Java aktuell

iJUG Verbund
www.ijug.eu

Java 24
Was ist neu?
Was ist besser?

Künstliche Intelligenz
AI Red-Teaming, ONNX-Runtime, Azure KI Services

Rund um Java
Gradle, Angular, Microcks, Testcontainers, Open Policy Agent

Ausgabe 03/2025
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
4 191978 307959 03

SUPER SAVER BIS 30.09.2025

JavaLand

10. - 12. MÄRZ 2026

im EUROPA PARK in Rust

#Javaland | www.javaland.eu

Präsentiert von: iJUG Verbund | heise medien | DOAG

Veranstalter: JavaLand

Liebe Leserinnen und Leser,

in dieser Ausgabe werfen wir einen Blick auf das aktuelle Java-Release 24. Falk Sippach geht mit uns alle Neuerungen, Änderungen und Verbesserungen im Detail durch.

Das Java-Tagebuch bringt euch auf den neuesten Stand rund um aktuelle Geschehnisse in der Java-Welt. Teil 7 der Goldenen Regeln widmet sich dem Trend der Agilität. In einem Sonderartikel gibt uns Shabnam Mayel von der Eclipse Foundation einen spannenden Einblick, was es Neues in Jakarta EE gibt und wie es in Zukunft weiter geht.

Weiter geht es dem aktuellen Top-Thema Künstliche Intelligenz. Den Anfang dazu macht Kai Müller ab Seite 28. Er zeigt uns eine Möglichkeit, wie wir mithilfe der ONNX-Runtime Machine-Learning-Modelle direkt in Java-Anwendungen einbetten können. Im Anschluss stellt Rico Komenda das AI Red-Teaming vor und zeigt, wie mithilfe dieses Ansatzes Schwachstellen in KI-Systemen identifiziert und Risiken minimiert werden können. Ziel ist es, dass die Systeme sicher, ethisch vertretbar und mit den Unternehmenszielen übereinstimmen. Johannes Illisei betrachtet mit uns gemeinsam die Microsoft Azure KI Services und präsentiert uns den praktischen Einsatz in einigen Beispielszenarien.

Yuna Morgenstern widmet sich in ihrem Beitrag dem populären Build-Tool Gradle und zeigt eine Reihe versteckter Kosten und Herausforderungen auf, die häufig übersehen werden. Im zweiten Teil seiner Artikelserie macht sich Jelmen Guhlke den Open Policy Agent zunutze, um ein dynamisches Policy-Management zu etablieren. Dr. Florian Rademacher und Sheila Kolodziej zeigen in ihrem Artikel ab Seite 60 eine Möglichkeit zur leichtgewichtigen und kosteneffizienten Realisierung zuverlässiger Testumgebungen für Integration-Testing. Dafür verwenden die beiden Microcks und Testcontainers. Im zweiten und letzten Teil ihrer Artikelreihe beleuchtet das Autorentrio Stephan Rauh, Alexander Pahn und Julian Schmitt weitere Neuigkeiten rund um Angular. Besonderen Fokus legen sie dabei auf die Themen Change Detection und Dependency Injection.

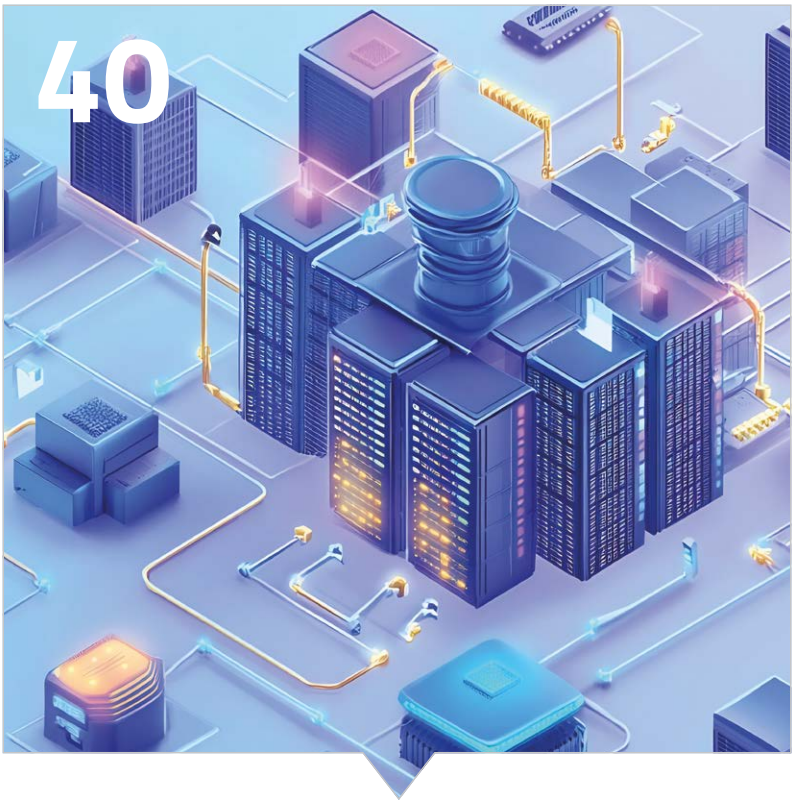
Wir wünschen euch viel Spaß beim Lesen!

Lisa Damerow
Redaktionsleitung Java aktuell

INHALT

Java 24: Ein Blick auf das aktuelle Release

Schwachstellenidentifizierung und Risikominimierung bei KI-Systemen mit AI Red-Teaming



Microsoft Azure KI Services: Angebot und praktische Einsatzbeispiele

Testumgebungen für Integration-Testing mit Microcks und Testcontainers

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

18. März 2025

30 Jahre Java

Herzlichen Glückwunsch zum 30. Geburtstag, Java! Eigentlich hättest du ja Oak heißen sollen, aber wie dein „Vater" James Gosling selbst gesagt hat: „There are only two hard things in computer science: cache invalidation and naming things." Aus Markenrechtsgründen brauchten er und sein Team also einen zweiten Anlauf – und ehrlich gesagt klingt Java auch viel cooler – nicht nur wegen der Kaffee-Anspielungen. Gosling werden aber auch (mindestens) zwei weitere wichtige Zitate zugeschrieben, die sich auf dich beziehen: „The beauty of Java lies in its ability to adapt and evolve with the changing needs of the industry." und ebenso: „If you want to go fast, go alone. If you want to go far, go together." Beides ist absolut wahr und hat deinen Erfolg maßgeblich bestimmt. Welche Programmiersprache hat ein derart großes Ökosystem um sich herum? Und wie viele sind schon ähnlich lange „auf dieser Welt" und gleichzeitig immer noch aktuell – und passen sich weiter an die Bedürfnisse dieses Ökosystems und vor allem der professionellen Software-Entwicklung an? Ich muss gestehen, dass ich dir in der jüngeren Vergangenheit öfter mal untreu geworden bin: mit Python, weil damit im ML-Umfeld einfach viele Dinge schneller von der Hand gehen; aber jedes Mal, wenn ich wieder wegen irgendwelcher Inkompatibilitäten zwischen Bibliotheken und Sprachversionen einen Nervenzusammenbruch bekomme, weiß ich, was ich seit 30 Jahren an dir habe!
Also bleib, wie du bist – auf viele weitere Jahre!

Veröffentlichung von JDK 24

Pünktlich zur Java-Geburtstagsfeier wurde das JDK 24 heute auf der JavaOne 2025 in der Oracle-Zentrale in Redwood Shores freigegeben und bringt viele spannende Features mit. Zum Beispiel den JEP 485 „Stream Gatherers" mit seinen neuen Möglichkeiten der Stream-Transformation. Einige sind allerdings nicht mehr ganz neu, insbesondere die Vector API, die in der mittlerweile neunten Inkubator-Version dabei ist (– rekordverdächtig). Im Performance-Bereich hat sich auch einiges getan – zum Beispiel der „Generational Mode", der jetzt experimentell auch für den Shenandoah Garbage Collector eingeführt wurde; das „Ahead-of-Time Class Loading & Linking" für schnellere Startup-Zeiten; oder das schon im letzten „Tagebuch" erwähnte signifikant reduzierte „Pinning" virtueller Threads bei synchronisierten Methoden (JEP 491), um Durchsatz und Speicherverbrauch insbesondere unter hoher Last zu optimieren. Passend dazu zeichnet auch der „Flugschreiber" JFR jetzt detailliertere Einblicke in Thread- und Speicheraktivitäten auf. Eclipse Temurin und andere Builds auf OpenJDK-Basis dürften wie üblich noch ein paar Tage benötigen, bevor sie zum Download bereitstehen.

Veröffentlichung von GraalVM für JDK 24

Parallel zum Release von JDK 24 hat Oracle die für dieses JDK optimierte Version von GraalVM veröffentlicht. Zu den Highlights gehört die Einführung von „SkipFlow", einem neuen Optimierungsansatz, der die Größe von Native Images weiter reduziert und gleichzeitig die Performance verbessert. Zudem wurden die Graal-Sprachen wie JavaScript, Python und Ruby mit neuen Features und besserer Interoperabilität ausgestattet. Im Machine-Learning-Umfeld gibt es auch spannende Neuigkeiten: die eingebaute Unterstützung für das Model Context Protocol (MCP) für eine einfachere Integration in KI-„Workflows". Wie andersherum ML eingesetzt wird, um die Performance von Native Images zu verbessern, ist in einem Blog-Eintrag zum Release beschrieben: „GraalVM verwendet ein vortrainiertes ML-Modell, um die Ausführungswahrscheinlichkeiten der Kontrollflussdiagramm-Zweige vorherzusagen." *[1]*

20. März 2025

MicroProfile und Jakarta EE: Hochzeit in Sicht?

Es kommt wieder Bewegung in die Beziehung zwischen MicroProfile und Jakarta EE, teils konkurrierende Projekte, aber allein personell schon immer in „freundschaftlicher Kooperation" verbunden. In einem Meeting des MicroProfile Steering Committees wurde vorgestern der Vorschlag diskutiert, das eigene Projekt in das Jakarta-EE-Projekt zu integrieren, das heißt, auch die eigene Working Group innerhalb der Eclipse Foundation aufzugeben. Inhaltlich würde daraus dann ein Jakarta MicroProfile in Analogie zum Core oder WebProfile entstehen. Ein Vorteil wäre auf jeden Fall die Synergie, zumal eh immer zu viel Arbeit auf zu wenigen Schultern lastet. Aber die Diskussionen stehen erst am Anfang. Ein Thema ist die Marke MicroProfile, die über viele Jahre aufgebaut wurde. Ein anderer wesentlicher Aspekt sind die – auch gewollten – unterschiedlichen Geschwindigkeiten und Dynamiken der beiden Projekte: Der „Innovator" MicroProfile und der „Standardisierer" Jakarta EE. Dem soll wohl unter anderem durch eine

weiterhin eigenständige „Release-Kadenz" einer MicroProfile-Platform-Spezifikation Rechnung getragen werden [2] [3].

27. März 2025

Quarkus 3.20 LTS

Das nächste Quarkus-LTS-Release 3.20 ist veröffentlicht worden. Es basiert feature-technisch auf dem Maintenance Release 3.19.4 – die parallel weiter entwickelten Features sind im gleichzeitig veröffentlichten Non-LTS-Release 3.21 gelandet. Quarkus bringt alle sechs Monate ein Long-Term-Support-Release heraus, das dann jeweils für zwölf Monate Bug und Security Fixes erhält. Die letzte Neuerung war, dass auch die hier sogenannten LTS „Micro-Releases" (also zum Beispiel 3.20.1, 3.20.2) jetzt in regelmäßigen Abständen kommen – alle zwei Monate. Alles andere, auch die monatlichen Minor Releases, bleibt gleich – wobei kleine Additionsfehler scheinbar nicht auszuschließen sind: Nach 3.8 LTS kam 3.15 LTS (+7), mit 3.20 (+5) wurde das dann nummerisch wieder korrigiert. Also das nächste LTS-Release nicht im Kopf ausrechnen, sondern immer schön auf den offiziellen LTS-Stempel achten.

Neue Features gibt's auch, dazu muss man dann auf Release 3.19 beziehungsweise alles seit 3.15 LTS schauen: Ganz neu ist zum Beispiel die Micrometer-Bridge zu OpenTelemetry; schon mit 3.18 kam eine komplette Neu-Implementierung der WebAuthn-Erweiterung auf Basis von WebAuthn4J, und eine Reihe von Verbesserungen für OIDC; 3.17 brachte unter anderem den Microprofile REST Client 4.0; und weitere Observability-Verbesserungen ziehen sich komplett durch [4].

4. April 2025

JavaLand-Rückblick

Die JavaLand 2025 am Nürburgring war erneut ein Highlight für die Java-Community. Knapp 1.200 Teilnehmerinnen und Teilnehmer kamen zusammen, um sich über die neuesten Trends und Technologien auszutauschen. Den kostenlosen Live-Stream von der Hauptbühne nutzten zusätzlich bis zu 800 Online-Gäste gleichzeitig.

Die Hauptkonferenz läutete Adam Bien mit seiner Eröffnungskeynote „30 Jahre Java – Hypes, Akronyme, Buzzwords und Paradigmen" ein – gefolgt von der gewohnt guten Mischung aus Vorträgen und interaktiven Formaten beziehungsweise Community-Aktivitäten. Begleitet natürlich auch wie immer von der Ausstellung und dem Rahmenprogramm, das der Location entsprechend wieder die Herzen aller Rennfahrt-Fans höherschlagen ließ, zum Beispiel mit einer Kart-Runde oder einer „Backstage"-Tour hinter die Kulissen der „Grünen Hölle". Die zweite JavaLand am Nürburgring wird aber vorerst auch die letzte sein. Im nächsten Jahr wird wieder ein Freizeitpark als Kulisse dienen: Dann erstmals der Europa-Park in… tam, tam… Rust! Sorry, den Kalauer konnte ich nicht vermeiden.

20. April 2025

Ausblick auf JDK 25

Seit Ende Februar sind eine ganze Reihe von Java Enhancement Proposals für das OpenJDK 25 angekündigt worden. Einer ist JEP 513 „Flexible Constructor Bodies", der Statements in einem Konstruktor vor dem Aufruf von `super()` oder `this()` erlauben soll. „Scoped Values" (JEP 506) zum Teilen unveränderlicher Daten insbesondere mit Child Threads sind nach mehreren Inkubator- und Preview-Versionen jetzt offizieller Bestandteil der API. JEP 511 bringt „Module Import Declarations", um alle von einem Modul exportierten Pakete mit einer einzigen Deklaration importieren zu können (auf Basis der gleichnamigen Preview in JDK 24 – dort JEP 494). Der „Generational Mode" für den Shenandoah Garbage Collector, in JDK 24 noch experimentell, ist jetzt auch produktionsreif. Eine Überarbeitung der „Structured Concurrency API" für die einfachere Handhabung von parallelen Tasks ist als 5. Preview dabei (JEP 505) [5].

10. Mai 2025

AQAvit-Initiative für Qualitätssicherung

Nichts geht mehr ohne AI, oder? Auch AQAvit, die Qualitätssicherung von Eclipse, will jetzt in Zusammenarbeit mit dem „Google Summer of Code" zwei „AI"-basierte Werkzeuge für die eigene Arbeit erstellen: Den CommitHunter und den GlitchWitcher (mein Favorit – ich gestehe, diesen Tagebucheintrag allein aufgrund des leicht schlüpfrig klingenden Namens geschrieben zu haben). Ersterer soll mit Hilfe von speziell trainierten ML-Modellen Commits aufspüren, die Probleme im Build verursachen – inklusive Performance-Problemen. Letzterer soll mit Hilfe eines „FixCache" beim Fixen von Bugs und anschließendem Testen den Fokus auf die relevantesten (sprich: weiterer Bugs höchstverdächtigen) Code-Teile lenken. Ob dabei tatsächlich Machine Learning (beziehungsweise „AI") eingesetzt wird, ist gar nicht klar. Aber der Marketing-Stempel ist ja auch wichtig… [6] [7].

15. Mai 2025

Microsoft: Entlassungswelle trifft Java-Evangelist

Microsoft spart 3 % seiner globalen Belegschaft ein und entlässt zirka 6.000 Angestellte, um mehr Geld für seine KI-Unternehmungen übrig zu haben. Das macht auch etwas mit den in den letzten Jahren umfangreichen Java-Aktivitäten – inhaltlich wie personell. Mindestens einen Prominenten hat es bereits „erwischt": Reza Rahman, unter anderem Jakarta EE Ambassador und bei Microsoft „Principal Program Manager for Java on Azure", wird den Konzern verlassen, will sich aber weiterhin im Java-Ökosystem engagieren. Hoffen wir, dass der Kahlschlag nicht zu arg wird.

17. Mai 2025

JavaOne Digest

Wer sich für die jüngsten Performance-Verbesserungen im JDK interessiert, ob bei der Garbage Collection, im JIT-Compiler, oder den Standardbibliotheken, der sollte sich diesen aufgezeichneten Talk von der JavaOne 25 anschauen [8]. Und dann gibt es noch diesen hier mit dem Titel „Java and AI" und der steilen (?) These: „AI shops grow up to be Java shops", in Abwandlung des älteren Slogans „Web shops grow up to be Java shops". Der Referent (ein Architekt aus der Java Platform Group bei Oracle) schränkt es ein bisschen ein: Es geht ihm um „Model Deployment" (und Nutzung), weniger um „Model Development" (diese Domäne dürfte noch lange von Python domi-

niert werden); damit klingt es deutlich realistischer. Die betrachteten Themen umfassen größtenteils JEPs aus den letzten OpenJDK Releases: Die Foreign Function and Memory API, die Vector API, Value Classes and Objects, und Code Reflection-Verbesserungen aus Project Babylon, mit denen (ähnlich Abstract Syntax Trees) auch auf die Bodies von Methoden/Lambdas zugegriffen werden kann *[9]*.

20. Mai 2025

JSON-API für das JDK in Planung

Das JDK soll jetzt auch eine native JSON-API erhalten. Die Pläne sind Teil von Projekt Babylon *[10]*. Eigentlich hatte ich erwartet, dass JSON jetzt komplett übersprungen wird, zugunsten von YAML. Ok, nicht ernst gemeint. Das Demo-Projekt – fast hätte man es erwartet – nutzt die neue API zur Kommunikation mit einem ML-Modell, dabei werden auch Unmengen an JSON durch die Gegend geschaufelt *[11]*.

30. Mai 2025

Fortschritte bei Jakarta EE

Was lange währt… Jakarta EE 11 soll nun wirklich kurz vor der Freigabe stehen (im Dezember war nur das Core Profile so weit). Der entsprechende Pull Request für das Plattform Release ist im Review und die TCK-Tests mit GlassFish auf Java 17 sowie 21-Basis laufen.

Jakarta EE 12 macht währenddessen langsam Fortschritte. Ziel-Termin für das Release ist das erste Halbjahr 2026. Neu dabei ist dann vermutlich auch Jakarta Query, das als Basis für Jakarta Persistence und das kürzlich dazugekommene Jakarta Data die gemeinsame objektorientierte Query Language für beide spezifizieren soll. Dahinter steht Gavin King, der hinreichend aus dem Hibernate-Projekt bekannt ist (und CDI und viele mehr).

Quellen

[1] *https://medium.com/graalvm/welcome-graalvm-for-jdk-24-7c829fe98ea1*
[2] *https://groups.google.com/g/microprofile/c/6HI5JjrZi3c*
[3] *https://docs.google.com/presentation/d/1ZXSArAU8g2FbkKJP-UOcukIIorFc0ZCFMUa_UkfjvWo*
[4] *https://quarkus.io/blog/quarkus-3-20-0-released*
[5] *https://openjdk.org/projects/jdk/25/*
[6] *https://gitlab.eclipse.org/eclipsefdn/emo-team/gsoc-at-the-ef/-/issues/11*
[7] *https://gitlab.eclipse.org/eclipsefdn/emo-team/gsoc-at-the-ef/-/issues/12*
[8] *https://www.youtube.com/watch?v=q_rmzH78WXI&list=PLX8CzqL3ArzVV1xRJkRbcM2tOgVwytJAi*
[9] *https://www.youtube.com/watch?v=-XnyJad88Ss*
[10] *https://openjdk.org/projects/babylon/*
[11] *https://mail.openjdk.org/pipermail/core-libs-dev/2025-May/145905.html*

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").

BIS ZUM 01.10.2025
EARLY BIRD TICKETS
SICHERN

KI Navigator

DER NAVIGATOR ZUR ANWENDUNG VON KI

→ 19. + 20. NOVEMBER 2025

IN NÜRNBERG

PROGRAMM IST ONLINE

KINAVIGATOR.EU

Die goldenen Regeln: Teil 7

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 7: Sei „agil"

Wasserfall ist böse, Agilität ist toll. Seit Jahren, gar seit Jahrzehnten, versuchen sie uns schon, es beizubringen. Aber ist das wirklich so? Wenn der Wasserfall durch ist, hat man am Ende einen großen Haufen Mist. Macht man das ganze agil und iterativ, hat man zwar am Anfang nur ein kleines Häufchen, aber der Mist wächst nach und nach. Mit jeder Iteration. Inkrementell. Oder vielleicht nennen wir es an dieser Stelle lieber „exkrementiell".

Aber es kann ja sein, dass in deinem Unternehmen jemand auf die Idee kommt, dass die Softwareentwicklung agil nach Scrum oder Ähnlichem arbeiten soll, und dann musst du halt mitmachen, die bezahlen dich schließlich.

Also spielst du das Spiel mit, und je nachdem, welche Rolle du im Projekt hast, gibt es verschiedene Wege, dein Unwesen zu treiben. Aber nicht jede Rolle ist da geeignet. Solltet ihr Scrum einsetzen, und du bist als Agile Coach beziehungsweise Scrum Master unterwegs, solltest du vom agilen Manifest und allem, was mit Scrum zu tun hat, überzeugt sein. Im Idealfall.

Aber was hat das nun mit Neulingen zu tun? Nun, unter Umständen haben sie noch nie ein agiles, geschweige denn ein Scrum-Projekt begleitet, vielleicht haben sie davon nur in der Theorie oder irgendwelchen agilen Simulationen gehört. Und hier kannst du ansetzen, du impfst ihnen direkt am ersten Tag ein, wie schlecht das alles ist, und da sie ja nach Orientierung suchen, werden sie für deine Meinungen offen sein, und für das, was du tust. Wie kannst du nun in der praktischen Anwendung noch weiter dafür sorgen, dass der Neuling ein falsches (bziehungsweise aus deiner Sicht richtiges) Bild von Agilität übernimmt.

Bleiben wir doch bei Scrum, denn das wird in der großen weiten Projektwelt am häufigsten verwendet. Und da können wir schon bei den Ritualen, die so anfallen, Verwüstung anrichten. Ein Daily, welches im Normalfall vielleicht 15 Minuten dauert, kann man durchaus ausarten lassen. Und bei einem Sprint Planning oder gar einer Retrospektive kann man auch immer wieder den Prozess in Frage stellen, weil „früher hat doch alles genauso gut geklappt, und außerdem sollte man doch mal den gesunden Menschenverstand gebrauchen." Hier kann man auch sehr gut den Agile Coach persönlich angreifen.

Soeben fiel auch schon das Wort „Sprint". Ein Sprint ist ein Zeitraum, in dem eine gewisse Anzahl an Features in Form von Stories umgesetzt werden soll, um ein Ziel, das Sprintziel, zu erreichen. Und hier kann der Spaß beginnen: Eigentlich sollte eine Story immer innerhalb eines Sprints abgeschlossen werden können. Aber das ist dir egal, du ziehst sie über mehrere Sprints hinweg, da du dich nicht nur auf eine Sache konzentrieren willst. Aber es wird noch besser: Um Sprints planbarer zu machen, versieht man sie mit sogenannten Storypoints, daraus lässt sich dann eine teamspezifische Velocity erzeugen und vieles mehr. Agile Coaches werden sagen: Storypoints stellen die Komplexität einer Story dar! Aber was ist denn schon Komplexität? Nein, deiner Meinung nach entspricht ein Storypoint einem Personentag, und das gibst du deinem Schützling, dem Neuling, so auch weiter. Das führt am Ende selbstverständlich auch dazu, dass du, oder vielleicht auch dein Team, dafür sorgst, dass Sprints regelmäßig überschätzt werden und damit am Ende gerissen werden. Da dies alles in der Regel ohne Konsequenz bleibt, wird sich auch der Neuling irgendwann denken: „Was soll das alles?" Ja, genau, das ist die Frage.

