die Rollen *user* und *admin*.

```
package org.openknowlegdehub.jakarta.security.rest;

import jakarta.annotation.security.DeclareRoles;
import jakarta.ws.rs.ApplicationPath;
import jakarta.ws.rs.core.Application;

@ApplicationPath("api")
@DeclareRoles({"user", "admin"})
public class RestApplication extends Application {}
```

Listing 3: Mit Hilfe von @DeclareRoles werden anwendungsrelevante Rollen definiert.

`@PermitAll` und `@DenyAll` schalten den Zugriff entweder für alle frei oder blockieren ihn für alle `Principals`. Wie die Annotation `@RolesAllowed` vermuten lässt, können Rollen angegeben werden, die in dem Set der Rollen des aktuell autorisierten Clients vorhanden sein müssen. Alle drei Annotationen können sowohl auf Klassen-, als auch Methoden-Ebene deklariert werden. Dabei **überschreibt** die Methoden-Konfiguration eine mögliche Klassen-Konfiguration.

In *Listing 4* wird auf Klassenebene definiert, dass die Rolle *user* notwendig ist, um auf die Ressourcen in diesem REST-Controller zuzugreifen. Bei der Implementierung der `POST`-Methode wird diese Konfiguration überschrieben. Um an den Endpunkt `.../dummy` einen POST zu senden, muss der aktuelle `Principal` in der Rolle *admin* sein. Andernfalls wird der Request abgelehnt.

```
@Path("/dummy")
@Produces(MediaType.APPLICATION_JSON)
@RolesAllowed("user")
  public class DummyRestController {

  public record DummyResponse(String message) {}
  public record DummyRequest(String message) {}

  @GET
  public DummyResponse getDummy() {
     return new DummyResponse("GET /dummy");
  }

  @POST
  @RolesAllowed("admin")
  @Consumes(MediaType.APPLICATION_JSON)
  public DummyResponse postDummy(DummyRequest dummyRequest){
     return new DummyResponse("POST /dummy with message
     %s".formatted(dummyRequest.message));
  }
}
```

Listing 4: Annotationsgetriebene Zugriffssteuerung

In den vorigen Beispielen wurde die deklarative Konfiguration lediglich in dem Anwendungsfall von *REST* gezeigt. Die Nutzung funktioniert identisch innerhalb von *Jakarta Enterprise Beans* (EJB) [14].

Programmatische Steuerung

Auch wenn die Verwendung der oben beschriebenen Methodiken je nach Komplexität der Anwendung schon genügen mag, kommt die einfache Reduktion auf Rollen gerade in größeren Anwendungen an ihre Grenzen. In den Beispielen wurden die Zugriffsregeln lediglich konfigurativ beschrieben. Der Anwendungsserver hat aus diesen

Konfigurationen im Zusammenspiel von *Jakarta Security* und *Jakarta Authorization* den eigentlichen Code für die Prüfung abgeleitet. Die *Security*-Spezifikation bietet noch ein weiteres mächtiges Interface, das die Handhabung der Zugriffsprüfung feingranularer in die Hände des Anwendungscodes legt: das `SecurityContext`-Interface.

Eine konkrete Implementierung des Interfaces kann mittels *Contexts and Dependency Injection* (CDI) mindestens in die vom Servlet- und EJB-Container verwalteten Klassen geladen werden. Das Interface stellt dabei Methoden zur Verfügung, um Informationen zu dem aktuellen autorisierten `Principal` zu erhalten. Mit `boolean isCallerInRole(String role)` kann geprüft werden, ob der aktuelle Client Mitglied in einer bestimmten Gruppe ist. Darüber hinaus kann mit `Principal getCallerPrincipal()` beziehungsweise `<T extends Principal> Set<T> getPrincipalsByType(Class<T> pType)` der aktuelle `Principal` geladen werden. Dabei bietet die `getPrincipalsByType`-Methode an, den zuvor beschriebenen applikationseigenen `Principal` zu laden. In *Listing 5* wird erst geprüft, ob es einen aktuell authentifizierten `UserPrincipal` gibt. Wenn es eine Instanz gibt, kann direkt auf die zusätzlichen Attribute zugegriffen werden.

```
Optional<UserPrincipal> optionalUserPrincipal =
securityContext.getPrincipalsByType(UserPrincipal.class)
.stream()
.findAny();

if (optionalUserPrincipal.isPresent()) {
UserPrincipal userPrincipal = optionalUserPrincipal.get();
long departmentId = userPrincipal.getDepartmentId();

// do some checks with the departmentId
}
```

Listing 5: Nutzen des eigenen Principals

Fazit

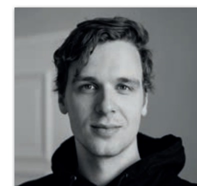
In der Jakarta-Welt erfreut sich das Thema der Authentifizierung und Autorisierung keiner allzu großen Beliebtheit. Historische Entwicklungen, die zu lange nebeneinander und ohne gemeinsames Ziel implementiert wurden, haben der Akzeptanz unter der Nutzendenschaft in diesem Bereich nachhaltig geschadet. Der JSR-375 *[5]* hat den Grundstein gelegt, um mit diesem Image aufzuräumen. Mit der für Jakarta EE 11 geplanten Version 4 der *Security*-Spezifikation gibt es ein erprobtes und solides Grundgerüst, auf dem sich die Sicherheit der eigenen Anwendung gut aufbauen lässt. Wie so oft in der community-getriebenen Open-Source-Welt: Die positive Weiterentwicklung, vor allem die Vereinheitlichung über alle Spezifikationen hinweg, hängt maßgeblich von der Mitwirkung und Rückmeldung aller ab.

Im nächsten Teil der Serie soll ein Blick auf den *Open Policy Agent [15]* geworfen werden. Insbesondere soll beschrieben werden, wie dieser aufbauend auf *Jakarta Security* angebunden und gewinnbringend genutzt werden kann.

Quellen

[1] Brian Foote und Joseph Yoder (1999): *Big Ball of Mud*. University of Illinois at Urbana-Champaign, *http://www.laputan.org/mud/*
[2] Cornelsen Verlag GmbH (2025): *Duden Online Berechtigung*. *https://www.duden.de/rechtschreibung/Berechtigung*
[3] Digitales Wörterbuch der deutschen Sprache (2025): *Präfix be-*. *https://www.dwds.de/wb/be-*
[4] Arjan Tijms, Teo Bais und Werner Keil (2022): *The Definitive Guide to Security in Jakarta EE*. Apress Media, New York.
[5] Inc. Oracle America (2017): *Java EE Security API Specification Version 1.0*. *https://javaee.github.io/security-spec/spec/jsr375-spec.html*
[6] Eclipse Foundation (2022): *Jakarta Security Version 3.0*. *https://jakarta.ee/specifications/security/3.0/jakarta-security-spec-3.0.html*
[7] Eclipse Foundation (2024): *Jakarta Security Version 4.0*. *https://jakarta.ee/specifications/security/4.0/jakarta-security-spec-4.0*
[8] Eclipse Foundation (2022): *Release Jakarta EE 10*. *https://jakarta.ee/release/10/*.
[9] Eclipse Foundation: *Jakarta Authentication*. *https://jakarta.ee/specifications/authentication/*.
[10] Eclipse Foundation: *Jakarta Authorization*. *https://jakarta.ee/specifications/authorization/*.
[11] Eclipse Foundation: *Jakarta Security, Jakarta Authorization, and Jakarta Authentication Explained*. *https://jakarta.ee/learn/specification-guides/security-authorization-and-authentication-explained/*
[12] National Institute of Standards und Technology: Risk Adaptive (Adaptable) Access Control. *https://csrc.nist.gov/glossary/term/Risk_Adaptive_Adaptable_Access_Control*.
[13] Eclipse Foundation (2021): *Jakarta Annotations Version 2.1.0*. *https://jakarta.ee/specifications/annotations/2.1/annotations-spec-2.1*.
[14] Eclipse Foundation: *Jakarta Enterprise Beans 4.0*. *https://jakarta.ee/specifications/enterprise-beans/4.0/jakarta-enterprise-beans-spec-core-4.0*.
[15] Cloud Native Computing Foundation: *Open Policy Agent*. *https://www.openpolicyagent.org/*.

Alle angegebenen URLs wurden zuletzt am 04.02.2025 besucht.

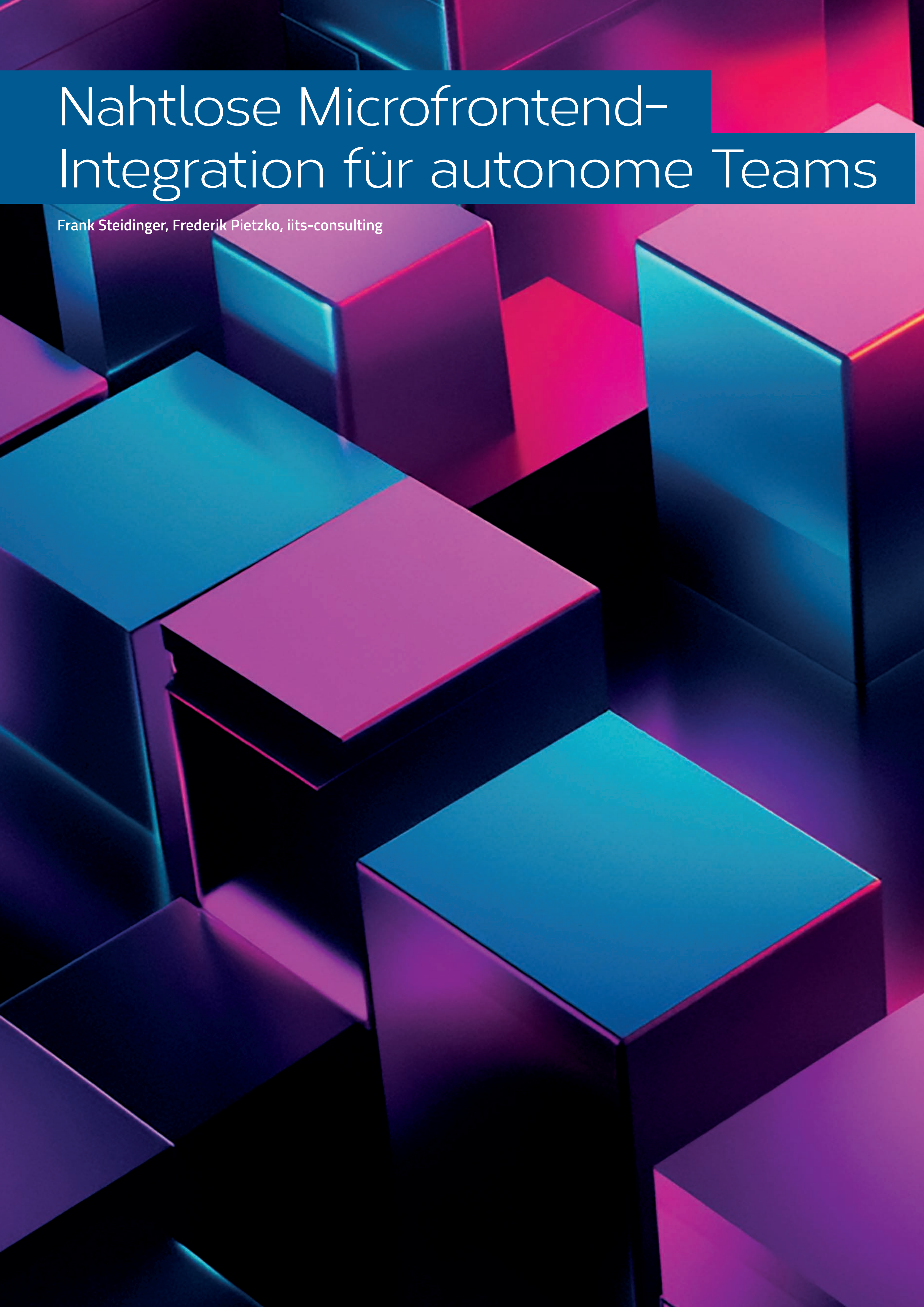


Jelmen Guhlke
Frachtwerk GmbH
mail@jguhlke.de

Jelmen Guhlke ist seit über 10 Jahren als zertifizierter Entwickler in der Java-Welt aktiv. Er arbeitet als Software Developer bei der Frachtwerk GmbH. Darüber hinaus ist er freiberuflich an verschiedenen Projekten in der Java- und Jakarta-Welt tätig. Dabei hat er einen besonderen Fokus auf Performance und Qualität. Der Transfer von Wissen stellt für ihn eine Grundsäule des gemeinsamen Arbeitens dar. Des Weiteren engagiert sich Jelmen bei dem Gemeinwohl-Ökonomie Berlin-Brandenburg e. V.

Nahtlose Microfrontend-Integration für autonome Teams

Frank Steidinger, Frederik Pietzko, iits-consulting



In der heutigen schnelllebigen Softwareentwicklung müssen Unternehmen skalierbare und wartbare Frontend-Lösungen schaffen. Solche Lösungen können Microfrontend-Architekturen sein, die die bewährten Prinzipien von Microservices auf das Frontend übertragen. Dieser Artikel untersucht die Motivation, grundlegende Konzepte und Implementierungsansätze von Microfrontends. Zudem werden die Vorteile sowie technische und organisatorische Herausforderungen beleuchtet und praxisnahe Empfehlungen zur effektiven Integration in bestehende Strukturen gegeben, um die Entwicklungseffizienz und Teamautonomie zu steigern.

Motivation für Microfrontendarchitekturen

Microfrontends versprechen, die Vorteile einer Microservice-Architektur ins Frontend zu übertragen. Während die bislang weit verbreiteten monolithischen Frontendanwendungen meist von einem speziellen Frontendteam entwickelt werden, ermöglicht eine Microfrontend-Architektur, die Verantwortung für das Frontend in die Microservice-Teams zu verlagern und zu dezentralisieren.

Grundlagen

Microfrontends zerlegen Anwendungen in voneinander unabhängige Module, die unabhängig voneinander entwickelt und deployt werden können. Das unabhängige Deployment ist der entscheidende Unterschied zwischen einem klassischen Frontend und einer Microfrontendarchitektur.

Die Microfrontends innerhalb einer Anwendung können mit unterschiedlichen Technologien realisiert werden. Das für ein Microfrontend zuständige Team hat die Freiheit, die aus ihrer Sicht optimale Technologie zu wählen. Diese Freiheit sollte aber mit Bedacht genutzt werden, jede neue Technologie bringt weitere Abhängigkeiten mit sich und erfordert Entwickler mit entsprechenden Kenntnissen, um diese zu warten. Mehr noch als bei Microservices empfiehlt es sich deshalb, sich bei Microfrontends auf möglichst wenige Technologien zu beschränken.

Soll eine Anwendung in Microfrontends zerlegt werden, muss zunächst entschieden werden, ob die Anwendung horizontal oder vertikal geschnitten wird. Bei einem horizontalen Schnitt teilen sich mehrere Microfrontends eine (logische) Seite der Anwendung. Ein typisches Beispiel aus dem E-Commerce ist, dass Produktdetails, Warenkorb und Empfehlungen als unterschiedliche Microfrontends realisiert werden. Bei einem vertikalen Schnitt hingegen übernimmt ein Microfrontend die Seite ganz oder größtenteils, zum Beispiel eine Trennung von Katalog und Check-out. Ein vertikaler Schnitt erfordert weniger Abstimmungen und Kommuni-

kation zwischen Teams und erleichtert die Frontend-Integration. Neben den Microfrontends wird meist eine Container- oder Shell-Anwendung benötigt, die gemeinsame Komponenten wie ein Hauptmenü bereitstellt und die erforderlichen Microfrontends für die aktuelle Ansicht integriert.

Nachdem die Anwendung in Microfrontends geschnitten wurde, müssen diese natürlich auch wieder zu einer Anwendung zusammengesetzt werden. Dies kann entweder vorab (im Build-Schritt) oder zur Laufzeit passieren, im letzten Fall wiederum wahlweise auf dem Server oder dem Client.

Bei der Integration im Build-Schritt werden Microfrontends als separate Bibliotheken veröffentlicht und in die Hauptanwendung eingebunden, die anschließend als ein einziges Artefakt (zum Beispiel ein JavaScript-Bundle) ausgeliefert wird. Dieser Ansatz verhindert das mehrfache Laden gemeinsamer Bibliotheken. Allerdings geht dies zulasten der Möglichkeit, Microfrontends unabhängig voneinander zu deployen, und erfordert die Koordination von Releases.

Bei der serverseitigen Laufzeitintegration fungiert die Shell-Anwendung als Template, in das die Microfrontends mittels Server-Side-Includes *[1]* als HTML-Fragmente eingebunden werden. Die Microfrontends werden anhand der Request-URL identifiziert und das entsprechende HTML sowie CSS und JavaScript abgerufen. Ein Nachteil dieses Ansatzes ist, dass die Ladezeit durch den langsamsten Server begrenzt wird. Für Single-Page-Anwendungen, bei denen das Rendering im Client erfolgt, ist dieser Ansatz ungeeignet, da diese normalerweise nur Daten vom Server abrufen.

Für Single-Page-Anwendungen bleibt somit nur die Integration auf dem Client. Im einfachsten Fall werden sowohl die Shell-Anwendung als auch die eingebundenen Microfrontends als eigenständige JavaScript-Bundles ausgeliefert. Der Client lädt zunächst die Shell-Anwendung. Diese bestimmt, welche Microfrontends für die aktuelle Ansicht geladen werden müssen, bindet sie ein und ruft die Einstiegspunkte der entsprechenden Bundles auf. Die Shell-Anwendung übergibt den Microfrontends dabei die Kontrolle über einen Bereich der Seite. Technisch erfolgt dies, indem die Shell-Anwendung dem Microfrontend ein HTML-DOM-Element zuweist. Damit die Shell-Anwendung die JavaScript-Bundles laden kann, müssen diese entweder über den gleichen Server ausgeliefert werden oder Cross-Origin-Resource-Sharing (CORS) verwenden. Die Auslieferung kann auch über einen Reverse-Proxy erfolgen, der die internen Server nach außen über die gleiche Basis-URL sichtbar macht.

Das beschriebene Zusammenbinden der Microfrontends zur Laufzeit auf dem Client hat den Nachteil, dass jedes Microfrontend alle notwendigen Bibliotheken selbst ausliefern muss. Das führt zu größeren JavaScript-Bundles und höherem Speicherbedarf im Browser. Module Federation ist ein Weg, dies zu umgehen *[2]*. Diesen Ansatz hat *Webpack* mit Version 5 eingeführt *[3]* und er wird auch durch andere Build-Tools unterstützt. Mit Module Federation können Shell-Anwendung und Microfrontends festlegen, welche Bibliotheken gemeinsam verwendet werden. Somit wird sichergestellt, dass diese Bibliotheken nur einmal ausgeliefert und zwischen den Microfrontends geteilt werden.

Vorteile bei gelungener Umsetzung

Eine Microfrontend-Architektur bietet ähnliche Vorteile wie Microservice-Architekturen in den Bereichen Entwicklung, Betrieb, Wartung und Organisationsstruktur. Sie schafft die technische Grundlage, große monolithische Frontend-Teams entlang fachlicher Domänen aufzuteilen. Dabei empfiehlt es sich, die klassische Trennung von Frontend- und Backend-Teams zugunsten interdisziplinärer, vertikal geschnittener Teams zu überdenken. Solche Teams können autonom entwickeln und ihr Domänenwissen optimal einbringen. Es ist wichtig, ihnen Autonomie und Kontrolle über ihre Microfrontends zu gewähren, während eine einheitliche Benutzererfahrung sichergestellt wird. Durch die Umstrukturierung in interdisziplinäre Teams wird der Abstimmungsaufwand zwischen Frontend- und Backendentwicklern minimiert. Neue Anforderungen können somit innerhalb eines Teams kollaborativ umgesetzt werden, was Effizienz und Produktivität erheblich steigert.

Die Nutzung von Module Federation ermöglicht unabhängige Deployments der Benutzeroberflächen. Dies führt zu schnelleren Entwicklungszyklen, was Nutzern zugutekommt, da sie kürzer auf Fehlerkorrekturen und neue Funktionen warten müssen. Zudem sind meist nicht alle Microfrontends von Programmierfehlern betroffen, sodass diese nur einen begrenzten Teil der Gesamtapplikation beeinträchtigen. Fehler lassen sich außerdem leichter dem verantwortlichen Team zuordnen, was die Ursachenbehebung beschleunigt. Dadurch steigt die Stabilität der Anwendung, genauso wie die Effizienz des Entwicklungsprozesses. Des Weiteren fördern unabhängige Deployments und die Isolation der Microfrontends moderne Entwicklungsprinzipien wie Continuous Integration und Continuous Deployment.

Außerdem sorgt die Aufteilung der Oberfläche in kleinere, vertikale Microfrontends für ein leichteres Verständnis der Teilsysteme. Dies erhöht die Wartbarkeit und erleichtert neuen Mitarbeitern die Einarbeitung. Sollte ein Microfrontend veralten oder sich Anforderungen drastisch verändern, kann es zu geringen Kosten neu geschrieben werden, ohne die Entwicklung anderer Teams zu beeinträchtigen. Ebenso steigen auch die Erweiterbarkeit und Flexibilität der Gesamtlösung. Diese Modularität ermöglicht es, die Software an sich ändernde Geschäftsanforderungen anzupassen und neue Funktionen nahtlos zu integrieren.