Damit endet diese Reihe. Fast. In den letzten sieben Beiträgen hast du das Handwerkzeug bekommen, damit du Neulingen definitiv das Leben schwerer machen kannst, sie gleichzeitig für ihr weiteres Berufsleben prägst (insbesondere, wenn es um Agilität geht), oder wie du mit ihnen den Erfolg deines Projekts gefährden kannst. Im Grunde kann man diese Reihe beliebig fortsetzen, aber wie heißt es so doch immer: Wenn es am schönsten ist, sollte man aufhören. Aber es wird noch einen Nachklapp geben: In der nächsten Ausgabe schauen wir mal, wie real diese Regeln wirklich sind, denn es gibt da durchaus valide Statistiken…

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.

Community-Konferenz organisiert von Java User Groups aus dem Norden

JAVA FORUM NORD

Hannover Congress Centrum - Dienstag, 16. September 2025

Das Java Forum Nord ist die Community Konferenz organisiert von Java User Groups (nicht nur) aus dem Norden. Informiere dich in unzähligen Vorträgen von hochkarätigen Speakern über neuste Technologien und die aktuellen Entwicklungen im Java Umfeld.

Das Java Forum Nord ist eine eintägige, nicht-kommerzielle Konferenz in Norddeutschland mit Themenschwerpunkt "Java für Entwickler und Entscheider". Mit über 30 Sessions wird ein vielfältiges Programm geboten. Der regionale Bezug bietet zudem interessante Networkingmöglichkeiten.

http://javaforumnord.de @JavaForumNord@ijug.social

Java 24 – Diese Neuerungen gibt es zum 30. Geburtstag

Falk Sippach, embarc Software Consulting GmbH

Am 23. Mai 1995 wurde Java veröffentlicht. Trotz dieser mittlerweile 30 Jahre ist Java lebendiger denn je. Halbjährig erscheinen neue Versionen. Im März dieses Jahres gab es bei Java 24 sogar einen neuen Rekord: Noch nie wurden seit der Umstellung auf den sechsmonatigen Release-Zyklus so viele JEPs umgesetzt. Darum können wir uns in diesem Artikel einige sowohl neue, aber auch schon länger in der Entwicklung befindliche Themen anschauen.

Für einen ersten Überblick ist wie immer die Projektseite des OpenJDK *[1]* ein guter Startpunkt. Dort sind diesmal immerhin 24 JEPs (JDK Enhancement Proposals) gelistet:

- 404: Generational Shenandoah (Experimental)
- 450: Compact Object Headers (Experimental)
- 472: Prepare to Restrict the Use of JNI
- 475: Late Barrier Expansion for G1
- 478: Key Derivation Function API (Preview)
- 479: Remove the Windows 32-bit x86 Port
- 483: Ahead-of-Time Class Loading & Linking
- 484: Class-File API
- 485: Stream Gatherers
- 486: Permanently Disable the Security Manager
- 487: Scoped Values (Fourth Preview)
- 488: Primitive Types in Patterns, instanceof, and switch (Second Preview)
- 489: Vector API (Ninth Incubator)
- 490: ZGC: Remove the Non-Generational Mode
- 491: Synchronize Virtual Threads without Pinning
- 492: Flexible Constructor Bodies (Third Preview)
- 493: Linking Run-Time Images without JMODs
- 494: Module Import Declarations (Second Preview)
- 495: Simple Source Files and Instance Main Methods (Fourth Preview)
- 496: Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism
- 497: Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm
- 498: Warn upon Use of Memory-Access Methods in sun.misc.Unsafe
- 499: Structured Concurrency (Fourth Preview)
- 501: Deprecate the 32-bit x86 Port for Removal

Ob die Zahl extra auf 24 JEPs angehoben wurde, lässt sich nicht genau sagen. Aber es ist zum Jubiläum zumindest ein netter Zufall.

Vereinfachungen in der Entwicklung mit Java

Trotz oder vielleicht gerade aufgrund seiner 30 Jahre zieht Java auch weiter viele Programmieranfänger an. Die JEPs 494 (Module Import Declarations) und 495 (Simple Source Files and Instance Main Methods) helfen aber nicht nur Neulingen, sie machen auch erfahrenen Entwicklern das Leben einfacher. Durch die Module Import Declarations lassen sich jetzt alle exportierten Packages eines Moduls auf einmal importieren. Das vereinfacht die Wiederverwendung modularer Bibliotheken. Bei diesem zweiten Preview gibt es zwei Erweiterungen: So wurden Beschränkungen bei den transitiven Abhängigkeiten vom Modul `java.se` (eine Art Aggregator-Modul ohne eigene Packages/Klassen) zu `java.base` aufgehoben. Dadurch kann man nun mit dem Import dieses einen Moduls die gesamte API von Java SE importieren. Außerdem ist es jetzt möglich, dass sogenannte Type-Import-on-Demand-Deklarationen (zum Beispiel `import java.util.*`) vorherige Modul-Import-Deklarationen überdecken. Wenn beispielsweise die Module `java.base` und `java.sql` importiert werden, gäbe es eine Unklarheit beim Verwenden der Klasse Date. Die gibt es als `java.util.Date` und als `java.sql.Date`. Durch die on-demand-Deklaration import `java.util.*` wird in dem Fall `java.util.Date` verwendet.

Der JEP zu „Simple Source Files and Instance Main Methods" hat das Ziel, insbesondere Anfängerinnen und Anfängern den Einstieg in Java zu erleichtern, aber auch erfahrenen Entwicklerinnen und Entwicklern die Möglichkeit zu geben, kleine Anwendungen einfach bauen und ausführen zu können. In dieser vierten Preview soll weiter Feedback gesammelt werden. Unter Beibehaltung der bestehenden Java-Toolchain und nicht mit der Absicht, einen separaten Dialekt für Java einzuführen, lassen sich in simplen Java-Quellcode-Dateien, die nur eine vereinfachte `main`-Methode (ohne Klassen-Deklaration) enthalten, einfache, skriptartige Programme erstellen. Dazu kommt die leichtere Verwendung von Standardein- und -ausgaben durch die neuen statischen Methoden `print()`, `println()` und `readln()` der Klasse `java.io.IO`. Diese wird zum Teil automatisch importiert, womit die Funktionen direkt einsatzbereit sind. Ein Beispiel zeigt *Listing 1*.

```
// > java --source 24 --enable-preview  Main.java
void main() {
  println("Hello, World!");
}
```

Listing 1: Instance Main Method

Mit diesen und vielen anderen (bereits früher im Rahmen des Projekts Amber veröffentlichten Neuerungen) Vereinfachungen könnte Java auch besonders im KI-Bereich den ein oder anderen aus der Python-Welt herüberlocken. Insbesondere die Typsicherheit, aber auch die sowieso schon hohe, aber ständig immer noch besser werdende Performance bei rechenintensiven Operationen sprechen für den Einsatz von Java. In Version 25 im September 2025 werden diese beiden Features dann übrigens finalisiert.

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

Der Grund für die lange Inkubationsphase der Vector API wird in den Zielen des JEP 489 *[2]* erklärt:

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

Man wartet also auf die Reformen am Typsystem. Bei Java ist es aktuell zweigeteilt – mit den primitiven und den Referenztypen (Klassen). Die primitiven Datentypen wurden ursprünglich aus Performanceoptimierungsgründen eingeführt, haben aber im Handling entscheidende Nachteile. Referenztypen sind auch nicht immer die beste Wahl, insbesondere was die Effizienz und den Speicherverbrauch angeht. Es braucht etwas dazwischen, was sich so schlank und performant wie primitive Datentypen verhält, aber auch die Vorzüge von selbst zu erstellenden Referenztypen in Form von Klassen kennt. Schon bald könnten daher aus dem Inkubator-Projekt Valhalla die Value Types (haben keine Identität) und Universal Generics (`List<int>`) ins JDK übernommen werden. Der JEP 401 (Value Classes and Objects) hat es leider noch nicht geschafft und wird wohl auch im OpenJDK 25 noch nicht dabei sein. Dementsprechend werden wir die Vector API wohl auch noch einige Releases als Inkubator- beziehungsweise dann hoffentlich bald als Preview-Feature wiedersehen.

Primitive Type Patterns

Ebenfalls schon lange in der Entwicklung ist das Pattern Matching. Hier wurden immer wieder Teile abgeschlossen, zuletzt die Unnamed Variables & Patterns (JEP 456) in Java 22. Bei den nun im zweiten Preview befindlichen „Primitive Types in Patterns, instanceof, and switch" (JEP 488) geht es um eine Erweiterung, durch die primitive Datentypen wie `int`, `byte` und `double` in allen Pattern-Kontexten (beim `instanceof` und im `switch`) verwendet werden dürfen. Entwickler haben dadurch weniger Limitierungen sowie Sonderfälle und können primitive und Referenzdatentypen auch im Kontext von Type Patterns oder als Komponenten in Record Patterns austauschbar verwenden. Seit dem ersten Preview gibt es keine Änderungen, die JDK-Entwickler wollen aber weiteres Feedback sammeln.

Beim Pattern Matching geht es darum, bestehende Strukturen mit Mustern abzugleichen, um komplizierte Fallunterscheidungen effizient und wartbar implementieren zu können. Ein Pattern ist dabei eine Kombination aus einem Prädikat (das auf die Zielstruktur passt) und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen und damit extrahiert. Die Intention des Pattern Matching ist die Destrukturierung von Datenobjekten, also das Aufspalten in die Bestandteile und das Zuweisen in einzelne Variablen zur weiteren Bearbeitung. Mit `instanceof` und `switch` können wir also überprüfen, ob ein Objekt von einem bestimmten Typ ist, und wenn ja, dieses Objekt einer Variable dieses Typs zuweisen und dann diese Variable in dem folgenden Programmpfad benutzen. Das funktionierte bisher aber nur mit Objekten und ließ sich nicht mit primitiven Datentypen kombinieren. Einzig im `switch` ließen sich bereits Variablen der primitiven Typen `byte`, `short`, `char` sowie `int` gegen Konstanten matchen und konnten sogar mit den neueren Type Patterns kombiniert werden *(siehe Listing 2)*.

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

Listing 2: Variablen mit primitiven Datentypen

JEP 488 verbessert nun die Typprüfung, Performance und Lesbarkeit in Java und macht Pattern Matching konsistenter, indem es primitive Typen direkt unterstützt. Dies reduziert überflüssiges Autoboxing und vereinfacht den Umgang mit primitiven Datentypen in `switch`-Anweisungen. Wie am Beispiel in *Listing 3* zu sehen ist, können Entwickler in Zukunft ganz einfach prüfen, ob ein ganzzahliger Wert in den Wertebereich eines `bytes` passt.

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

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

Listing 3: Prüfung auf primitiven Datentyp

Finalisierung der Class-File API und der Stream Gatherers

Im aktuellen OpenJDK-Release wurden auch wieder frühere Previews finalisiert. Dazu zählt zum Beispiel die Class-File API (JEP 484). Sie ermöglicht das Lesen und Schreiben von `class`-Dateien, also von kompiliertem Bytecode. Sowohl das JDK selbst als auch viele Bibliotheken und Frameworks haben für diese Aufgabe bisher auf ASM gesetzt, ein universelles Java-Bytecode-Manipulations- und Analyse-Framework *[3]*. Es kann sowohl zum Modifizieren existierender sowie zum dynamischen Generieren von Klassen im Binärformat verwendet werden. Neben dem OpenJDK kommt ASM unter anderem auch beim Groovy- und Kotlin-Compiler, einigen Test-Coverage-Tools (Cobertura, JaCoCo) und Build-Management-Werkzeugen (Gradle) zum Einsatz. Mockito verwendet es indirekt, um per Byte Buddy Mock-Klassen zu generieren. Aufgrund der kürzeren Release-Zyklen des OpenJDK fällt es nicht leicht, dass ASM mit den Änderungen des Bytecodes mithält. Das führt dann wiederum zu Abhängigkeiten und somit können die oben genannten Tools und Frameworks nicht schnell genug mit neuen OpenJDK-Releases umgehen. Mit der Entwicklung der JDK-internen Class File API sollen solche Abhängigkeiten minimiert werden.

ASM hat bestehenden Bytecode auf Basis des Visitor-Patterns analysiert und modifiziert. Das Visitor-Pattern ist aber sperrig und unflexibel. Durch die direkte Unterstützung von Pattern Matching in Java kann der notwendige Code in der neuen Class File API direkter und konsistenter ausgedrückt werden. *Listing 4* zeigt ein Beispiel, das auch im JEP 484 beschrieben ist.

```
CodeModel code = ...
Set<ClassDesc> deps = new HashSet<>();
for (CodeElement e : code) {
  switch (e) {
     case FieldInstruction f   -> deps.add(f.owner());
     case InvokeInstruction i -> deps.add(i.owner());
     ... and so on for instanceof, cast, etc ...
  }
}
```

Listing 4: Parsen von Klassen mit Pattern Matching

Das Erzeugen von Klassen erfolgt im Gegensatz zum Visitor-Ansatz mit Buildern. Um beispielsweise eine Methode `foobar` *(Listing 5)* zu erzeugen, kann das ebenfalls aus dem JEP 484 entnommene Code-Beispiel verwendet werden *(Listing 6)*.

```
void fooBar(boolean z, int x) {
  if (z)
     foo(x);
  else
     bar(x);
}
```

Listing 5: Zu erzeugende Methode

Es kann auch existierender Code verändert werden. *Listing 7* zeigt beispielhaft, wie bei einer bestehenden Klasse alle Methoden gelöscht werden, die mit `debug` beginnen.

Nach den zwei Previews (in Java 22 und 23) wurde die Class-File API nun finalisiert. Es gab nur ein paar kleine Änderungen, die im JEP nachverfolgt werden können.

Streams um benutzerdefinierte Intermediate Operationen erweitern

Die Stream-API wurde in Java 8 eingeführt. Ein Stream wird erst bei Bedarf ausgewertet und kann potenziell eine unbeschränkte Anzahl

```
ClassFile cf = ClassFile.of();
ClassModel classModel = cf.parse(bytes);
byte[] newBytes =
  cf.build(
     classModel.thisClass().asSymbol(),
     classBuilder -> {
        for (ClassElement ce : classModel) {
           if (!(ce instanceof MethodModel mm
                 && mm.methodName().stringValue()
                    .startsWith("debug"))) {
                    classBuilder.with(ce);
           }
        }
     });
```

Listing 7: Bestehende Klasse transformieren

von Werten enthalten. Sie können sequenziell oder parallel verarbeitet werden. Eine Stream Pipeline besteht typischerweise aus drei Teilen: der Quelle, ein oder mehreren Intermediate Operationen und einer abschließenden Operation *(siehe Listing 8)*.

Im JDK gibt es eine begrenzte Anzahl an vordefinierten Intermediate Operationen wie `filter`, `map`, `flatMap`, `mapMulti`, `distinct`, `sorted`, `peak`, `limit`, skip, takeWhile und dropWhile. Es kommt immer wieder der Wunsch nach weiteren Methoden wie `window` oder `fold` auf. Aber anstatt nur genau die geforderten Operationen bereitzustellen, wurde eine API (Stream Gatherers) entwickelt und im JDK 24 nun final bereitgestellt. Sie ermöglicht es sowohl den JDK-Entwicklern als auch den normalen Anwendern, beliebige Intermediate Operations selbst zu implementieren.

Es werden die folgenden Gatherer mitgeliefert, erreichbar über die Klasse `java.util.stream.Gatherers` *(Listing 9 zeigt einige Beispiele)*:

```
CodeBuilder classBuilder = ...;
classBuilder.withMethod("fooBar",
  MethodTypeDesc.of(CD_void, CD_boolean, CD_int),
  flags,
  methodBuilder -> methodBuilder
     .withCode(codeBuilder -> {
        codeBuilder
           .iload(codeBuilder.parameterSlot(0))
           .ifThenElse(
              b1 -> b1.aload(codeBuilder.receiverSlot())
                      .iload(codeBuilder.parameterSlot(1))
                      .invokevirtual(
                         ClassDesc.of("Foo"),
                         "foo",
                         MethodTypeDesc.of(CD_void, CD_int)),
              b2 -> b2.aload(codeBuilder.receiverSlot())
                      .iload(codeBuilder.parameterSlot(1))
                      .invokevirtual(
                          ClassDesc.of("Foo"),
                             "bar",
                             MethodTypeDesc.of(
                                CD_void, CD_int))
           .return_();
});
```

Listing 6: Erzeugen der Methode aus Listing 5

```
var number = Arrays.asList("abc1", "abc2", "abc3").stream() // Quelle
        .skip(1) // 1. Intermediate Operation
        .map(element -> element.substring(0, 3)) // 2. Intermediate Operation
        .sorted() // 3. Intermediate Operation
        .count(); // Terminal Operation
System.out.println(number);
```

Listing 8: Verschiedene Stufen der Stream Pipeline

- `fold`: zustandsbehafteter N-1-Gatherer, baut ein Aggregat inkrementell auf und gibt dieses Aggregat am Ende zurück
- `mapConcurrent`: zustandsbehafteter 1-1-Gatherer, der die übergebene Funktion nebenläufig für jedes Input-Element aufruft
- `scan`: zustandsbehafteter 1-1-Gatherer, wendet eine Funktion auf dem aktuellen Zustand und dem aktuellen Element an, um das nächste Element zu erzeugen
- `windowFixed`: zustandsbehafteter N-N-Gatherer, der Eingabeelemente in Listen vorgegebener Größe gruppiert
- `windowSliding`: ähnlich zu `windowFixed`, nach dem ersten Rahmen wird der nächste Rahmen erzeugt, in dem das erste Element gelöscht wird und alle weiteren Werte nachrutschen

Der Blick auf den Aufbau eines Stream Gatherers zeigt die Funktionsweise. Er kann einen Status besitzen, sodass Elemente abhängig von den vorherigen Aktionen unterschiedlich transformiert werden können. Und er kann einen Stream auch vorzeitig terminieren, wie beispielsweise `limit()` oder `takeWhile()`. Der Gatherer startet mit einem optionalen Initializer, der den Status bereitstellen soll. Dann folgt der Integrator, der jedes Element des Streams verarbeitet und gegebenenfalls den Status aktualisiert. Dann folgt der optionale Finisher, der nach der Verarbeitung des letzten Elements aufgerufen wird und je nach Status weitere Elemente an die nächste Stufe der Stream-Pipeline sendet. Und zu guter Letzt kombiniert ein optionaler Combiner den Status von parallel ausgeführten Transformationen.

Um eine eigene Implementierung zu schreiben, muss man vom Interface Gatherer ableiten und mindestens die Methode `integrator()` implementieren. Die anderen bringen eine Default-Implementierung mit und sind daher optional *(siehe Listing 10)*.

```
interface Gatherer<T, A, R> {
     default Supplier<A> initializer() {
           return defaultInitializer();
     };

     Integrator<A, T, R> integrator();

     default BinaryOperator<A> combiner() {
           return defaultCombiner();
     }

     default BiConsumer<A, Downstream<? super R>> finisher() {
           return defaultFinisher();
     };
     [...]
}
```

Listing 10: Interface Gatherer

Der JEP 485 zeigt ein vollständiges Beispiel für den Gatherer `windowFixed`. Die Stream Gatherers erhöhen durch ihre Ausdrucksstärke nicht nur die Lesbarkeit und Modularität des Codes. Sie erlau-

```
// will contain: Optional["12345"]
Optional<String> numberString =
           Stream.of(1, 2, 3, 4, 5)
                      .gather(
                                 Gatherers.fold(() -> "", (string, number) -> string + number)
                      )
                      .findFirst();
System.out.println(numberString);

// will contain: ["1", "12", "123"]
List<String> numberStrings = Stream.of(1, 2, 3).gather(
                Gatherers.scan(() -> "", (string, number) -> string + number)
           ).toList();
System.out.println(numberStrings);

// will contain: [[1, 2, 3], [4, 5, 6], [7, 8]]
List<List<Integer>> windows =
           Stream.of(1, 2, 3, 4, 5, 6, 7, 8).gather(Gatherers.windowFixed(3)).toList();
System.out.println(windows);

// will contain: [[1, 2], [2, 3], [3, 4], [4, 5]]
List<List<Integer>> windows2 =
           Stream.of(1, 2, 3, 4, 5).gather(Gatherers.windowSliding(2)).toList();
System.out.println(windows2);

// will contain: [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
List<List<Integer>> windows3 =
           Stream.of(1, 2, 3, 4, 5).gather(Gatherers.windowSliding(3)).toList();
System.out.println(windows3);
```

Listing 9: Mitgelieferte Gatherers

ben auch komplexe Operationen ohne zusätzliche Transformationen oder Collect-Schritte und ermöglichen es, insbesondere mehrere, möglicherweise zustandsbehaftete Intermediate-Operationen ohne teure Zwischenschritte effizient auszuführen.

Flexible Konstruktor Inhalte

Dank des JEP 492 (Flexible Constructor Bodies) dürfen in Konstruktoren Anweisungen nun bereits vor einem expliziten Konstruktoraufruf (`super()` oder `this()`) erscheinen. Diese Anweisungen dürfen zwar nicht auf die zu konstruierende Instanz verweisen, können jedoch Parameter validieren beziehungsweise transformieren oder auf Felder der Oberklasse zugreifen. Die Initialisierung von Feldern vor dem Aufruf des Super-Konstruktors macht die Klasse zuverlässiger, wenn beispielsweise Methoden überschrieben werden müssen. Insgesamt erhöht sich sowohl die Lesbarkeit als auch die Performance, da unnötige Aufrufe bei der Validierung oder Weiterverarbeitung von Konstruktor-Parametern vermieden werden. In diesem dritten Preview gibt es keine nennenswerten Änderungen, auch hier möchte man weiter Feedback sammeln.

Listing 11 zeigt ein Beispiel für die Validierung und die Transformation von Eingabeparametern und das Aufspalten eines Parameters in Einzelteile.

Neuerungen im Umfeld von Virtual Threads

Für die in Java 21 finalisierten virtuellen Threads sind weiterhin APIs (JEP 487 – Scoped Values und JEP 499 – Structured Concurrency) in der Bearbeitung, die eine effizientere, sichere und besser strukturierte Nebenläufigkeit ermöglichen sollen.

Ein `ScopedValue<T>` ist eine Alternative zu `ThreadLocal<T>`, die speziell für Virtual Threads optimiert wurde. Er ermöglicht es, unveränderbare Werte sicher über einen Code-Bereich hinweg zu propagieren, ohne dabei die Nachteile von `ThreadLocal` zu übernehmen. Diese sind bei Virtual Threads problematisch, da Speicherplatz nicht automatisch freigegeben wird. Da Virtual Threads sehr leichtgewichtig sind und potenziell tausende parallele Aufgaben ausgeführt werden können, würden `ThreadLocal`-Variablen schlecht skalieren. Denn jeder Thread würde seinen eigenen Wert speichern. Bei `ScopedValues` sind die Inhalte nur innerhalb eines bestimmten Bereichs gültig und verursachen dadurch keine Speicherprobleme. `ScopedValues` sind sicherer, performanter und expliziter in ihrer Lebensdauer. Durch ihre Unveränderbarkeit können keine ungewollten Nebenwirkungen auftreten. *Listing 12* zeigt ein Beispiel, bei dem USER_ID nur innerhalb des `run()`-Blocks gültig ist und dann auch automatisch aufgeräumt wird.

```
class Person {
      private final String firstname;
      private final String lastname;
      private final LocalDate birthdate;

      public Person(String firstname, String lastname, LocalDate birthdate) {
            this.firstname = firstname;
            this.lastname = lastname;
            this.birthdate = birthdate;
      }
}

class Employee extends Person {
      private final String company;

      public Employee(String name, LocalDate birthdate, String company) {
            if (company == null || company.isEmpty()) {
                  throw new IllegalArgumentException("company is null or empty");
            }
            String[] names = name.split("\\s" );
            super(names[0], names[1], birthdate);
            this.company = company;
      }
}

System.out.println(new Employee("Dieter Develop", LocalDate.now(), "embarc"));
```

Listing 11: Flexible Constructor Bodies

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

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

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

Listing 12: Kompilieren und Ausführen von Preview-Funktionen

Die Structured Concurrency (JEP 499) sorgt dafür, dass nebenläufige Tasks innerhalb eines klar definierten Scopes gestartet und beendet werden. Das hilft bei der Fehlerbehandlung (wenn eine Aufgabe fehlschlägt, werden alle anderen koordiniert abgebrochen) und bei der Lesbarkeit (explizite Nebenläufigkeitsstrukturen). Alternativ konnten Entwicklerinnen und Entwickler für diesen Zweck bisher die Parallel Streams, den `ExecutorService` oder reaktive Programmierung einsetzen. Alles sehr mächtige Ansätze, die aber gerade einfache Umsetzungen unnötig kompliziert und fehleranfällig machen. Structured Concurrency behandelt Gruppen von zusammengehörigen Aufgaben als eine Arbeitseinheit, wodurch die Fehlerbehandlung sowie das Abbrechen der Aufgaben vereinfacht und die Zuverlässigkeit sowie die Beobachtbarkeit erhöht werden. *Listing 13* zeigt ein Beispiel, bei dem im Falle eines Fehlers alle laufenden Aufgaben abgebrochen werden. Änderungen zum letzten Preview gibt es keine. Die Macher des OpenJDK wünschen sich vielmehr weitere Rückmeldungen aus der „realen" Welt.

```
try (var scope =
  new StructuredTaskScope.ShutdownOnFailure()) {

  Future<String> task1 = scope.fork(() -> { … }
  Future<String> task2 = scope.fork(() -> { … }
  Future<String> task3 = scope.fork(() -> { … }

  scope.join();
  scope.throwIfFailed();

  System.out.println(task1.get());
  System.out.println(task2.get());
  System.out.println(task3.get());
}
```

Listing 13: Structured Concurrency

Mit dem JEP 491 (Synchronization für Virtual Threads ohne Pinning) wird ein weiteres Problem der Virtual Threads in Angriff genommen. Wenn ein Virtual Thread einen `synchronized`-Block betritt, blockiert er den Carrier-Thread. Das nennt man Pinning: Der Virtual Thread bleibt an einen bestimmten Carrier-Thread „fixiert", bis der `synchronized`-Block verlassen wird. Das reduziert die Skalierbarkeit, da blockierte Carrier-Threads keine anderen Virtual Threads ausführen können. Mit dem JEP 491 wurde die virtuelle Thread-Architektur so angepasst, dass Threads, die auf Monitore warten, nun den zugrunde liegenden Plattform-Thread freigeben können, ohne die Funktionalität von `synchronized` zu beeinträchtigen. Das erhöht die Anzahl der verfügbaren virtuellen Threads für die Bearbeitung von offenen Aufgaben. Diese Änderungen sollen die Effizienz bei der Nutzung von virtuellen Threads erheblich steigern, indem fast alle Fälle von Thread-Pinning vermieden werden. Damit wird eine weitere wichtige Baustelle im Bereich der Virtual Threads geschlossen.

Weitere Änderungen unter der Haube

Damit wir mit Virtual Threads so einfach arbeiten können, musste in den vergangenen Releases viel unter der Haube angepasst werden. Viele der anderen aktuellen Neuerungen beziehen sich auf Themen wie zum Beispiel Garbage Collectors oder Optimierungen beim Start beziehungsweise im laufenden Betrieb von Java-Anwendungen sowie Maßnahmen, die die Robustheit der Java-Plattform steigern. So erhält im JEP 404 der Shenandoah Garbage Collector ähnlich wie der Z Garbage Collector (ZGC) einen „Generational Mode". Dabei wird zwischen neuen und alten Objekten unterschieden. Die junge Generation enthält eher kurzlebige Objekte und wird häufiger bereinigt. Die alte Generation muss dagegen nur selten aufgeräumt werden. Damit verbessert sich der Durchsatz sowie die Widerstandsfähigkeit gegen Lastspitzen und die Speichernutzung wird optimiert. Beim Z Garbage Collector wird mit dem JEP 490 der nicht-generationale Modus nun entfernt, da der generationale Modus inzwischen leistungsfähiger und für die meisten Anwendungen besser geeignet ist. Die Maßnahme soll den Wartungsaufwand reduzieren und die Weiterentwicklung des auf Generationen basierenden Modus beschleunigen. Dies reflektiert den Fokus auf modernere Ansätze bei der Speicherbereinigung, die sowohl Effizienz als auch Wartbarkeit erhöhen sollen.

Der JEP 475 (Late Barrier Expansion for G1) führt eine Optimierung für den Standard Garbage Collector G1 ein, indem Speicherbarrieren später im Kompilierungsprozess generiert werden. Diese Änderung verbessert die Qualität der maschinennahen Instruktionen, reduziert die Komplexität in der Implementierung und erleichtert die Pflege. Die Motivation liegt in der Identifizierung von Schwächen bei der bisherigen Platzierung von Barrieren, insbesondere im Hinblick auf ihren Einfluss auf die Leistung und den Speicherverbrauch. Durch diese spätere Platzierung können unnötige Barrieren eliminiert werden, was zu einer effizienteren Programmausführung beiträgt. Diese Optimierung stellt einen weiteren Schritt dar, die Performance und Wartbarkeit des G1-Garbage-Collectors zu verbessern und langfristig die Anpassungsfähigkeit an moderne Hardwarearchitekturen sicherzustellen.

Der JEP 450 führt kompakte Objekt-Header in Java ein, um Speicherplatz effizienter zu nutzen und den Speicherverbrauch von Java-Objekten zu reduzieren. Der Header, der bisher 128 Bit auf 64-Bit-Plattformen einnahm, wird auf 64 Bit verkleinert. Dies geschieht durch komprimierte Klassenreferenzen und eine optimierte Verwaltung von Synchronisationsdaten. Der Ansatz zielt darauf ab, die Speicherbelastung bei datenintensiven Anwendungen mit vielen kleinen Objekten zu minimieren, ohne die Leistung der JVM zu beeinträchtigen. Da jedoch tiefgreifende Änderungen an grundlegenden Datenstrukturen vorgenommen werden, bleibt das Feature zunächst experimentell, um mögliche Probleme zu identifizieren und Feedback zu sammeln. Dies könnte langfristig die Grundlage für eine noch effizientere Speicherverwaltung in zukünftigen Java-Versionen schaffen.

Der JEP 483 führt das Konzept des Ahead-of-Time (AOT) Class Loading & Linking ein, wodurch die Startzeit und die Effizienz von Java-Anwendungen verbessert wird. Die Idee besteht darin, häufig genutzte Klassen bereits zur Build-Zeit vorzubereiten, statt diese erst zur Laufzeit zu laden und zu verlinken. Dies geschieht durch die Integration einer AOT-Caching-Lösung, in der Metadaten wie Konstanten, Methoden-Handles oder Lambda-Ausdrücke vorgeladen und gespeichert werden. Das Ziel ist es, die Kosten und Latenzen des dynamischen Ladens zur Laufzeit zu minimieren, was insbesondere bei wiederkehrenden Anwendungsfällen in Cloud-nativen oder ressourcenbeschränkten Umgebungen Vorteile bringt. Die Motivation hinter dieser Änderung liegt in der zunehmenden Bedeutung von schnell startenden Anwendungen, die oft auf Containerplattformen

wie Kubernetes ausgeführt werden. In solchen Umgebungen ist die Optimierung von Startzeiten und Speicherverbrauch entscheidend. Durch die Verwendung eines Cache-Mechanismus wird die Effizienz deutlich gesteigert, ohne die Flexibilität des Java-Ökosystems zu beeinträchtigen.

Im Bereich des Toolings wird mit dem JEP 493 (Linking Run-Time Images without JMODs) die Größe einer vom Benutzer erstellten Laufzeitumgebung (JRE) mit jlink um etwa 25 % verringert. Bei der Erzeugung der Images werden keine JMOD-Dateien inkludiert. Diese Funktion muss allerdings bei der Erstellung des JDKs aktiviert werden. Außerdem entscheiden sich einige JDK-Anbieter möglicherweise dafür, diese Option nicht anzubieten. Als Motivation wird darauf verwiesen, dass in Cloud-Umgebungen die installierte Größe des JDK auf dem Dateisystem wichtig ist. Gerade Container-Images, die ein installiertes JDK enthalten, werden automatisch und häufig über das Netzwerk aus Container-Registrierungen heruntergeladen. Eine Verringerung der Größe des JDK würde die Effizienz dieser Vorgänge verbessern.

Alte Zöpfe müssen irgendwann abgeschnitten werden

Java ist bekannt dafür, besonderen Wert auf die Abwärtskompatibilität zu legen. Das führte dazu, dass zwar einerseits auch vor 30 Jahren geschriebene Anwendungen weiter lauffähig sind. Das führt andererseits aber zu vielen Altlasten, die seit vielen Jahren mitgeschleift werden. Doch seit einigen Jahren werden nun nicht nur manche dieser alten Zöpfe als veraltet markiert (deprecated), sondern auch konsequent abgeschnitten.

So zielt zum Beispiel der JEP 472 (Prepare to Restrict the Use of JNI) darauf ab, die Nutzung der Java Native Interface (JNI) zu beschränken, um die Integrität und Sicherheit der Java-Plattform zu erhöhen. JNI erlaubt den Zugriff auf private Felder und Methoden sowie den direkten Speicherzugriff, was grundlegende Prinzipien wie Kapselung und Speichersicherheit untergräbt. Ziel ist es, die Verwendung von JNI standardmäßig einzuschränken, es jedoch für Anwendungen explizit aktivieren zu können, die es benötigen. Dies folgt dem langfristigen Ansatz, Java von unsicheren APIs zu befreien und alternative Mechanismen wie die Foreign Function & Memory API zu fördern. Die Motivation liegt in der Verbesserung der Robustheit, Wartbarkeit und Sicherheit der Plattform, um Risiken wie Speicherkorruption und unvorhergesehenes Verhalten zu minimieren und die Modernisierung von Java-Programmen zu erleichtern.

Durch den JEP 498 (Warn upon Use of Memory-Access Methods in sun.misc.Unsafe) gibt Java 24 eine Laufzeitwarnung aus, wenn erstmals eine der unsicheren Speicherzugriffsmethoden in `sun.misc.Unsafe` aufgerufen wird. Diese Methoden wurden bereits in JDK 23 zur Entfernung markiert. Sie wurden durch sicherere Alternativen ersetzt, zum Beispiel `VarHandle` (JEP 193, JDK 9) für Speicherzugriffe auf dem Heap und `MemorySegment` (JEP 454, JDK 22) für Off-Heap-Speicher. Das Ziel ist es, Entwickler frühzeitig auf die Entfernung dieser Methoden in zukünftigen JDK-Versionen vorzubereiten und sie zum Umstieg auf standardisierte APIs zu bewegen.

Der JEP 479 (Remove the Windows 32-bit x86 Port) entfernt den Windows 32-Bit x86-Port aus dem OpenJDK, da diese Architektur zunehmend veraltet ist und keine neue Hardware mit diesem Format mehr produziert wird. Windows 10, das letzte Betriebssystem mit Unterstützung für 32-Bit-Betrieb, erreicht 2025 das Ende seines Lebenszyklus, was die Relevanz dieser Plattform weiter verringert. Durch die Entfernung dieses Ports können Ressourcen bei der Weiterentwicklung von Java effizienter genutzt werden. Gleichzeitig wird die Wartung durch die Reduzierung von Komplexität vereinfacht. Diese Änderung entspricht den aktuellen Trends in der Industrie, wo 64-Bit-Architekturen klar dominieren. Bestehende Nutzer können weiterhin ältere Versionen des JDK nutzen oder durch den Einsatz von Remote-APIs für 32-Bit-Funktionen migrieren.

Neben Windows werden bald auch andere 32-Bit-Implementierungen entfernt. Mit dem JEP 501 (Deprecate the 32-Bit x86 Port for Removal) wird der 32-Bit-x86-Port in Java 24 als veraltet markiert und auf dessen Entfernung in einer zukünftigen Version vorbereitet. Betroffen ist insbesondere die letzte verbliebene Implementierung für Linux auf 32-Bit-x86. Die Wartung dieses Ports verursacht ebenfalls hohe Kosten und blockiert die Implementierung neuer Features wie Project Loom, die Foreign Function & Memory API oder die Vector API. Nach der Entfernung bleibt als einzige Möglichkeit zur Ausführung von Java-Programmen auf 32-Bit-x86-Prozessoren der architekturunabhängige Zero-Port.

Sicherheitsrelevante und kryptographische Themen

Mit dem JEP 486 wird der Security-Manager dauerhaft deaktiviert. Der Security-Manager wurde häufig zur Absicherung von clientseitigem Java-Code (Rich-Clients, Applets), aber nur selten für die Server-Seite verwendet. Zudem ist seine Wartung teuer. Mit Java 17 (2021) wurde er als „Deprecated for Removal" markiert. Nun wird er intern ausgebaut. Er kann jetzt nicht mehr aktiviert werden und andere Klassen der Java-Plattform verweisen nicht mehr darauf. Diese Änderung wird aber vermutlich keine Auswirkungen auf die große Mehrheit der Anwendungen, Bibliotheken und Tools haben. In einer der zukünftigen Java-Versionen wird die Security-Manager-API endgültig entfernt.

Der JEP 478 führt eine API für Key Derivation Functions (KDFs) als Preview ein, um kryptografische Schlüssel aus einem geheimen Schlüssel und zusätzlichen Informationen zu generieren. Das basiert auf Standards wie RFC 5869 (HMAC-based Extract-and-Expand Key Derivation Function, HKDF). Ziel ist es, eine standardisierte, gut integrierte Lösung für Java-Entwickler bereitzustellen, die interoperabel und vielseitig einsetzbar ist. Die Motivation ist die Erleichterung bei der Umsetzung von Verschlüsselung, Authentifizierung und digitalen Signaturen, da bestehende Methoden oft auf benutzerdefinierten Implementierungen beruhen, die potenziell unsicher oder weniger effizient sind. Die API bietet ein standardisiertes Framework, das die Sicherheit und Benutzerfreundlichkeit erhöht, während es gleichzeitig mit den aktuellen Sicherheitsbibliotheken von Java kompatibel bleibt.

Die JEPs 496 (Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism – ML-KEM) und 497 (Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm – ML-DSA) führen Implementierungen von Algorithmen für Schlüsselaustauschverfahren und digitale Signaturen ein. ML-KEM ist ein von NIST in FIPS 203 standardisierter Algorithmus, der sichere Schlüsselaustauschverfahren gegen zukünftige Angriffe durch Quantencomputer ermöglicht. Dies ist besonders relevant, da Quantencomputer

herkömmliche kryptografische Verfahren wie RSA und Diffie-Hellman in Zukunft brechen könnten. Java 24 unterstützt ML-KEM mit den Parametern ML-KEM-512, ML-KEM-768 und ML-KEM-1024, um die Sicherheit von Anwendungen langfristig zu gewährleisten. ML-DSA ist ein von NIST in FIPS 204 standardisierter Algorithmus zur digitalen Signatur, der ebenfalls gegen zukünftige Angriffe durch Quantencomputer abgesichert ist. Digitale Signaturen werden zur Authentifizierung und zur Erkennung von Datenmanipulationen genutzt. Java 24 unterstützt ML-DSA mit den Parametern ML-DSA-44, ML-DSA-65 und ML-DSA-87, um eine langfristig sichere Signaturprüfung zu ermöglichen.

Fazit und Ausblick

Nach dem Release ist bekanntlich vor dem Release. Bereits im September 2025 wird das OpenJDK 25 *[4]* erscheinen. Da werden wir einige der hier angesprochenen Preview-Themen erneut wiedersehen. Aber es sind auch schon neue Themen geplant, zum Beispiel Stable Values. Das sind Objekte, die unveränderliche Daten enthalten und vom JVM als Konstanten behandelt werden können. Damit ermöglichen sie dieselben Performance-Optimierungen wie bei `final`-Feldern, bieten aber eine höhere Flexibilität in Bezug auf den Initialisierungszeitpunkt. Die Ziele der neuen APIs sind ein schnellerer Start von Java-Anwendungen durch Aufbrechen monolithischer Initialisierung (Lazy Initialization), die Trennung von Erzeugung und Initialisierung ohne nennenswerte Performanceeinbußen, die Garantie, dass Stable Values höchstens einmal initialisiert werden (besonders wichtig bei nebenläufigen Szenarien) und interne Verbesserungen wie zum Beispiel das Constant Folding für benutzerdefinierten Code – eine Optimierung, die bisher nur JDK-intern verfügbar war.

Wer sich vorab über alle zukünftigen Themen informieren möchte, kann sich schon mal im JEP-Index unter „Draft and submitted JEPs" *[5]* umschauen. Ansonsten sind wie immer die Release Notes des OpenJDK 24 *[6]* und der Java Almanac *[7]* ein guter Startpunkt, um auch die vielen kleinen API-Änderungen nachverfolgen zu können.

Java ist und bleibt weiterhin sehr relevant. Während derzeit allerorten noch das 30-jährige Jubiläum gefeiert wird, entwickelt sich die Sprache und vor allem auch die Plattform immer weiter. Auch im Umfeld von KI wird gegenüber Python aufgeholt. Es gibt mittlerweile einige spannende Bibliotheken, die die Einbindung von Large Languages Models ermöglichen. An der dafür nötigen Performance wird es in naher Zukunft weitere Optimierungen geben. An Vereinfachungen für Einsteiger in Java wird auch gearbeitet. Mit den Value Objects steht zudem die nächste große Änderung vor der Tür. Im letzten Jahr haben diverse Mitarbeiter von Oracle auf verschiedenen Konferenzen nach über zehn Jahren der Entwicklung angedeutet, dass man bei der Umsetzung dieses Bindeglieds von primitiven zu Referenztypen mittlerweile den Durchbruch geschafft hat. Das wird die Art, wie wir zukünftig in Java Code schreiben, nochmal gründlich verändern. Und es geht dabei nicht nur um die Value Objects an sich, sondern auch um Null-Restricted Types (ähnliche Idee wie bei Kotlin: `String! name`) und Generics über primitive Datentypen (`List<int>`).

Referenzen

[1] *https://openjdk.java.net/projects/jdk/24/*
[2] *https://openjdk.org/jeps/489*
[3] *https://asm.ow2.io/*
[4] *https://openjdk.java.net/projects/jdk/25/*
[5] *https://openjdk.org/jeps/0#Draft-and-submitted-JEPs*
[6] *https://jdk.java.net/24/release-notes*
[7] *https://javaalmanac.io/jdk/24/apidiff/23/*

Falk Sippach
embarc Software Consulting GmbH
falk@jug-da.de

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

SUPER-SAVER

Bis 31.10.2025

Heide Park Soltau

APEX connect

18. - 20. Mai 2026

+

DOAG 2026 Datenbank

mit Cloud Infrastructure

18. - 19. Mai 2026

Jakarta EE 11: Advancing Cloud Native Java

Shabnam Mayel, Eclipse Foundation

The Jakarta EE [1] community has reached significant milestones with the release of Jakarta EE 11 Core Profile and Web Profile, setting a strong foundation for cloud-native Java development. These updates modernize enterprise Java with lightweight runtimes, modular APIs, and enhanced compatibility testing through updated Technology Compatibility Kits (TCKs). Together, they mark a major step in making enterprise Java more efficient, agile, and aligned with today's deployment needs.

Jakarta EE 11 Core Profile: A Lean Foundation

Released in December 2024, the Jakarta EE 11 Core Profile *[2]* caters to microservices and lightweight runtimes. It includes essential specifications such as Jakarta Contexts and Dependency Injection (CDI) 4.1, Jakarta RESTful Web Services 4.0, Jakarta JSON Processing 2.1.2, Jakarta Annotations 3.0, and Jakarta Interceptors 2.2. This profile is designed for applications that require a minimal set of functionalities, promoting faster startup times and reduced resource consumption.

The Core Profile aligns with the industry's shift towards microservices and serverless architectures. By providing a streamlined set of APIs, it enables developers to build efficient, scalable applications without the overhead of unnecessary components.

Jakarta EE 11 Web Profile: Enhancing Web Application Development

Following the Core Profile, the Jakarta EE 11 Web Profile was released in March 2025. This profile builds upon the Core Profile by adding specifications that are essential for developing robust web applications. Key additions include Jakarta Servlet 6.1, Jakarta Server Pages (JSP) 4.0, Jakarta Faces 4.1, Jakarta Security 4.0, Jakarta Expression Language 6.0, and Jakarta Persistence 3.2.

The Web Profile's enhancements focus on modern web development needs, such as improved security mechanisms, support for Java SE 17 and 21, and better integration with contemporary web technologies. These updates facilitate the development of secure, high-performance web applications that can seamlessly integrate into cloud environments.

TCK Updates: Ensuring Compatibility and Compliance

The Technology Compatibility Kit (TCK) plays a crucial role in maintaining the integrity and compatibility of Jakarta EE implementations. Significant efforts have been made to refactor and update the TCK to align with the new profiles. This includes modularizing the TCK to match the Core and Web Profiles, simplifying the testing process, and ensuring that implementations can be certified efficiently. The updated TCKs provide a more streamlined approach to verifying compliance, reducing the complexity and time required for certification. This encourages broader adoption of Jakarta EE 11 by making

Image 1: Jakarta EE 11 Platform Overview (© Eclipse Foundation AISBL and contributors. Made available under CC-BY-SA 4.0 International)

it more accessible for vendors and developers to certify their implementations.

Jakarta EE 12

Due to the one-year delay of Jakarta EE 11, the work with Jakarta EE 12 has been going on in parallel. All the plan reviews have been completed and approved, and we are looking at a release of Jakarta EE 12 in Q2, 2026. This is aligned with the 2-year release cadence established since the release of Jakarta EE 9.

While Jakarta EE 10 and Jakarta EE 11 had updates of approximately half of the component specifications, most will be updated for Jakarta EE 12. There will also likely be some new specifications added to the Platform. Jakarta Query, Jakarta NoSQL, and Jakarta MVC are among the candidates.

The plan is to base Jakarta EE 12 on Java SE 21. The TCK will be adjusted to be able to verify the specifications on both Java SE 21 and 25.

Since Jakarta EE is a community-driven effort, and the work with Jakarta EE 12 is in its early stages, the plans and goals may change as the project progresses.

Stay Connected with the Jakarta EE Community

The Jakarta EE community is open to everyone interested in contributing, learning, or staying informed. You can join discussions on mailing lists, connect via Slack, subscribe to the newsletter, and follow project updates *[3]*.