Ein weiterer potenzieller Vorteil von Module Federation ist die freie Wahl und Kombination mehrerer JavaScript-Frameworks. Während man sich bei einem monolithischen Frontend in der Regel für ein JavaScript-Framework entscheiden muss, ermöglicht Module Federation die Kombination mehrerer Frameworks miteinander. Voraussetzung dafür ist lediglich eine kompatible Implementierung des Module-Federation-Standards, was durch ein einheitliches Build-Tooling (beispielsweise *Webpack*) leicht sichergestellt werden kann. Diese relative Technologieunabhängigkeit bietet das Potenzial für mehr Flexibilität in der Entwicklung und ermöglicht isolierte Experimente mit neuen Technologien.

Außerdem führt sie zu einer höheren Entwicklerzufriedenheit, da Teams die Möglichkeit haben, die eingesetzten Technologien selbst zu bestimmen. Darüber hinaus erweitert sich der verfügbare Talentpool, da Entwickler mit unterschiedlichen Framework-Kenntnissen und technischen Hintergründen flexibel in die Architektur integriert

JUG SAXONY DAY 2025

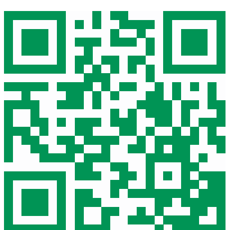
JETZT TICKETS SICHERN! #JSD2025

Sei dabei auf der beliebtesten IT-Community-Konferenz in Sachsen!

25. - 26. September 2025

Radisson Blu Park Hotel & Conference Center in Radebeul bei Dresden

Infos & Tickets: jugsaxony.day



werden können. Allerdings ist der Einsatz mehrerer JavaScript-Frameworks mit zusätzlichem technischem und organisatorischem Aufwand verbunden. Deshalb sollte immer eine gründliche Abwägung des jeweiligen Nutzens erfolgen.

Technische und organisatorische Herausforderungen

Microfrontendarchitekturen bringen nicht nur Vorteile mit sich. So muss darauf geachtet werden, dass technologische Wahlfreiheit nicht zur Entstehung eines Zoos konkurrierender Frontendtechnologien führt. *ThoughtWorks* hat dies treffend als „Micro frontend anarchy" bezeichnet *[4]*. Eine Einigung auf eine gemeinsame Basistechnologie ist hierbei noch wichtiger als bei Microservices.

Verwendet man zum Zusammenführen der Microfrontends Module Federation, so bindet man sich damit an ein Build-Tool. Angesichts der Geschwindigkeit, mit der sich JavaScript-Tools weiterentwickelt haben, sollte dieser Schritt gut überlegt sein. Ein Umstieg auf andere Build-Tools erfordert bei Module Federation einen koordinierten Aufwand in allen Microfrontends und stellt damit eine größere Herausforderung dar als gewöhnlich. Außerdem könnten sich durch gemeinsame Bibliotheken implizite Abhängigkeiten bilden, sodass ein Umstieg auf eine neuere Version nur koordiniert erfolgen kann.

Insbesondere vertikal geschnittene Microfrontends werden unterschiedliche Implementierung für ähnliche Anwendungsbestandteile enthalten. Im Enterprise-Umfeld sind dies häufig Tabellen und Formulare, deren Aussehen und Bedienung möglichst einheitlich sein sollte. Zu beachten sind beispielsweise die Reihenfolge von Buttons („Abbrechen" immer rechts oder immer links), größere Funktionalitäten wie speicherbare Tabellenfilter, automatisches Sichern von Formulareingaben und Validierungsverhalten. Sollen diese Gemeinsamkeiten nicht in Bibliotheken auslagert werden, und damit Abhängigkeiten zwischen den Microfrontends eingeführt werden, müssen Funktionen mehrfach implementiert werden. Führt ein Team neue Funktionalitäten oder ein neues Styling ein, sollten andere Teams darüber informiert werden, um ihre Oberflächen gegebenenfalls anzupassen. Man kann gemeinsam genutzte Funktionalität auch in die Shell-Anwendung auslagern. Damit mutiert die Shell-Anwendung aber mehr und mehr zu einer gemeinsam verwendeten Bibliothek und bringt mehr eigene Abhängigkeiten mit sich. Änderungen an der Shell-Anwendungen müssen aber zwingend mit den Microfrontend abgestimmt werden, um Laufzeitfehler zu vermeiden. Je mehr eigene Funktionalität die Shell-Anwendung mitbringt, desto größer wird der Abstimmungs- und Testaufwand.

Für die lokale Entwicklung wird die vollständige Shell-Anwendung oft durch eine eingeschränkte Version ersetzt. Dies führt zu abweichendem Verhalten im Vergleich zum Produktivbetrieb und erhöht den Aufwand für automatisierte sowie manuelle Integrationstests. Entwickler müssen ihre Microfrontends häufiger auf Testumgebungen deployen, um auch kleine Änderungen zu überprüfen, was die Entwicklungszyklen verlängert.

Werden gemeinsam genutzte Funktionalitäten wie zum Beispiel eine Benachrichtigungsfunktion in ein eigenes Microfrontend ausgelagert, erfordert dies Kommunikation zwischen den Microfrontends. Hier muss dann abgewogen werden, ob diese Kommunikation über die Backends oder über Custom Events im Browser erfolgen soll.

Umsetzungsempfehlungen

Es ist sinnvoll, sich von qualitativen Anforderungen an die Architektur leiten zu lassen, wenn Sie eine Microfrontend-Architektur einsetzen möchten. Sei es für ein neues Projekt oder zur schrittweisen Modernisierung einer bestehenden Benutzeroberfläche. Die Zielarchitektur sollte eine flexible, wartbare und verständliche Softwareentwicklung ermöglichen und gleichzeitig gut mit der Teamanzahl skalieren.

Eine der größten Effizienzsteigerungen ergibt sich aus der Schaffung von interdisziplinären, autonom agierenden Teams, die das Domänenwissen auf Entwicklerseite bündeln können. Ähnlich wie bei der Gestaltung von Microservice-Architekturen sollten Systemgrenzen anhand fachlicher Domänen definiert werden. Dafür müssen technische Stakeholder ein tiefes Verständnis der Fachlichkeit mitbringen. Fehlt dieses Verständnis, können Workshop-Methoden wie beispielsweise EventStorming *[5]* oder Domain Storytelling [6s] aus dem Domain-Driven Design eingesetzt werden.

Außerdem müssen Technologieentscheidungen abgewogen und getroffen werden. Unterschätzen Sie dabei bitte nicht die Wichtigkeit, diese zu dokumentieren und zu begründen. Entschieden werden muss unter anderem Folgendes:

- Legen Sie sich auf ein oder mehrere JavaScript-Frameworks fest oder ermöglichen Sie Ihren Teams vollständige Wahlfreiheit?
- Wie stellen Sie eine einheitliche Benutzererfahrung sicher? Stellen Sie Komponenten oder Stylesheets zur Verfügung, oder legen Sie fest, welche existierende Komponenten-Bibliothek von allen Teams genutzt wird?
- Wie gehen Sie mit Querschnittsbelangen um? Muss jedes Team diese selbständig lösen, stellen Sie Kopiervorlagen bereit oder entwickeln Sie zentrale Bibliotheken?

Die Frage, inwieweit Sie die technologische Vielfalt einschränken wollen, lässt sich nicht pauschal beantworten. Technische Freiheit erhöht den Integrationsaufwand, bietet jedoch schwer quantifizierbare Vorteile wie Mitarbeiterzufriedenheit und einen größeren Talentpool. Zu Beginn ist es jedoch sinnvoll, sich auf ein JavaScript-Framework zu beschränken, um die Komplexität beim Erstellen der Plattform zu verringern.

Nachdem die technischen und fachlichen Rahmenbedingungen festgelegt wurden, kann entschieden werden, wie zentrale Komponenten und Querschnittsbelange definiert und bereitgestellt werden sollen.

Aufgrund ihrer Vielzahl empfiehlt sich die Gründung eines Plattform- oder Integrationsteams. Dieses Team ist verantwortlich für die Schaffung einer nahtlos integrierten zentralen Plattform und für Querschnittsaufgaben wie eine einheitliche Benutzererfahrung durch UI-/UX-Designs, Styleguides oder gemeinsame Komponenten. Zudem stellt das Team vorkonfigurierte Werkzeuge wie Bundler, Transpiler (zum Beispiel *Webpack*) und CI-Pipelines bereit. Es integriert und verbessert die Microfrontends und -backends iterativ durch gemeinsame Bibliotheken und umfassende Dokumentation, sowie Lösungen für Authentifizierung, Autorisierung, resiliente Schnittstellenkommunikation und Frontend-Message-Passing. Bei Bedarf ermöglicht es auch, websocket-basierte Integ-

rationen zwischen Front- und Backends über *STOMP* und *RabbitMQ* zu verwenden.

Das Plattformteam sollte die Bereitstellung neuer Microservices und Microfrontends für andere Teams vereinfachen, beispielsweise durch automatisierte Deployments in *Kubernetes* mit *GitOps* und *Helm Charts*. Zudem sollten Applikationslogs, -metriken und -traces zentral über Tools wie *Elasticsearch* und *Grafana* aggregiert sowie anwendungsspezifische Alerts ermöglicht werden. Diese Aufgaben und deren Lösungen müssen dokumentiert und kontinuierlich verbessert werden, um die Autonomie der Entwicklungsteams zu gewährleisten und die selbständige Integration ihrer Software in die Plattform zu unterstützen.

Ein Plattformteam trägt somit nicht nur die Verantwortung für die Bereitstellung, sondern auch für eine nahtlose Integration mit der Plattform sowie deren Dokumentation. Darüber hinaus unterstützt es die anderen Teams bei technischen Problemen, kommuniziert proaktiv Änderungen und Updates und dient als Anlaufstelle für Feedback und Bedenken.

Auf technischer Ebene ist es empfehlenswert, die gemeinsame Frontend-Shell möglichst schlank zu halten und nicht zu viele Querschnittsaufgaben zu übernehmen. Die Shell-Anwendung sollte lediglich eine einfache Menüführung bieten. Andere funktionale Querschnittsanforderungen (zum Beispiel Benachrichtigungen), können in eigenen Microfrontends und -services implementiert werden. Dadurch erhöht sich die Flexibilität und Abhängigkeiten der Microfrontends auf die Shell-Anwendung werden vermieden, was komplizierte und koordinationsbedürftige Deployments verhindert.

Fazit

Microfrontend-Architekturen übertragen die Vorteile von Microservices auf das Frontend, indem sie die Benutzeroberfläche in unabhängige, leicht integrierbare Module aufteilen. Dies ermöglicht autonomen Teams schnellere Entwicklungszyklen und verbesserte Wartbarkeit. Module Federation erleichtert dabei besonders die Integration verschiedener Technologien. Eine erfolgreiche Umsetzung erfordert sorgfältige Technologieentscheidungen und ein spezialisiertes Plattform-Team, das für eine konsistente Benutzererfahrung und effiziente Integration sorgt. Insgesamt fördern Microfrontends skalierbare, wartbare und dynamische Frontend-Architekturen, die den Anforderungen moderner Softwareentwicklung gerecht werden.

Quellen

[1] *https://en.wikipedia.org/wiki/Server_Side_Includes*
[2] *https://module-federation.io/*
[3] Webpack Team, 2020: *https://webpack.js.org/blog/2020-10-10-webpack-5-release/#module-federation*
[4] ThoughtWorks, 2020: *https://www.thoughtworks.com/de-de/radar/techniques/micro-frontend-anarchy*
[5] DDD-Crew: Eventstorming Glossary *https://github.com/ddd-crew/eventstorming-glossary-cheat-sheet*
[6] Domain Story Telling *https://domainstorytelling.org/*

Frederik Pietzko
iits-consulting
frederik.pietzko@iits-consulting.de

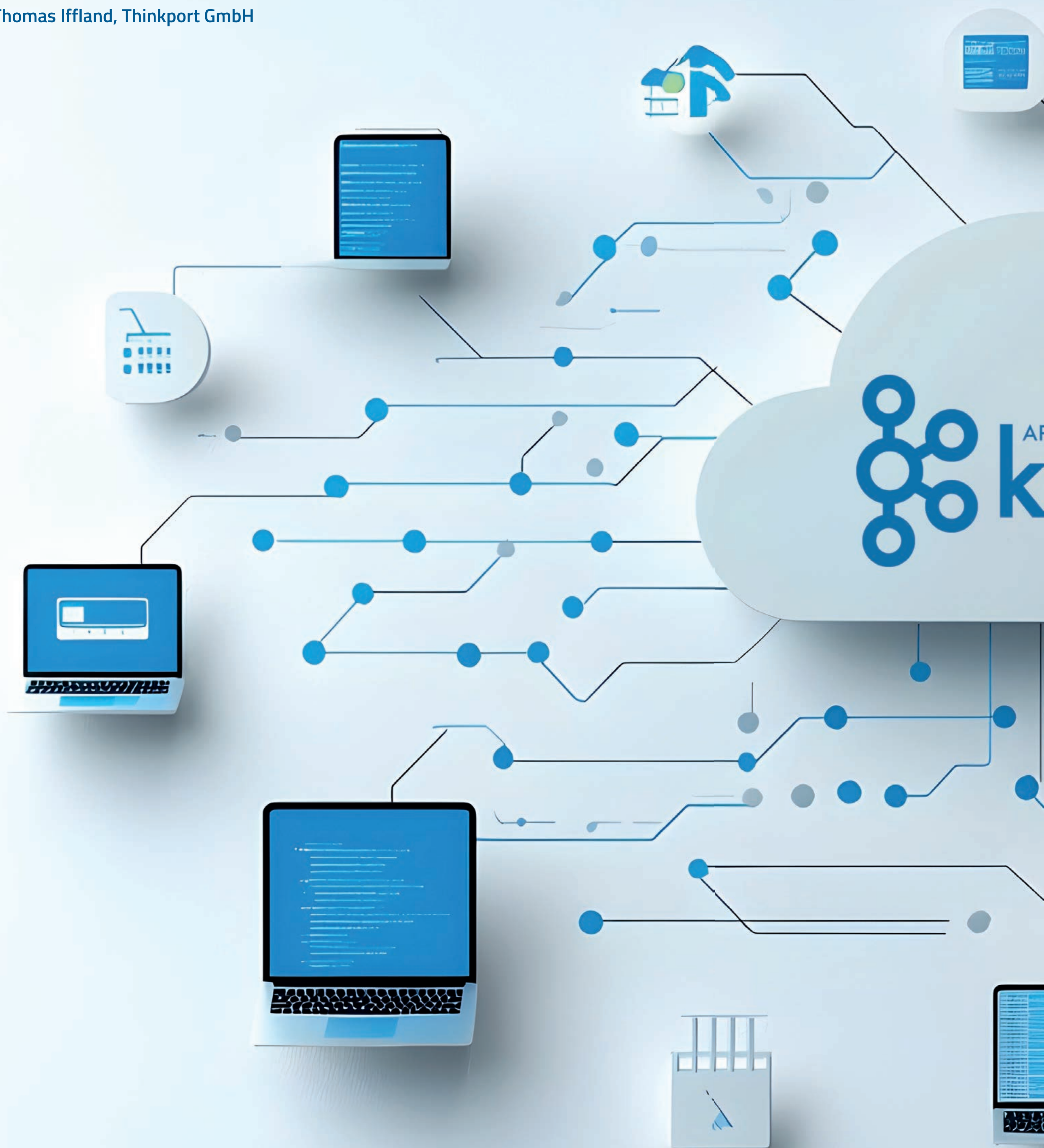
Frederik ist ein Full-Stack-Entwickler mit sieben Jahren Berufserfahrung in Java und Kotlin. Er entwickelt Cloud-native Anwendungen für Kunden in den Bereichen Automotive, Logistik und Energie. Zudem verfügt er über umfangreiche Erfahrungen mit Frontend-Frameworks wie React und der Entwicklung von Microfrontends.

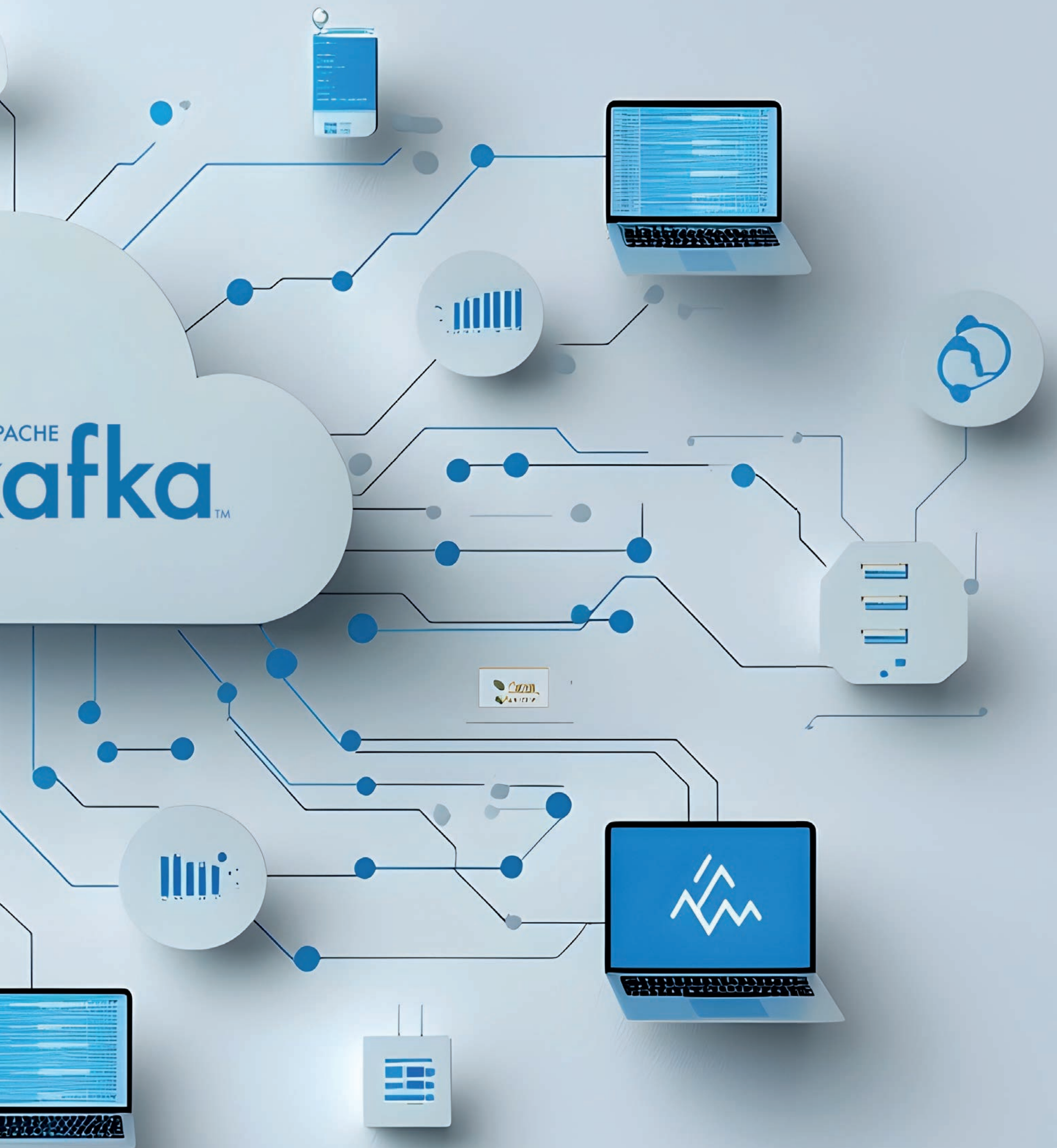
Frank Steidinger
iits-consulting
frank.steidinger@iits-consulting.de

Frank ist Full-Stack-Entwickler bei der iits-consulting. Mit der Web-Entwicklung hat er vor über 25 Jahren angefangen und ist ihr über die Jahre treu geblieben. Dabei hat er server- und client-seitige Frameworks in Java, Kotlin und JavaScript nicht nur verwendet, sondern auch mitentwickelt.

Apache Kafka intern: Wie man das Kafka-Wire-Protokoll spricht

Thomas Iffland, Thinkport GmbH




PACHE
afka

Apache Kafka ist eine Technologie, die in vielen Organisationen nicht mehr wegzudenken ist. Trotzdem gilt sie oftmals immer noch als große Blackbox. In diesem Artikel werden die Interna beleuchtet und gezeigt, wie die Kommunikation zwischen Broker und Clients auf Byteebene passiert.

Kafka – eine kleine Übersicht

Apache Kafka ist eine Open-Source-Plattform, die für das Verarbeiten, Speichern und Übertragen von Event-Streaming-Daten entwickelt wurde. Es wird oft verwendet, um große Mengen an Daten in Echtzeit zu verarbeiten und ist besonders geeignet für Anwendungsfälle wie Benutzeraktivitäten, Log-Daten oder Sensormessungen. Kafka organisiert Daten in sogenannten Topics, in denen Nachrichten, aber auch Events von Producern geschrieben und von Consumern gelesen werden können.