Referenzen

[1] *http://jakarta.ee/*

[2] *https://jakartaee.github.io/platform/jakartaee11/JakartaEE11ReleasePlan*

[3] *https://jakarta.ee/connect/*

Shabnam Mayel

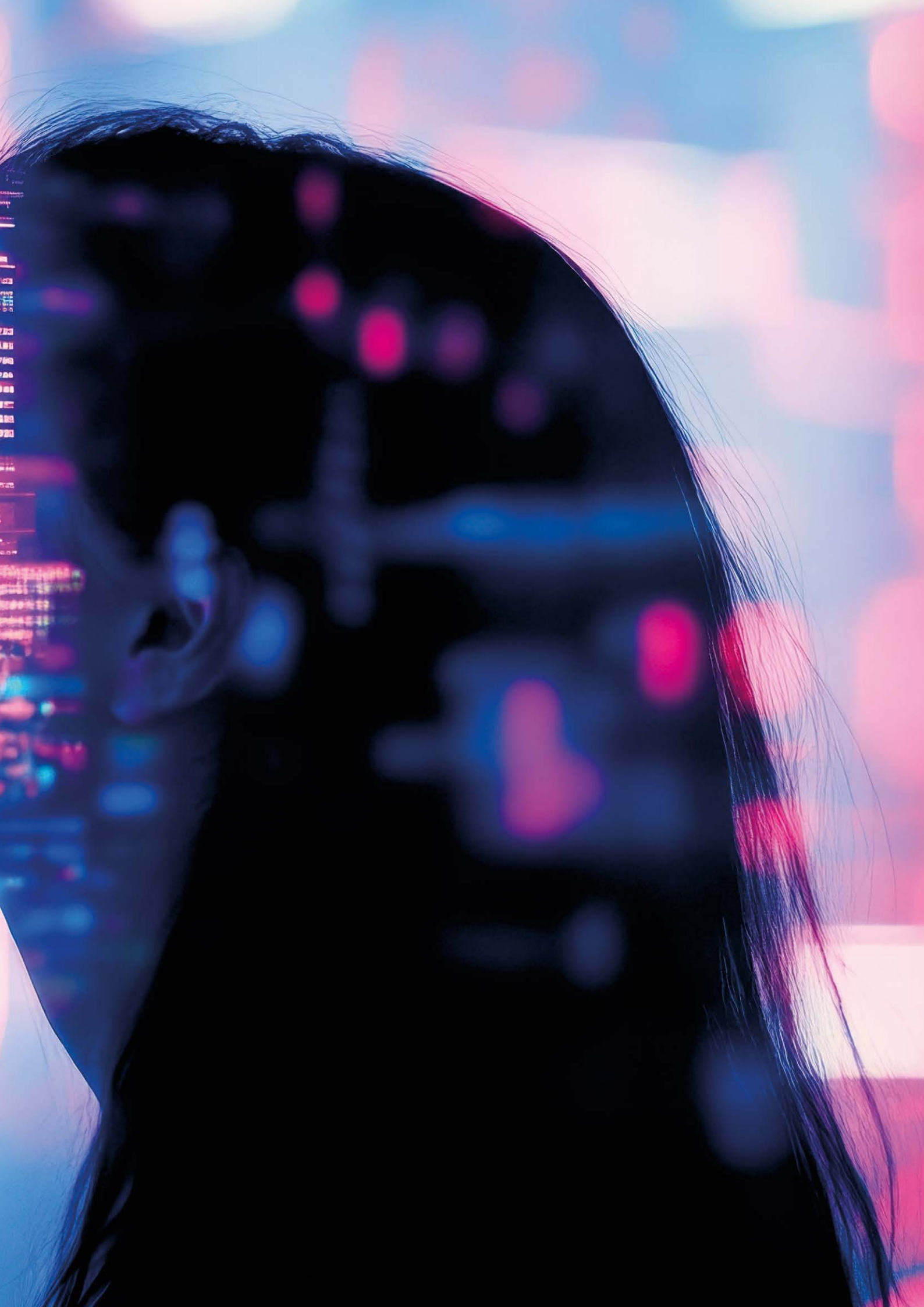
Eclipse Foundation

Shabnam Mayel is the Senior Marketing Manager for Cloud Native Java at the Eclipse Foundation. With over 15 years of experience in marketing leadership, product marketing, and business development across Southeast Asia and North America, she brings a strategic and global perspective to open-source communities. Shabnam specializes in driving brand growth and community engagement in technology-focused and open-source organizations.

Machine-Learning-Modelle in Java-Anwendungen einbetten – mit ONNX-Runtime

Kai Müller, Exxeta AG

Während Large Language Modells die Schlagzeilen beherrschen, bieten einfachere, individualisierte Machine-Learning-Modelle ein breites Anwendungsfeld. Um diese zu erstellen und zu trainieren, sind Python und R großartige Sprachen. Doch können wir diese Modelle direkt in unsere Java-Anwendungen einbetten, ohne einen externen Service anzusprechen? Eine Lösung hierfür ist die ONNX-Runtime.



Künstliche Intelligenz (KI) sorgt derzeit nahezu täglich für Schlagzeilen, weshalb auch in der Java-Welt das Interesse an diesem Thema in letzter Zeit stark gestiegen ist. Verantwortlich dafür sind vor allem die Durchbrüche in der generativen KI (GenAI) und hier primär die Large Language Modells (LLMs) wie ChatGPT, Claude oder DeepSeek. Auf diesen basieren eine Vielzahl an Tools, die wir als Java-Entwickler:innen nutzen können und sie eröffnen diverse Anwendungsfälle, die sich in Softwarelösungen integrieren lassen.

Allerdings sind diese LLMs nur ein kleiner Teil des Teilgebiets der KI namens Machine Learning (ML) und nicht für jeden Anwendungsfall nutzbar. Außerdem sind sie im Normalfall nur über eine externe API abrufbar, was die Anwendungsfälle zusätzlich einschränkt. Deshalb möchte ich das aktuelle Interesse der Java-Community nutzen, um den Blick auf das große Feld des Machine Learning zu richten und zu fragen, wie wir eine Vielzahl unterschiedlicher Machine-Learning-Modelle direkt in Java-Anwendungen einbetten können.

Was ist Machine Learning?

Machine Learning ist ein Teilbereich der Künstlichen Intelligenz, der sich mit der Entwicklung und Anwendung von Algorithmen beschäftigt, die es Computern ermöglichen, aus Daten zu lernen und basierend darauf Entscheidungen oder Vorhersagen zu treffen, ohne explizit dafür programmiert zu sein. Der Fokus liegt auf der Entwicklung von Modellen, die Muster und Zusammenhänge in Daten erkennen und aus dieser Erkenntnis heraus generelle Regeln oder Modelle ableiten können.

Machine Learning ist somit ein großes Feld in der Künstlichen Intelligenz. Es umfasst sowohl generative KI und Deep Learning als auch klassische statistische Methoden wie Random Forest oder K-Means *(siehe Abbildung 1)*. Auch wenn das große Interesse aktuell bei der generativen KI liegt, so beherrschen diese Modelle bisher nur einen kleinen Teil der multiplen Anwendungsmöglichkeiten von Machine Learning und andere spezialisiertere Methoden bieten oft stabilere, nachvollziehbarere Ergebnisse.

Warum direkt in eine Java-Anwendung einbetten?

Wenn wir an Machine Learning denken, so ist die Sprache der Wahl meist Python. Ich würde auch nicht empfehlen, ML-Modelle in Java zu trainieren. Für das Training ist Python eine ideale Sprache. Das Training von ML-Modellen ist vor allem Data-Science, und Python ermöglicht es, direkt mit wenigen Zeilen Code auf Daten zu arbeiten, ohne sich um Details wie Datentypen kümmern zu müssen. Deshalb existiert in Python ein sehr gutes Ökosystem an Bibliotheken und Tools, um Daten zu bearbeiten, zu visualisieren und ML-Modelle zu trainieren.

Die Stärken von Java liegen in der Anwendungsentwicklung. Aber auch in unseren Java-Anwendungen möchten wir ML-Modelle nutzen. Meist wird dies gelöst, indem das Modell in einem separaten Python-Service deployt wird und die Vorhersagen des Modells über eine API abgerufen werden. Jedoch kann es Vorteile haben, ein Modell direkt in eine Anwendung einzubinden:

- Verfügbarkeit
- Datenschutz
- weniger Wartung

Optimal wäre also eine Lösung, die es uns erlaubt, in einer beliebigen Programmiersprache trainierte ML-Modelle direkt in Java-Anwendungen einzubetten, und, die für das gesamte Feld des Machine Learnings anwendbar ist. Eine solche Lösung ist die Kombination von ONNX und der ONNX-Runtime.

ONNX und ONNX-Runtime

ONNX (Open Neural Network Exchange) ist ein offenes Austauschformat für ML-Modelle. Als solches Austauschformat ist es Framework-unabhängig. ONNX definiert eine Menge aus mathematischen Operationen und ein festes Dateiformat. Ein einmal trainiertes Modell kann in ONNX exportiert werden, solange die mathematischen Operationen eines Modells auf die in ONNX spezifizierten Operationen übersetzbar sind. Alle gängigen Machine Learning Frameworks unterstützen den Export nach ONNX entweder nativ oder es sind Tools für den Export vorhanden. Als beliebtes Format existiert auch ein Ökosystem an Tools, das uns zur Verfügung steht, um mit ONNX-Modellen zu arbeiten. Beispielsweise Netron *[3]* zur Visualisierung und Olive (ONNX-Live) *[4]* zur Optimierung.

Wie können wir die exportierten Modelle in Java ausführen? Hier kommt die ONNX-Runtime ins Spiel. Diese ist, wie der Name schon sagt, eine Runtime für ML-Modelle, die im ONNX-Format bereitste-

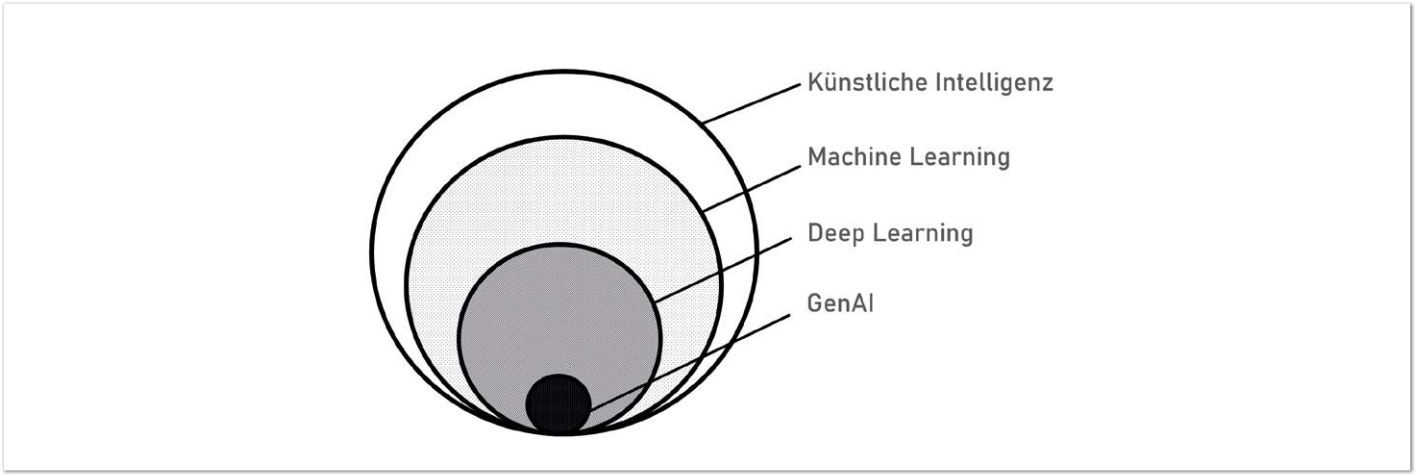


Abbildung 1: Machine Learning als Teilbereich der KI

hen. Sie hat eine ähnliche Idee wie das „Build Once, Run Anywhere"-Konzept der JVM und erlaubt es, in ONNX gespeicherte Modelle plattformunabhängig auf einer Vielzahl von Umgebungen auszuführen. Dabei bietet sie als optimierte Runtime einige Vorteile gegenüber der Ausführung des Modells über ein Framework, das für das Training von Modellen optimiert ist:

- Größe
- Geschwindigkeit
- Plattformunabhängigkeit

Gemeinsam bieten ONNX und die ONNX-Runtime die Möglichkeit, Modelle in beliebigen Frameworks zu trainieren und auf jeder gängigen Hardware auszuführen. Dies gestattet, Training und Deployment sauber voneinander zu trennen.

Die ONNX-Runtime kann für verschiedene Umgebungen angepasst und optimiert werden. Sei es in der Cloud oder im Edge-Computing. Hierzu können Entwickler:innen aus einer Vielzahl von Execution Providern wählen, den Hardware-Bibliotheken der ONNX-Runtime. Sind diese konfiguriert, so lädt die ONNX-Runtime ein übergebenes Modell als Graph in den Speicher. Dieses wird dann in Teilgraphen zerlegt und parallelisiert auf den vorhandenen Execution Providern mit den passenden Eingabedaten ausgeführt *(siehe Abbildung 2)*.

ONNX-Runtime in Java nutzen

Die ONNX-Runtime ist in mehreren Sprachen einsetzbar und hat eine gute Java-Unterstützung. Dabei ist die ONNX-Runtime selbst eine C++-Bibliothek, die in Java durch JNI angesprochen wird. Um sie zu nutzen, benötigt es jedoch nicht mehr, als das passende Maven-Paket (`com.microsoft.onnxruntime:onnxruntime`) in eine Anwendung einzubinden. Hierin sind alle benötigten Abhängigkeiten gebündelt. Eine Vorhersage auf dem Modell zu erfragen, kann dann in wenigen Zeilen Code geschehen *(siehe Listing 1)*.

Hierbei wird zuerst die Umgebung für die ONNX-Runtime initialisiert. Dieser wird das ML-Modell zur Ausführung übergeben. Um eine Vorhersage auf dem Modell zu erstellen, müssen die Daten den Eingängen des Modells entsprechend angepasst werden. Diese werden dem Modell übergeben und wir erhalten ein Ergebnis der Vorhersage.

Auch wenn die ONNX-Runtime über die Konfiguration der Session viele Möglichkeiten zur Optimierung bietet, abstrahiert sie uns die Details und wir können sie wie eine API nutzen, indem wir uns nur auf die Ein- und Ausgaben konzentrieren. Bei der ONNX-Runtime bestehen die Ein- und Ausgabedaten immer aus Tensoren. Ein Tensor ist im Grunde ein multidimensionaler Array mit einer festen Zahl an Dimensionen, bei dem jedes Element gleichen Typs sein muss. Im Normalfall ist dies ein Float, um effizient von Grafikkarten verarbeitet werden zu können. Wie diese Tensoren konkret spezifiziert sind

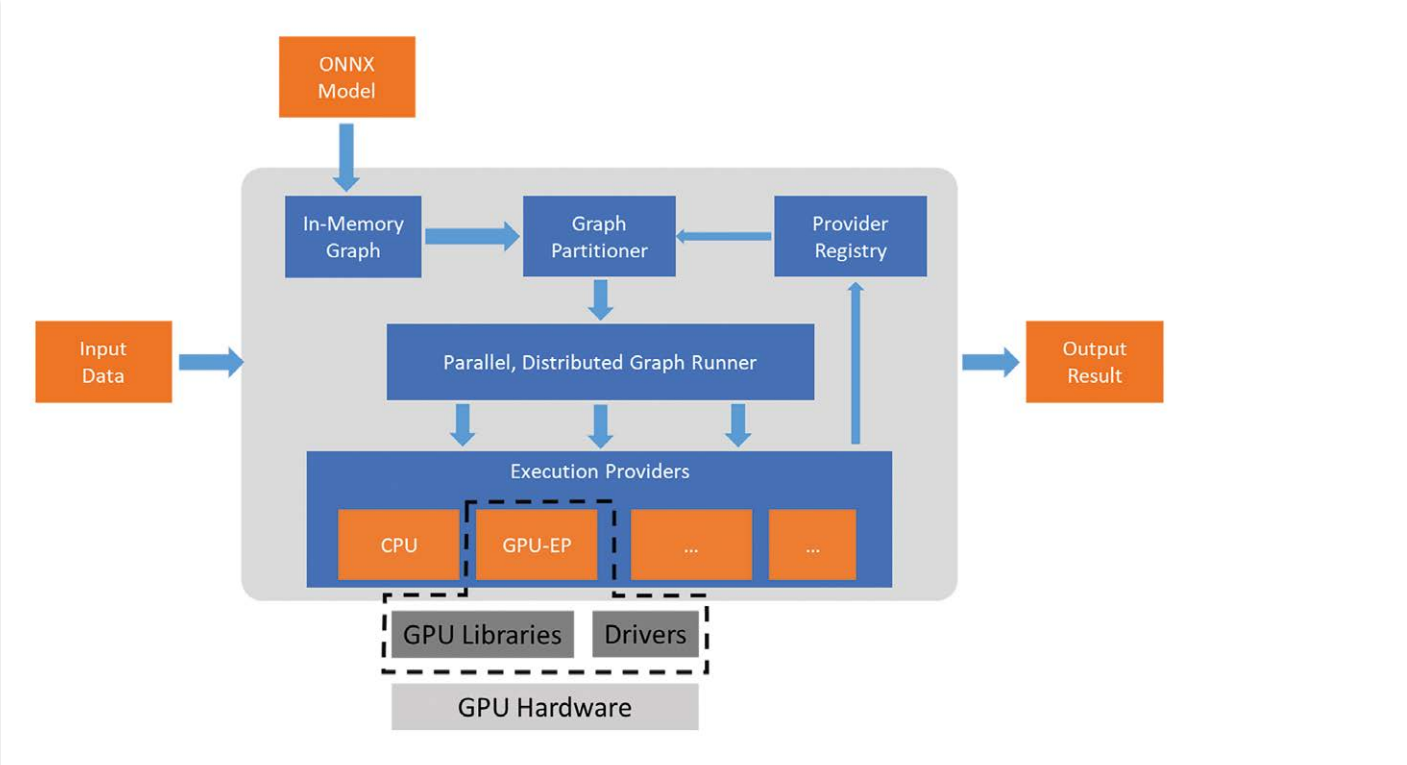


Abbildung 2: ONNX-Runtime [2]

```
this.env = OrtEnvironment.getEnvironment();
this.session = env.createSession(("path_to/your_model.onnx");
Map<String, OnnxTensor> input = yourData.toTensorMap(env);
OrtSession.Result result = session.run(input);
```

Listing 1: Nutzen der ONNX-Runtime

und wie die Daten entsprechend umgewandelt werden müssen, hängt von dem auszuführenden Modell ab.

Deep Learning

Deep Learning ist eine Unterkategorie des Machine Learnings, die sich auf die Verwendung von tiefen neuronalen Netzwerken konzentriert. Der Begriff „tief" in diesem Kontext bezieht sich auf die Vielzahl an Schichten in den neuronalen Netzwerken. Deep Learning ermöglicht es, Zusammenhänge aus großen komplexen Datenmengen zu extrahieren. Die großen Durchbrüche waren entsprechend vor allem in der Bild- und Spracherkennung zu finden.

Wie der Name schon sagt, wurde Open Neural Network Exchange primär als Austauschformat für neuronale Netze entworfen. Somit existieren in ONNX eine Reihe mathematischer Operatoren, um Deep-Learning-Modelle darzustellen und zu verarbeiten. Es ist also für die meisten Modelle gegeben, dass die ONNX-Runtime diese effizient ausführen kann.

Wir müssen uns also nur noch um die Ein- und Ausgabe des Modells kümmern. Wie die Eingabedaten vorliegen müssen, muss für jedes neuronale Netz dokumentiert sein, um es überhaupt nutzen zu können. Wie die Daten dazu umgewandelt werden müssen, ist aufgrund der Beliebtheit meist in Python-Code dokumentiert. Deshalb betrachten wir einmal beispielhaft, wie ein solches Umwandeln der Daten für ein neuronales Netz in Java aussehen könnte. Dies wird im Folgenden an einem Beispiel aus der Computer-Vision erläutert. Wir nutzen dazu das Ultraface-Modell *[5]* zur Objekterkennung aus dem ONNX Model Zoo *[6]*, um zu erkennen, wo auf einem Bild sich Gesichter befinden.

Nehmen wir als Ausgangspunkt ein Java-`BufferedImage`, so müssen wir die darin gespeicherten Daten in einen Tensor für das Modell überführen, der die für das Modell spezifische Form besitzt *(siehe Listing 2)*. Hierzu muss zuerst die Höhe und Breite auf die Eingabegröße des Modells angepasst werden. Dann müssen aus dem Bild die einzelnen RGB-Werte in Bytes extrahiert werden und schließlich in den für das Modell dokumentierten Bereich normalisiert werden. Liegen die Bilddaten in der vordefinierten Form vor, so kann ein `OnnxTensor` daraus erstellt werden, um die Daten dem Modell zu übergeben.

Die Ergebnisse der Vorhersage sind wieder ein multidimensionaler Array. Dieser wird ebenfalls je nach Modell anders interpretiert. In diesem speziellen Fall existieren zwei Ausgänge in Form von Wahrscheinlichkeiten und Bounding Boxes, die die Koordinaten einer Box um ein Gesicht auf einem Bild beschreiben. Diese können wir nach einer Mindestwahrscheinlichkeit filtern und die Wahrscheinlichsten zur Weiterverarbeitung auswählen.

Klassische Modelle aus scikit-learn

Trotz der großen Durchbrüche in künstlichen neuronalen Netzen spielen klassische Machine-Learning-Methoden noch immer eine große Rolle. Sie sind kleiner, effizienter und sind auch auf kleinen Datensätzen trainierbar. Außerdem ist leichter nachzuvollziehen, aufgrund welcher Prämissen sie zu einer bestimmten Vorhersage kommen.

Auch klassische Modelle können beispielsweise aus scikit-learn *[11]* in ONNX exportiert und mit der ONNX-Runtime ausgeführt werden. Jedoch nutz scikit-learn als Eingabe für seine ML-Modelle normalerweise keine Tensoren, sondern tabellarische Daten. Diese müssen also entsprechend umgewandelt werden. Dies geschieht durch die Aufteilung der Eingabe in eine Map aus Tensoren, mit den Spaltennamen als Key und den Einträgen als Value. Jede Spalte wird dann an einen eigenen Eingang des ML-Modells gegeben werden. *Listing 3* stellt ein Beispiel mit einer Zeile als Eingabe einer einzelnen Zahl als Ausgabe-Parameter dar.

GenAI

In der generativen KI machen vor allem die großen proprietären Large Language Models wie ChatGPT Schlagzeilen. Da diese proprietär sind und aufgrund ihrer Größe spezialisierte Hardware benötigen, müssen sie über eine API genutzt werden, beispielsweise über Langchain4J *[7]*.

Die Fortschritte machen aber auch den Einsatz kleinerer LLMs immer valider. Von diesen sind diverse Modelle als Open Source ver-

```
this.env = OrtEnvironment.getEnvironment();
this.session = env.createSession("path_to/your_model.onnx");
width = 640;
height = 480;
BufferedImage img = ImageUtils.resizeImage(yourimage, width, height);

float[][][][] normalizedImg = new float[1][3][height][width];
for (int y = 0; y < height; y++) {
     for (int x = 0; x < width; x++) {
            int rgb = img.getRGB(x, y);
            float r = ((rgb >> 16) & 0xFF);
            float g = ((rgb >> 8) & 0xFF);
            float b = (rgb & 0xFF);

            normalizedImg[0][0][y][x] = (r - 127.0f) / 128.0f;
            normalizedImg[0][1][y][x] = (g - 127.0f) / 128.0f;
            normalizedImg[0][2][y][x] = (b - 127.0f) / 128.0f;
     }
}
OnnxTensor tensor = OnnxTensor.createTensor(env, normalizedImg)
OrtSession.Result result = session.run(Map.of("input", tensor))
```

Listing 2: Skalieren und Normalisieren für Ultraface [8].

```
import static ai.onnxruntime.OnnxTensor.createTensor;
Map<String, OnnxTensor> input  = Map.of(
            "Gehalt", createTensor(env, new double[][]{{6800}}),
            "Abteilung",createTensor(env, new String[][]{{"A"}}),
            "Alter", createTensor(env, new int[][]{{38}})
);
OrtSession.Result output = session.run(input);
try (
            var output = output.get("output-name").orElseThrow();
) {
     var result = ((float[][]) output.getValue())[0][0];
}
```

Listing 3: Datenumwandlung für mehrere Eingänge

fügbar und auch für spezielle Aufgaben nachtrainierbar. Auf Hugging Face *[6]* findet sich eine Vielzahl an Open-Source-Modellen zum Download, die für spezielle Einsatzgebiete optimiert sind. Bei diesen kleineren Modellen kann es sinnvoll sein, sie direkt in eine Anwendung einzubinden.

Die Basis eines LLMs ist ein neuronales Netz, das wie jedes andere auch über die ONNX-Runtime ausgeführt werden kann. Jedoch besteht ein generatives KI-Modell nicht nur aus dem neuronalen Netz, denn die Ausgaben entstehen aus dem Zusammenspiel mehrerer Elemente. Diese wollen wir nicht für jedes Modell selbst implementieren oder einzeln einbinden.

Auch hier hat die ONNX-Runtime eine Lösung. Diese besteht in der `gernerate()`-API von onnx-genai *[7]*. Diese bietet eine einheitliche API zur Ausführung von generativen KI-Modellen und ihrer Tools auf der ONNX-Runtime – inklusive Vorhersage durch die ONNX-Runtime, Optimierung der Ausgabe, Cache-Verwaltung und Text-Token-Umwandlung.

Die `generate()`-API unterstützt alle aktuell verfügbaren Open-Source-Sprachmodelle wie Llama, Phi oder DeepSeek. Leider ist für die ONNX-GenAI in Java bisher noch kein Maven-Paket veröffentlicht worden. Somit muss das Paket noch selbst gebaut und eingebunden werden. Ein Java-Build über Cmake wird jedoch über das GitHub-Repo angeboten.

Im Normalfall würde ich gemäß dem heutigen Stand aber noch auf Alternativen ausweichen. Eine solche Alternative, um generative KI-Modelle lokal auszuführen, ist Ollama *[10]*. Mit Ollama lässt sich lokal oder über Container ein Server für ein LLM starten und über eine API aufrufen. In Java lässt sich diese beispielsweise bequem über SpringAI *[11]* aufrufen.