Jedes Topic in Kafka ist in Partitionen unterteilt. Dieses Partitionierungskonzept ermöglicht es Kafka, Daten parallel zu verarbeiten. Innerhalb einer Partition bleibt die Reihenfolge der Nachrichten erhalten. Nachrichten, die denselben Schlüssel (Key) haben, werden immer in der gleichen Reihenfolge verarbeitet, um Konsistenz in der Reihenfolge zu garantieren.

Kafka ist ein verteiltes System, das aus mehreren Brokern besteht, die Nachrichten speichern und verwalten. Mehrere Broker werden zu einem Cluster zusammengeschaltet. Produzenten senden Daten an Topics und Consumer lesen diese Daten entweder individuell oder als Teil einer Consumer Group. Dies ermöglicht eine Parallelisierung der Verarbeitung. Um Datenverlust zu verhindern, repliziert Kafka Events über mehrere Broker hinweg. Dadurch bleibt das System auch bei Ausfällen einzelner Broker funktionsfähig.

Übersicht über das Kafka-Wire-Protokoll

Das Kafka-Wire-Protokoll bildet die Grundlage für die Kommunikation zwischen Brokern, Producern und Consumern. Es definiert die Art und Weise der Kommunikation und das Format der einzelnen Nachrichten.

Apache Kafka verwendet ein schlankes, binäres Protokoll über TCP, bei dem Anfragen und Antworten als Paar übertragen werden. Ein Handshake beim Verbindungsaufbau entfällt, wodurch Verbindungen schnell und effizient hergestellt werden können. Um die Kosten des TCP-Handshakes zu minimieren, ist es jedoch sinnvoll, Verbindungen möglichst lange offen zu halten. Da die Partitionen in Kafka in der Regel auf mehrere Broker verteilt sind, braucht ein Client oft Verbindungen zu verschiedenen Brokern, aber in der Regel nur eine Verbindung pro Broker. Kafka garantiert, dass Anfragen in der Reihenfolge bearbeitet werden, in der sie gesendet wurden, und die Antworten entsprechend zurückkommen *[1]*.

Verwendete Datentypen

Das Protokoll unterstützt in der aktuellen Version 21 verschiedene Datentypen. Neben primitiven Datentypen werden auch komplexere Datenstrukturen wie Records oder Arrays unterstützt. Eine Auswahl der gängigen Datentypen sieht man hier:

BOOLEAN	Boolscher Wert 0 oder 1. Alle Werte ungleich 0 werden als „true“ interpretiert.
INT32	Vorzeichenbehafteter Integer-Wert zwischen -231 und 231-1.
VARINT	Stellt eine Ganzzahl zwischen -2^{31} und $2^{31}-1$ dar. Die Kodierung erfolgt nach dem Zig-Zag-Encoding von Google Protocol Buffers [2].
STRING	Zeichensequenz. Zuerst wird die Länge n als INT16 angegeben. Dann folgen n Bytes als Payload im UTF-8 Format.
COMPACTED_STRING	Wie der String, jedoch repräsentiert der Wert -1 in der Payload, dass der Wert null ist.
BYTES	Enthält eine Sequenz von Bytes. Dabei wird die Länge N zuerst übergeben. Im Anschluss folgen N Bytes.
RECORDS	Repräsentiert ein Batch von Kafka-Nachrichten als NULLABLE_BYTES.
ARRAY	Ein Array beinhaltet eine Sammlung von Objekten. Zuerst wird die Anzahl N als INT32 übergeben. Danach folgen N Instanzen des Typs T.
COMPACTED_ARRAY	Im Gegensatz zum normalen Array, wird zuerst die Anzahl N+1 als VARINT geschrieben. Anschließend folgen N Instanzen des Typs T.

Der Header

Ein wichtiger Bestandteil der Kommunikation, besonders des Requests, sind die Header. Darin sind essenzielle Informationen enthalten, die die Kommunikation und korrekte Verarbeitung des Brokers überhaupt erst ermöglichen.

In der Dokumentation sind die Formate der Nachrichten jeweils in der Backus-Naur-Form (BNF) angegeben. Diese formale Sprache dient dazu, Grammatiken zu beschreiben, wie beispielsweise die Syntax von Programmiersprachen *[3]*.

In diesem Format sieht der Request-Header in der aktuellen Version wie in *Listing 1* gezeigt aus.

```
Request Header v2 => request_api_key request_api_version correlation_id client_id TAG_BUFFER
  request_api_key => INT16
  request_api_version => INT16
  correlation_id => INT32
  client_id => NULLABLE_STRING
```

Listing 1: Der Request-Header (V2) in BNF-Schreibweise

```
private static byte[] createHeader(short apiKey, short apiVersion, int correlationId, String clientId) {
     ByteBuffer headerBuffer = ByteBuffer.allocate(2 + 2 + 4 + 2 + clientId.length() + 1);
     headerBuffer.putShort(apiKey);
     headerBuffer.putShort(apiVersion);
     headerBuffer.putInt(correlationId);
  headerBuffer.putShort((short) clientId.length());
     headerBuffer.put(clientId.getBytes());
     writeUnsignedVarint(0, headerBuffer);
  return headerBuffer.array();
}
```

Listing 2: Die createHeader-Methode

Erklärung der Bestandteile:

- `request_api_key` (`INT16`): Gibt die API-Operation an, die durch die Anfrage ausgelöst wird, beispielsweise „Produce" oder „Fetch".
- `request_api_version` (`INT16`): Spezifiziert die Version der API, die verwendet wird. Dies ermöglicht Abwärtskompatibilität und neue Funktionalitäten.
- `correlation_id` (`INT32`): Ein eindeutiger Identifikator für die Anfrage, der es dem Client ermöglicht, Antworten mit entsprechenden Anfragen zu verknüpfen.
- `client_id` (`NULLABLE_STRING`): Ein optionaler String, der den Client identifiziert, der die Anfrage sendet.
- `TAG_BUFFER` (`COMPACT_ARRAY`): Ein zusätzlicher, optionaler Speicherbereich für weitere Metainformationen. Die Länge 0 als VARINT übergeben repräsentiert, dass keine weiteren Informationen folgen.

Der Header eines `CreateTopics`-Requests, der dazu dient, ein neues Topic anzulegen, könnte dann so aussehen. Zu beachten ist, dass Big-Endian als Byte-Reihenfolge gewählt wird – was heißt, dass alles vom höchstwertigen Byte zum niedrigstwertigen Byte *[4]* angeordnet ist.

Feldname	Dezimal-/UTF-8-Wert	Byte-Darstellung
request_api_key	19	00 19
request_api_version	17	00 07
correlation_id	1	00 00 00 01
client_id	my-client	99, 108, 105, 101, 110, 116, 45, 105, 100
TAG_BUFFER	[]	00

Wenn wir den Header als Byte-Array darstellen, ergibt sich der folgende Wert:
[0, 19, 0, 7, 0, 0, 0, 1, 0, 9, 99, 108, 105, 101, 110, 116, 45, 105, 100, 0]

Um dieses Byte-Array in Java zu erstellen, kann die in *Listing 2* gezeigte Methode genutzt werden.

Nach der Initialisierung des `ByteBuffer`-Objekts mit der entsprechenden Größe, werden die Werte hinzugefügt. Die Implementierung der `writeUnsignedVarint` wird als gegeben hingenommen, kann aber beispielsweise im Kafka-Repository nachgeschlagen werden *[5]*.

Die Payload

Nachdem der Header zusammengebaut wurde, können die Payload-Daten folgen. Diese enthalten die zur Verarbeitung des Requests notwendigen Informationen. In diesem Artikel soll der `CreateTopics`-Request näher beleuchtet werden. Er bietet eine mittlere Komplexität und nutzt die üblichen Datenstrukturen. Die Grammatik in BNF sieht wie in *Listing 3* dargestellt aus.

Erklärung der Bestandteile auf der obersten Ebene:
`topics` (`COMPACT_ARRAY`): Das Topics-Array enthält die Informationen über die zu erstellenden Topics. Die genauere Erläuterung der Felder folgt unten.

- `timeout_ms` (`INT32`): Die Zeit in Millisekunden, die gewartet wird, bevor der Request mit einem Timeout abbricht.
- `validate_only` (`BOOLEAN`): Überprüft nur, ob das Topic angelegt werden könnte, ohne dies jedoch wirklich zu tun.
- `TAG_BUFFER` (`COMPACT_ARRAY`): Ein zusätzlicher, optionaler Speicherbereich für weitere Metainformationen. Die Länge 0 wird als übergebener `VARINT` repräsentiert, sodass keine weiteren Informationen folgen.

Erklärung der Bestandteile des Topics:

- `name` (`COMPACT_STRING`): Der Name des anzulegenden Topics.
- `num_partitions` (`INT_32`): Die Anzahl der Partitionen, die im Topic erstellt werden sollen, oder -1, wenn entweder eine ma-

```
CreateTopics Request (Version: 7) => [topics] timeout_ms validate_only TAG_BUFFER
  topics => name num_partitions replication_factor [assignments] [configs] TAG_BUFFER
     name => COMPACT_STRING
     num_partitions => INT32
     replication_factor => INT16
     assignments => partition_index [broker_ids] TAG_BUFFER
        partition_index => INT32
        broker_ids => INT32
     configs => name value TAG_BUFFER
        name => COMPACT_STRING
        value => COMPACT_NULLABLE_STRING
  timeout_ms => INT32
  validate_only => BOOLEAN
```

Listing 1: Der CreateTopics-Request (V7) in BNF-Schreibweise

nuelle Partitionierung zugewiesen wird oder die Standardanzahl an Partitionen verwendet wird.

- `replication_factor` (`INT_16`): Die Anzahl der Replikate, die für jede Partition im Topic erstellt werden sollen, oder -1, wenn eine manuelle Partitionierung zugewiesen wird oder der Standard-Replikationsfaktor genutzt wird.
- `assignments` (`COMPACT_ARRAY`): Eine Sammlung von manuellen Partitionszuordnungen. Dabei wird ein Mapping Partitionsindex (`INT32`) zu einem oder mehreren Brokern über die Broker-Ids (Array vom Typ `INT32`) durchgeführt. Außerdem können auch hier zusätzliche Metadaten als `TAG_BUFFER` übergeben werden.
- `configs` (`COMPACT_ARRAY`): Zusätzliche Konfigurationsparameter auf Topic-Ebene. Dabei handelt es sich um eine Sammlung von `name` (`COMPACT_STRING`) und `value` (`COMPACT_NULLABLE_STRING`) Paaren. Anschließend folgt auch hier ein TAG_BUFFER.
- `TAG_BUFFER` (`COMPACT_ARRAY`): Ein zusätzlicher, optionaler Speicherbereich für weitere Metainformationen. Die Länge 0 wird als übergebener VARINT repräsentiert, sodass keine weiteren Informationen folgen.

Der Java-Code, um einen solchen `CreateTopics`-Request zusammenzubauen, ist in *Listing 4* zu sehen.

In *Listing 4* sehen wir zwei Neuerungen. Zum einen wird die `writeCompactString`-Methode genutzt. Diese schreibt zuerst die Länge des Strings als VARINT und anschließend den Wert selbst. Zum anderen wird mit Hilfe der `trimByteBuffer`-Methode der erstellte `ByteBuffer` auf die tatsächliche Größe gekürzt. Das ermöglicht es, bei der Allokation etwas großzügiger zu sein (in diesem Beispiel 512 Bytes) ohne die genaue Größe im Vorhinein zu kennen. Ansonsten werden hier keine Assignments oder Configs übergeben. Zusätzlich sind die `TAG_BUFFER` null.

Die Response

Um Informationen und mögliche Fehler zu übertragen, werden über denselben Kommunikationskanal auch Daten vom Broker an den Client zurückgesendet. Diese Response besteht ebenfalls aus der Angabe der Größe und einem Header. Die eigentliche Payload sieht man in *Listing 5*.

Auf eine vollständige Erklärung aller Felder verzichten wir an dieser Stelle. Einige seien jedoch hervorzuheben. Besonders `error_code` und `error_message` sind Felder, die man auch in den anderen Response-Typen der Protokollbeschreibung findet. Diese Felder sind wichtig, wenn es zu einem Fehlerfall kommen sollte. Beispielsweise wird beim Versuch der Erstellung eines bereits existierenden Topics, die Fehlerkonstante 36 – `TOPIC_ALREADY_EXISTS` mit der näheren Fehlerbeschreibung `Topic 'topic1' already exists.` zurückgeliefert. Eine genaue Übersicht aller Fehlercodes ist auf der Kafka-Website *[6]* zu finden.

```
private static byte[] createPayloadCreateTopic() throws IOException {
     String topicName = "topic1";
     int numPartitions = 1;
     short replicationFactor = 1;
     int timeoutMs = 10000;
     byte validateOnly = 0; // false
     int numberOfTopics = 1;
     int numberOfAssignments = 0;
     int numberOfConfigs = 0;
     int emptyTaggesFields = 0;
     ByteBuffer payloadBuffer = ByteBuffer.allocate(512);
     writeUnsignedVarint(numberOfTopics + 1, payloadBuffer);
     writeCompactString(topicName, payloadBuffer);
     payloadBuffer.putInt(numPartitions);
     payloadBuffer.putShort(replicationFactor);
     writeUnsignedVarint(numberOfAssignments + 1, payloadBuffer);
     writeUnsignedVarint(numberOfConfigs + 1, payloadBuffer);
     writeUnsignedVarint(emptyTaggesFields, payloadBuffer);
     payloadBuffer.putInt(timeoutMs);
     payloadBuffer.put(validateOnly);
     writeUnsignedVarint(emptyTaggesFields, payloadBuffer);
     return trimByteBuffer(payloadBuffer).array();
}
```

Listing 4: Die Methode zur Erstellung der Payload des Create-Topics-Request

```
CreateTopics Response (Version: 7) => throttle_time_ms [topics] TAG_BUFFER
  throttle_time_ms => INT32
  topics => name topic_id error_code error_message num_partitions replication_factor [configs] TAG_BUFFER
     name => COMPACT_STRING
     topic_id => UUID
     error_code => INT16
     error_message => COMPACT_NULLABLE_STRING
     num_partitions => INT32
     replication_factor => INT16
     configs => name value read_only config_source is_sensitive TAG_BUFFER
        name => COMPACT_STRING
        value => COMPACT_NULLABLE_STRING
        read_only => BOOLEAN
        config_source => INT8
        is_sensitive => BOOLEAN
```

Listing 5: Das Format der Payload der Create-Topics-Response

```
private static void parseCreateTopicsResponse(InputStream inputStream) throws Exception {
            int responseSize = ByteBuffer.wrap(inputStream.readNBytes(4)).getInt();
            byte[] responseBody = inputStream.readNBytes(responseSize);
            ByteBuffer buffer = ByteBuffer.wrap(responseBody);


            int throttleTimeMs = buffer.getInt();
            System.out.println("throttleTimeMs: " + throttleTimeMs);


            int topicsCount = readVarInt(buffer) - 1;
            System.out.println("Number of Topics: " + topicsCount);

            for (int i = 0; i < topicsCount; i++) {
                  String name = readCompactString(buffer);
                  System.out.println("Topic Name: " + name);


                  short errorCode = buffer.getShort();
                  System.out.println("Error Code: " + errorCode);

                  String errorMessage = readCompactNullableString(buffer);
                  System.out.println("Error Message: " + (errorMessage == null ? "null" : errorMessage));


                  for (int j = 0; j < configsCount; j++) {
                        ...
                  }

            }
      }
}
```

Listing 6: Die Methode zum Lesen und Printen der Antwort

Der Code, um die Response zu verarbeiten und die gelesenen Informationen auf der Konsole auszugeben, ist in *Listing 6* zu finden. Das Beispiel wurde gekürzt.

Fazit

Kafka ist aus modernen verteilten Systemen kaum noch wegzudenken, doch die zugrundeliegende Kommunikation bleibt oft eine Blackbox. Dieser Artikel hat einen Blick hinter die Kulissen geworfen und gezeigt, wie die Interaktion zwischen Clients und Brokern auf Byte-Ebene funktioniert.

Besonders spannend ist, wie schlank und effizient das Kafka-Wire-Protokoll aufgebaut ist. Die Requests und Responses sind so optimiert, dass sie mit minimalem Overhead auskommen, was zur hohen Performance von Kafka beiträgt. Der detaillierte Blick auf den `CreateTopics`-Request hat verdeutlicht, wie Clients direkt mit Kafka sprechen können – ohne auf High-Level-Bibliotheken angewiesen zu sein.

Wer tiefer in Kafka einsteigen möchte, kann sich die offizielle Protokolldokumentation anschauen oder mit eigenen Tests experimentieren. Ein Verständnis des Wire-Protokolls hilft nicht nur beim Debuggen, sondern eröffnet auch neue Möglichkeiten zur Optimierung und individuellen Anpassung.

Der gesamte Sourcecode ist in GitHub zu sehen *[7]*.

Quellen

[1] Apache Software Foundation: Kafka Protocol Guide (URL: *https://kafka.apache.org/protocol*)

[2] *https://protobuf.dev/programming-guides/encoding/*

[3] Wikipedia: Backus-Naur-Form (URL: *https://de.wikipedia.org/wiki/Backus-Naur-Form*)

[4] Wikipedia: Byte-Reihenfolge (URL: *https://de.wikipedia.org/wiki/Byte-Reihenfolge*)

[5] Apache Software Foundation: Kafka Github-Repository (URL: *https://github.com/a0x8o/kafka/blob/54eff6af115ee647f60129f2ce6a044cb17215d0/clients/src/main/java/org/apache/kafka/common/utils/ByteUtils.java#L294*)

[6] Apache Software Foundation: Kafka Protocol Guide – Error Codes (URL: *https://kafka.apache.org/protocol#protocol_error_codes*)

[7] Thomas Iffland: Sourcecode zum Artikel (URL: *https://github.com/tiffland/kafka-wire-protocol*)

Thomas Iffland
Thinkport GmbH
tiffland@thinkport.digital

Thomas Iffland ist Cloud Architect bei der Thinkport GmbH. Neben dem Projektgeschäft mit dem Schwerpunkt auf Event-basierten Technologien wie Apache Kafka, hält er gerne Vorträge auf Konferenzen und Meetups und organisiert auch das Apache-Kafka-Meetup in Frankfurt am Main.

Angular – The Silent Revolution

Stephan Rauh, Alexander Pahn, Julian Schmidt

Angular erfindet sich gerade von Grund auf neu. Das ist eine bemerkenswerte Entwicklung für ein Framework, das auf die harte Tour lernen musste, wie wichtig Kompatibilität ist. Und es scheint Wirkung zu zeigen. Zum ersten Mal seit Jahren steigt die Beliebtheit von Angular in den State-of-JS-Umfragen [8]. Wir zeigen euch in dieser zweiteiligen Serie die wichtigsten Änderungen, erzählen euch etwas über die Hintergründe und darüber, welches Potenzial wir uns von den Verbesserungen erhoffen.

Vielleicht erinnert ihr euch noch an jenen denkwürdigen Moment im Jahr 2014, als das *Angular*-Team auf der ng-conf seine Pläne für *Angular* 2 bekanntgegeben hat. Die Ideen waren gut, das hat die Entwicklung von *Angular* im letzten Jahrzehnt deutlich bewiesen. Dummerweise lief die Kommunikation unglücklich. Der kleine Satz „Wir haben uns noch keinen Migrationspfad überlegt" sorgte in zahlreichen Managementetagen für Panik[1] .

Das ist auch heute noch spürbar. Viele Architekten haben damals die Konsequenz gezogen, *Angular* zu vermeiden – mit der Begründung, dass sie nicht wissen könnten, ob die Quelltexte auch nach dem nächsten Update noch liefen.

Die Realität sieht anders aus. Das *Angular*-Team hat aus der Kommunikationspanne gelernt. Sie legen großen Wert auf Rückwärtskompatibilität, und wo diese nicht machbar ist, beschreiben sie einen klaren Migrationspfad. In den meisten Fällen stellen sie sogar ein Migrationstool bereit. Das Update von einer Version zur nächsten ist normalerweise kein Problem. Selbst ein Update über mehrere Versionen ist möglich. Dafür stellt das *Angular*-Team ein Online-Tool zur Verfügung, das euch basierend auf eurer Start- und Zielversion die einzelnen Schritte auflistet und erklärt. Dabei können sogar weitere Parameter, wie die Komplexität der App, berücksichtigt werden. Problematisch sind in der Regel nur die übrigen Bibliotheken im Projekt[2].

Dennoch wagt das *Angular*-Team gerade einen ähnlichen Schritt wie vor zehn Jahren. In den letzten Jahren war die Build-Infrastruktur im Fokus, aktuell verändert sich das Programmiermodell deutlich, und sogar die Test-Frameworks Jasmine und Karma werden demnächst durch *Jest*, *Web Test Runner* oder *Vitest* ersetzt *[7]*.

Was bedeutet das für mich?

Viele dieser Änderungen passieren im Hintergrund, ohne dass ihr euch darum kümmern müsst. War es in den Anfangsjahren von *Angular* noch üblich, sich eine eigene Build-Tool-Chain aufzu-

---

[1] Wir würden euch gerne einen Quellenverweis dafür geben. Leider sind die entsprechenden Artikel und YouTube-Videos inzwischen aus dem Netz verschwunden. Die (verwaisten) Links könnt ihr immer noch leicht recherchieren, nur zeigen sie ins Leere. Ein Echo der großen Verblüffung damals könnt ihr auf Stephan Rauhs Blog finden – inklusive der toten Links *[9]*.

bauen, hat sich in den letzten Jahren die *Angular* CLI als Standard durchgesetzt[3]. Das wiederum ermöglichte dem *Angular*-Team, weitgehend geräuschlos von *Webpack* auf *esbuild* und *Vite* umzustellen. Ihr merkt das hauptsächlich daran, dass das Kompilieren heute viel schneller geht als früher. Bei kleinen und mittleren Projekten wie unserem Open-Source-Projekt *ngx-extended-pdf-viewer* sind die Zeiten von rund 30 Sekunden auf unter 5 Sekunden gesunken.

Ein anderes Highlight war die Einführung des Ivy-Compilers. Zugegeben, das ist schon einige Zeit her – im Februar 2020, als *Angular* 9 herauskam, dachten wir bei „Corona" noch an Bier. Es war eine beeindruckende technische Leistung, die überraschend geräuschlos über die Bühne ging.

Von der Ivy-Einführung habt ihr vermutlich weniger mitbekommen als ich (Stephan). Eure Anwendung lief vorher und sie lief auch nach der Umstellung auf Ivy. Ihr konntet den Unterschied allenfalls sehen, wenn ihr den generierten JavaScript-Code debuggt habt. Für mich als Bibliotheksentwickler war die Umstellung schmerzhafter. Ich musste eine Zeit lang beide Varianten unterstützen und mit rätselhaften Fehlermeldungen, in denen der griechische Buchstabe Theta ( ) vorkam. Den Kampf habe ich gerne aufgenommen: der Ivy-Compiler lieferte einfach viel bessere Ergebnisse.

Heutzutage fokussieren sich die Neuerungen auf das Programmiermodell und das hat Auswirkungen auf euch. Es sei denn, ihr entscheidet euch, die Änderungen zu ignorieren. Das ist – zumindest im Moment – eine valide Option. Doch schaut euch die Neuerungen erst einmal an. Ihr werdet sie lieben!

Bye, bye, Modules – Hello Standalone Components

Angular schafft die Module ab! Auf den ersten Blick ergibt das überhaupt keinen Sinn. Entwicklerinnen und Entwickler lieben Module. Modularisierung ist in der IT Trumpf. Die Idee, große Aufgaben in kleinere Module aufzuteilen, die wiederum aus kleineren Paketen von Algorithmen bestehen, hat sich einfach bewährt.

Es sei denn, es handelt sich um *Angular*-Module. Mit diesem Konzept fremdeln viele Leute. Das Problem ist weniger die Modularisierung als solche, sondern dass ihr in *Angular* die Module manuell und aufwändig konfigurieren müsst. Vereinfacht ausgedrückt, ist ein Modul eine Auflistung von Komponenten und Services, die ihr verwenden wollt. Komponenten, die in keinem Modul stehen, könnt ihr nicht verwenden. Die Dependency-Injection von *Angular* findet die Komponente einfach nicht. Das fühlt sich in den meisten Fällen nach unnötiger Schreibarbeit an. Java-Entwicklerinnen und -Entwickler haben es besser: Sie müssen sich nicht damit abplagen.

Wenn ihr im Zusammenhang mit *TypeScript* und *JavaScript* nach Modulen sucht, entdeckt Ihr ein zweites Problem. Das Wort „Modul" wird in der JavaScript-Welt bereits für etwas anderes benutzt. Grob vereinfacht gesagt, ist ein Modul eine JavaScript-Datei, in der das Schlüsselwort „export" vorkommt. Das doppelt verwendete Wort führt erstaunlich selten zu Problemen – ist aber eine potenzielle Quelle von Missverständnissen. Das seht ihr zum Beispiel dann, wenn ihr eine TypeScript-Datei importiert, in der das Wort „export" nicht vorkommt. Die Fehlermeldung von TypeScript lautet dann „File 'xyz.ts' is not a module.ts (2306)" – und dürfte alle *Angular*-Entwicklerinnen und -Entwickler verwirren, die keinen Zusammenhang zu NgModule sehen.

Seit *Angular* 14 können wir Standalone-Komponenten definieren und seit *Angular* 19 sind sie der Default. Diese Komponenten gehören zu keinem Modul mehr. Das bedeutet auch, dass sie nicht mehr einfach so andere Komponenten, Services, Pipe und Direktiven verwenden können. Jede Abhängigkeit zu anderen Komponenten muss explizit im `@Component`-Decorator definiert werden.

Ist das ein Nachteil oder ein Vorteil?

Als langjähriger *Angular*-Trainer würde ich (Stephan) sagen, dass das ein Vorteil ist. Ich hatte immer große Probleme, meinen Trainees zu erklären, was ein Modul ist. Wenn es schwierig wird, ein Konzept zu erklären, weist das klar darauf hin, dass das Konzept Schwächen hat.

Andererseits bedeutet der Verzicht auf Module auch einen Verzicht auf eine Strukturierungsmöglichkeit. Und es bedeutet, dass jetzt jede Komponente eine Vielzahl von anderen Dingen importieren muss. Der `@Component()`-Decorator wird mit Standalone-Komponenten raumgreifender.

Vielfach wird die große Anzahl an Imports kritisiert, da sie die Lesbarkeit reduziert. Das lässt sich allerdings in weiten Teilen durch eine bessere Aufteilung in Komponenten beheben. In allen anderen Fällen hilft auch hier eine gute IDE für eine bessere Darstellung. Möglicherweise ist die Diskussion aber auch müßig: In den kommenden *Angular*-Versionen sollen die Imports aus dem Decorator entfernt werden. Minko Gechev schreibt *[7]*, dass sich das *Angular*-Team des Problems bewusst sei, und dass sie das Problem des „doppelten Imports" im Rahmen des RFCs für „selectorless Components" lösen wollen. Wie das konkret aussehen wird, wissen wir noch nicht, aber hier zeichnet sich eine weitere spannende Änderung ab.

Auf technischer Ebene hat es auch Vorteile, alle Komponenten, Direktiven und anderes einzeln zu importieren. Ein Modul zu importieren, bedeutet eben, das gesamte Modul zu importieren. Auch die Teile, die ihr gar nicht verwendet. Das Tree-Shaking von *Angular* ist ziemlich gut darin, überflüssige Programmteile wieder hinauszuwerfen – aber noch besser ist es, sie gar nicht erst zu importieren.

[2] „Nur" ist vielleicht untertrieben: Das ist eine sehr große Hürde. Viele Projekte mussten leidvoll erleben, auf die falsche Library gesetzt zu haben. Egal, wie innovativ eure favorisierte Library ist, das nützt euch wenig, wenn sie mit der nächsten Version von Angular nicht kompatibel ist! Das gilt natürlich auch für die Mitbewerber von Angular – wobei Angular hier sogar punkten kann, da es einen großen Teil der benötigten Infrastruktur bereits mitbringt. Die oft kritisierte Bevormundung („being opinionated") hat auch ihre Vorteile.

[3] Zumindest weitgehend. Es gibt noch zahlreiche Projekte, die ganz oder teilweise auf eine eigene Build-Pipeline setzen. Man denke nur an ngx-build-plus, das euch erlaubt, die Webpack-Konfiguration feingranular zu optimieren. JHipster ist ein anderes prominentes Beispiel. Das Angular-Team hat diese Zielgruppe nach wie vor auf dem Radar. Aber sie wird kleiner. Zumindest, wenn man nach den Issues in Stephans Open-Source-Framework gehen darf. Dort begegnen mir (Stephan) nur noch selten individuelle Build-Chains.

Standalone-Components haben einen großen Vorteil: Wenn ihr so schusselig seid wie ich (Stephan), legt ihr dauernd Komponenten an, nur um sie ein paar Stunden später wieder zu löschen. Mit Modulen ist das umständlich – da müsst ihr die Module-Datei öffnen und die Komponente entfernen. Bei Standalone-Komponenten braucht ihr nur den Ordner zu löschen und die Komponente ist rückstandsfrei verschwunden. Sehr praktisch!

Signale

Kommen wir zu einem Thema, das richtig Furore macht: Signals *[3]*.

Bei Signalen geht es um State Management und um Kommunikation. Dafür gibt es in *Angular* eigentlich schon Lösungen: *RxJS* für die Kommunikation und Services sowie *NgRX/Store* für das State Management. Und natürlich Change Detection – jene Magie, die *Angular* groß gemacht hat.

Signale ermöglichen es euch, Daten von einem Sender zu beliebig vielen Empfängern zu transportieren. Auf dieser Flughöhe klingt es genauso wie die Definition von *RxJS*. Im Detail gibt es aber große Unterschiede *[10]*. *RxJS* kümmert sich um die Verarbeitung von Datenströmen und bietet eine große Palette von reaktiven, funktionalen Operatoren, mit denen ihr die Daten transformieren könnt.

Falls ihr bisher hauptsächlich in der Java-Welt unterwegs seid, könnt ihr *RxJS* grob mit Hadoop vergleichen, oder auch mit der reaktiven Programmierung, die vor der Corona-Pandemie kurz en vogue war. *Hadoop* (und vergleichbare Frameworks wie *Spark* oder *Snowflake*) hat seine Nische im Bereich Big Data gefunden und ist dort sehr erfolgreich. Die reaktive Programmierung ist trotz aller Vorteile in der Java-Welt weniger erfolgreich. Die traditionelle imperative Programmierung ist für die meisten Entwicklerinnen und Entwickler einfacher, und seit Java 21 machen die virtuellen Threads die reaktive Programmierung in den meisten Fällen überflüssig. Die Laufzeitumgebung sorgt selbst dafür, dass I/O nicht mehr blockiert und befreit euch von der Notwendigkeit, aktiv non-blocking I/O zu implementieren.

Bei *Angular* ist das anders. *RxJS* erfreut sich großer Beliebtheit. So groß, dass *RxJS* in vielen Projekten zur Kommunikation zwischen Komponenten eingesetzt wird – und das ist oft eine relativ schwergewichtige Lösung mit unnötig vielen Abstraktionslayern. Faustregel: Wenn ihr ein Observable als `@Input()`-Parameter übergebt, dann habt ihr entweder sehr gute Gründe, oder ihr macht es Neueinsteigern im Projekt schwer.

Das Problem bei Observables ist, dass ihr eine Abstraktionsschicht hinzufügt. Ihr betrachtet nicht mehr die Daten, sondern die Änderungen der Daten. Das ist ein sehr mächtiges und wertvolles Werkzeug. Es hat nur den Nachteil, nicht so unmittelbar intuitiv zu sein wie die direkte Arbeit mit den Daten. Es erfordert Übung.

Signale sind in vielen Fällen eine leichtgewichtige Alternative. Besonders deutlich wird das, wenn ihr die OnPush-Strategie der Change Detection verwendet oder sogar ganz auf *zone.js* verzichtet *[2]*. Traditionell gibt es viele Situationen, in denen ihr den Change Detector manuell aktivieren müsst. Mit Signalen ist das anders: Wenn ein Signal einen neuen Wert bekommt, rendert *Angular* automatisch alle Komponenten neu, die das Signal verwenden. Dafür braucht *Angular* nicht den kompletten Komponentenbaum zu überprüfen. Signale liefern euch eine feinkörnige Reaktivität, die der Performance eurer Anwendung zugutekommt.

Im Prinzip sind Signale auch eine Abstraktionsschicht, aber die Erfahrung zeigt, dass sie leichter zu verstehen sind. Computed Signals sind ein gelungenes Beispiel dafür. Wenn ihr eine Implementation mit Signalen *(siehe Listing 1)* mit der entsprechenden *RxJS*-Lösung *(siehe Listing 2)* vergleicht, werdet ihr keinen großen Unterschied feststellen, aber die Signal-Variante ist für Einsteiger leichter zu verstehen.

```
@Component({
  selector: 'app-counter-signals',
  template: `
     <p>Zähler: {{ count() }}</p>
     <button (click)="increment()">Erhöhen</button>
  `,
  standalone: true
})
export class CounterSignalsComponent {
  count = signal(0);

  increment() {
     this.count.update(c => c + 1);
  }
}
```

Listing 1: Komponente mit Signalen

```
@Component({
  selector: 'app-counter-RxJS',
  template: `
     <p>Zähler: {{ count$ | async }}</p>
     <button (click)="increment()">Erhöhen</button>
  `,
  standalone: true
})
export class CounterRxJSComponent {
  private countSubject = new BehaviorSubject<number>(0);
  count$ = this.countSubject.asObservable();

  increment() {
     this.countSubject.next(this.countSubject.value + 1);
  }
}
```

Listing 2: Die Komponente aus Listing 1, diesmal mit RxJS

Ersetzen Signale *RxJS*?

Signale sind nicht der Versuch, *RxJS* zu ersetzen*[1]*. Für *RxJS* gibt es nach wie vor valide Einsatzgebiete. Wer jedoch gar nicht damit klarkommt, kann dennoch in vielen Fällen auf Signale umsteigen. Die *Angular University* fasst das in diesem bemerkenswerten Satz zusammen: „Wenn es darum geht, Datenänderungen in der Anwendung bekannt zu geben, sind Signale erheblich leichter zu verwenden und zu verstehen als *RxJS*." *[3]*

Das ist tatsächlich ein Problem bei *RxJS*: so faszinierend und leistungsfähig es ist, es ist eine große Hürde für Einsteiger. Insbesondere dann, wenn sie aus einer Welt ohne funktionale Programmierung kommen.

So gesehen ist es nur konsequent, dass in letzter Zeit immer mehr

„InterOp"-Pakete zwischen Signalen und *RxJS* aufkommen. Für viele Entwicklerinnen und Entwickler scheint es doch interessant zu sein, *RxJS* durch Signale zu ersetzen. In einfachen Fällen reicht das in der Regel auch aus. Und seien wir mal ehrlich: die meisten Business-Anwendungen bestehen aus einer Reihe einfacher REST-Calls. Es ist gut, die Magie von *RxJS* in der Hinterhand zu haben, aber für den Alltag reicht eine einsteigerfreundliche Lösung.

@Input() vs. input()

Vermutlich kommt kein *Angular*-Projekt ohne eine Komponente aus, die ein Input benötigt. Insbesondere Presentation-Komponenten, also Komponenten, die ausschließlich etwas anzeigen, machen intensiven Gebrauch davon. Nicht selten sollen diese dann auch darauf reagieren, wenn sich der Input-Wert ändert.

Bisher hat man dafür seinen Input über den `@Input()`-Decorator definiert und dann mittels dem Lifecycle-Hook *ngOnChanges* auf dessen Änderung geprüft. Dort hat man dann bei Bedarf andere Werte aktualisiert oder andere Funktionen aufgerufen. Seit *Angular* 17.1 sind Signal Inputs stabil und können produktiv genutzt werden. Mit diesen lassen sich Inputs in einer Zeile und ohne Decorator definieren.

Um die Unterschiede zu veranschaulichen, stellen wir uns mal eine Spieler-Komponente vor. Diese hat unter anderem die Aufgabe den Spielernamen im Format „Spieler: {{spieler.tag}} / Level: {{spieler.level}}" anzuzeigen. Für die Veranschaulichung gehen wir außerdem davon aus, dass wir diesen String nicht direkt im Template zusammenbauen wollen oder können.

Vor *Angular* 17.1 sähe diese Komponente so aus wie in *Listing 3*.

Mit *Angular* 18 können wir es vereinfachen, wie in *Listing 4* zu sehen ist.

Nicht nur sparen wir uns ein paar Zeilen Code, wir haben für unsere Inputs auch alle weiteren Vorzüge von Signalen. Wir könnten sogar, wenn es unbedingt sein muss, über `effect()` Side-Effects auslösen. Einen Input als Pflicht zu markieren, ist mit Input Signals auch deutlich leichter.

Es kommt noch besser. Traditionell rendert *Angular* immer gleich die komplette Komponente neu. Bei Signalen kann es gezielt die Teile der Komponente aktualisieren, die vom Signal abhängen *[3]*. Alles andere bleibt unverändert.

@if und @for

Habt ihr euch schon einmal gefragt, warum *Angular* so merkwürdige HTML-Erweiterungen wie *ngIf oder (click) verwendet?

```
@Component({
      selector: 'app-spieler',
      template: `
          <span>{{ spielerNameLevel }}</span>
          <!-- Weitere Darstellungen -->
      `,
})
export class SpielerComponent implements OnChanges {

      // Seit Angular 16 geht auch @Input({required: true})
      @Input()
      spieler!: Spieler;

      spielerNameLevel = '';

      ngOnChanges(changes: SimpleChanges) {
          if(changes.spieler){
               const spieler = changes.spieler;
               this.spielerNameLevel = `Spieler: ${spieler.tag} / Level: ${spieler.level}`;
          }
      }
}
```

Listing 3: Komponente mit @Input()-Decorator

```
@Component({
  selector: 'app-spieler',
  imports: [],
  template: `
      <span>{{ spielerNameLevel() }}</span>
      <!-- Weitere Darstellungen -->
  `,
})
export class SpielerComponent {
  spieler = input.required<Spieler>()
  spielerNameLevel = computed(() =>`Spieler: ${this.spieler().tag} / Level: ${this.spieler().level}`)
}
```

Listing 4: Komponente mit Signal Input

```
<span>Rangliste</span>
<ol *ngIf="isLoggedIn; else notLoggedInMessage">
  <ng-container *ngFor="let spieler of spielerRangliste; index as i; first as isFirst">
     <li [ngClass]="{'first-place': isFirst}">[{{ i + 1 }}] - {{ spieler.tag }}</li>
  </ng-container>
</ol>

<ng-template #notLoggedInMessage>
  <span>Bitte melde dich an um die Rangliste zu sehen!</span>
</ng-template>
```

Listing 5: Template mit alter Control-Flow-Syntax (*ngIf. *ngFor)

```
<span>Rangliste</span>
@if(isLoggedIn){
  <ol>
     @for (spieler of spielerRangliste; track spieler; let i = $index; let isFirst = $first){
        <li [ngClass]="{'first-place': isFirst}">[{{ i + 1 }}] - {{ spieler.tag }}</li>
     }
  </ol>
} @else {
  <span>Bitte melde dich an um die Rangliste zu sehen!</span>
}
```

Listing 6: Template mit neuer Control-Flow-Syntax (@if. @for)

Tatsächlich war das das Ergebnis einer längeren Diskussion, und es war auch keineswegs die einzige Lösung, die diskutiert wurde. Das *Angular*-Team hätte auch einfach „ng-if“ verwenden können, ohne das Sternchen. Oder es hätte – ähnlich wie es React.js macht – eine neue, dedizierte Syntax verwenden können. Das war seinerzeit gar nicht ungewöhnlich. Nur ein Beispiel von vielen: *Handlebars* war seinerzeit eine populäre Templating-Engine und verwendete die Syntax `{{#if condition}}`.

Das wäre aus heutiger Sicht eine Möglichkeit. Doch damals war *Angular* – in der Tradition von *AngularJS* – darauf angewiesen, kompatibel zu HTML zu bleiben. *Angular* war eine JavaScript-Bibliothek, die im Browser lief. Das bedeutete, dass der Browser den HTML-Code erst einmal laden und akzeptieren musste. Es war wichtig, nur gültiges HTML zu verwenden.

Dafür gab es verschiedene Möglichkeiten. Beispielsweise hätte man alles mit HTML-artigen Elementen umsetzen können, wie zum Beispiel <ng-if>. Das wären aber zusätzliche DOM-Elemente gewesen – mit dem entsprechenden Overhead. Eine andere Möglichkeit wären HTML-artige Attribute wie „ng-if“ gewesen. Das Team hat sich dagegen entschieden, weil die Gefahr von Kollisionen mit bestehenden Attributen in Kundenprojekten bestand.

Was damals niemand auf dem Radar hatte, war, dass nicht nur englische Buchstaben für HTML-Attribute verwendet werden dürfen. Der HTML-Standard ist da sehr inklusiv. Eure Attribute dürfen Umlaute genauso enthalten wie chinesische Schriftzeichen – oder auch Sonderzeichen wie das Sternchen, die runde oder die geschweifte Klammer. So etwas hat damals niemand gemacht.

Damit war die Idee für *ng-if, *ngFor und [(ng-model)] geboren. In der finalen Version von *Angular* 2.0 wurde daraus *ngIf, *ngFor und [(ngModel)] (siehe Listing 5) [4].

Heute hat sich vieles geändert: *Angular* liefert seinen eigenen Compiler mit; und wenn ihr eure HTML-Templates im Browser sucht, habt ihr schlechte Karten. Da ist kein HTML-Code. Stattdessen wird der DOM-Baum heutzutage per JavaScript erzeugt. *Angular* ist so weit verbreitet, dass es von jeder IDE unterstützt wird. Es ist nicht mehr notwendig, HTML-kompatibel zu sein. Das bietet neue Freiheitsgrade und modernes *Angular* nutzt sie, um die Lesbarkeit zu verbessern. Seit *Angular* 17 könnt ihr `@if` und `@for` verwenden. Das ist ein großer Fortschritt. Endlich werden `for`-Schleifen und `if`-Bedingungen nicht in Attributen versteckt, sondern prominent im Quelltext angezeigt (siehe Listing 6). Als Sahnehäubchen verbessert sich die Performance eurer Anwendung ein wenig [5].

Resümee

Die Innovationskraft von *Angular* ist nicht nur ungebrochen – im Moment verändert sich *Angular* in ungeahntem Tempo. Und das, ohne seinen Charakter zu verändern. Dabei ist es wichtig, die Community mitzunehmen, und deswegen werden auch die bisherigen Konzepte bis auf weiteres unterstützt.