Fazit

Durch die Kombination von Framework und Plattformunabhängigkeit ermöglicht die ONNX-Runtime Entwickler:innen, eine Vielzahl von Machine-Learning-Modellen direkt in Java-Anwendungen einzubetten. Dabei ist es egal, ob es sich um ein neuronales Netz oder klassische Machine-Learning-Modelle handelt.

Außerdem ist die ONNX-Runtime durchaus in der Lage, generative KI-Modelle auszuführen. Da diese jedoch nicht nur aus dem Modell selbst bestehen und es eine sehr schnelle Entwicklung gibt, bleibt abzuwarten, ob sie sich in diesem Fall als Plattform der Wahl durchsetzt.

Quellen

[1] *https://onnx.ai/*
[2] *https://onnxruntime.ai/*
[3] *https://netron.app/*
[4] *https://microsoft.github.io/Olive/*
[5] *https://github.com/onnx/models/tree/main/validated/vision/body_analysis/ultraface*
[6] *https://github.com/onnx/models*
[7] *https://github.com/langchain4j/langchain4j*
[8] *https://huggingface.co/*
[9] *https://github.com/microsoft/onnxruntime-genai*
[10] *https://ollama.com/*
[11] *https://spring.io/projects/spring-ai*
[11] *https://scikit-learn.org/stable/index.html*

Kai Müller
Exxeta AG
me@kaimueller-code.com

Kai Müller ist IT-Berater bei der Exxeta AG und beschäftigt sich dort primär mit Java-Backend-Technologien. Außerdem ist der Machine-Learning-Enthusiast besonders im Bereich Computer-Vision unterwegs.

Sicherheitstests von KI-Systemen (AI Red-Teaming)

Rico Komenda, adesso SE

Generative KI-Systeme (GenAI), wie beispielsweise Large Language Models (LLMs), bieten neue Möglichkeiten, bringen aber auch einzigartige Sicherheitsherausforderungen und neue Risiken mit sich, die spezielle Testansätze erfordern. AI Red-Teaming bietet einen strukturierten Ansatz zur Identifizierung von Schwachstellen und zur Risikominimierung bei KI-Systemen, wobei der Schwerpunkt auf Sicherheit, Schutz und Vertrauen liegt. Dabei werden herkömmliche gegnerische Tests mit KI-spezifischen Methoden kombiniert, um Risiken wie Prompt Injection, toxische Ausgaben, Modellextraktion, Verzerrungen, Wissensrisiken und Halluzinationen zu vermeiden. AI Red-Teaming stellt sicher, dass die Systeme sicher und ethisch vertretbar sind und mit den Unternehmenszielen übereinstimmen.

Da generative KI-Systeme zunehmend in den Unternehmensbetrieb und die Arbeitsabläufe von Produktionsanwendungen integriert werden, müssen Sicherheitsexperten robuste Methoden entwickeln, um potenzielle Schwachstellen zu identifizieren und zu entschärfen. Beim AI Red-Teaming werden sowohl die KI-Modelle, die als zentrale Komponenten für die Anwendungen dienen, als auch die Systeme, die während des gesamten Lebenszyklus der Anwendung verwendet werden, systematisch untersucht – von der Modellentwicklung und dem Training über die Staging-Pipelines der Anwendung bis hin zu den Laufzeitumgebungen der Produktion. Adversarial Testing hilft den Entwicklern, zu überprüfen, ob die Sicherheit, Zuverlässigkeit und die Übereinstimmung mit den Unternehmenswerten in verschiedenen Angriffsszenarien gewährleistet sind. Red-Teaming ist ein altes Instrument der Cybersicherheitsbranche. Mit GenAI wurde der Ansatz um KI-spezifische Überlegungen wie Prompt Injection, Modellextraktion und Output-Manipulation sowie Bewertungen erweitert. AI Red-Teaming befasst sich auch mit neuen Aspekten wie Toxizität, der Generierung schädlicher Inhalte und Halluzinationen.

Was ist ein LLM in diesem Zusammenhang?

Große Sprachmodelle (Large Language Models, LLMs) sind eine Art von KI-Systemen, die Sprache verarbeiten und generieren, traditionell mit Text als Eingabe und Ausgabe.

Der Begriff „groß" hat sich im Laufe der Jahre weiterentwickelt: Er bezog sich zunächst auf Modelle mit Millionen von Parametern, dann auf Milliarden und umfasst heute hochmoderne Basismodelle mit über einer Billion Parametern. Per Definition ist ein LLM singlemodal – es nimmt ausschließlich Sprache als Input und produziert Sprache als Output. Der Begriff „Multimodal LLM" ist ungenau – eine genauere Bezeichnung für Modelle, die mehrere Arten von Eingaben und Ausgaben verarbeiten oder erzeugen können, ist Large Multimodal Model (LMM). Ähnlich werden Modelle, die Aktionen oder Agenten als Eingabe oder Ausgabe verarbeiten, als Large Action Models (LAMs) bezeichnet. Zusammen gehören diese Modelle zu einer breiteren Kategorie, die als Large Transformer Models (LTMs) bezeichnet wird. Nicht alle generativen KI-Modelle basieren auf der Transformer-Architektur. Es werden auch alternative Ansätze erforscht, zum Beispiel Diffusionsmodelle oder V-JEPA. Small Language Models (SLMs) verfügen ebenfalls über generative Fähigkeiten. Sie sind speziell für ressourcenschonende Anwendungen konzipiert, etwa auf mobilen Geräten oder in eingebetteten Systemen. Darüber hinaus werden SLMs zunehmend als spezialisierte Werkzeuge eingesetzt. Eine wichtige Rolle spielen sie bei der Bewertung großer Sprachmodelle, zum Beispiel zur Erkennung von Halluzinationen oder inhaltlichen Fehlern.

Trotz dieser technischen Unterscheidungen können all diese Modelle im Großen und Ganzen als generative KI-Technologien eingestuft werden – KI-Systeme, die Eingaben akzeptieren (zum Beispiel Text, Bilder, Audio, numerische Diagramme) und neue Inhalte als Ausgabe generieren (zum Beispiel Text, Bilder, Videos, Diagramme, Aktionen, Plansequenzen). Aus der Risiko- und Red-Teaming-Perspektive überwiegen die Gemeinsamkeiten dieser generativen KI-Technologien gegenüber ihren Unterschieden. Daher werden sie umgangssprachlich alle als „LLMs" bezeichnet, was in den meisten Kontexten auch ausreicht.

Was ist AI Red-Teaming?

AI Red-Teaming ist eine strukturierte Methode, die menschliches Fachwissen mit Automatisierungs- und KI-Tools kombiniert, um die Sicherheit der Nutzer, die Sicherheit des Betreibers und das Vertrauen (von Nutzern und Partnern) zu stärken sowie Leistungslücken in Systemen aufzudecken, die generative KI-Komponenten enthalten. Diese strenge Bewertung umfasst sowohl die grundlegenden Modelle als auch alle miteinander verbundenen Anwendungsebenen und gewährleistet eine umfassende Risikobewertung des gesamten KI-getriebenen Ökosystems. In vielen Fällen wird die breitere Bewertung durch geltende Anforderungen, Vorschriften und Standards vorgeschrieben.

Der Begriff „Red-Teaming" bezeichnet in der Cybersicherheit gegnerische Simulationen zur Prüfung der Verteidigungsmaßnahmen eines Unternehmens. Im Kontext von GenAI-Systemen erweitert sich dieser Ansatz auf die Inhalte, die ein Modell erzeugen. Neben klassischen Angriffen rücken neue Risiken wie toxische Ausgaben, Halluzinationen, Voreingenommenheit und ethische Verstöße in den Fokus. AI Red-Teaming baut auf etablierten Sicherheitsprozessen auf, ergänzt diese jedoch um KI-spezifische Prüfungen. Dabei werden alle sicherheitsrelevanten Aspekte generativer Modelle in den Fokus genommen – was eine enge Zusammenarbeit verschiedener Disziplinen erfordert, etwa zwischen Fachleuten für soziotechnische Risiken und klassischen IT-Sicherheitsexperten.

Scope eines AI-Red-Team-Assessments

AI Red-Teaming ist eine breit angelegte Untersuchung, die herkömmliche Sicherheitstestmethoden mit Techniken kombiniert, die auf die spezifischen und neuartigen Risiken generativer KI zugeschnitten sind. Dabei wird die Definition des Gegners nicht nur auf externe Angreifer beschränkt, sondern auch auf das Modell selbst und die Inhal-

te, die es erzeugt, ausgeweitet. Ein zentrales Ziel ist die Bewertung der Risiken durch schädliche oder irreführende Antworten, die von den zugrunde liegenden Systemen erzeugt werden können.

Die Prüfung umfasst Tests auf unsicheres Material, Verzerrungen und Ungenauigkeiten in den Ausgaben des Modells. Entscheidend ist dabei, dass das gesamte System mit all seinen Komponenten betrachtet wird. Ein besonderer Schwerpunkt liegt auf dem Umgang mit Fehlinformation – ein Bereich, der eine kritische Herausforderung darstellt. Da generative Modelle potenziell gefährliche oder irreführende Inhalte erzeugen können, müssen Red-Teams gezielt und mit Nachdruck testen, um diese Risiken frühzeitig zu erkennen und zu reduzieren.

Dazu gehört auch die Analyse, wie leicht sich das Modell manipulieren lässt, ob es versehentlich sensible oder vertrauliche Informationen preisgibt und ob seine Ausgaben Voreingenommenheiten enthalten oder gegen ethische Prinzipien verstoßen. Um möglichen Schaden zu vermeiden, müssen die Tests gründlich und vorausschauend gestaltet sein. Nur so können problematische Fälle rechtzeitig identifiziert und behoben werden. Neben der Perspektive potenzieller Angreifer wird auch die Sicht betroffener Nutzer berücksichtigt. Teil des Vorgehens ist zudem die Überprüfung von Sicherheitsmaßnahmen, die Angriffe abwehren oder erschweren sollen. Ergänzend können auch die Fähigkeit zur Erkennung und die Reaktion auf Sicherheitsvorfälle getestet werden.

Der Strukturierungsprozess kann zum Beispiel Gespräche mit Risikomanagementteams beinhalten, um Risikotoleranzen auf der Grundlage der oben genannten Kriterien festzulegen, oder mit Systembesitzern, um auf der Grundlage des getesteten Anwendungsfalls die für die Organisation wichtigsten Aspekte zu ermitteln. Beispielsweise befürchten Besitzer den Diebstahl benutzerdefinierter Modelle, daher sollte die Aufdeckung dieses Problems Teil der Festlegung des Umfangs der Übung sein. Bei der Festlegung des Umfangs sollten die Testteams Experten konsultieren, die sich mit dem zu bewertenden Risiko auskennen. Experten können Nutzer, Fachleute, die mit dem Zweck und Inhalt der Anwendung vertraut sind, Cybersicherheitsexperten und Vertreter der Zielgruppen sein. Die Teams müssen sich je nach Risiken geeignete Tools besorgen, wie Datensätze für Tests, gegnerische Modelle für Tests, Testumgebungen zur Weiterleitung von Tests, zur Erfassung von Testergebnissen und zu deren Auswertung. Schließlich sollte die Methodik zur Festlegung des Umfangs den Standards für die Autorisierung von Tests, Datenprotokollierung, Berichterstattung, Konfliktlösung, Kommunikation oder Opsec und Datenentsorgung entsprechen.

Risiken, die durch AI Red-Teaming adressiert werden

Ein zentraler Bestandteil von AI Red-Teaming ist die Identifizierung und Bewertung von Risiken, die im Zusammenhang mit generativen KI-Systemen entstehen. Der Ansatz ist ganzheitlich und deckt mehrere Ebenen ab – von der Modellausgabe über technische Implementierung bis hin zu systemischen Abhängigkeiten. Dabei stehen vor allem drei zentrale Zielbereiche im Fokus: **Sicherheit** (für Betreiber und Infrastruktur), **Schutz** (für Nutzer) und **Vertrauen** (in das System). Diese korrespondieren mit den Grundwerten moderner Sprachmodelle wie Harmlosigkeit, Hilfsbereitschaft, Ehrlichkeit, Fairness und Kreativität.

Im Rahmen von Red-Teaming-Aktivitäten werden insbesondere vier technische Perspektiven berücksichtigt:

- **Modellbewertung:** Hier geht es um die gezielte Prüfung auf inhärente Schwachstellen wie Voreingenommenheit, mangelnde Robustheit oder Halluzinationen.
- **Implementierungstests:** Die Effektivität von Schutzmaßnahmen wie Inhaltsfiltern, Prompt-Vorgaben oder Guardrails wird unter realen Bedingungen überprüft.
- **Systemanalyse:** Diese bezieht sich auf Schwachstellen in der Architektur, etwa in Lieferketten, Deployment-Pipelines, API-Schnittstellen oder der Datenhaltung.
- **Laufzeitanalyse:** Betrachtet werden Risiken aus der Interaktion zwischen Nutzern, Modellausgaben und angebundenen Systemen. Dazu zählen unter anderem Social-Engineering-Vektoren oder übermäßige Nutzerabhängigkeit.

Diese Prüfungen adressieren eine Reihe kritischer Risikokategorien:

1. *Sicherheits-, Datenschutz- und Robustheitsrisiken*
   Neben klassischen Bedrohungen treten in GenAI-Systemen neue Angriffsmuster auf – etwa Prompt Injection, Datenlecks oder Datenschutzverletzungen. Solche Risiken entstehen häufig durch manipulierte Eingaben oder kompromittierte Trainingsdaten und können schwerwiegende Folgen für Integrität und Vertraulichkeit haben.
2. *Toxizität, schädliche Inhalte und Interaktionsrisiken*
   Generative Modelle sind in der Lage, Inhalte mit hoher Ausdrucksstärke zu erzeugen, inklusive Hassrede, Diskriminierung, Obszönitäten oder extremer Ideologie. Diese Outputs gefährden nicht nur die Nutzererfahrung, sondern auch die rechtliche und ethische Vertrauenswürdigkeit des Systems. Red-Teams simulieren gezielt Szenarien, in denen solche Inhalte entstehen könnten.
3. *Verzerrung, Fehlinformation und Integritätsrisiken*
   Die Zuverlässigkeit automatisch erzeugter Inhalte spielt eine entscheidende Rolle. Neben bekannten Problemen wie erfundenen Fakten stehen dabei auch die Fragen nach sachlicher Richtigkeit, Relevanz und klarer Quellenangabe im Mittelpunkt. Ebenso wird kritisch geprüft, wie stabil das System auf leicht veränderte Eingaben reagiert.

Neue Komplexität durch autonome Agenten

Ein wachsendes Risikofeld ergibt sich durch den Einsatz sogenannter autonomer Agenten, die große Sprachmodelle nicht nur zur Textgenerierung, sondern als Entscheidungsmaschinen verwenden. Diese Systeme:

- verknüpfen mehrere Modelle miteinander,
- greifen auf externe Tools, APIs und Datenquellen zu
- und führen mehrschrittige Prozesse automatisiert aus.

Dadurch entstehen neuartige Angriffsflächen, darunter:

- mehrstufige Angriffsketten über mehrere Dienste,
- Manipulation von Entscheidungsprozessen,
- Berechtigungsumgehung durch Agenteninteraktion
- und Datenexfiltration über natürliche Spracheingabe (zum Beispiel in RAG-Systemen).

Strategie und Umsetzung

Die Entwicklung einer wirksamen Red-Teaming-Strategie im Kontext generativer KI erfordert weit mehr als reine Angriffssimulation.

Entscheidend ist ein strukturierter, risikoorientierter Ansatz, der reale Bedrohungsszenarien berücksichtigt und gleichzeitig unternehmensspezifische Anforderungen wie Compliance, Datenschutz und ethische KI-Prinzipien einbezieht. Eine solche Strategie stellt sicher, dass Angriffsflächen nicht nur erkannt, sondern auch systematisch priorisiert und adressiert werden können.

Simulieren statt spekulieren

Generative KI-Systeme unterscheiden sich fundamental von klassischen IT-Systemen – insbesondere in ihrer Interaktivität, Adaptivität und der Breite potenzieller Einsatzszenarien. Red-Teaming in diesem Bereich orientiert sich daher an konkreten Taktiken, Techniken und Verfahren (TTPs), die von Angreifern genutzt werden könnten. Ziel ist es, Sicherheitsmechanismen nicht unter Laborbedingungen, sondern unter realitätsnahen Umständen zu testen.

Risikobasierter Umfang und Priorisierung

Die Festlegung des Testumfangs erfolgt auf Grundlage einer Risikoabschätzung. Dabei werden Anwendungen und Schnittstellen priorisiert, die kritische Unternehmensprozesse steuern oder sensible Daten verarbeiten. Typische LLM-Rollen wie Klassifikator, Agent, Übersetzer oder RAG-basierter Assistent werden dabei jeweils unterschiedlich bewertet. Ein bewährter methodischer Rahmen hierfür ist die Kombination aus Impact-Analyse, Responsible-AI-Kriterien und dem NIST AI Risk Management Framework (AI RMF).

Funktionsübergreifende Koordination

Effektives AI Red-Teaming ist keine isolierte Sicherheitsdisziplin. Es erfordert die enge Zusammenarbeit zwischen Datenschutz, Recht, Informationssicherheit, Risikomanagement und KI-Entwicklung. Durch abgestimmte Metriken, Schwellenwerte und Eskalationsprozesse entsteht ein gemeinsames Verständnis für Risiken – und eine gemeinsame Verantwortung für deren Bewältigung.

Passende Bewertungsansätze

Nicht jede LLM-Anwendung eignet sich für ein standardisiertes Testverfahren. Während Black-Box-Tests für externe Services sinnvoll sein können, liefern Gray-Box- oder Assumed-Breach-Ansätze bei stark integrierten Business-Systemen oft tiefere Einsichten. Die Wahl der Methodik sollte immer die Komplexität und Kontexttiefe der KI-Nutzung widerspiegeln.

Zielgerichtete Red-Teaming-Szenarien

Ein klar definierter Zielkatalog ist essenziell. Mögliche Fragestellungen lauten:

- Kann das Modell zur Datenexfiltration missbraucht werden?
- Lassen sich vertrauliche Informationen ungewollt offenlegen?
- Werden durch gezielte Eingaben ungewollte Systemzustände erzeugt?

Bedrohungsmodellierung & Schwachstellenanalyse

Die Basis jeder guten Red-Teaming-Strategie ist ein fundiertes Bedrohungsmodell. Dieses orientiert sich sowohl an regulatorischen Anforderungen als auch an unternehmensspezifischen Geschäftsrisiken. Es beantwortet Fragen wie:

- Welche Geschäftsprozesse sind KI-gestützt?
- Wo liegen Schwachstellen in der Modell- oder Systemarchitektur?
- Welche Angriffspfade sind realistisch – intern wie extern?

Modellerkundung und Systemzerlegung

Ein technischer Fokus liegt auf der systematischen Analyse des eingesetzten Sprachmodells: Architektur, API-Verhalten, Konfigurationen und zugrunde liegende Mechanismen (zum Beispiel Layer-Design, Attention-Mechanismen). Diese Erkundung legt die Grundlage für realistische und präzise Angriffssimulationen.

Angriffssimulation und TTP-Nachbildung

Die eigentliche Angriffssimulation orientiert sich an realen TTPs: Datenmanipulation, Prompt Injection, Output-Leakage, Jailbreaks oder Social Engineering durch Modellinteraktion. Die Simulation dient dabei nicht nur dem Nachweis von Schwachstellen, sondern auch der Entwicklung wirksamer Gegenmaßnahmen und technischer Guardrails.

Risikoanalyse und Reporting

Abschließend werden alle gefundenen Schwachstellen systematisch bewertet und priorisiert. Die Ergebnisse fließen in strukturierte Berichte ein, samt konkreter Handlungsempfehlungen, Eskalationswegen und strategischer Empfehlungen. Damit unterstützt AI Red-Teaming nicht nur die Absicherung einzelner Anwendungen, sondern stärkt die gesamtheitliche KI-Governance im Unternehmen.

Fazit

AI Red-Teaming ist ein zentraler Baustein zur Sicherstellung von Sicherheit, Vertrauenswürdigkeit und ethischer Vertretbarkeit von KI-Systemen. Angesichts der einzigartigen Risiken durch LLMs und verwandte Modelle wie Prompt Injection, Halluzinationen, toxische Inhalte oder Modellmanipulation, reicht klassisches Sicherheitstesten nicht mehr aus. Vielmehr ist ein interdisziplinärer, risikobasierter Ansatz erforderlich, der technische, regulatorische und gesellschaftliche Anforderungen berücksichtigt. AI Red-Teaming kombiniert traditionelle Sicherheitsmethoden mit KI-spezifischen Prüfungen und simuliert reale Bedrohungsszenarien, um Schwachstellen frühzeitig zu identifizieren und Gegenmaßnahmen zu entwickeln. Dabei ist die enge Zusammenarbeit zwischen Entwicklung, Sicherheit, Datenschutz und Governance essenziell. Unternehmen, die generative KI produktiv einsetzen wollen, sollten AI Red-Teaming als festen Bestandteil ihrer Sicherheitsstrategie etablieren – nicht nur zur Gefahrenabwehr, sondern auch als Beitrag zu verantwortungsvoller und nachhaltiger KI-Nutzung.

Rico Komenda
adesso SE
rico.komenda@adesso.de

Rico Komenda beschäftigt sich bei adesso allumfassend mit der regulatorischen und technischen Sicherheit von KI. Als IT-Sicherheitsexperte arbeitet er eng mit Kunden zusammen, um maßgeschneiderte Sicherheitslösungen zu entwickeln und umzusetzen, die ihren spezifischen Bedürfnissen entsprechen.

iJUG Verbund

NEWSLETTER

Anmeldung

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

Abonniere ihn kostenfrei und bleibe immer auf dem Laufenden!

https://meine.doag.org/newsletter/groupSet.ijug/

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

Azure KI Services: Umfassende Werkzeuge für maßgeschneiderte und leistungsstarke KI-Lösungen

Johannes Ilisei, Accenture GmbH

Mit den Azure KI Services bietet Microsoft Unternehmen jeder Größe die Möglichkeit, moderne Softwarelösungen im KI-Umfeld aufzusetzen. Durch das umfassende Portfolio an Werkzeugen und Diensten lassen sich KI-Funktionalitäten mit geringem Aufwand in bestehende Anwendungen integrieren. In diesem Artikel werfen wir einen Blick auf einzelne Bereiche der Azure KI Services und betrachten deren praktischen Einsatz in unterschiedlichen Szenarien.

Bereits 2015 veröffentliche Microsoft unter dem Namen „Project Oxford" *[1]* eine Sammlung intelligenter Technologien mit Lösungen zur Ausführung kognitiver Aufgaben wie Gesichtserkennung, Spracherkennung und Sprachverständnis. Durch die stetige Weiterentwicklung und Ergänzung neuer Plattformen und Services ist die Auswahl an verfügbaren KI-Tools innerhalb der Azure Cloud mittlerweile enorm. Daher konzentrieren wir uns in diesem Artikel auf einige ausgewählte Services, die für die Vermittlung eines praxisnahen Bezugs anhand von Beispielanwendungen vorgestellt werden. Im Azure-KI-Foundry-Portal erstellen wir unseren eigenen Chat-Bot und verbinden ihn mit einer externen Datenquelle. Mit Prompt Flow greifen wir auf Ergebnisse aus einer externen API zu und verknüpfen diese mit einem KI-Modell. Abschließend lernen wir mittels des Azure Machine Learning Designer zwei unterschiedliche Möglichkeiten für das Trainieren eigener Modelle kennen.

Azure KI Foundry

Azure stellt mit KI Foundry (ehemals Azure KI Studio) eine Plattform für das Erstellen, Verwalten und Betreiben von KI-Lösungen bereit *[2]*. Foundry ermöglicht die Entwicklung produktionsreifer Anwendungen für den Unternehmenseinsatz. Hierfür stehen eine Vielzahl an KI-Modellen, Diensten und Funktionen bereit – darunter Zugriff auf modernste Modelle für Text- und Bildgenerierung, leistungsstarke Tools zur Feinabstimmung und Anpassung dieser Modelle an spezifische Anwendungsfälle sowie robuste Deployment-Optionen für einen zuverlässigen und skalierbaren Betrieb. Außerdem sind Funktionalitäten für gemeinsames Kollaborieren sowie Monitoringtools inkludiert. In Foundry stehen über 1.800 KI-Modelle von Anbietern wie beispielsweise OpenAI, Microsoft, Meta oder Hugging Face bereit. Über unterschiedliche Modelle können gängige KI-Aufgaben wie beispielsweise Text- und Bildgenerierung, Objekterkennung oder Bildklassifizierung abgebildet werden.

Für den Einsatz eines KI-Modells muss es zunächst bereitgestellt werden. Die Bereitstellung ermöglicht den Zugriff innerhalb wie auch außerhalb von Foundry, beispielsweise über REST. Dann steht das Modell zur Interaktion bereit. Wie wir in *Abbildung 1* sehen, ist dies in Foundry mit dem Chat-Playground möglich. Das hier gewählte Modell ist GPT-4o von OpenAI. Über die System-Message werden Anweisungen zu Verhalten und Kontext mitgegeben. Auf der rechten Seite ist die Interaktion mit dem Modell über Chat möglich.

Die Konfiguration eines Modells ist durch verschiedene Parameter möglich *(siehe Abbildung 2)*. Diese sind beispielsweise die Anzahl der vorherigen Nachrichten, die in eine neue Antwort einfließen, die maximale Antwortlänge oder die Temperatur, die die Zufallszufälligkeit steuert. Eine niedrigere Temperatur führt eher zu sich wiederholenden und deterministischen Antworten. Eine Erhöhung führt eventuell zu unerwarteten oder kreativeren Antworten. Die Spezifikation von Variablen oder die Erstellung von Mustern für beispielhafte Fragen und Antworten sind ebenfalls möglich.

KI-Modell mit Python-Bibliothek einbinden

Für die Verwendung der Azure KI Services im eigenen Unternehmen bedarf es typischerweise der Einbindung in hauseigene IT-Systeme. Hierfür werden je nach Azure Service und Modellanbieter unterschiedliche APIs, SDKs oder Bibliotheken bereitgestellt. Für die Kommunikation mit KI-Modellen von OpenAI (zum Beispiel GPT) stehen SKDs und Bibliotheken für nahezu alle gängigen Programmiersprachen zur Verfügung *[3]*. Über REST ist die Kommunikation mit jedem

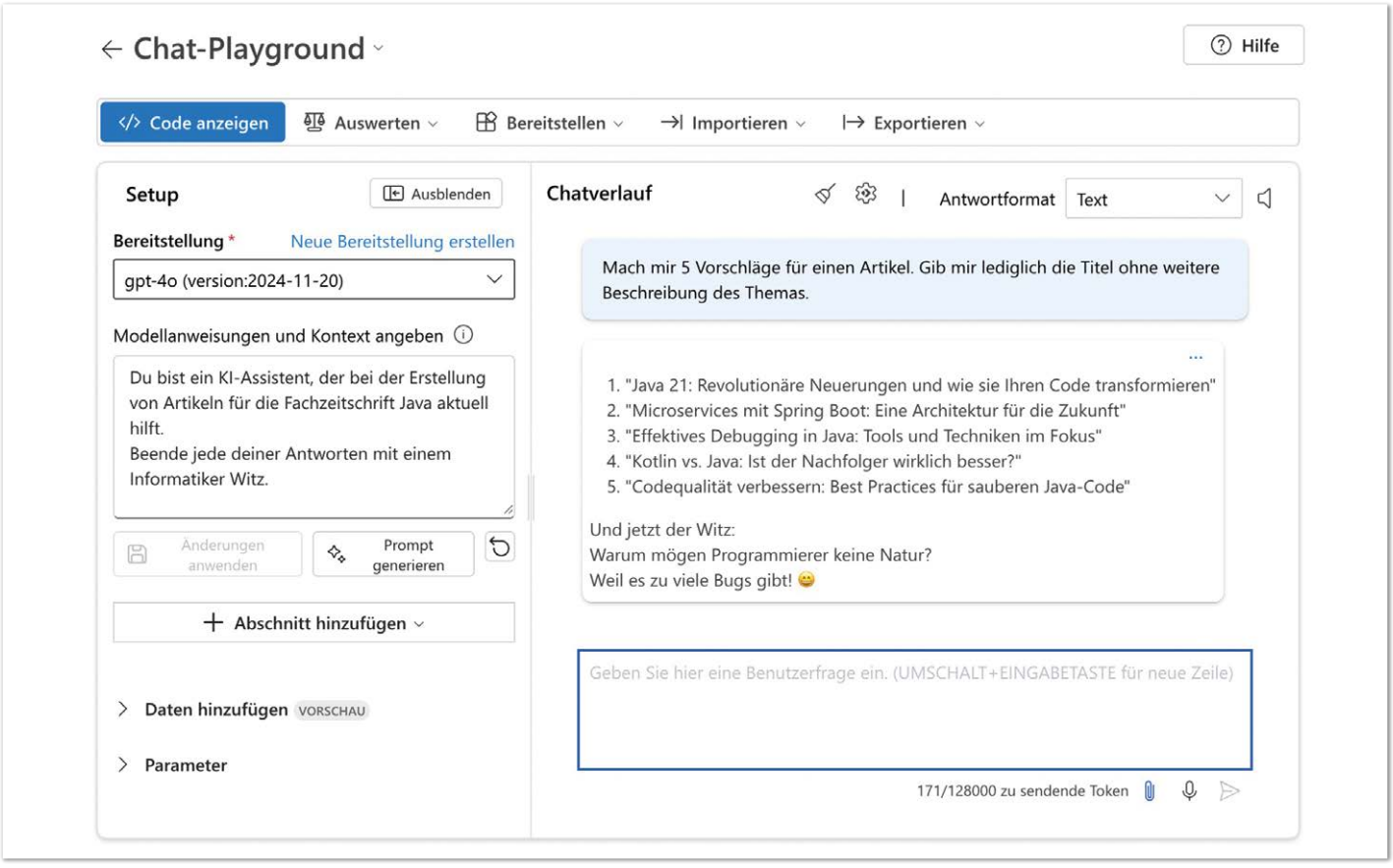


Abbildung 1: Chat-Playground in Azure KI Foundry (© Microsoft Azure)

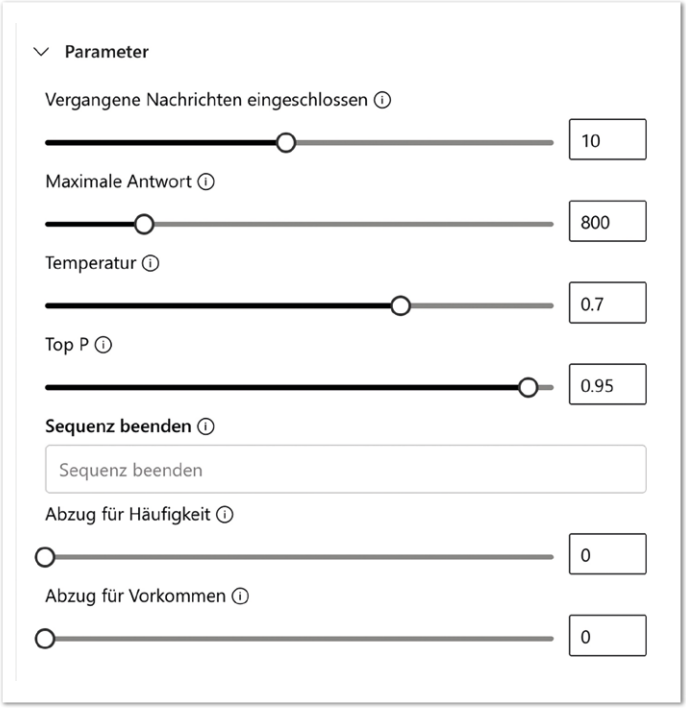


Abbildung 2: Parameter in Chat-Playground (© Microsoft Azure)

bereitgestellten Modell möglich. In *Listing 1* sehen wir den Einsatz der OpenAI-Python-Bibliothek. Zunächst wird ein neuer Client initialisiert. Beim Aufruf des Clients besteht die Möglichkeit, die gleichen Parameter, die wir bereits im Chat-Playground gesehen haben, zu übergeben.

KI-Modell in KI Foundry mit externen Daten verknüpfen

Interessant wird es, wenn wir das Modell mit einer externen Datenquelle versorgen. Statt sich lediglich auf das Wissen zu verlassen, das während des Trainings in das LLM (Large Language Model) eingeflossen ist, greift ein RAG-System (Retrieval Augmented Generation) bei der Beantwortung von Fragen auf Informationen aus externen Quellen zu. Das ist nützlich, um beispielsweise einen KI-Chat-Bot mit unternehmensinternen Informationen anzureichern. Das Neutrainieren eines LLMs würde einen hohen Aufwand bedeuten – mittels RAG hingegen ist der Einfluss externer Daten in die Generation von Antworten möglich. Eine externe Wissensquelle kann eine Vektordatenbank, eine herkömmliche Datenbank, eine Sammlung von Dokumenten oder das Internet sein. Bei Dokumenten wie PDFs wird in der Regel eine Vektordatenbank aufgebaut. Hierfür werden die Dokumente zunächst aufbereitet, bevor deren Inhalte an LLMs weitergegeben werden. Das geschieht in fünf Schritten:

1. **Chunking:** Dokumente werden zunächst in kleinere Abschnitte oder „Chunks" zerlegt. Das ist wichtig, da LLMs eine begrenzte Kontextgröße haben und es effizienter ist, nur die relevanten Chunks an das KI-Modell zu übergeben.
2. **Embeddings:** Jedes der Chunks wird durch ein Embedding-Modell (ein spezielles neuronales Netzwerk) in einem hochdimensionalen Vektorraum abgebildet. Diese Vektoren stellen die semantische Bedeutung des Textes dar. Textabschnitte mit ähnlicher Bedeutung liegen im Vektorraum nah beieinander.
3. **Indizierung in Vektordatenbank:** Die generierten Embeddings werden in einer Vektordatenbank gespeichert und für einen schnellen Zugriff indiziert.
4. **Abfrage (Retrieval):** Die eingehende Frage wird durch dasselbe Embedding-Modell in einen Vektor umgewandelt. In der Vektordatenbank wird anschließend eine Ähnlichkeitssuche durchgeführt, um Chunks zu finden, die relevante Informationen für die Antwort beinhalten.
5. **Anreicherung und Generierung (Augmentation & Generation):** Die relevantesten Chunks werden zusammen mit der eingehen-

```
from openai import AzureOpenAI

# Initializiere Azure OpenAI Client mit API-Key-Authentifikation
client = AzureOpenAI(
      api_version="2024-10-01",
      azure_endpoint="https://ai-java-aktuell-artikel-us-west-5434.openai.azure.com/",
      api_key="API_KEY_GOES_HERE"
)
# Aufruf mit System Message und allen vorherigen Nachrichten
# In der Response ist unter anderem die Antwort der letzten Frage enthalten
response = client.chat.completions.create(
      messages=[
            {
                  "role": "system",
                  "content": "Gib Hilfe bei der Erstellung von Artikeln für die Zeitschrift Java aktuell.",
            },
            {
                  "role": "user", # Vorherige Frage an LLM
                  "content": "Mach mir Vorschläge für Artikel über Java-Trends.",
            },
            {
                  "role": "assistant", # Vorherige Antwort von LLM
                  "content": "Hier sind fünf spannende Themenvorschläge: ...",
            },
            {
                  "role": "user", # Neue Frage an LLM
                  "content": "Gib mir mehr Infos zu: 'Künstliche Intelligenz mit Java'",
            }
      ],
      max_tokens=4096,
      model="gpt-4o" # Hier können weitere Parameter angegeben werden
)
```

Listing 1: Aufruf eines bereitgestellten KI-Modells über OpenAI-Python-Bibliothek

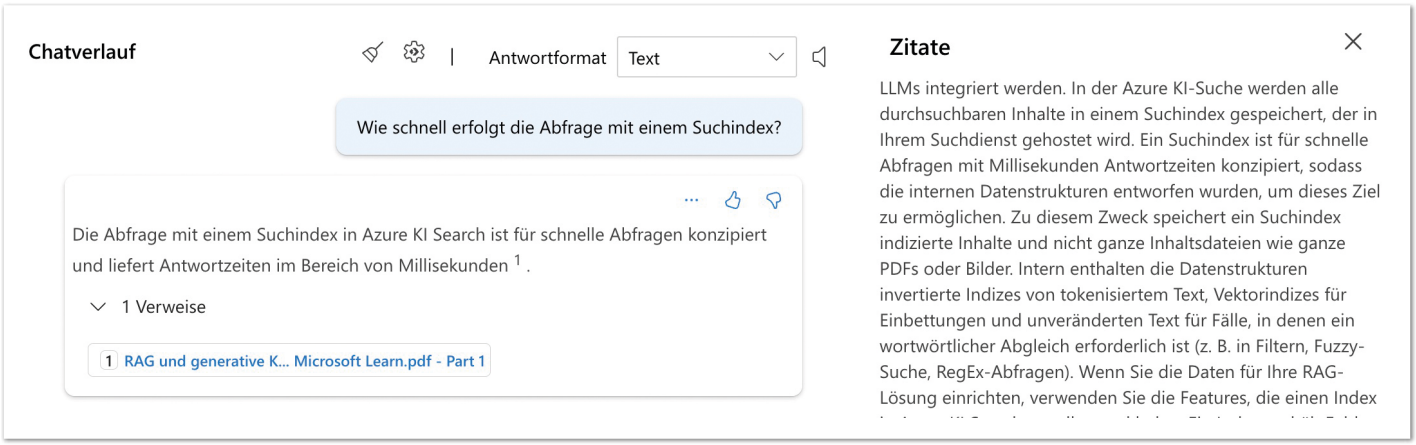


Abbildung 3: Chat-Playground mit verknüpfter Datenquelle (© Microsoft Azure)

den Frage als Prompt an das LLM übergeben. Das KI-Modell nutzt den zusätzlichen Kontext, um eine Antwort zu generieren.

Im Chat-Playground von KI Foundry gibt es unterschiedliche Möglichkeiten für das Verknüpfen mit einer externen Datenquelle. Der Upload von PDFs ist direkt möglich, wobei deren Inhalt automatisch in eine Vektordatenbank übertragen wird. Die Einbindung einer Azure-KI-Suche oder die Verknüpfung von Azure Blob Storage sind ebenfalls möglich. Je nach gewählter Methode kann ein Intervall für den automatischen Neuaufbau des Suchindex angegeben werden. Das ist besonders praktisch, um beispielsweise firmeninterne Informationen aktuell zu halten.

Abbildung 3 zeigt eine Anfrage an ein KI-Modell mit Zugriff auf eine externe Datenquelle. In diesem Beispiel wurde ein Dokument von Azure mit Informationen zu „RAG in Azure KI Search" in einer Vektordatenbank abgelegt. Zusätzlich zur Antwort werden Verweise auf die Daten ausgegeben.

Prompt Flow in KI Foundry

Prompt Flow ist ein leistungsstarkes Entwicklungstool, das die Ausführung einzelner Schritte (sogenannter Tools) als Workflow darstellt und mit KI-Modellen kombiniert. Ein „Flow" besteht aus mehreren Tools, die miteinander verknüpft sind und ihre Ausgabeparameter jeweils als Eingabeparameter an den nächsten Schritt weitergeben. Tools sind die grundlegenden Bausteine eines Flows und können beispielsweise ein LLM-Aufruf oder die Ausführung eines Python-Skripts sein *[4]*. Die Verbindung aus Workflow-Engine und KI-Modellen ermöglicht die Realisierung zahlreicher Anwendungsmöglichkeiten für unterschiedlichste Industriezweige.

Das folgende Beispiel zeigt die Stärken von Prompt Flow. Über einen Chat-Bot werden Daten von Wikipedia abgefragt, die mit in die Generierung der Antwort einfließen. Die Anwendung ist ein RAG-System, bei dem eine externe API aufgerufen wird und die Response einem LLM für die weitere Verarbeitung zur Verfügung gestellt wird. Ein ähnliches System wäre in der Lage, einen beliebigen anderen Endpunkt aufzurufen. Mögliche Szenarien sind beispielsweise eine Suche gegen das firmeninterne Confluence oder die Integration eines Mitarbeiterportals. Ebenfalls möglich ist das Ausführen von Aktionen. So könnte etwa die Chat-Nachricht „Ich beantrage Urlaub vom 15.09.25 bis 26.09.25" einen Aufruf an die interne HR-Software auslösen, der den Urlaubsantrag erstellt. Auf diese Art und Weise wird Mitarbeitern durch einen einzigen Chat-Bot Zugriff auf eine Vielzahl an firmeninternen Informationen und Tools geboten.

Eine Ausführung der Beispielanwendung ist in *Abbildung 4* zu sehen. Die Frage „Was ist Java?" wurde mit Informationen beantwortet, die von Wikipedia-Seiten stammen. Die entsprechenden Wikipedia-Seiten werden unter Sources gelistet.

Welche Schritte für dieses Ergebnis notwendig sind, schauen wir uns jetzt im Detail an. Hierbei hilft ein Blick auf den zugehörigen Graphen, der in *Abbildung 5* zu sehen ist. Der Graph ist die visuelle Darstellung der Workflowstruktur. Er veranschaulicht die Konnek-

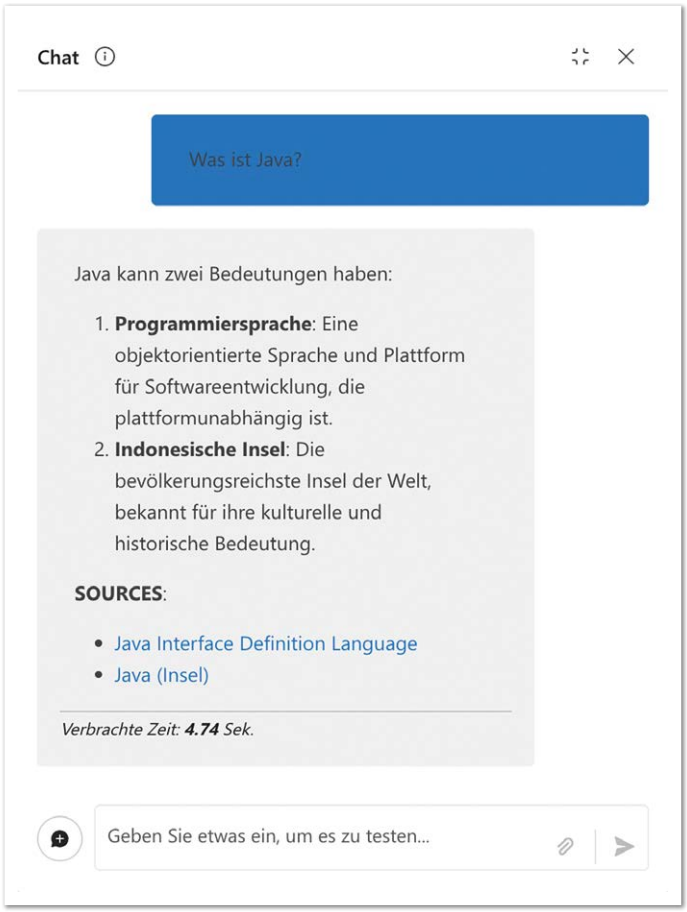


Abbildung 4: Ausführung Wikipedia-Chat-Prompt-Flow-Anwendung (© Microsoft Azure)

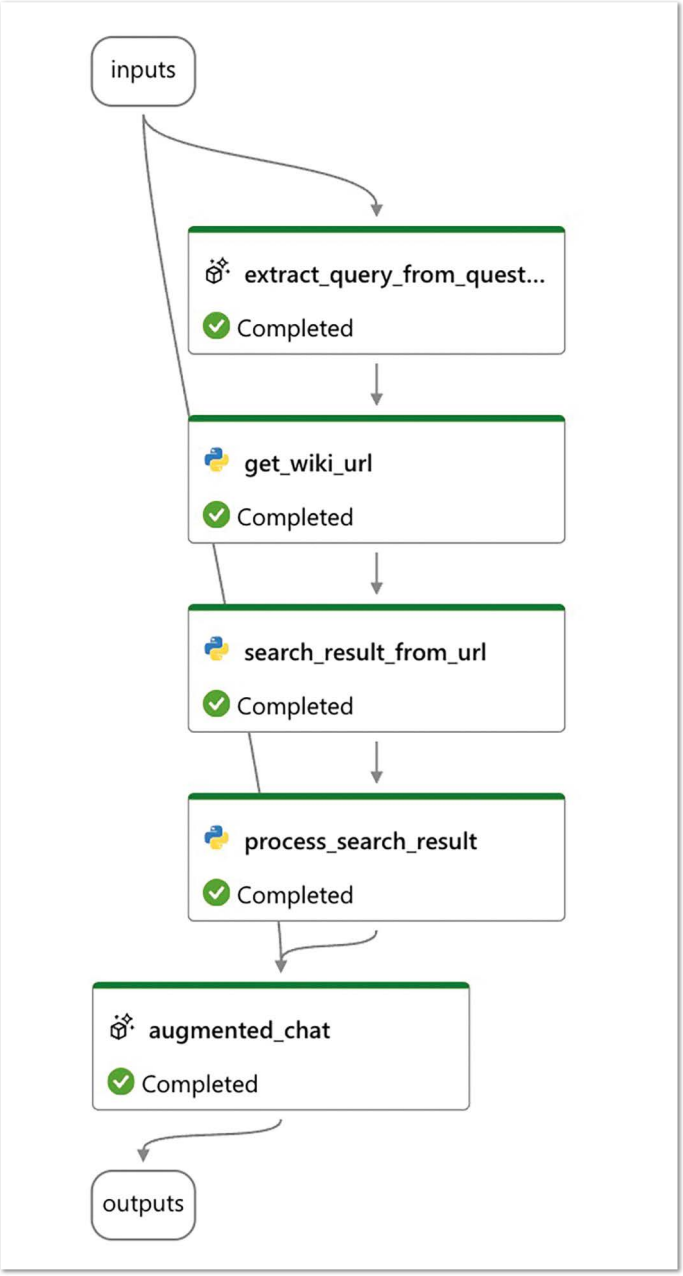


Abbildung 5: Graph der Wikipedia-Chat-Prompt-Flow-Anwendung (© Microsoft Azure)

tivität und die Abhängigkeiten zwischen den Tools und bietet eine Übersicht über den gesamten Workflow. Die Wikipedia-Chat-Anwendung besteht aus insgesamt 7 Schritten:

1. **inputs:** Die Eingabeparameter für die weitere Verarbeitung. In diesem Beispiel die aktuelle Frage sowie die Chat-Historie.
2. **extract_query_from_question:** Ein LLM mit der Aufgabe, aus der Frage den Wikipedia-Suchparameter zu bestimmen. Über eine Eingabeaufforderung erhält das LLM entsprechende Instruktionen. Im Fall von „Was ist Java?" wurde „Java Definition" ermittelt.
3. **get_wiki_url:** Ein Python-Skript, das die Wikipedia-Suche mit dem zuvor ermittelten Such-Parameter aufruft. Das Ergebnis sind Wikipedia-Seiten, wie zum Beispiel *https://en.wikipedia.org/wiki/Java*.
4. **search_result_from_url:** Ein Python-Skript, das den Inhalt der zuvor ermittelten Wikipedia-URLs abruft.
5. **process_search_result:** Ein Python-Skript für die Aufbereitung des zuvor gelesenen Inhalts.
6. **augmented_chat:** Ein LLM, das zusammen mit der initialen Frage und den Wikipedia-Daten aufgerufen wird. Entsprechende Instruktionen sorgen dafür, dass die korrekte Antwort generiert wird. Ausschnitt aus den Instruktionen: *„Given the following extracted parts of a long document and a question, create a final answer with references ('SOURCES')."*
7. **outputs:** Das Ergebnis aus Punkt 6, das im Chat als Antwort ausgegeben wird.

Ist die Erstellung des Flow abgeschlossen, kann er für externe Aufrufe bereitgestellt werden. Bei der Bereitstellung wird der Endpunkt, die Ausstattung des virtuellen Computers und Weiteres konfiguriert. Nach erfolgreicher Bereitstellung kann der Flow über REST mit entsprechender Authentifizierung (zum Beispiel über API-Key) aufgerufen werden.