Der Vergleich mit anderen Frameworks wie React.js ist spannend. *Angular* und *React* nähern sich in vielerlei Hinsicht an, bleiben aber dennoch unverwechselbar und behalten ihren eigenen Stil. Wenn wir noch weiter über den Tellerrand schauen, sehen wir ganz neue Konzepte wie *Astro*, *Svelte* oder *HTMX*. Diese bringen neue Ideen in die Welt der Webentwicklung und sorgen indirekt dafür, dass die Innovationskraft von *Angular*, *React* und Co. nicht erlahmt.

Das heißt aber nicht, dass wir in einer Update-Hölle landen. Erinnert ihr euch noch an den berühmten Artikel „How it feels to learn JavaScript in 2016“ [6]? Heute können wir uns darauf verlassen,

dass unsere Programme auch nächstes Jahr noch kompilierbar sind. Die JavaScript-Welt ist erwachsen geworden. Und dennoch entwickelt sie sich weiter, bleibt spannend und ist immer für eine Überraschung gut!

Es sind gleich so viele Überraschungen, dass wir einen Folgeartikel für euch vorbereitet haben. Habt ihr die Themen zoneless *Angular*, functional Inject und server-side Rendering vermisst? Stay tuned!

Quellen

[1] Alexander Möller 2023: *https://thecattlecrew.net/2023/11/06/Angular-signals-uebersicht-und-aktueller-stand/*

[2] Phillip Escher 2023: *https://angular.de/artikel/developer-ergonomics-hier-kommen-angular-signals/*

[4] Stephan Rauh, 2014: *https://www.beyondjava.net/angularjs-2-0-sneak-preview-data-binding*

[5] Kim Pham: *https://www.tevpro.com/blog/exploring-the-new-angular-17-if-and-for-directives*

[6] Jose Aguinaga 2016: *https://hackernoon.com/how-it-feels-to-learn-javascript-in-2016-d3a717dd577f*

[7] Minko Gechev 2025, *https://blog.Angular.dev/Angular-2025-strategy-9ca333dfc334*

[8] Sacha Greif 2025, *https://2024.stateofjs.com/en-US/libraries/front-end-frameworks/*

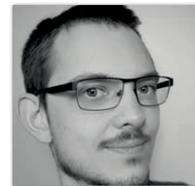
[9] Stephan Rauh 2014/2015, *https://www.beyondjava.net/angular-2-0-courageous-plans*

[10] Angular Architects 2025, *https://www.Angulararchitects.io/blog/Angular-signals/*

Stephan Rauh

Articles@beyondjava.de

Stephan Rauh ist Software-Architekt und führt unter anderem Workshops und Trainings für KI, Angular, Spring und Microservices durch. In seiner Freizeit kümmert er sich um die Open-Source-Frameworks ngx-extended-pdf-viewer, BootsFaces und natürlich um seinen Blog *https://www.beyondjava.net*, der – noblesse oblige! – eine Angular-Anwendung ist. Ihr könnt Stephan unter *articles@beyondjava.de* erreichen.



Alexander Pahn

Alexander.pahn@outlook.com

Alexander Pahn ist Software-Entwickler und zertifizierter Senior Angular Entwickler. Sein Herz brennt für alles rundum Angular. In seiner Freizeit werkelt er meist an eigenen Apps oder probiert neue Frameworks und Sprachen aus. Nachdem er die Dokumentationsseite des ngx-extended-pdf-viewer neugestaltet hat, plant er nun ein eigenes Open-Source Forms Framework für Angular.

Julian Schmidt

Info@julianb.de

Julian Schmidt ist passionierter Software-Entwickler mit besonderer Begeisterung für Webtechnologien. Er verfolgt aktuelle Entwicklungen aufmerksam und freut sich über jedes neue Release. Mit fundierter Erfahrung und einem Auge fürs Detail entwickelt er innovative Lösungen und teilt sein Wissen über moderne Technologien.

Java aktuell

JAHRESABO

FÜR 29,00 €
BESTELLEN

www.ijug.eu

Mehr Informationen zum Magazin und Abo unter:

www.ijug.eu/de/java-aktuell

Die Kunst hinter ChatGPT und DeepSeek: Geheimnisse der Transformer-Architektur

Adrian Füller

ChatGPT und DeepSeek sind sogenannte Large Language Models (LLMs), deren Grundlage die Transformer-Architektur ist. In diesem Artikel werden die wesentlichen Komponenten dieser Architektur, insbesondere die Feed-Forward-Netze und der Attention-Mechanismus, beschrieben, die durch Matrix-Multiplikationen realisiert werden. Auf Basis dessen kann erklärt werden, warum diese Architektur allen vorherigen Methoden, wie LSTM („Long short-term memory"), überlegen ist. Es wird verständlich, wie ChatGPT oder DeepSeek funktionieren und warum die Funktionsweise von LLMs nicht die eigentlich angedachte Idee der Entwickler der Transformer-Architektur ist.

Vor Transformern wurden bevorzugt „Long short-term memory" (LSTM) für sequenzielle Vorhersagen verwendet. Das Problem daran ist, dass LSTMs eine Komplexität von $\Theta(n*d^2)$ haben, was ihre Skalierung sehr ressourcenaufwändig macht. Transformer hingegen haben eine Komplexität von $\Theta(n^2*d)$, was ähnlich problematisch klingt, aber aufgrund der „Verschiebung" der quadratischen Komplexität weniger aufwändig ist. Wie die Komplexität zustande kommt, und weshalb die Komplexität von Transformern eine neue Ära in der KI eingeleutet hat, wird im Folgenden erklärt.

Neuronale Netze

Neuronale Netze (NN) versuchen das Denken eines Gehirns durch einen ähnlichen Aufbau nachzustellen. Ein NN besteht aus einem Geflecht von Nervenzellen (Neuronen), die durch Synapsen (Gewichte) verschieden miteinander verbunden sind. Sofern alle Neuronen auf einer Seite der Synapse gleichzeitig stimuliert werden, feuern die Neuronen, wodurch die Gewichte verstärkt werden. Wenn ein Neuron nie stimuliert wird, stirbt die Synapse irgendwann ab und die Verbindung bricht beziehungsweise die Gewichte fallen auf null. Zusätzlich können Neuronen einen optionalen Bias haben, wodurch

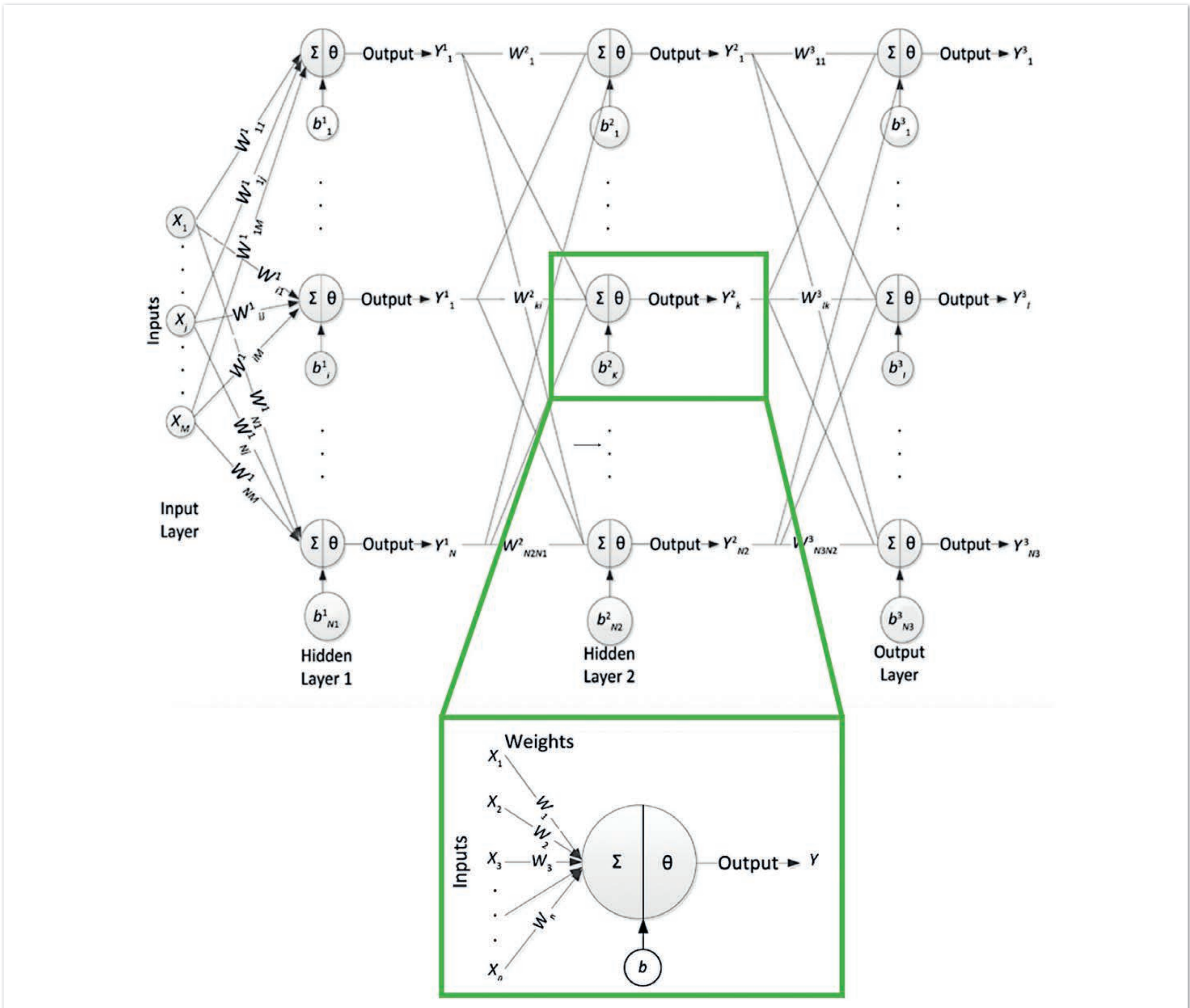


Abbildung 1: Neuronales Netz mit Fokus auf ein einzelnes Neuron [1]

ein Eigengewicht erzeugt werden kann. Eine visuelle Darstellung eines NN wird in *Abbildung 1* dargestellt.

Eine Schicht von Neuronen beschreibt eine vertikale Linie an Neuronen, die jeweils mit einer Schicht Neuronen vor und hinter sich verbunden sind. Beim Training werden die Werte der Neuronen, Gewichte und Bias zu Beginn eines Trainingslaufs initialisiert und können anschließend vom Benutzer nicht mehr angepasst oder beeinflusst werden. Die Anpassung der Werte passiert ausschließlich durch die Backpropagation beim Training des NN. Deswegen werden diese Neuronenschichten „Hidden Layer" genannt. Bei klassischen NN wird der verwendete Datensatz zum Training in Feature Set und Targets aufgeteilt. Dabei beschreibt das Feature Set die Daten, die anfänglich in das NN als Input gegeben werden und Targets die Werte, die durch das NN vorhergesagt werden sollen. Dadurch erzeugt das NN einen Prediction-Wert und das Target ist der IST-Wert.

Im Trainingslauf gibt es nach der Initialisierung der Gewichte einen Forward-Pass, bei dem die Trainingsdaten durch das NN geschickt werden und einen Prediction-Wert abgeben. Anschließend wird ein Fehler (oder Loss) abhängig von der Differenz zwischen Prediction- und IST-Wert berechnet. Durch die Backpropagation wird dieser Loss rückwärts durch das NN geschickt, um die Gewichte anzupassen.

Sobald alle Trainingsdaten einmal durchlaufen sind, ist ein Durchlauf (genannt Epoche) beendet. Hierbei muss darauf geachtet werden, dass das NN nicht zu viele Epochen trainiert, sonst lernt das NN die Trainingsdaten auswendig und zeigt während der Inferenz auf neuen Daten schlechte oder falsche Prediction-Werte. Dieses Phänomen nennt man Overfitting. Im Gegensatz zum Training durchläuft bei der Inferenz das Modell einmal den Forward-Pass und gibt einen Prediction-Wert ab, aber macht keine Backpropagation, also keine Anpassung der Gewichte. Die Inferenz wird klassischerweise für Modelle in Produktion angewandt *[1]*.

Eine Erweiterung zu NN ist LSTM. Das sind NN, die keine klassische Klassifizierung durchführen, sondern versuchen, den nächsten Wert für die eingegebene Sequenz zu berechnen. Diese rekursive Technik wird auch Sequence-to-Sequence bezeichnet und wird beispielhaft in *Abbildung 2* dargestellt.

Bei LSTMs werden jüngere Werte höher gewichtet als alte Werte. Ein Vorteil bei dieser Art der Vorhersage ist, dass mehrere Werte vorhergesagt werden können. Möchte man beispielsweise das Wetter vorhersagen, dann kann der Wert für morgen berechnet werden und basierend darauf der Wert für übermorgen. Mit dieser Technik wird auch Textgenerierung durchgeführt. Mit genügend Trainingsdaten können für einfache Sätze die nächsten Worte generiert werden oder es können Antworten auf Fragen gegeben werden.

Der Nachteil bei LSTMs ist der hohe Rechenaufwand, der für das Training benötigt wird, weil LSTMs eine Komplexität von (n*d^2) haben. Dabei ist n die Sequenzlänge, also die Anzahl der Werte, die für die Ein- und Ausgabe vorhanden sind und d die Dimensionalität beziehungsweise die Länge der Vektoren, die die Wörter darstellen, also wie groß das NN durch die Neuronen und Schichten ist. Hierbei ist das Problem, dass sich die quadratische Komplexität wegen des sequenziellen Durchlaufs nicht parallelisieren lässt (zum Beispiel mit GPUs), weshalb die Komplexität nicht verkleinert werden kann *[2]*.

Der Ursprung der Transformer und LLMs

Der Google-Übersetzer benutzt für die Übersetzungen ebenfalls NN. Dort wurde 2016 eine eigene Variante von LSTM mit dem Namen „Google Neural Machine Translation" (GNMT) vorgestellt, die seinen Vorgänger in der Übersetzung übertroffen hat *[3]*.

Trotzdem ist auch bei GNMT nach wie vor die Problematik, dass es sich aufgrund der Rekursivität nicht parallelisieren lässt. Dadurch müssen die Wörter in einer Sequenz eingelesen werden und können nicht durch GPUs beschleunigt werden. Ein weiteres Problem ist der fehlende Kontext des Textes. LSTMs übersetzen Texte immer satzweise und können nur bedingt auf Sachverhalte früherer Sätze zugreifen. Beispielsweise im Satz „Der Lehrer belehrt im Unterricht seine Schüler." weiß das LSTM nicht, dass das Wort „seine" auf „Lehrer" bezogen ist. Aufgrund dieser Einschränkungen sind LSTMs und auch die Übersetzungen auf wenige Wörter bis Sätze limitiert und die Übersetzungen sind nur teilweise korrekt, weil kein Kontext besteht *[4]*.

Deswegen setzte sich ein Team von Google-Mitarbeitenden zusammen und versuchte, die oben genannten Probleme mit der Parallelität und mit dem Kontext zu lösen, indem sie versuchten, eine Architektur zu entwickeln, die nicht auf die Rekursivität angewiesen ist. Auf Basis des Papers „Sequence to Sequence Learning with Neural Networks" von Sutskever et al. *[5]* war die Idee, die Rekursivität zu entfernen und damit eine parallele Entwicklung zu ermöglichen. Das Ergebnis war das Paper „Attention Is All You Need" von Vaswani et al. *[6]*.

Da Google diesem Paper und den vorgestellten Lösungen keine große Aufmerksamkeit schenkte, ging unter anderem Sutskever damit

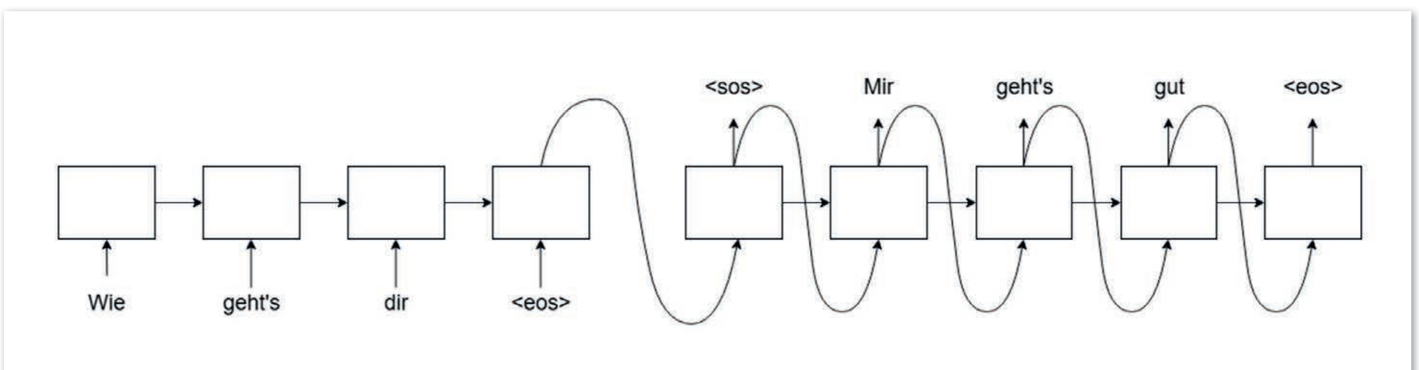


Abbildung 2: Beispiel Sequence-to-Sequence (© Adrian Füller)

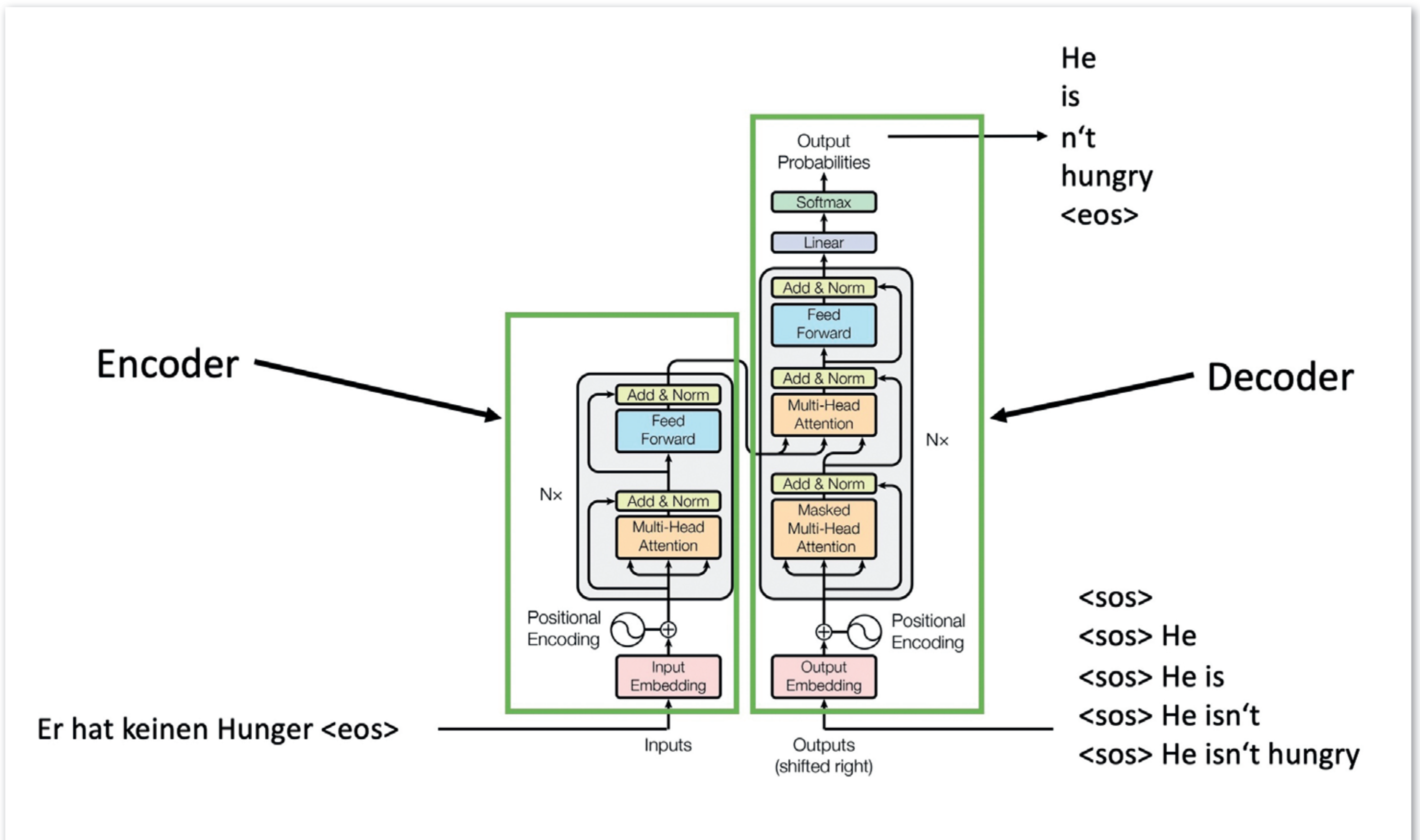


Abbildung 3: Transformer-Architektur [6]

eigene Wege und gründete die Firma OpenAI mit. Dort wurde die Transformer-Architektur aus dem Paper weiterentwickelt, vergrößert und mit großen Datenmengen wochenlang trainiert. Diese Art von Model wird LLM genannt und aus dem Produktspektrum von OpenAI ist aktuell ChatGPT das bekannteste Produkt.