Azure Machine Learning Studio

Die zweite Plattform, die wir uns anschauen, ist Azure Machine Learning Studio. Sie ermöglicht das Erstellen, Trainieren und Bereitstellen von Machine Learning Modellen, ohne Code schreiben zu müssen [5]. Teil von Studio ist unter anderem der Designer – eine visuelle Drag-and-Drop-Oberfläche, die die Erstellung von Machine-Learning-Pipelines ermöglicht. Im Vergleich zu Azure KI Foundry, das sich stärker auf die Entwicklung und Bereitstellung generativer KI-Anwendungen konzentriert, beschäftigt sich Azure Machine Learning Studio mit dem gesamten Machine-Learning-Lebenszyklus. Im Folgenden wird Machine Learning Studio anhand eines Beispiels für automatisiertes Machine Learning und einer Beispielanwendung für eine Designer Pipeline vorgestellt.

Automatisiertes Machine Learning (AutoML)

AutoML zielt darauf ab, den oft zeitaufwändigen und iterativen Prozess der Entwicklung von Machine-Learning-Modellen zu automatisieren. Hochwertige Modelle können in kurzer Zeit erstellt werden, ohne dabei tiefgreifende Expertise in Machine Learning zu benötigen [6].

Zunächst erstellen wir in AutoML einen neuen Trainingsauftrag und wählen die Vorgangsart aus, wobei Machine-Learning-Algorithmen wie Klassifizierung, Regression oder maschinelles Sehen verfügbar sind. In diesem Beispiel trainieren wir ein Klassifizierungsmodell, das vorhersagen soll, ob ein Kunde Geld bei einer Bank anlegen wird.

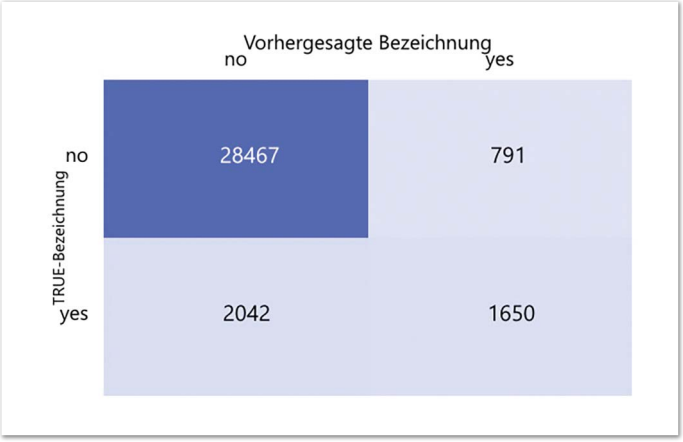


Abbildung 6: Konfusionsmatrix aus AutoML (© Microsoft Azure)

Das zu ermittelnde Ergebnis (das sogenannten Label) ist entweder „der Kunde legt Geld an" (yes) oder „der Kunde legt kein Geld an" (no). Für das Trainieren werden Trainingsdaten bereitgestellt, die als CSV-Datei vorliegen. Input-Parameter (die sogenannten Features) sind zum Beispiel Alter der Person, Beruf, Familienstand oder Einkommen.

Der Auftrag führt nach und nach mehrere Trainingsexperimente durch, bei denen mit verschiedenen Machine-Learning-Algorithmen KI-Modelle erstellt werden. Die Modelle werden anschließend tabellarisch inklusive einer Metrik zur Bewertung der Leistung aufgelistet. Auf der Detail-Seite eines erstellten Modells werden Informationen zu Metriken, Testergebnissen und Weiteres angezeigt. Die Metriken sind umfangreich und umfassen eine Vielzahl an ermittelten Werten und Grafiken. *Abbildung 6* zeigt beispielsweise die Konfusionsmatrix des Modells. Sie visualisiert die Leistung des Klassifikationsmodells, indem die Anzahl der korrekten und inkorrekten Vorhersagen im Vergleich zu den tatsächlichen Werten angezeigt wird. So wurde bei 28.467 Einträgen korrekt bestimmt, dass ein potenzieller Kunde kein Geld anlegen wird. Allerdings wurde bei 791 Einträgen fälschlicherweise vorhergesagt, dass der Kunde Geld anlegen wird.

Ein trainiertes Modell kann über AutoML als Webdienst mit REST-API bereitgestellt werden. Neue Input-Daten können an den Endpunkt übermittelt werden und eine Vorhersage wird als Antwort zurückgegeben. Ein Finanzinstitut könnte das trainierte Modell einbinden und als Weblösung zur Identifizierung potenzieller Kunden verwenden. Eine zweite Möglichkeit ist die Bereitstellung als Batch-Endpunkt. Hier ist die asynchrone Generierung von Vorhersagen großer Datenmengen möglich.

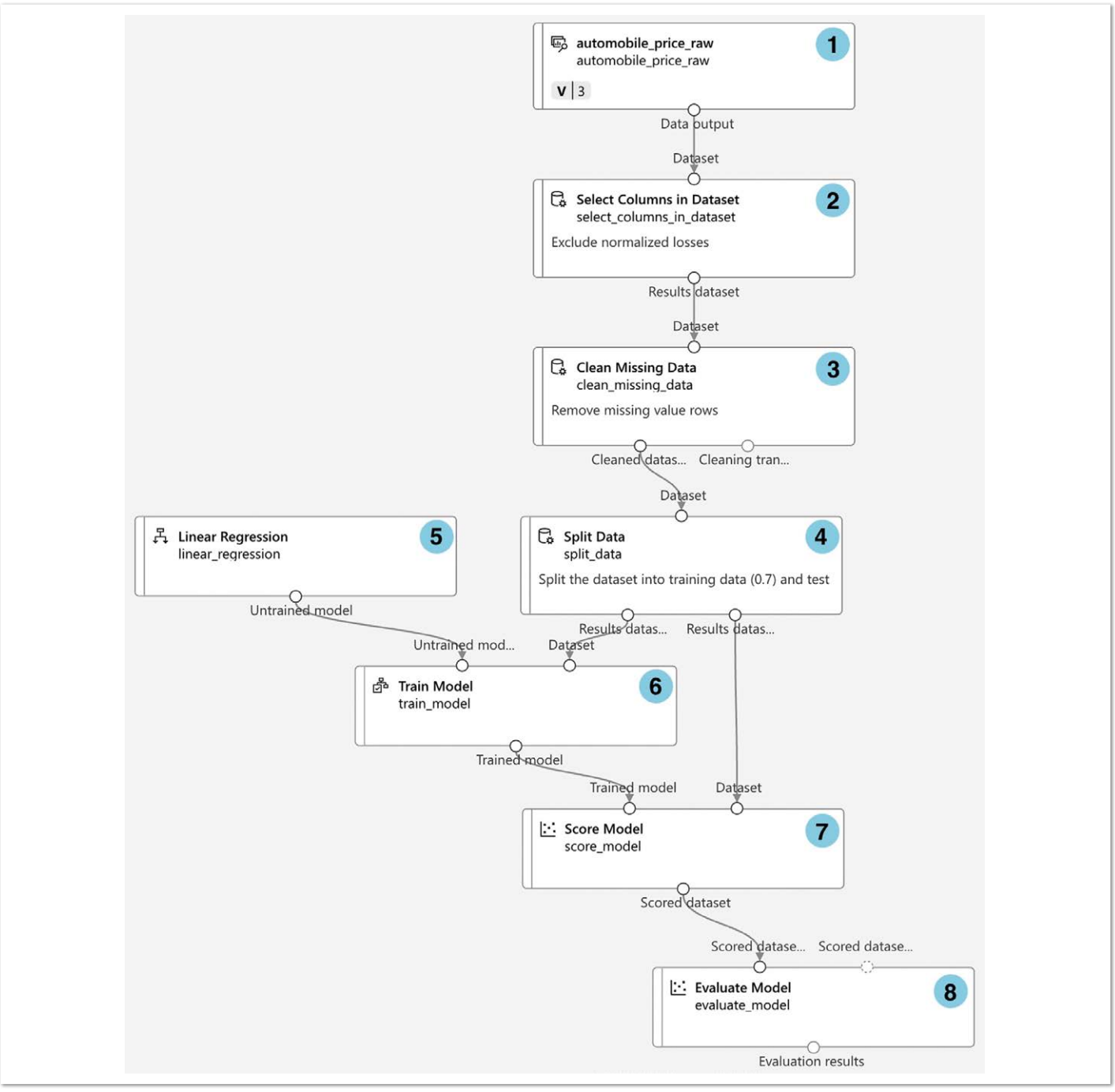


Abbildung 7: Linear Regression Pipeline aus Machine Learning Designer (© Microsoft Azure)

Machine Learning Designer

Der Machine Learning Designer ist ebenfalls Teil von Azure Machine Learning Studio und verfügt über eine grafische Drag-and-Drop-Oberfläche zur Erstellung von Machine Learning Pipelines. Pipelines werden aus einzelnen Komponenten erstellt, die von Dateneingangsfunktionen bis zu Trainings-, Bewertungs- und Validierungsprozessen reichen. Sowohl AutoML als auch der Designer trainieren KI-Modelle, allerdings setzt AutoML auf Effizienz und einfache Nutzung, während der Designer Flexibilität und Transparenz bietet, allerdings einer höheren Einarbeitung bedarf.

In der folgenden Beispielanwendung wird ein Regressionsmodell mit der Aufgabe, den Preis von Autos zu bestimmen, trainiert *[7]*. Die vollständige Pipeline besteht aus acht Komponenten und ist in *Abbildung 7* zu sehen. Die einzelnen Schritte funktionieren wie folgt:

1. **Automobile price data** beinhaltet ein Beispieldataset, das im Designer zur Verfügung steht und sich für Experimente anbietet.
2. **Select Columns in Dataset** selektiert alle Spalten des Datasets. Einzelne Spalten können individuell ein- oder ausgeschlossen werden.
3. **Clean Missing Data** bereinigt das Dataset. Datenreihen, bei denen ein Attribut nicht gesetzt ist, werden komplett entfernt.
4. **Split Data** teilt die Daten in Trainingsdaten (70 %) und Testdaten (30 %) auf. Das Aufteilen von Daten ist eine gängige Aufgabe beim maschinellen Lernen. Ein Dataset trainiert das Modell und das andere Dataset testet die Ergebnisqualität.
5. **Linear Regression** ist das zu trainierende KI-Modell. Bei diesem Ansatz wird die Beziehung zwischen mehreren, unabhängigen Variablen (Features) und einer Zielvariable (Label) modelliert. Hierdurch wird die Vorhersage von numerischen Werten (hier der Preis) basierend auf Eingabevariablen ermöglicht.
6. **Train Model** trainiert das Modell, indem sowohl die Trainingsmethode (Linear Regression) als auch das bereinigte Dataset übergeben werden.
7. **Score Model** bewertet, wie gut das Modell funktioniert, indem das trainierte Modell mit den restlichen 30 % des Datasets (der Testdaten) ausgeführt wird.
8. **Evaluate Model** stellt eine Reihe von Statistiken bereit, um die Leistung des trainierten Modells zu bewerten. Beispielsweise werden der „Mean Absolute Error" oder der „Coefficient of Determination" angezeigt.

Nach Ausführung der Pipeline stellt Score Model die einzelnen Testdaten-Durchläufe dar. Pro Durchlauf ist ein Vergleich zwischen dem tatsächlichen und dem ermittelten Autopreis direkt möglich. Über Evaluate Model können zudem die Evaluierungsergebnisse betrachtet werden. Der „Mean Absolute Error" beträgt beispielsweise 1773, was bedeutet, dass im Durchschnitt die vom Modell vorhergesagten Preise der Autos um 1713 vom tatsächlichen Preis abweichen.
Die Bereitstellung erfolgt ähnlich zu AutoML, wobei zunächst die Trainingspipeline in eine Echtzeit-Rückschlusspipeline konvertiert werden muss. Hierbei werden automatisch die Trainingskomponenten entfernt und durch passende APIs ersetzt.

Fazit

Azure stellt mit seinen Plattformen wie KI Foundry und Machine Learning Studio mächtige Werkzeuge für die Entwicklung und den Betrieb von KI-Anwendungen bereit. Durch die gezeigten Anwendungen konnten wir die Vielschichtigkeit von Azure im KI-Bereich erkennen – das Trainieren und Bereitstellen von eigenen KI-Modellen lässt sich je nach Anwendungsfall individuell sowie komplex konfigurieren und mittels Prompt Flow können nahezu beliebige KI-getriebene Abläufe abgebildet werden. Hierbei überzeugen die Services durch ihre Robustheit, Flexibilität und Skalierbarkeit. Die Verfügbarkeit von zahlreichen Beispielanwendungen und die umfangreiche Dokumentation erleichtern zudem den Einstieg. Azure KI Services integrieren sich zudem nahtlos in das bestehende Microsoft-Ökosystem sowie in die Azure Cloud und sind daher speziell für bestehende Microsoft-Nutzer eine interessante Wahl. Angesichts der stetigen Weiterentwicklung im Bereich der künstlichen Intelligenz bleibt Azure somit eine zukunftssichere Plattform für innovative KI-Projekte.

Quellen

[1] *https://blogs.microsoft.com/ai/microsofts-project-oxford-helps-developers-build-more-intelligent-apps/*
[2] *https://learn.microsoft.com/de-de/azure/ai-foundry/what-is-azure-ai-foundry*
[3] *https://github.com/openai*
[4] *https://learn.microsoft.com/de-de/azure/ai-foundry/concepts/prompt-flow*
[5] *https://learn.microsoft.com/de-de/azure/machine-learning/overview-what-is-azure-machine-learning*
[6] *https://learn.microsoft.com/de-de/azure/machine-learning/tutorial-first-experiment-automated-ml*
[7] *https://learn.microsoft.com/de-de/azure/machine-learning/tutorial-designer-automobile-price-train-score*

Johannes Ilisei
Accenture GmbH
johannes.ilisei@accenture.com

Johannes Ilisei ist bei Accenture GmbH als Softwareentwickler und Berater stets auf der Suche nach neuen innovativen Technologien, mit denen er seine Kunden und Kollegen begeistern kann. Seit 2017 ist er überwiegend in agilen Softwareentwicklungsprojekten im Java- und Angular-Umfeld in Kombination mit Cloud-Technologien tätig. Sein Interesse für KI treibt ihn an, ständig neue Möglichkeiten zu erforschen, um seine Arbeit als Softwareentwickler zu verbessern und zu optimieren.

Die versteckten Kosten von Gradle

Yuna Morgenstern

Gra

dle

Die Softwareentwicklungsgemeinschaft neigt dazu, moderne Werkzeuge schnell zu übernehmen, ohne deren langfristige Auswirkungen vollständig zu berücksichtigen. Ein prominentes Beispiel hierfür ist die weitverbreitete Einführung von Gradle als Standard-Build-Tool in vielen Projekten. Obwohl Gradle als „Build-Tool der nächsten Generation" vermarktet wird, bringt seine Nutzung eine Reihe versteckter Kosten und Herausforderungen mit sich, die oft übersehen werden.

Warum existiert Gradle?

Gradle wurde als Reaktion auf die wahrgenommenen Einschränkungen von Apache *Maven* entwickelt. Während *Maven* auf deklarative Konfiguration mittels XML setzt, wollte *Gradle* durch eine flexible, skriptbasierte Lösung auf Basis von *Groovy* und später *Kotlin* mehr Anpassungsmöglichkeiten bieten. Ein weiteres Hauptversprechen war eine verbesserte Performance durch inkrementelle Builds und Caching-Mechanismen. In der Praxis zeigt sich, dass *Gradle* in vielerlei Hinsicht auf den Konzepten von *Maven* aufbaut, jedoch zusätzliche Features mit sich bringt.

Was brauchen wir wirklich?

Letztendlich möchten wir ein einfaches JAR-File oder ein anderes funktionierendes Artefakt erstellen. Theoretisch könnte dies mit einem einfachen javac-Befehl erreicht werden. Doch Build-Tools bieten Komfort und Automatisierung. Nur zu welchem Preis?

Versionsbeschränkungen

Eine der verwirrendsten Einschränkungen von *Gradle* ist seine strikte Abhängigkeit von spezifischen Versionen wie zum Beispiel zu *Java*, *Groovy*, *Kotlin* und *Spring Boot* [1] [2]. Während *Maven* keinerlei Abhängigkeiten zur Entwicklungsumgebung benötigt, zwingt *Gradle* uns dazu, mehrere Versionen über Projekte hinweg zu verwalten. Dies ist keine Flexibilität, sondern ein Kopfschmerz. Obwohl Tools wie *SDKMAN* [3], spezielle Java-Fix-Tools [4] oder andere Workarounds existieren, beheben sie ein Problem, das ohne *Gradle* nicht existieren würde. *Gradle* schafft harte Abhängigkeiten zu Tools, was zukünftige Updates erschweren kann.

Langfristige Wartung

Maven-Projekte bleiben auch nach einigen Jahren stabil, mit Artefakten, die in *Maven Central* erhalten bleiben. *Gradle*-Projekte hingegen können Probleme haben: Plug-ins verschwinden, Abhängigkeiten brechen, und Build-Skripte zerfallen unter dem Gewicht der Obsoleszenz. Syntax-Änderungen sind keine Ausnahme. *Gradle* ist eine „funktioniert für jetzt"-Lösung ohne Garantie für Langlebigkeit. Es fühlt sich an, als würde man eine weitere Anwendung programmieren, denn *Gradle*-Build-Dateien benötigen deutlich mehr Wartung.

Dependency Updates

Die Aktualisierung von Dependencies ist ein Eckpfeiler für sichere und wartbare Software. *Maven* glänzt hier mit Plug-ins wie:

- `mvn versions:use-latest-versions` zur Aktualisierung von Abhängigkeiten
- `mvn license:add-third-party` zur Generierung und Überprüfung von Lizenzinformationen
- `mvn org.owasp:dependency-check-Maven:check` für Sicherheitsprüfungen.

Diese Plug-ins erfordern keine zusätzliche Konfiguration und lassen sich problemlos in CI/CD-Pipelines per CLI integrieren. Es gibt nur wenige Plug-ins, die in *Maven* unbedingt eine XML-Konfiguration benötigen.

Gradle hingegen bietet keine äquivalente Lösung. Die Aktualisierung von Abhängigkeiten erfordert oft externe Tools wie *Dependabot* [5] oder *Renovate* [6]. Da jedoch *Gradle* oft Versionen von *Java*, *Groovy*, *Kotlin* oder *Spring Boot* mit spezifischen *Gradle*-Versionen verknüpft, kann dies den manuellen Wartungsaufwand deutlich erhöhen.

Der Gradle Wrapper: Ein zweischneidiges Schwert

Ein Beispiel dafür ist der *Gradle Wrapper*. Im Gegensatz zum *Maven Wrapper*, der sich selbst herunterlädt, erfordert *Gradle* das Einchecken von Binärdateien in das Repository. Diese Binärdateien können das Git-Repository im Laufe der Zeit aufblähen und zu Performance-Problemen führen. Für eine Branche, die nach sauberen und leichten Lösungen strebt, scheint dies ein Rückschritt zu sein.

Fazit: Eine bewusste Wahl treffen

Wie so oft ist die Wahl zwischen Tools stark von den individuellen Projektanforderungen abhängig. Oft verlieren wir jedoch aus den Augen, was unser eigentliches Ziel ist – YAGNI [7] („You Aren't Gonna Need It", zu Deutsch: „Du wirst es nicht brauchen"). *Gradle* bietet hohe Flexibilität und mehr Features, benötigt jedoch eine tiefere Einarbeitung und kann durch seine hohe Dynamik zu höherem Wartungsaufwand führen.

Ein wichtiger Punkt in der Softwareentwicklung ist die Reduzierung unnötiger Komplexität. In manchen Fällen kann eine einfachere Lösung, die sich auf die Kernaufgaben beschränkt, vorteilhafter sein.

Es empfiehlt sich daher, nicht nur nach Popularität zu entscheiden, sondern objektiv abzuwägen, welches Tool die beste Langzeitlösung für das jeweilige Projekt darstellt.

Quellen

[1] *https://docs.gradle.org/current/userguide/compatibility.html*: Gradle, Java, Kotlin, Groovy Kompatibilitätseinschränkungen

[2] *https://stackoverflow.com/a/78784652/3907616*: Gradle Spring Boot Kompatibilitätseinschränkungen

[3] *https://sdkman.io*: Java Version Manager

[4] *https://github.com/YunaBraska/gradle-java-fix*: Gradle Java Version Manager

[5] *https://docs.github.com/en/code-security/getting-started/dependabot-quickstart-guide*: Dependency updater of GitHub

[6] *https://docs.renovatebot.com*: Independent dependency updater

[7] *https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it*: YAGNI principle

Yuna Morgenstern

io@yuna.berlin

Ich bin Team- und Tech-Lead und brenne für Einfachheit sowie Automatisierung. Ich contribute in Open-Source-Projekten wie NATS und habe bereits ein JetBrains-Plug-in geschrieben. Meine Erfahrung in der OSS-Welt hat meine Qualitätsstandards geschärft und mich kritisch gegenüber Marketing und Dokumentationen gemacht. Stattdessen analysiere ich Code und Javadocs direkt, um zu verstehen, wie Dinge im Hintergrund funktionieren. Weniger ist oft mehr – ein Prinzip, das ich nicht nur in der Softwareentwicklung, sondern auch in meinem minimalistischen Lebensstil verfolge.

Dynamisches Policy-Management mit dem Open Policy Agent

Jelmen Guhlke

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.

Im letzten Artikel zum Thema „Moderne Berechtigungssteuerung" in der Java aktuell 02/25 wurden die grundlegenden Funktionen und Methodiken von Access Control (zu Deutsch „Zugriffssteuerung") im Allgemeinen und in der Jakarta-EE-Welt im Speziellen betrachtet. Besonders hervorzuheben sind dabei die Jakarta-Security- *[1]*, Jakarta-Authentication- *[2]* und Jakarta-Authorization-Spezifikationen *[3]*. In der Regel wird im Zuge der Applikationsentwicklung nicht direkt mit den beiden Low-Level Service Provider Interfaces („SPI") von Jakarta Authentication und Jakarta Authorization kommuniziert, sondern mit dem Top-Level API von Jakarta Security. Dabei stellt die Spezifikation Funktionen zur Verfügung, um die zentralen Herausforderungen der Zugriffssteuerung angehen zu können: Authentifizierung und Autorisierung *(siehe Abbildung 1)*. Wie schon im ersten Artikel dieser Reihe soll der Fokus weiterhin auf der Autorisierung liegen und nicht auf der Authentifizierung.

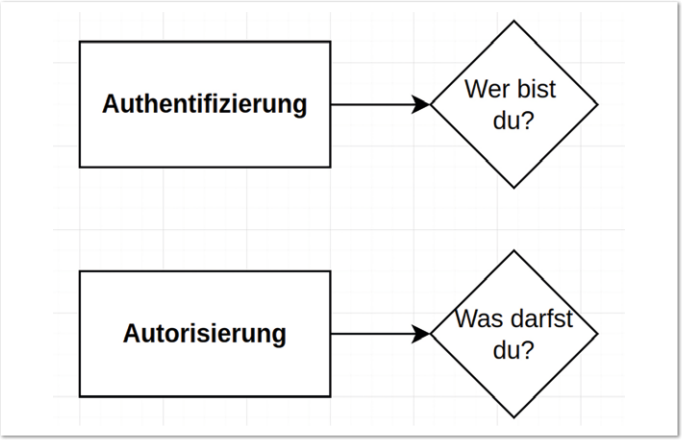


Abbildung 1: Authentifizierung vs. Autorisierung (© Jelmen Guhlke)

Historisch orientiert sich die Jakarta-EE-Umwelt *[4]* (vormals Java EE) an einer monolithischen Entwicklungsstruktur. Alle Deployment-relevanten Artefakte wie Beans, Servlets und Views werden zusammen mit dem Deployment Descriptor verpackt und in einem Applikationsserver ausgerollt. Nachvollziehbarerweise orientieren sich die Spezifikationen rund um das Thema Sicherheit und Zugriffssteuerung an der gleichen Vorgehensweise, sodass auch der Jakarta-Security-Code verpackt und mit in das Monolithen-Archiv geschnürt wird.

Wenngleich mit Jakarta Security der Anwendungsentwicklung eine mächtige Spezifikation an die Hand gegeben wird, um viele der gängigen Herausforderungen im Zuge der Zugriffskontrolle zu begegnen, bleibt ein zentrales Manko zurück: Anwendungs-Code und Zugriffskontrolle sind sehr eng miteinander verbunden. Durch Design-Ansätze wie aspektorientierte Programmierung wird zwar einiges dafür getan, dass die beiden Belange so lose wie möglich miteinander gekoppelt sind – eine gewisse Verbundenheit bleibt aber durch die gemeinsame Paketierung in ein Archiv immer.