Transformer-Architektur

Wie in *Abbildung 3* gezeigt, besteht ein Transformer aus den zwei Blöcken Encoder (linke Hälfte) und Decoder (rechte Hälfte). Kurz zusammengefasst generiert der Encoder aus einem Eingabetext Embeddings, die in Vektorform zurückgegeben werden. Die Werte der Vektoren sind so gewählt, dass ähnliche Werte (wie Prinz und König) nah beieinander liegen und gegensätzliche Werte (wie Baum und Würfel) möglichst weit auseinander zeigen. Der Decoder generiert auf Basis des Eingabetexts das nächste Wort, von dem der Decoder glaubt, dass es am wahrscheinlichsten als nächstes kommt und fügt es dem Eingabetext an. Danach läuft der Decoder wieder durch bis das wahrscheinlichste Wort das „End-of-Sequence"-Wort ist.

In *Abbildung 3* kommen links unten die Inputs in den Transformer. Das kann beispielsweise ein deutscher Text sein, den der Transformer ins Englische übersetzen soll. Im Schritt „Input Embedding" werden die eingegebenen Texte in Token embedded. Grundsätzlich werden Wörter vor dem Modell-Training immer in Zahlen (oder Vektoren) konvertiert, da Maschinen mit Zahlen besser umgehen beziehungsweise rechnen können.

Wie die Tokenisierung funktioniert, ist vom jeweiligen Transformer-Model abhängig. Die einfachste Variante ist, ein Token aus einem Wort zu erzeugen, geläufiger ist es aber, dass ein Token einer Silbe entspricht. Auch Sonderzeichen, wie Punkt und Fragezeichen, können ein eigenes Token sein. Dadurch entsteht eine Matrix, die für jeden Token einen Vektor aus einem vorab definierten Vokabular des Transformers darstellt.

Bei trainierten LLMs haben Wörter mit ähnlicher Bedeutung einen ähnlichen Vektor und gleiche Wörter den gleichen Vektor. Im Positional Encoding wird jedem Token abhängig von seiner Position ein Wert hinzuaddiert. Dadurch wird sichergestellt, dass beispielsweise im Satz „Das große Kind spielt mit dem kleinen Kind." die zwei Wörter „Kind" nicht als der gleiche Wert angesehen werden, sondern das erste „Kind" ist näher am „große" und das zweite „Kind" ist näher am „kleinen".

Das Ziel des Attention-Mechanismus soll es sein, einen Kontext zwischen den Wörtern in einem Satz oder Text herzustellen. Bevor die Attention berechnet wird, werden die Token mit den vortrainierten Matrizen W_Q, W_K und W_V verrechnet. Dadurch entstehen aus den Token-Vektoren die Matrizen Q, K und V. K und V stehen für Key und Value, ähnlich wie es von Key-Value-Pairs bekannt ist. Q steht für Query und versucht für jedes Token die besten Matches (beziehungsweise Wörter mit Attention zum Query) in den Keys zu finden. Der Attention-Mechanismus lässt sich mit der Formel zusammenfassen:

$$z = Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$ *[6]*

Dabei werden die Queries mit den Keys multipliziert, um ein möglichst hohes Match zu finden, wie in *Abbildung 4* oben und links unten dargestellt.

Dabei wird jeder Token mit jedem Token multipliziert. Da die ursprüngliche Matrix der Token mit den vortrainierten W-Matrizen

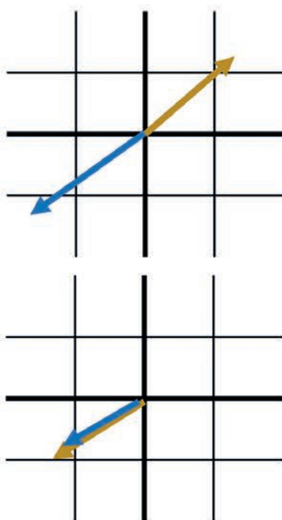
multipliziert wurde, können damit Token gefunden werden, die einen Kontext zueinander haben. Wenn es beispielsweise für Q_3K_5 ein hohes Match gibt, dann hat der Token 5 einen Kontext zu Token 3. Danach wird das Ergebnis normalisiert, damit zu große Werte nicht zu groß werden und kleine Werte nicht zu klein.

Mithilfe des Softmax werden die Ergebnisse jedes Queries auf die Summe 1 skaliert, um Wahrscheinlichkeiten für jeden möglichen Wert zu bekommen. Damit lässt sich sagen, welche Token wie viel relativen Kontext für den jeweiligen Query haben. Anschließend können durch die Multiplikation mit der Value die passenden Werte gefunden werden. Ein Beispiel zur Veranschaulichung ist in *Abbildung 4* rechts unten zu sehen.

Damit aus der Attention (oder Single-Head-Attention) eine Multi-Head-Attention wird, wird der Attention-Mechanismus mehrfach parallel durchgeführt und das Ergebnis mit einer vortrainierten Matrix W^O multipliziert, um alle Heads zusammen zu nehmen. Es wird davon ausgegangen, dass sich jeder Head auf einen anderen Bereich spezialisiert, zum Beispiel auf Verben oder auf die Verbindung von Adjektiven und Nomen. Eine visuelle Darstellung dieser Heads ist in *Abbildung 5* zu sehen.

Die Blöcke „Add & Norm" beschreiben Residual Neural Nets, die den ursprünglichen Wert mit dem berechneten Wert addieren. Der Block „Feed Forward" ist ein klassisches NN mit zwei Hidden Layers, die eine größere Dimensionalität haben, als die Anzahl der Token. Der

	Q_1	Q_2	Q_3	Q_4	Q_5	$\cdots$	Q_n
K_1	$\frac{Q_1 \cdot K_1}{\sqrt{d_k}}$	$\frac{Q_2 \cdot K_1}{\sqrt{d_k}}$	$\frac{Q_3 \cdot K_1}{\sqrt{d_k}}$	$\frac{Q_4 \cdot K_1}{\sqrt{d_k}}$	$\frac{Q_5 \cdot K_1}{\sqrt{d_k}}$	$\cdots$	$\frac{Q_n \cdot K_1}{\sqrt{d_k}}$
K_2	$\frac{Q_1 \cdot K_2}{\sqrt{d_k}}$	$\frac{Q_2 \cdot K_2}{\sqrt{d_k}}$	$\frac{Q_3 \cdot K_2}{\sqrt{d_k}}$	$\frac{Q_4 \cdot K_2}{\sqrt{d_k}}$	$\frac{Q_5 \cdot K_2}{\sqrt{d_k}}$	$\cdots$	$\frac{Q_n \cdot K_2}{\sqrt{d_k}}$
K_3	$\frac{Q_1 \cdot K_3}{\sqrt{d_k}}$	$\frac{Q_2 \cdot K_3}{\sqrt{d_k}}$	$\frac{Q_3 \cdot K_3}{\sqrt{d_k}}$	$\frac{Q_4 \cdot K_3}{\sqrt{d_k}}$	$\frac{Q_5 \cdot K_3}{\sqrt{d_k}}$	$\cdots$	$\frac{Q_n \cdot K_3}{\sqrt{d_k}}$
K_4	$\frac{Q_1 \cdot K_4}{\sqrt{d_k}}$	$\frac{Q_2 \cdot K_4}{\sqrt{d_k}}$	$\frac{Q_3 \cdot K_4}{\sqrt{d_k}}$	$\frac{Q_4 \cdot K_4}{\sqrt{d_k}}$	$\frac{Q_5 \cdot K_4}{\sqrt{d_k}}$	$\cdots$	$\frac{Q_n \cdot K_4}{\sqrt{d_k}}$
K_5	$\frac{Q_1 \cdot K_5}{\sqrt{d_k}}$	$\frac{Q_2 \cdot K_5}{\sqrt{d_k}}$	$\frac{Q_3 \cdot K_5}{\sqrt{d_k}}$	$\frac{Q_4 \cdot K_5}{\sqrt{d_k}}$	$\frac{Q_5 \cdot K_5}{\sqrt{d_k}}$	$\cdots$	$\frac{Q_n \cdot K_5}{\sqrt{d_k}}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$



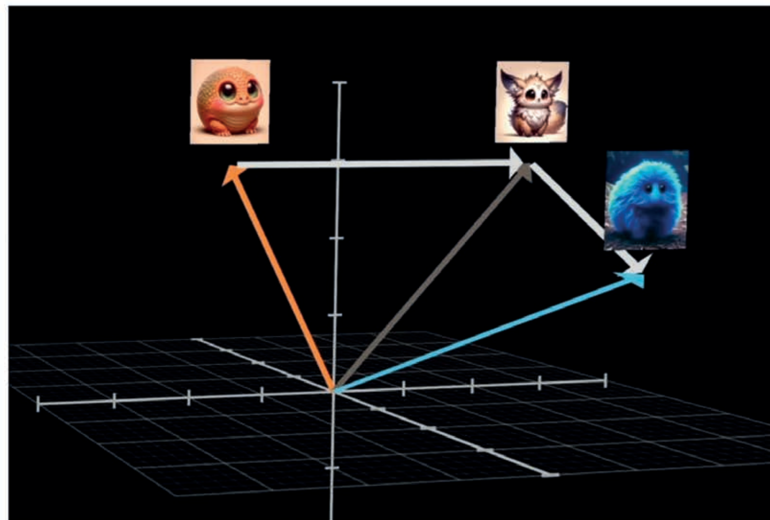


Abbildung 4: Multiplikation der QK-Werte mit anschließender Division der gewurzelten Dimensionalität (oben), visuelle Darstellung der QK-Multiplikation: weit entfernte Pfeile beschreiben Token mit wenig semantischer Übereinstimmung (wie Würfel und Baum) und geben einen niedrigen Wert zurück, ähnliche Token mit ähnlicher Semantik (wie König und Prinz) haben nah beieinander liegende Vektoren und geben einen hohen Wert zurück (links unten), visuelle Darstellung der Multiplikation mit V: der Vektor für „Monster" wird durch die Value zu einem „flauschigen blauen Monster" (rechts unten) [7]

Encoder und Decoder werden jeweils N-mal in Sequenz geschaltet, also mehrfach hintereinander ausgeführt. Abschließend wird in dem Block Linear die Tokensequenz auf die Länge aller möglichen Token des Transformers umgerechnet und ein Softmax angewendet. Damit bekommt jeder mögliche Output-Wert eine Wahrscheinlichkeit. Der höchste Wert ist das generierte Output-Token.

Zusammengefasst wird der Eingabetext im Encoder in Token aufgeteilt, die zum Ende des Durchlaufs einen Vektor erzeugen, der für jeden möglichen Token eine Wahrscheinlichkeit erzeugt. Zu Beginn bekommen die Token nach ihrer Tokenisierung einen Positionswert addiert und anschließend wird für die Token der Attention-Mechanismus durchgeführt. Die Vektoren der Token werden parallel in mehreren Heads jeweils mit drei Matrizen multipliziert, wodurch die Matrizen Query, Key und Value entstehen.

Das Ergebnis der Heads wird wieder zusammengefasst und mit einem Residual Neural Net normalisiert. Das Ergebnis durchläuft ein klassisches NN mit zwei Hidden Layers und wird nochmal mit einem Residual Neural Net normalisiert. Dieser Attention-Mechanismus wird N-mal wiederholt. Das Ergebnis wird dem Decoder bereitgestellt. Dieser führt ebenfalls den Attention-Mechanismus N-mal mit der Ausgabe des Encoders und dem bereits generierten Text durch *[6]*.

Warum der Hype um die Transformer?

Wie zuvor beschrieben, haben LSTMs eine Komplexität von (n*d^2), da die Sequenz einmal durchlaufen wird, aber wegen der Rekursivität eine quadratische Komplexität für die Größe der Vektoren hat. Bei den Transformern ist die Komplexität Θ(n^2*d). Die Länge der Sequenz ist quadratisch, weil die Länge mit sich selbst in die Multiplikation von Query und Key berechnet wird, wie zuvor in *Abbildung 4* gezeigt. Die Dimensionalität ist aber nicht quadratisch, weil die Werte nicht rekursiv durchgegangen werden, sondern nur einmal in die Matrix geladen werden. Der Grund, weshalb Θ(n*d^2) (LSTM) leistungsintensiver ist als Θ(n^2*d) (Transfomer) liegt in den großen Zahlen. Bei kleinen Netzen sind die Netze ungefähr gleich performant, aber je größer sie werden, umso performanter sind die Transformer. Das n beschreibt die Sequenzlänge des eingegebenen Textes. Dieser ist im Normalfall ein paar hundert bis tausend Wörter lang *[4]*.

Die Dimensionalität wächst bei LSTMs quadratisch, aber bei Transformern linear, wodurch viel größere Netze möglich sind. Mit der Dimensionalität ist die Länge der Vektoren der einzelnen Token gemeint. Das heißt, je mehr Werte im Vektor für die Definierung des Tokens stehen, umso akkurater ist das System. Durch den Attention-Mechanismus kann der Transformer auch Adjektive oder Artikel einem Nomen zuordnen und einen Kontext in Sätzen erkennen.

Ein weiterer Vorteil ist das Gedächtnis des Transformers. Da alle Eingabewerte gleichzeitig eingegeben werden, verlieren ältere Werte über Zeit nicht an Bedeutung. Jeder Wert steht in der Matrix, die sich aus den Token mit den jeweiligen Vektoren zusammensetzt, und wird für die Berechnung der Attention betrachtet. Der einzige Flaschenhals ist die quadratische Komplexität der Sequenz-Länge. Dadurch ist ein Transformer auf eine bestimmte Tokenlänge limitiert und kann keine Bücher oder mehrseitigen Aufsätze auf einmal verarbeiten, beziehungsweise ignoriert er einen Teil des Textes *[4, 7]*.

ChatGPT ist kein klassischer Transformer

Ein klassischer Transformer, wie er im Paper „Attention Is All You Need" vorgestellt wird, bekommt eine Eingabe, die vom Encoder verarbeitet wird und Embeddings erzeugt. Die Embeddings werden dem Decoder gegeben. Dieser erzeugt mithilfe der Embeddings und der bisher generierten Ausgabewerte weitere Ausgabewerte bis das <eos>-Token (End of Sequence) die höchste Wahrscheinlichkeit hat.

Ein Beispiel für einen solchen Transformer ist ein Übersetzer wie der Google Übersetzer. ChatGPT dagegen hat keinen Encoder – er ist ein sogenannter Decoder-only. ChatGPT wird der Input (Prompt), der hier gegeben wird, als bisher generierte Output-Werte gegeben. Durch sein Training weiß ChatGPT, dass er auf den Prompt eine Antwort geben soll und generiert darauf basierend die wahrscheinlichsten Token und gibt sie aus, bis das <eos>-Token kommt. Ein Beispiel für einen Encoder ist „Bert" von Google, der Embeddings (untereinander abhängige Vektoren) aus den Eingabetexten erzeugen kann *[9]*.

Was DeepSeek anders macht als ChatGPT

Um die Genauigkeit und Leistung von LLMs messen zu können, werden sie auf Benchmarks getestet und darauf basierend verglichen. DeepSeek ist ein neues LLM aus China, das auf populären Benchmarks bessere Ergebnisse erzielt als ChatGPT und einen Bruchteil von ChatGPT gekostet haben soll.

Vor DeepSeek war man der Meinung, dass für bessere Leistung größere Modelle gebraucht werden, die mehr Hardwareressourcen für eine längere Zeit benötigen. Diese Trainingsleistung ist von einer Kette großer Firmen wie NVIDIA für GPUs oder TSMC für Halbleiter abhängig. Als durch DeepSeek klar wurde, dass die gleichen Ergebnisse für einen Bruchteil der Kosten und des Trainingsaufwands möglich sind, sind die Aktien der Firmen entsprechend gefallen.

Ein weiterer Punkt, der für noch weniger Trainingsleistung sorgt, ist die Technik-Distillation. Dabei werden die Inferenz-Antworten eines großen Models genommen und als Trainingswerte für kleinere Modelle verwendet *[10]*. Im Fall von DeepSeek wurde das Modell DeepSeek-R1 mit 671 Milliarden Parametern genommen und mit seinen Inferenz-Antworten ein Llama-Model von Meta mit 8 Milliarden Parametern trainiert.

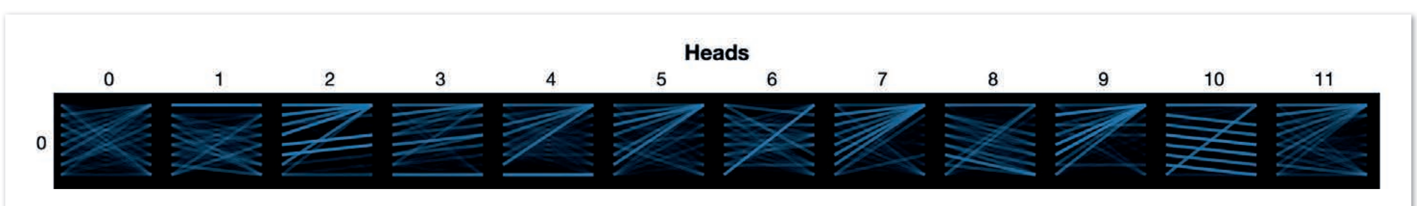


Abbildung 5: Multi-Head-Attention mit elf Heads, wobei jeder Head einen anderen Fokus hat. Die Dicke der Linien beschreibt die Attention der Token an der jeweiligen Position zueinander [8].

Zum Vergleich hat ein NN mit 1000 Eingabewerten, zwei Hidden Layers mit 128 und 64 Neuronen und 4 Ausgabewerten zirka 130.000 Parameter. Dadurch lernt das kleine Model die Antworten des großen Models und kann bei genügend Training dieses nachahmen, benötigt aber nur einen Bruchteil der Leistung. Das daraus resultierende Model DeepSeek-R1-Distill-Llama-8B ist so klein, dass es auf den meisten Heimrechnern mit einer neueren NVIDIA-Grafikkarte laufen kann *[11]*.

Gepaart mit dem Fakt, dass DeepSeek komplett Open Source verfügbar ist, sorgte DeepSeek bei den bisherigen großen Anbietern von LLMs für eine entsprechende Überraschung.

Fazit

Auslöser des Transformer-Hypes war in der Tech-Branche die Verschiebung der Komplexität von der Dimensionalität auf die Sequenzlänge. Damit konnten große Modelle mit fast linearer Komplexität trainiert werden. In den Medien und der breiten Gesellschaft begann der Hype mit ChatGPT und seinen menschen-ähnlichen Antworten und schnellen Generierung von ausführbarem Quellcode für verschiedene Programmiersprachen. Seitdem kommen fast täglich neue Meldungen zu KI in den Nachrichten. Nachdem es bis vor kurzem ein technisches und finanzielles Wettrüsten zwischen den großen Tech-Firmen um das größte Model gab, begann mit DeepSeek ein Wettkampf um das effizienteste Modell zwischen verschiedenen Ländern, das nicht nur präzise Antworten ausgeben kann, sondern auch möglichst wenige Ressourcen verbraucht.

Quellen

[1] Awad und Khanna 2015: Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers, Springer-Verlag, Berkley

[2] Hochreiter und Schmidhuber 1997: Long Short-Term Memory, MIT Press, Cambridge

[3] Johnson et al. 2016: Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation, eprint arXiv

[4] *https://www.youtube.com/watch?v=rBCqOTEfxvg* am 01.02.2025

[5] Sutskever et al. 2014: Sequence to Sequence Learning with Neural Networks, eprint arXiv

[6] Vaswani et al. 2017: Attention Is All You Need, Curran Associates Inc., Red Hook

[7] *https://www.youtube.com/watch?v=wjZofJX0v4M* am 01.02.2025

[8] *https://github.com/jessevig/bertviz* am 01.02.2025

[9] *https://www.youtube.com/watch?v=XfpMkf4rD6E* am 01.02.2025

[10] *https://www.youtube.com/watch?v=gY4Z-9QlZ64* am 01.02.2025

[11] *https://huggingface.co/deepseek-ai/DeepSeek-R1* am 01.02.2025

Adrian Füller

adrian.fueller@deutschebahn.com

Adrian ist IT-Innovationsmanager bei der DB Fernverkehr AG. Dort sollen die Vertriebsplattformen DB Navigator und bahn.de zukunftsfähig und modern gehalten werden. Nebenbei studiert er Data Science im Master, um auch die theoretischen Hintergründe zu verstehen. Privat werden verschiedene IT-Themen, wie HomeLab oder App-Entwicklung ausprobiert, die im beruflichen Umfeld zu kurz kommen.

Metaversum: Ein neuer Ansatz zur digitalen Transformation im öffentlichen Sektor

Malte Teichmann, Universität Potsdam, Georg Ritterbusch, Universität Potsdam, Sarah Victoria Mohr, Hochschule Hof, Collin Croome, Eldar Sultanow, Capgemini

Die Bundesagentur für Arbeit bietet an ihren Agenturstandorten ein umfangreiches Karriereangebot, doch wie praktisch wäre ein Ort, an dem alle Services und Informationen von überall und für alle einfach abrufbar wären? Ein Traum, der durch das Stichwort Metaversum vielleicht in naher Zukunft Wirklichkeit werden könnte und damit eine digitale Transformation im öffentlichen Sektor darstellt. Eine Analyse beleuchtet die Push- und Pull-Faktoren des Metaversums zur Verbesserung der Karriereberatung und Verwaltungsprozesse.

Der Weg in eine virtuelle Parallel-Welt

Buzzwörter wie Metaversum sind in aller Munde, doch der Begriff ist noch nicht vollständig definiert *[1] [2]*. Die Grundidee des Metaversums kann jedoch als eine Art digitale Parallelwelt verstanden werden. Ein besonderes Merkmal besteht in der dreidimensionalen Darstellung, die durch immersive Technologien wie Virtual-Reality-Brillen einen immersiven Charakter erhält. In dessen Räumen werden interaktive Erlebnisse durch Avatare erlebbar. Die wohl bekanntesten Plattformen mit Metaverse-Charakter sind Roblox und Decentraland, die sich durch soziale Interaktion und einen Verbund von unterschiedlichen, miteinander verbundenen virtuellen Räumen oder Welten auszeichnen *[1]*.

Für den öffentlichen Sektor entsteht eine Vielzahl von Möglichkeiten, etwa die Bereitstellung virtueller Dienstleistungen oder die Einbindung unterschiedlicher Interessengruppen in partizipative Prozesse. Beispiele wie aus der südkoreanischen Hauptstadt Seoul in Form des „Metaverse Seoul" zeigen, dass innovative Anwendungen machbar sind und von Bürger*innen genutzt werden. Mit der lauffähigen kommunalen Metaverse-Lösung werden unterschiedliche staatliche Services virtuell und rund um die Uhr für Nutzer*innen zur Verfügung gestellt *[3] [4]*.

In Deutschland experimentiert die Bundesagentur für Arbeit (BA) bisher im Zuge von Pilotprojekten, etwa mit VR-Videos und Cardboard-Lösungen, um Schüler*innen bei der Ausbildungswahl zu unterstützen und damit aufzuzeigen, wie digitale Tools zur Berufsberatung eingesetzt werden können.

Ein Blick in die Zukunft

Das Metaverse der Zukunft entfaltet sich als eine grenzenlose digitale Landschaft. Im Zuge einer tiefgreifenden Analyse von Interviewgesprächen mit Mitarbeitenden der Bundesagentur konnten von den Experten Push-Faktoren identifiziert werden, die eine Implementierung des Metaversums vorantreiben, aber auch Pull-Faktoren, die gegen die Implementierbarkeit sprechen.

Push Faktoren für das Metaversum im öffentlichen Sektor

Das Metaverse der Zukunft bietet unterschiedliche Services an, darunter Beratungsangebote, Unterstützung bei der Jobvermittlung sowie Fort- und Weiterbildungsangebote. Die **Barrierefreiheit**, also der offene Zugang für alle Personen, orts- und zeitunabhängig, stellt einen klaren Vorteil dar, da geografische und logistische Barrieren überwunden werden können. So erhalten Menschen in abgelegenen Regionen oder mit Mobilitätseinschränkungen neue Möglichkeiten, professionelle Beratung und Unterstützung in Anspruch zu nehmen.

> *„... es gibt hier Möglichkeiten, diese Technologie zu nutzen, um eben vielleicht auch einen Job zu finden oder auch eine Qualifizierung zu erhalten."*
>
> *Operativer Mitarbeiter der BA*

Diese Dezentralisierung schafft eine bislang ungekannte **Flexibilität** in der Beratung und Schulung. Sie ermöglicht es, auch in Krisenzeiten wie Pandemien, Naturkatastrophen oder anderen außergewöhnlichen Umständen, Beratungs- und Schulungsangebote aufrechtzuerhalten, ohne dass physische Treffen erforderlich sind. Selbst bei persönlichen Hindernissen wie gesundheitlichen Einschränkungen, familiären Verpflichtungen oder geografischen Barrieren können Menschen weiterhin Zugang zu wichtigen Dienstleistungen erhalten.

Darüber hinaus erweitert diese Flexibilität die **Reichweite der Angebote** erheblich, insbesondere für benachteiligte Gruppen, Langzeitarbeitslose, geflüchtete Personen und Menschen mit Behinderungen. Dies gilt insbesondere für ländliche Gebiete, die oft von zentralisierten Angeboten ausgeschlossen bleiben. Regionen, die zuvor aufgrund infrastruktureller Schwächen oder fehlender Ressourcen nicht erreicht werden konnten, profitieren nun von digitalen Lösungen.

> „... Metaverse-Schulung würde diesen Menschen erst mal die Sozialität wiedergeben. Das bedeutet, sie wären dabei, aber wären immer noch geschützt."
>
> *Mitarbeiterin in der Organisationsabteilung der BA*

Virtuelle Plattformen fördern eine **effizientere Zusammenarbeit** zwischen Berater*innen, Klient*innen und verschiedenen Abteilungen. In einer zunehmend digitalisierten Arbeitswelt spielen effektive Besprechungen eine zentrale Rolle. Der Einsatz virtueller Räume im Metaverse könnte durch die immersive Umgebung das Gefühl persönlicher Anwesenheit stärken und so die Effektivität und Qualität der Interaktion deutlich verbessern.

Zeit- und ortsunabhängige Schulungen im Metaverse zeichnen sich durch ihre hohe **Individualisierbarkeit** aus und könnten nicht nur die Karrierechancen der Teilnehmenden erhöhen, sondern auch das Wohlbefinden und die Lebensqualität nachhaltig verbessern. Besonders in Phasen erhöhter familiärer Anforderungen, wie etwa in der Kinderbetreuung oder Pflege von Angehörigen, bieten diese digitalen Lösungen einen wertvollen Mehrwert. Dank der Möglichkeit, Inhalte und Lernprozesse individuell anzupassen, wird das Gefühl der Selbstbestimmung gestärkt, sodass berufliche Ziele erreicht werden können, ohne Abstriche im Privatleben machen zu müssen.

Virtuelle Realität ermöglicht die Nachbildung komplexer Szenarien, in denen Fachkräfte praktische Fertigkeiten erlernen und **realitätsnahe Erfahrungen** sammeln können, ohne Risiken einzugehen. Diese Form des Trainings bietet nicht nur eine effektive Vorbereitung auf anspruchsvolle Aufgaben, sondern ist auch flexibel einsetzbar, selbst in Regionen mit begrenzten Ressourcen. Besonders in Bereichen, in denen Sicherheit eine zentrale Rolle spielt, erweist sich diese Technologie als besonders vorteilhaft.

Unterstützende humanoide KI-Agenten im Metaverse könnten den Prozess des Ausfüllens von Anträgen durch interaktive und effiziente Unterstützung vereinfachen. Sie bieten personalisierte Hilfestellung, prüfen Eingaben und geben nützliche Hinweise, wodurch der Aufwand für Antragstellende reduziert und gleichzeitig die Verwaltung entlastet werden könnte.

Pull-Faktoren gegen das Metaversum im öffentlichen Sektor

Die Vision birgt jedoch auch Schattenseiten. Die Integration in bestehende öffentliche Verwaltungsprozesse gestaltet sich **komplex**. Bestehende Prozesse und Zugriffsrechte müssen analysiert werden und es müssen sinnvolle Schnittstellen geschaffen werden, die eine einfache Integration ermöglichen.

Darüber hinaus müssen aufgrund der geringen Verbreitung **Akzeptanzprobleme** sowie **mangelnde digitale Kompetenzen** innerhalb der Belegschaft und der Gesellschaft überwunden werden. Diese erfordern umfassende Schulungsmaßnahmen durch fachkundiges Personal, das den Umgang mit neuen Medien beherrscht.

Zusätzlich stellen die **technologischen Anforderungen** eine erhebliche Herausforderung dar: Ohne flächendeckendes Hochgeschwindigkeitsinternet und robuste IT-Systeme könnten viele Regionen von dieser Zukunft ausgeschlossen bleiben – insbesondere strukturschwache Gebiete, für die dies ein entscheidender Nachteil wäre.

Neben **erheblichen Investitionen** in die Aus- und Weiterbildung der Mitarbeitenden sind auch der technologische Ausbau sowie fortlaufende Kosten für Wartung und Schulung essenziell. Eine klare Finanzierungsstrategie ist unerlässlich, um die langfristige Nutzung des Metaverse sicherzustellen.

> „Da hilft mir die beste Technologie nichts, wenn der Zugang zu schwierig … oder zu kostenintensiv ist."
>
> *Technischer Mitarbeiter der BA*

Darüber hinaus gewinnt die Frage des **Datenschutzes** an Bedeutung: Wie sicher sind persönliche Informationen in einer Welt, die zunehmend von Algorithmen und vernetzten Systemen gesteuert wird? Die Implementierung des Metaverse im öffentlichen Sektor muss komplexe rechtliche Rahmenbedingungen berücksichtigen, insbesondere beim Umgang mit sensiblen Daten. Im Falle einer nicht sorgfältig ausgeführten Planung und Umsetzung können kritische Stimmen laut werden, die die Akzeptanz neuer Technologien beeinträchtigen.

Auch die Gefahr, dass das Metaverse mehr Spaltung als Integration bewirkt, ist nicht zu unterschätzen. Ohne gezielte Maßnahmen zur Förderung von **Inklusion** könnten insbesondere ältere, weniger technikaffine Nutzer*innen oder Menschen mit kognitiven Einschränkungen Schwierigkeiten bei der Nutzung haben. Zusätzlich

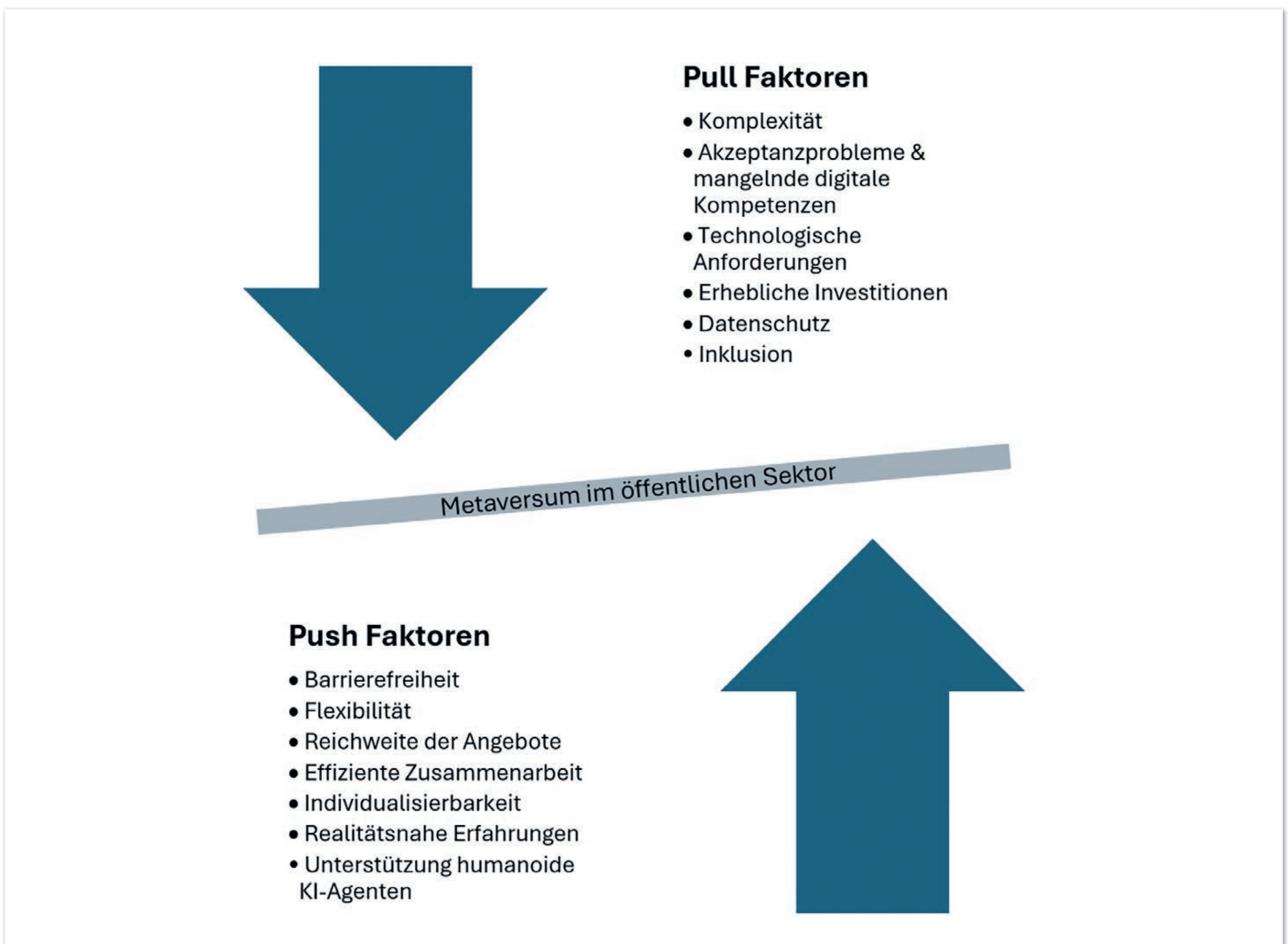


Abbildung 1: Push- und Pull-Faktoren für das Metaversum des öffentlichen Sektors ©Sultanow et al. 2025

könnten die hohen Kosten für technische Hardware einkommensschwache Personen von den Vorteilen des Metaverse ausschließen. Damit das Metaverse zugänglich und nutzbar für alle bleibt, sind umfassende Anstrengungen erforderlich, um soziale und technologische Barrieren abzubauen.

> „Also ich glaube, das größte Hemmnis wäre, wenn der Zugang ein recht komplexer Prozess ist."
>
> *Technischer Mitarbeiter der BA*

Fazit und Blick in die Zukunft

Das Metaverse repräsentiert eine vielversprechende Perspektive für die digitale Transformation des öffentlichen Sektors. Die sich daraus ergebenden Chancen, insbesondere im Bereich der Karriereberatung und Verwaltung, gilt es zu nutzen. Hierfür sind jedoch Strategien erforderlich, die sowohl technologische als auch soziale Aspekte gleichermaßen berücksichtigen.

Obwohl die Entwicklung des Metaversums noch in den Anfängen steckt, ist bereits erkennbar, dass es den öffentlichen Sektor barrierefreier und zugänglicher gestalten könnte, sofern bestehende Herausforderungen überwunden werden.

Referenzen

[1] Ritterbusch, G. D., & Teichmann, M. R. (2023). Defining the Metaverse: A Systematic Literature Review. *IEEE Access, 11*,

Dr. Malte Teichmann
Universität Potsdam und Weizenbaum Institut
malte.teichmann@wi.uni-potsdam.de

Dr. Malte Rolf Teichmann ist erfolgreich von der Erziehungswissenschaft in die Wirtschaftsinformatik gewechselt. Darüber hinaus ist er Forschungsgruppenleiter der FG „Bildung für die digitale Welt" am Weizenbaum Institut. Seine Forschungsschwerpunkte liegen in der erfolgreichen Gestaltung von Lehr- und Lernangeboten, der humanzentrierten Digitalisierung von Geschäftsprozessen und der Verbindung von Lern- und Wissenstheorien.

Collin Croome
Internet- und Metaverse-Pionier
collin@croome.de

Collin Croome ist Internet- und Metaverse-Pionier sowie ein gefragter Experte und Keynote-Speaker für Zukunftstrends wie Generative Künstliche Intelligenz. In seiner 30-jährigen Karriere hat der Marketingprofi über 800 digitale Projekte für namhafte Unternehmen und Marken realisiert und dabei einen enormen Erfahrungsschatz aufgebaut.

Dr. Eldar Sultanow
IT-Stratege bei Capgemini
eldar.sultanow@capgemini.com

Dr. Eldar Sultanow ist IT-Stratege bei Capgemini. Der promovierte Wirtschaftsinformatiker blickt auf eine mehr als zwanzigjährige Erfahrung in der Softwareentwicklung zurück: von der Programmierung bis hin zum KI-Design. Zuletzt ist von Eldar Sultanow das Buch „Sie werden mit dem nächsten freien Chatbot verbunden" im Redline Verlag erschienen.

Georg Ritterbusch
Universität Potsdam und Weizenbaum Institut
georg.ritterbusch@wi.uni-potsdam.de

Georg David Ritterbusch ist wissenschaftlicher Mitarbeiter am Lehrstuhl für Wirtschaftsinformatik, Prozesse und Systeme der Universität Potsdam und am Weizenbaum Institut in Berlin. Seinen Forschungsschwerpunkt setzt er in der Human-Computer Interaction (HCI), insbesondere Extended Reality (VR/MR/AR), Metaverse, sensorisches Feedback und Human-AI Interaction.

12368–12377. *https://doi.org/10.1109/ACCESS.2023.3241809*

[2] Sultanow, E., Chircu, A., Wüstemann, S., Schwan, A., Lehmann, A., Sept, A., Szymanski, O., Venkatesan, S., Ritterbusch, G. D., & Teichmann, M. R. (2022). *Metaverse Opportunities for the Public Sector. 8th Annual Meeting of AIS SIG BD: International Research Workshop on Big Data for Business Ecosystems.*

[3] Kshetri, N., Dwivedi, Y. K., & Janssen, M. (2024). Metaverse for advancing government: Prospects, challenges and a research agenda. *Government Information Quarterly, 41*(2), 101931. *https://doi.org/10.1016/j.giq.2024.101931*

[4] Lnenicka, M., Rizun, N., Alexopoulos, C., & Janssen, M. (2024). Government in the metaverse: Requirements and suitability for providing digital public services. *Technological Forecasting and Social Change*, 203, 123346. *https://doi.org/10.1016/j.techfore.2024.123346*

Sarah Victoria Mohr
Universität Bayreuth und Hochschule Hof
sarah.victoria.mohr@iisys.de

Sarah Victoria Mohr ist wissenschaftliche Mitarbeiterin in der Forschungsgruppe Empirical Research and User Experience an der Hochschule Hof und promoviert an der Universität Bayreuth. Ihr Forschungsschwerpunkt liegt auf der psychologischen und physiologischen Wirkung olfaktorischer Stimuli in immersiven Virtual-Reality-Erlebnissen.

MITMACHEN UND AUTORIN ODER AUTOR WERDEN!

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

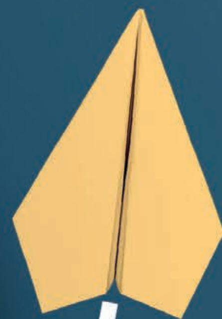
Nimm Kontakt zu uns auf
und sende deinen Artikelvorschlag an:
redaktion@ijug.eu

Wir freuen uns, von dir zu hören!

Weitere Informationen

Endlich was bewegen!

Alexander Domene, G&H Bankensoftware AG



Warum machen wir das eigentlich so? Habt ihr euch das auch schonmal gefragt? Oft sind es Arbeitsweisen, Prozesse und Methoden, die sich über Jahre etabliert und wahrscheinlich auch festgefahren haben. In einer solchen Situation ist es für euch vermutlich mühsam, eure „neuen" Ideen einzubringen, obwohl sie wunderbar passen würden?

Hand aufs Herz, wie oft ist es euch auch so gegangen, dass eure Vorschläge oder Erfahrungen aus einem vorherigen Job oder der Uni nicht verstanden oder abgelehnt wurden? Das ist selten von den Kollegen böse oder abwertend gemeint, sondern hat oft eine Vielzahl von Gründen. Es ist daher wichtig, die verschiedenen Gründe zu verstehen und zu erkennen, warum Menschen gegenüber Veränderung meist zurückhaltend sind. Nachfolgend beschreibe ich zehn häufige Erklärungen dafür:

1. **Angst vor Kontrollverlust**
   Wenn eine Veränderung von außen auf uns zukommt, oder uns auferlegt wird, haben wir manchmal das Gefühl, dass wir die Autonomie über unseren Arbeitsbereich verlieren könnten. Diese oft unbegründete Angst vor dem Verlust der Entscheidungsbefugnis oder von Einfluss oder Macht führt zu unmittelbarem Widerstand, selbst wenn die Veränderung für uns und unseren Alltag von Vorteil sein könnte.

2. **Übermäßige Ungewissheit**
   „Lieber das bekannte Unglück als das unbekannte Glück" – ohne klare Informationen über künftige Schritte und Auswirkungen stellen wir uns gerne erstmal die schlimmsten Szenarien vor. Das Unbekannte erzeugt Unruhe und Angst und führt dazu, dass der Status quo bevorzugt wird.

3. **Überraschungsentscheidungen**
   Plötzliche Änderungen ohne vorherige Einbeziehung oder Ankündigung lösen Abwehrreaktionen aus. Wir fühlen uns überrumpelt und nicht respektiert, wenn wir nicht in den Entscheidungsprozess einbezogen wurden.

4. **Der „Anders-Effekt"**
   Veränderungen, die sich ungewohnt anfühlen oder erheblich von der Routine abweichen, verursachen bei uns Unbehagen. Wir haben Schwierigkeiten, wenn sich neue Prozesse zu sehr von unseren gewohnten Arbeitsweisen unterscheiden – insbesondere, wenn wir „genau so" schon seit Jahren „erfolgreich" arbeiten.

5. **Gesichtsverlust**
   Wenn die derzeitigen Prozesse oder Arbeitsweisen als ineffizient oder veraltet bezeichnet werden, fühlen sich diejenigen, die diese eingeführt, befürwortet oder angewandt haben, in ihrem Ansehen, ihrer Kompetenz und ihrem Urteilsvermögen angegriffen.

6. **Bedenken hinsichtlich der Kompetenz**
   Veränderungen erfordern oft neue Fähigkeiten, Denk- oder Arbeitsweisen. Die Angst, in der neuen Arbeitsstruktur unzulänglich zu sein oder als inkompetent zu gelten, erzeugt Widerstand.

7. **Mehr Arbeit**
   Seien wir ehrlich: Veränderungen – denkt zum Beispiel nur an einen vergleichsweise einfachen Wechsel von Jira nach GitLab! – erfordern in der Regel zusätzlichen Aufwand: das Erlernen neuer Systeme/Tools, die Teilnahme an Schulungen, die Anpassung von Prozessen. Diese zusätzliche Arbeitsbelastung führt zu Unmut, denn die Welt bleibt nicht stehen und Releases müssen weiterhin ausgeliefert werden, Kundenanfragen müssen beantwortet werden und so weiter.

8. **Strahleffekte**
   Die meisten Veränderungen haben oft unerwartete Folgen für alle Abteilungen und deren Beziehung und Arbeitsweise untereinander. Menschen widersetzen sich, wenn sie diese kaskadenartigen Auswirkungen nicht vorhersehen oder kontrollieren können.

9. **Ressentiments von früher**
   „Das haben wir schon x-mal probiert und das wird nichts oder geht nicht", kommt dir bekannt vor, oder? Frühere gescheiterte Veränderungen oder negative Erfahrungen mit dem Führungsteam oder den Entscheidern führen zu Zynismus. Historische Missstände und Erfahrungen tauchen entsprechend bei neuen Initiativen zu Veränderungen immer wieder auf.

10. **Echte Bedrohungen**
    Manchmal ist der Widerstand rational und nachvollziehbar – Veränderungen könnten tatsächlich Arbeitsplätze, Machtstrukturen oder geschätzte Arbeitsbeziehungen bedrohen. Menschen schützen ihre Interessen und ihre Position.

Jetzt ist klarer, was Veränderung mit uns macht und auf welchen Ebenen Veränderung wirkt, aber es ist immer noch unklar, was denn nun eine Veränderung für uns als Unternehmen mit dem Fokus Softwareentwicklung ist. Und es mag überraschen; meines Erachtens nach ist es eben nicht vordergründig Technologie. Nachfolgend eine Liste von Gründen für Veränderungsdruck – und keine Sorge, *GenAI* wird nicht auftauchen:

- **Veränderung mit Nutzerverhalten**
  Kennt ihr noch die Zeiten, in denen man mit Fax und Unterschrift eine Reisebuchung bestätigt hat? Genau, ich auch nicht. Jede Entwicklung im Nutzerverhalten die mit „** first" beginnt, zum Beispiel „web first" oder „mobile first", ist eine Revolution für die Anbieter der entsprechenden Services. Im B2B-Bereich war mit Sicherheit der Wechsel von On-Premises zu Cloud ein wesentlicher Treiber für Veränderung. Auch dabei stand nicht die Technologie im Vordergrund, sondern der Wunsch, ohne große IT-Kenntnis oder IT-Freigabe Software einzusetzen. Außerdem gibt es auch den Bedarf, via Web auf die Dienste zuzugreifen – am besten noch direkt via Mobile. Anbieter, die nicht oder sehr lange für diesen Change gebraucht haben, sind zurückgefallen.

- **Kosten**
Wettbewerb, Marktlage oder Feature Set sind nur ein paar der Gründe, die mit Druck auf den Verkaufspreis wirken können. Um diesen Druck aufzufangen, muss sich auch die Kostenseite verändern. Die Kosten in der Softwareentwicklung, abseits von Lizenzen oder Patenten, sind vor allem der Personalaufwand. Mit berücksichtigt werden sollten auch immer die Time-to-Market oder Leadtime-to-Change, die Kosten für die CI/CD und Test sowie Entwicklungsumgebungen und, je nach Unternehmen, der Betrieb beziehungsweise das Betriebsmodell. Auf all diese Faktoren haben der Tech-Stack und die Architektur wesentlichen Einfluss, aber der Auslöser für diese Veränderungen ist das Ziel, Kosten zu minimieren. Ein aktuelles Beispiel ist der enorme Preisanstieg von VMware-Lizenzen, der dazu führt, dass einige Hoster im großen Stil und mit viel Aufwand auf andere Virtualisierer migrieren *[1]*.

- **Einkaufsverhalten „The Next Shiny Thing"**
Ok, vielleicht geht es doch ein wenig um GenAI. Aber es geht nicht um die oft geführten Diskussionen, ob und wie AI eine Revolution ist, die unsere Industrie nachhaltig verändern wird, sondern schlicht darum, dass seit rund zwei Jahren so gut wie jede Software AI enthalten muss. Teilweise regt das auch zum Schmunzeln an, wie zum Beispiel der AI-Chatbot meiner Waschmaschine, mit dem ich mich über das richtige Programm austauschen kann. Es macht keinen Sinn, es spart auch keine Zeit, ist aber für den inneren Nerd ein großer Spaß. Ein paar Beispiele von vergangenen Hypes findet man hier: Big Data (zirka 2010), Internet of Things/ IoT (zirka 2013), Blockchain/Crypto (zirka 2017), Metaverse (zirka 2021).

Manche davon sind auch heute noch relevant, aber natürlich mit deutlich geringerer Aufmerksamkeit als GenAI. Mit jedem neuen Hype verlangt der Markt, dass Produkte oder Services entsprechend angepasst oder ergänzt werden. Wie sinnvoll das ist (siehe mein Beispiel mit der Waschmaschine), kann man ausführlich diskutieren, jedoch wird es nicht die Erwartungen der Kunden wegnehmen. Man muss sehr genau überlegen, ob man sich wirklich einem Hype verwehren kann, und falls nicht, hat dies direkte Auswirkung auf die Produktentwicklung und, je nachdem wie weitreichend das Thema ist, auf das ganze Unternehmen und dessen Geschäftsmodell.

- **Veränderte Marktbedingungen, Zölle, Sanktionen**
Ein leider immer aktuelleres Thema: Zölle oder Sanktionen gegenüber Ländern führen unmittelbar dazu, dass Unternehmen, die Handel mit den betroffenen Ländern betreiben, unter Handlungsdruck stehen. Die Einschränkung beim Einkauf von Dienstleistungen, zum Beispiel Nearshoring in (Weiß-)Russland, oder auch vom Verkauf von Software, zum Beispiel Kaspersky Labs, können unmittelbar Einfluss auf den Tech-Stack oder die Architektur haben.

- **Regulatorik/Gesetze:**
Neue oder veränderte Standards (zum Beispiel ISO/DIN, US- oder EU-Recht) können Auswirkungen auf das Geschäftsmodell haben, jedoch ist fast sicher, dass das Unternehmen sich mit dem Produkt oder dem Service befassen und gegebenenfalls Anpassungen vornehmen muss.

Jeder der genannten Punkte hat vordergründig nichts mit dem Tech-Stack, der Architektur oder Prozessen und Methoden zu tun, dennoch haben diese oft umfassende Auswirkung auf das gesamte Unternehmen.

Es gilt also getreu dem Motto „Wer nicht mit der Zeit geht, geht mit der Zeit", dass sich alle Unternehmen, aber insbesondere wir in unserer Industrie, stetig verändern müssen. Doch wie gestaltet man nun stetige Veränderung so, dass möglichst wenig der zehn genannten Widerstände/Ablehnungen zutreffen?

Wichtig ist es, ein gemeinsames Verständnis für die anstehende Veränderung zu entwickeln. Am besten anhand des Musters der fünf W-Fragen, die man auch vom Erste-Hilfe-Kurs kennt: Wer? Was? Wann? Wie? Wo? Soweit der einfache Teil, aber wie kann man es dann in die Organisation einbringen und vor allem umsetzen?

Kurzer Exkurs: Architecture Decision Records

Wenn man sich nur auf die Softwareentwicklung und die Arbeit an der Architektur bezieht, hat sich ein Vorgehen über die Zeit bewährt: Die Architecture Decision Records (ADRs) wurden 2011 von Michael Nygard eingeführt und in seinem Artikel „Documenting Architecture Decisions" *[2]* erstmals beschrieben. Das Konzept entstand aus der Erkenntnis, dass traditionelle Softwarearchitekturdokumentation oft veraltet und schwer nachvollziehbar gewesen ist. Nygard entwickelte ADRs als leichtgewichtigen Ansatz, der eng mit der agilen Softwareentwicklung verzahnt ist. Und nein, falls ihr jetzt Agilität oder Scrum im Kopf habt: Der vorgeschlagene Weg ist unabhängig vom Vorgehensmodell der Softwareentwicklung. Es geht mir dabei also nicht um die eine weitere Variante für eine agile Transformation!

Ein ADR anzulegen ist sehr einfach und braucht auch keine fancy Tools. Ob ihr diese in einer Text-Datei speichert, als Markdown oder in Jira pflegt, ist unerheblich. Wichtig ist, dass es eine Form ist, die für die meisten leicht zugänglich ist. Technisch, aber auch toolseitig.

Nachfolgend ein einfaches Template für einen ADR und dessen Kernbestandteile mit einem kurzen Beispiel:

```
# Title
- Fortlaufende Nummer und prägnante Überschrift
- "ADR-0001: Verwendung von PostgreSQL als Datenbank"

# Status
- Aktuelle Phase der Entscheidung. Mögliche Werte:
proposed, accepted, rejected, deprecated, superseded
- "proposed"

# Context
- Ausgangssituation, Problemstellung, Rahmenbedingun-
gen und Einschränkungen
- "Die Performance der aktuellen SQLite ist unzure-
ichend"

# Decision
- Getroffene Entscheidung; Konkrete technische De-
tails der Lösung
- "Migration von SQLite auf Postgres"

# Consequences
- Positive und negative Auswirkungen; Resultierende
Abhängigkeiten; Notwendige Folgeentscheidungen
- "erwartet wird eine deutliche bessere Performance,
jedoch erhöht sich der Ressourcenbedarf im Betrieb"
```

```
# Optional
- Date: Entscheidungsdatum
- Participants: Beteiligte Personen
- Technical Story: Verweis auf zugehörige Stories/
Tickets
- Related ADRs: Verweise auf verbundene ADR
```

Natürlich gibt es noch ganz viele andere Ideen und Weiterentwicklungen, wie man mit ADRs arbeiten kann. Wer das gerne übernehmen oder einführen möchte, dem empfehle ich als Startpunkt das gut gepflegte Repository von Joel Parker Henderson *[3]*.

Die Vorteile, wenn man Architekturentscheidung so trifft und vor allem dokumentiert, liegen auf der Hand. Man kann jederzeit später nachvollziehen, warum wir uns damals für *Talwind* statt *Bulma*, *Postgres* statt *MongoDB* oder für oder gegen den Architekturshift auf Event-Streaming mit *Kafka* entschieden haben. Jeder kann die Motivation, Überlegungen und den Kontext nachvollziehen.

Falls dann erneut (wir alle wissen, dass es kein „ob", sondern nur eine Frage des „wann" ist) eine Entscheidung oder Technologie in Frage gestellt wird, kann man den ADR von damals zu Rate ziehen und analysieren und bewerten, was sich zwischen damals und heute verändert hat. Haben sich die User verzehnfacht? Die Datenbank kommt ans Limit? Die Design- oder UX-Ideen lassen sich aktuell nur sehr schwer umsetzen und so weiter? Oder ist es einfach, dass ein neues Framework/Produkt („Shiny New Thing") sich cool anhört und zumindest in den Videos und Artikeln sehr viel verspricht.

Wenn ihr das mehrfach wiederholt, werdet ihr sehen, wie sich eure Entscheidungen qualitativ verbessern und besser nachvollziehbar sind – und als Bonus: Es fühlt sich für alle besser an!

Der Weg von ADRs zu ODRs

Ausgestattet mit dieser Erfahrung habe ich mir überlegt, dass es ähnlich vielversprechend sein kann, wenn man die gleichen Konzepte und Vorgehensweisen für die Organisationsentwicklung einsetzt und damit für die berühmt berüchtigten Veränderungen. Der Einfachheit halber und als Hommage an die ADRs nenne ich diesen Weg gerne „Organizational Decision Records" (ODRs).

Was ist ein ODR?

Ein ODR sollte genauso schlank sein wie ein ADR und ebenso keine besondere Anforderung an Tool oder Tooling stellen. Wichtig ist, das zu wählen, womit ihr aber auch eure Kollegen sich wohl fühlen, was im Unternehmen häufig für Kollaboration eingesetzt wird und worauf die allermeisten Zugriff haben. Während bei den ADRs zum Beispiel auch Markdown-Dateien im Git-Repository liegen können, wäre das wiederum für ODRs unpraktisch, denn ich kenne nur wenige Kollegen aus Finance oder HR, die Zugriff auf Git-Repos haben oder haben wollen/sollten. In vielen Unternehmen wird es vermutlich Confluence oder ein anderes Wiki sein.

Kurz zum Aufbau und ein Vorschlag für ein sehr einfaches Template für ODRs – exemplarisch ausgefüllt mit einem bewusst nicht ernst gemeinten Beispiel.

Das Template soll nur eine Anregung für euch sein und deckt die aus meiner Sicht wichtigsten Elemente ab. Erweitert es gerne um

```
Titel (+ID)
# Ein griffiger Titel mit dem die allermeisten direkt
etwas anfangen können.
zum Beispiel: #1 Austausch der Kaffeemaschine mit
einer Softeismaschine

Motivation
# Die kompakte Antwort auf die Frage: Warum sollten
wir das tun?
Der Sommer naht und im Büro wird es so heiß, dass eine
Softeismaschine bestimmt zu einer besseren Stimmung
und Produktivität beiträgt

Kurze Beschreibung
# Hier sollte eine kurze Zusammenfassung stehen, was
die Idee ist? Was dafür getan werden muss, und falls
es Geld kostet auch das Budget.

Letztes Jahr war es im Büro sehr heiß und die Kaf-
feemaschine wurde so selten benutzt, dass sie sogar
eingetrocknet ist und gewartet werden musste. Außer-
dem ist die Kaffeemaschine in die Jahre gekommen und
müsste ohnehin bald erneuert werden.

Eine Softeismaschine kostet rund 500 Euro und könnte
gemeinsam beim Sommerfest aufgebaut werden. Außerdem
würden Kollegen die alte Kaffeemaschine entsorgen.

Mehrwert für uns
# Hier geht es um Vereinfachungen, Einsparpotentiale,
Wohlfühlfaktoren, nicht zuletzt um alles, was die
Produktivität erhöhen kann.

Die Softeismaschine wird der zentrale soziale Anker-
punkt im Büro und Informationen und Ideen können dann
dort schneller und freier ausgetauscht werden.
Außerdem hebt es bestimmt die Stimmung im Sommer: Und
gute Stimmung gleich mehr Leistung.

Mehrwert für unsere Kunden
# Hier geht es um Vereinfachungen, Einsparpotentiale,
Wohlfühlfaktoren nicht zuletzt alles, was die Kunden
zufriedener macht und den Umsatz erhöhen kann.

Stellt Euch mal vor, wie einfach Sales wird, wenn wir
unseren Kunden in Meetings im Sommer Softeis anbieten
würden statt dem üblichen Kaffee und Keksen.

Qualität:
# Als Softwarehersteller sollte sich alles um das
Produkt drehen. Jeder Vorschlag kann implizit oder
explizit Auswirkung auf die Qualität haben. Hier gilt
es, kurz zu beschreiben, ob Auswirkungen zu erwarten
sind und wenn ja, wie man diese adressieren oder ab-
mildern kann.

Die Softeismaschine wird keinen Einfluss auf die
Qualität von Lieferungen oder sonstigen Deliverables
haben.

Nachhaltigkeit:
# Eine wichtige Frage und Check bei jeder Veränderung
sollte sein, ob und wie sie die Organisation heraus-
fordert und ob sie dauerhaft leistbar ist. Kann die
Organisation diese Idee „unendlich" fortsetzen?

Die Softeismaschine stellt keine Belastung oder Meh-
raufwand für die Organisation dar. Der Betrieb ist zu
jeder Jahreszeit möglich.

Umkehrbarkeit:
# Ist diese Veränderung einfach umkehrbar? Und wenn
ja, zu welchem Aufwand/zu welchen Kosten?
```

```
Die Softeismaschine kann jederzeit wieder abgebaut
werden und durch eine neue Kaffeemaschine ersetzt
werden. Vermutlich entstehen auch kaum Mehrkosten,
sofern die Softeismaschine verkauft wird.

Lernen
# kontinuierliches Lernen ist eine zentrale Aufgabe
für jeden von uns und damit natürlich auch für Or-
ganisationen. Hier geht es um die Frage, ob und wie
die Organisation und die Mitarbeiter sich durch den
Vorschlag weiter entwickeln können.

Die Softeismaschine wird nicht Teil des Weiterbildung-
sprogramms.
```

Punkte, die euch wichtig sind oder speziell in eurem Unternehmen zu berücksichtigen sind. Ihr solltet aber genauso auf die Einfachheit achten und nicht wie wild Punkte hinzufügen. Es funktioniert dann gut, wenn die Schwelle, ODRs zu erstellen, möglichst gering ist.

Der letzte Schritt hört sich erstmal einfach an, kann aber der anstrengendste sein. Ihr beruft einen Termin oder Meeting ein oder erweitert die Agenda von bestehenden Formaten, um die eingereichten Vorschläge vorzustellen und abzustimmen, ob und wann sie umgesetzt werden. Warum soll ausgerechnet dieser Schritt anstrengend sein? Weil ihr einige Zeit durchhalten müsst. Eventuell gibt es erstmal keine oder wenige Vorschläge, es gibt mit Sicherheit Kritiker (weil Kritik immer einfacher als ein konstruktiver Vorschlag ist) und es gibt die Mehrheit, die Hoffnung hegt, aber noch schweigt.

Die Transparenz und Nachvollziehbarkeit, warum Veränderungen umgesetzt werden sollen, werden über die Zeit aber dazu führen, dass die „stille Mehrheit" sich aktiver beteiligt, die Kritiker merken, dass der Resonanzkörper fehlt und spätestens mit der ersten umgesetzten Veränderung bewiesen ist, dass sich „endlich was bewegt!"

Quellen

[1] *https://www.theregister.com/2025/02/05/rackspace_vmware_planet9_migration/*

[2] *https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions*

[3] *https://github.com/joelparkerhenderson/architecture-decision-record*

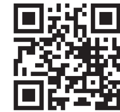
Alexander Domene
G&H Bankensoftware AG
alexander.domene@gmail.com

Alexander Domene ist Vorstand bei der G&H Bankensoftware AG, dort verantwortlich für alles rund um die Produktentwicklung und HR. Er befasst sich schon seit Jahren mit dem Aufbauen, Weiterentwickeln und Beschleunigen von Produktteams. Der innere Nerd ist ihm nie verloren gegangen, darf aber meistens nur nach Feierabend oder am Wochenende raus.

Mitglieder des iJUG

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

www.ijug.eu



Impressum

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

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

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

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

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

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

Bildnachweis:
Titel: Bild © Designed by macrovector
https://freepik.com
S. 2: Bild © DOAG
https://doag.org
S. 10 + 11: Sloth McSloth
https://stock.adobe.com
S. 14 + 15: Bild © U2M Brand
https://stock.adobe.com
S. 18 + 19: Bild © Paper Trident
https://stock.adobe.com
S. 24 + 25: Bild © Designed by freepik
https://freepik.com
S. 30 + 31: Bild © Suriyo
https://stock.adobe.com
S. 36 + 37: Bild © Srinard
https://stock.adobe.com
S. 42 + 43: Bild © Designed by bunny
https://freepik.com
S. 50 + 51: Bild © yourapechkin
https://stock.adobe.com
S. 58 + 59: Bild © Designed by freepik
https://freepik.com
S. 64 + 65: Bild © Christian Horz
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

DOAG e. V. — S. 8-9, U 3, U 4
iJUG e.V. — S. 13, S. 23, S. 49, S. 63
JavaLand GmbH — U 2
Java User Group Stuttgart e.V. — S. 23
JUG Saxony e. V. — S. 33

LINE-UP ONLINE

#CLOUDLAND2025

DAS CLOUD NATIVE FESTIVAL

1. – 4. JULI 2025 • IM HEIDE PARK IN SOLTAU

GOOGLE CLOUD

AWS

HYPER-SCALER

AZURE

- VIELE INSIGHTS FÜR DIE COMMUNITY
- VERTIEFT EUER WISSEN ÜBER HYPERSCALE COMPUTING
- LERNT DIE SCHLÜSSELROLLEN EINER DIGITALEN TRANSFORMATION KENNEN
- OPTIMIERT EUER SYSTEM MIT INSIDERWISSEN

IT-Systemhaus

Red Hat

CloudLand
WWW.CLOUDLAND.ORG

18. – 19. NOVEMBER 2025

LOW-CODE CREATOR

NÜRNBERG CONVENTION CENTER (NCC OST)

LOW-CODE.DOAG.ORG