Leserinnen und Leser der Java aktuell (oder anderer Fachliteratur) werden den in den letzten Jahren vorherrschenden Trend der losen Kopplung von Concerns (Angelegenheiten; Belange) und Abhängigkeiten bis hin zu Microservice-Infrastrukturen mit Dutzenden von verschiedenen Services unweigerlich mitbekommen haben. Jedoch steht das Ziel einer möglichst vollumfänglichen Entkopplung einzelner Services oftmals konträr gegenüber Querschnittsbelangen, wie die der Zugriffssteuerung. Im schlimmsten Fall führt dies zu einer notwendig gewordenen Redundanz von Logik und Code über zahlreiche Artefakte hinweg.

Eine fachliche Policy = eine technische Implementierung?

Zugriffsbeschränkungen entstehen in den seltensten Fällen in einem luftleeren Raum. Vielmehr bilden sogenannte Policies (frei übersetzt „Richtlinien") aus der fachlichen Domäne die Grundlage für die konkrete technische Implementierung einer Prüfung. Hierbei kommt es in modularisierten Anwendungen öfter vor, dass sich eine Richtlinie nicht explizit und ausschließlich auf ein Modul oder noch kleiner auf einen Service eines Moduls beschränkt. Vielmehr umfasst das Thema Zugriffssteuerung oft einen breiten Pool an Modulen und wird nicht immer wieder neu definiert. Dieses Phänomen kann, wie weiter oben schon ausgeführt, dazu führen, dass die Realisierung einer Policy in mehreren Deployment-Einheiten redundant implementiert werden muss.

Wie angehenden Entwicklerinnen und Entwicklern schon früh eingeflößt wird, bringt Redundanz in Code einige Herausforderungen mit sich, die hier nicht noch einmal wiederholt werden müssen. Eleganter wäre es für diesen Fall, wenn die Realisierung der Policy nur an einer Stelle erfolgen müsste und von beliebigen Services oder Modulen genutzt werden kann.

Genau hier versucht der Open Policy Agent *[5]* anzusetzen. Die unter der Apache 2.0 lizensierte und in Go geschriebene Software *[6]* orientiert sich an einem einfachen Grundprinzip: eine klare Trennung von „Policy decision making" und „Policy enforcement". Ins Deutsche übertragen, lässt sich das Prinzip als eine Trennung der prüfenden Logik einer Richtlinie und der durchführenden Logik einer Richtlinie beschreiben.

Dieses im ersten Moment etwas kryptisch anmutende Konzept lässt sich leicht durch ein Schaubild verdeutlichen *(siehe Abbildung 2)*. Das Subjekt Bob will auf die Resource A zugreifen. Der Prüfprozess ist als einfaches Flussdiagramm dargestellt. Werden die Schritte in

einzelne Tasks untergliedert, lässt sich oftmals schnell der Teil der prüfenden Logik (im Schaubild lila dargestellt) von der durchführenden Logik (im Schaubild rot dargestellt) unterscheiden. Bei der Nutzung von Frameworks wie Jakarta Security wird in der Regel die durchführende Logik von dem Framework unter der Haube verwaltet. Einem unautorisierten Zugriff auf einen *REST*-Endpunkt wird scheinbar „automatisch" mit dem richtigen 403-HTTP-Code geantwortet, oder ein Browser-Aufruf auf eine unerlaubte Seite wird „wie von Geisterhand" weitergeleitet.

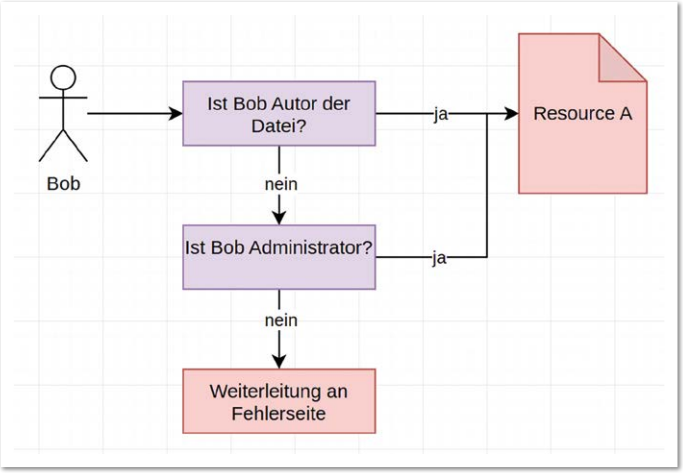


Abbildung 2: Beispielhafter Autorisierungsfluss (© Jelmen Guhlke)

Das Durchführen der Richtlinie ist hierbei oft stark von der verwendeten Technologie und dem Einsatzzweck abhängig, sodass dies auch in den Händen der Frameworks und Libraries bleiben sollte. Die prüfende Logik hingegen hängt in der Regel an keiner bestimmten Technologie, sondern nur an der aus der fachlichen Domäne kommenden Policy. Genau diese lässt sich also sehr gut auslagern und der Open Policy Agent ist dabei eine willkommene Hilfe.

Funktionsweise des OPA

Bei der Nutzung des Open Policy Agents (OPA) werden die Richtlinien mit Hilfe der deklarativen Policy-Sprache „Rego" *[7]* (ausgesprochen als „ray-go") geschrieben und implementiert. Diese zeichnet sich durch sehr klare sowie – nach etwas Gewöhnung – einfach zu lesende und schreibende Ausdrücke aus. Dabei werden Entscheidungen auf Basis hierarchisch strukturierter Daten getroffen.

Die Informationen, die der Agent zur Entscheidungsfindung benötigt, können über verschiedene Interfaces per Push oder Pull synchron oder asynchron geladen werden. Alle Daten, die von extern kommen, werden im Datenmodell von OPA als „base documents" angesehen. Die Bezeichnung „document" hat seinen Ursprung aus den dokumentenbasierten Datenbanken und sollte nicht mit einer Datei auf dem Dateisystem gleichgesetzt werden. Neben den von außen kommenden Daten können auch Regeln innerhalb des Agents von anderen Regeln abhängen. Diese werden als „virtual documents" bezeichnet, um zu unterstreichen, dass sie von dem Agenten berechnet wurden. Daten, die asynchron mit Push und Pull verarbeitet wurden, sind in allen Policies unter der globalen Variable `data` erreichbar. Informationen, die jedoch synchron zur Findung einer konkreten Entscheidung an den Agenten gesendet werden, sind unter der Variable `input` referenzierbar.

Erste Schritte mit dem Agenten

Der Agent kann als Stand-alone-Anwendung auf dem lokalen Host *[8]*, als Docker Container *[9]* oder als Library direkt in Go-Anwendungen ausgeführt werden. Da im Zuge des Artikels ein Blick auf die dynamischen Möglichkeiten bei der Nutzung der Agents geworfen werden soll, wird die letzten Option nicht näher beleuchtet. Bei den ersten beiden Optionen gibt es sowohl ein sehr umfangreiches Command-Line-Interface als auch einen Server-Modus, in dem ein REST-API zur Verfügung gestellt wird. Läuft der Agent einmal, können über den Endpunkt `PUT v1/data` statische Daten („base documents") an den Server übergeben werden *(siehe Listing 1)*. Auf diese kann dann im Zuge der eigentlichen Regelprüfung zugegriffen werden. Die Daten werden im RAM gehalten und lassen sich auch partiell ändern oder wieder ganz löschen.

```
{
  "departments": [
      {
          "id": 123,
          "timed_access": "full"
      },
      {
          "id": 456,
          "timed_access": "full"
      },
      {
          "id": 789,
          "timed_access": "day"
      }
  ],
  "timed_access": [
      {
          "id": "full",
          "from": 0,
          "to": 24
      },
      {
          "id": "day",
          "from": 9,
          "to": 18
      }
  ]
}
```

Listing 1: Basis-Daten für die Policy-Berechnung

Wie schon erwähnt, werden die eigentlichen Policies in Rego geschrieben. Hierbei werden diese in für Java-Entwicklungen vertraute Pakete gruppiert. Dies dient bei der Nutzung der Policies später der eindeutigen Identifizierung. Um den Umfang des Artikels nicht zu sprengen, wird an dieser Stelle auf eine detaillierte Beschreibung der Sprache von Rego verzichtet. Eine ausführliche Dokumentation steht hierfür auf der Projektseite zur Verfügung *[10]*.

Eine simple Policy besteht aus mindestens einer Regel. Wie aus dem Java-Umfeld von Methoden und Funktionen bekannt, besteht eine Regel aus einem Rule Header und Rule Body. Dabei spiegelt der Rule Header eine Referenzierungsmöglichkeit als virtuelles Dokument wider, um diese auch in anderen Regeln nutzen zu können. Eine simple Policy kann somit so wie in *Listing 2* aussehen.

Die fertige Policy wird mit der Endung `.rego` abgespeichert und kann an den Endpunkt `PUT v1/policies/{id}` an den Agenten übermittelt werden. In der in *Listing 2* gezeigten Policy werden mehrere

```
package timed

import rego.v1

default allow := false

allow := true if {
     input.department in allowed_department_access
}

allowed_timed_access contains allowed_time.id if {
     request_time := time.parse_rfc3339_ns(input.requestTime)
     request_hour := time.clock(request_time)[0]
     some allowed_time in data.timed_access
        request_hour >= allowed_time.from
        request_hour < allowed_time.to
}

allowed_department_access contains department.id if {
     some department in data.departments
        department.timed_access in allowed_timed_access
}
```

Listing 2: Eine simple Policy-Implementierung in Rego

Regeln definiert. Zentral hierbei ist die Regel `allow`, die per `default` auf `false` gesetzt wird. Sie löst sich auf `true` auf, wenn der Wert aus `input.department` in der Liste `allowed_department_access` vorhanden ist. `allowed_department_access` ist hierbei ein virtuelles Dokument und setzt sich wiederum aus dem virtuellen Dokument `allowed_timed_access` zusammen. Aus der fachlichen Domäne bildet die Policy die Anforderung ab, dass bestimmten Departments nur in bestimmten Zeitfenstern ein Zugriff gewährt wird. Mit einer erfolgreichen Übertragung lassen sich jetzt `POST`-Anfragen abschicken, die als Body den für die Auswertung der Rule relevanten Input mitliefern *(siehe Listing 3)*. Die URL setzt sich aus dem Package der Policy und der Referenz der Rule zusammen.

```
POST /v1/data/timed/allow
Content-Type: application/json

{
  "input": {
     "department": 123,
     "requestTime": "2025-05-15T15:12:18.514567787+02:00"
  }
}
```

Listing 3: Die eigentliche Abfrage einer Policy kann mittels POST-Anfrage erfolgen.

Die Möglichkeiten, Regeln zu erstellen, sind dabei mannigfaltig. Darüber hinaus gibt es zahlreiche Funktionen, die vom OPA bereitgestellt werden, um zum Beispiel mathematische Probleme zu lösen oder einen weiteren HTTP-Aufruf zu starten *[7]*.

Anbindung an Jakarta EE

Um den Agenten an eine Jakarta-Anwendung anzubinden, bedarf es nicht viel Aufwand, da die Kommunikation auf standardisierten Wegen mittels *REST* und *JSON* (oder anderem Format) vonstattengehen kann. Spannender hingegen ist eine sinnvolle Integration in die Anwendung. Out-of-the-box bietet Jakarta Security im Gegensatz zu Spring keine `PreAuthorize-` oder `PostAuthorize-` Annotationen *[11]* an, die genutzt werden könnten, um sich in den Aufruf einhängen zu können. Die schon im vorherigen Artikel kennen gelernten Annotationen `RolesAllowed`, `PermitAll` und `DenyAll` *[12]* sind für diesen Anwendungsfall zu statisch. Mit Hilfe eines oder mehrerer CDI-(Context and Dependency Injection)-Interceptoren *[13]* kann diese Lücke aber geschlossen werden. So lässt sich zum Beispiel ein `OpaSecured` Interceptor implementieren, der an den notwendigen Methoden beziehungsweise Klassen ausgeführt werden soll *(siehe Listing 4 und 5)*.

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD, TYPE})
public @interface OpaSecured {}
```

Listing 4: Die Definition eines Interceptors, der später an den relevanten Stellen im Code genutzt werden kann.

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

Listing 5: Beispielhafte Nutzung des Interceptors

Die konkrete Implementierung des Interceptors kann dann nach Belieben mit anderen Services interagieren, um die eigentliche Abfrage an den Agenten zu schicken. Hierbei ist zu beachten, dass der `SecurityContext` *[14]* aus dem `jakarta.security.enterprise` Package nicht garantiert injiziert werden kann. Der Glassfish-Applikationsserver wirft bei einer direkten Verwendung eine Exception. In dem Beispiel in *Listing 7* wird das Problem mit einer CDI gemanagten Bean gelöst *(siehe Listing 6)*. Das Extrahieren des applikationseigenen `UserPrincipal` sollte aus dem ersten Artikel noch bekannt sein.

```
@RequestScoped
public class SecurityContextProviderImpl implements SecurityContextProvider {

  @Inject private SecurityContext securityContext;

  @Override
  public UserPrincipal getCurrentPrincipal() {
     return securityContext.getPrincipalsByType(UserPrincipal.class)
   .stream()
            .findAny()
            .orElseThrow(() -> new IllegalStateException("No user principal found"));
  }
```

Listing 6: Eine eigene CDI Bean, um den custom UserPrincipal auszulesen.

```
@OpaSecured
@Interceptor
@Priority(Interceptor.Priority.APPLICATION)
public class OpaInterceptor {

  @Inject private SecurityContextProvider securityContextProvider;

  @Inject private OpaClient opaClient;

  @AroundInvoke
  public Object checkSecurity(InvocationContext invocationContext) throws Exception {

     OpaSecured securedMetaData = invocationContext.getMethod().getAnnotation(OpaSecured.class);

     if (Objects.isNull(securedMetaData)) {
        securedMetaData = invocationContext.getTarget().getClass().getAnnotation(OpaSecured.class);
     }

     if (Objects.isNull(securedMetaData)) {
        return invocationContext.proceed();
     }

// Extrahieren von Meta-Informationen zu dem OPA Check aus der securedMetaData Annotation

     boolean allowed = opaClient.isAllowed(securityContextProvider.getCurrentPrincipal());

     if (allowed) {
        return invocationContext.proceed();
     }

     throw new SecurityException();
  }
}
```

Listing 7: Die konkrete Interceptor-Implementierung

Damit ist alles angerichtet, um in dem `OpaClient` die Query-Parameter in ein Request zu verpacken und mit dem favorisierten HTTP-Client abzuschicken *(siehe Listing 8)*.

Bei den Beispielen in den Listings ist zu beachten, dass sie stark vereinfacht wurden. Es wurde die komplette Autorisierung der Jakarta-Anwendung gegenüber des Open Policy Agent vernachlässigt. Dieser sollte im Produktivbetrieb nicht einfach aus dem Netzwerk erreichbar sein. Ebenfalls untergräbt die statische Verwendung eines Interceptors, der nur eine OPA Rule prüft, die erhoffte Dynamik durch die Nutzung des Agenten. Dies kann über die in *Listing 7* angedeutete Implementierung von Metadaten in der Interceptor-Annotation verhindert werden. So kann zum Beispiel mittels Enums angegeben werden, welche Rule beziehungsweise Policy geprüft werden soll.

Fazit

Die Nutzung einer dedizierten Policy Engine wie dem Open Policy Agent bringt im ersten Schritt eine gewisse Komplexität mit sich. Neben der Aneignung der Sprache Rego, müssen auch Themen wie eine Deployment-Strategie des neuen Services und die Erweiterung des bestehenden Codes berücksichtigt werden. Der Return on Investment ist jedoch nicht zu unterschätzen. Neben der Möglichkeit einer zentralen Verwaltung von Policies, die technologieübergreifend von allen Services genutzt werden können, wird die Nachvollziehbarkeit und Wartbarkeit einer der zentralsten Angelegenheiten des domänenspezifischen Business Codes spürbar erhört. Somit kann sich die Nutzung des Agenten auch bei Deployments mit nur einem Artefakt anbieten.

Verändern sich jedoch die Basisdaten der Policies schnelllebig oder ist die notwendige Datenbasis zum Berechnen einer Entscheidung

```
@Stateless
public class SimpleOpaClient implements OpaClient {

  private static final String OPA_URL = "http://localhost:8888/v1/data/timed/allow";

  public record OpaRequest(OpaAllowedRequest input) {}

  public record OpaAllowedRequest(long department, OffsetDateTime requestTime) {
     private OpaAllowedRequest(long department) {
        this(department, OffsetDateTime.now());
     }
  }

  public record OpaAllowedResponse(boolean result) {}

  @Override
  public boolean isAllowed(UserPrincipal userPrincipal) {

     OpaRequest opaRequest = new OpaRequest(new OpaAllowedRequest(userPrincipal.getDepartmentId()));

     OpaAllowedResponse opaResponse =
           SimpleHttpClient.sendPostRequest(OPA_URL, opaRequest, OpaAllowedResponse.class);

     if (Objects.nonNull(opaResponse)) {
        return opaResponse.result();
     }

  throw new IllegalStateException("OPA response is not valid");
  }
}
```

Listing 8: Die Anbindung des Agenten erfolgt über standardisierten REST-Request.

sehr groß, ist der Open Policy Agent nicht zwingend die beste Option. Im nächsten und letzten Artikel dieser Serie soll ein Blick auf die *SpiceDB* [15] geworfen werden, die eine Open-Source-Implementierung von Googles internem Autorisierungssystem *Zanzibar* [16] ist und eindrucksvolle Specs verspricht.

Quellen

[1] Eclipse Foundation (2024): Jakarta Security Version 4.0. *https://jakarta.ee/specifications/security/4.0/jakarta-security-spec-4.0*

[2] Eclipse Foundation (2024): Jakarta Authentication Version 3.1. *https://jakarta.ee/specifications/authentication/3.1/jakarta-authentication-spec-3.1*

[3] Eclipse Foundation (2024): Jakarta Authorization Version 3.0. *https://jakarta.ee/specifications/authorization/3.0/jakarta-authorization-spec-3.0*

[4] Eclipse Foundation (2024): Jakarta EE. *https://jakarta.ee/*

[5] Open Policy Agent (2025): Open Policy Agent. *https://www.openpolicyagent.org/*

[6] Open Policy Agent (2025): Open Policy Agent GitHub. *https://github.com/open-policy-agent*

[7] Open Policy Agent (2025): Rego Language Reference. *https://www.openpolicyagent.org/docs/latest/policy-language/*

[8] Open Policy Agent (2025): Open Policy Agent Releases. *https://github.com/open-policy-agent/opa/releases*

[9] Open Policy Agent (2025): Open Policy Agent Docker Hub. *https://hub.docker.com/r/openpolicyagent/opa*

[10] Open Policy Agent (2025): Open Policy Agent Dokumentation. *https://www.openpolicyagent.org/docs/latest/*

[11] Broadcom (2025): Spring Method Security. *https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html*

[12] Eclipse Foundation (2024): Jakarta Annotations Version 3.0. *https://jakarta.ee/specifications/annotations/3.0/annotations-spec-3.0*

[13] Eclipse Foundation (2024): Jakarta Contexts and Dependency Injection Version 4.1 – Interceptors. *https://jakarta.ee/specifications/cdi/4.1/jakarta-cdi-spec-4.1#interceptors*

[14] Eclipse Foundation (2024): Jakarta Security Version 4.0 – Security Context. *https://jakarta.ee/specifications/security/4.0/jakarta-security-spec-4.0#security-context*

[15] AuthZed (2025): AuthZed SpiceDB. *https://authzed.com/spicedb*

[16] Auth0 (2025): Zanzibar. *https://zanzibar.academy/*

Alle angegebenen URLs wurde das letzte Mal am 17.05.2025 besucht.

Jelmen Guhlke

mail@jguhlke.de

Jelmen Guhlke ist seit über 10 Jahren als zertifizierter Entwickler in der Java- und Jakarta-Welt aktiv. 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.

CLOUD NATIVE FESTIVAL

im Heide Park Soltau

CloudLand
www.cloudland.org

19. – 22. MAI 2026

#CLOUDLAND2026

Leichtgewichtiges Integration-Testing mit Microcks und Testcontainers

Dr. Florian Rademacher, Sheila Kolodziej, codecentric AG

Der Artikel zeigt die leichtgewichtige und kosteneffiziente Realisierung zuverlässiger Testumgebungen für das Integration-Testing mit Microcks und Testcontainers.

Einleitung

Testing ist maßgeblich für hohe Software-Qualität. Im Laufe der Zeit entstanden verschiedene Arten von Tests, die Software-Bestandteile und ihre Interaktionen auf unterschiedlichen Granularitätsebenen prüfen *[1]*. Unit-Tests stellen die korrekte Funktion der kleinsten, isoliert testbaren Bestandteile einer Anwendung, zum Beispiel Methoden, sicher. Component-Tests konzentrieren sich hingegen auf das Zusammenspiel von Komponenten.

Besonders komplex sind Integration-Tests, die die Integration einer Software mit externen Systemen validieren. Hierzu muss für möglichst realitätsnahe Testszenarien in der Regel das Verhalten externer Systeme passend simuliert werden, ohne ihre Produktivumgebungen zu beeinflussen. Dadurch ergeben sich für das Team der zu testenden Software unter anderem folgende Herausforderungen:

- Zuverlässige Simulation externer Systeme, insbesondere wenn die Verfügbarkeit ihrer Testumgebungen nicht garantiert ist.
- Kosteneffiziente Realisierung eigener Testumgebungen, wenn extern keine verlässlichen Testumgebungen bereitgestellt werden, und damit verbunden die Implementierung ihrer Schnittstellen.

Das Open-Source-Tool Microcks *[2]* adressiert diese Herausforderungen, indem es aus API-Definitionen kompatible Mocks bereitstellt, die sich vergleichsweise mit wenig Aufwand um simuliertes Verhalten anreichern lassen. In Kombination mit Testcontainers *[3]* unterstützt Microcks zudem virtualisiertes Integration-Testing in tendenziell verlässlichen lokalen oder Cloud-Systemen. Der Artikel führt in die entsprechenden Konzepte und Nutzung von Microcks ein.

Test Doubles

Test Doubles *[4]* sind eine zentrale Technik des Integration-Testings. Sie ersetzen die Abhängigkeiten der zu testenden Software für bestimmte Zwecke, etwa zur Steigerung von Testgeschwindigkeit oder -zuverlässigkeit. Test Doubles stehen für ihre zweckmäßige Anpassung unter der Kontrolle des Teams der zu testenden Software und stellen Schnittstellen bereit, mit denen die Software in unveränderter Form wie mit ihren tatsächlichen Abhängigkeiten interagieren kann. Stubs und Mocks sind verbreitete Arten von Test Doubles *[4]*. Während Stubs ausschließlich vordefinierte Antworten auf bestimmte Anfragen liefern, prüfen Mocks auch, ob Aufrufe wie erwartet stattfinden.

API-Mocking mit Microcks

Microcks fokussiert mockbasiertes Integration-Testing auf Basis von API-Definitionen. Im Vergleich zu etwa WireMock *[5]* bietet Microcks eine breite Unterstützung von Standards zur API-Definition, hohe Automatisierung und große Flexibilität bei der Mock-

Anpassung. Des Weiteren setzt es konsequent auf Virtualisierung, beispielsweise durch einen spezialisierten Kubernetes-Operator *[6]* oder Testcontainers-Module *[7]*. Microcks ist daher prädestiniert für Cloud-natives Entwickeln.

Mit diesen Eigenschaften adressiert Microcks die in der Einleitung beschriebenen Herausforderungen des Integration-Testings. Voraussetzung ist jedoch eine korrekte API-Definition des externen Systems, dessen Integration zu testen ist. Microcks leitet aus einer solchen API-Definition automatisiert anpassbare Mocks ab und unterstützt neben OpenAPI *[8]* und AsyncAPI *[9]* viele weitere Standards zur API-Definition *[10]*.

Ablauf des Integration Testings mit Microcks

Mockbasiertes Integration-Testing mit Microcks besteht aus folgenden Schritten:

1. Start einer Microcks-Instanz.
2. Upload von Main Artifacts, also der Definitionen zu mockender APIs in die Microcks-Instanz.
3. Upload deklarativer Secondary Artifacts, die abgeleitete API-Mocks verfeinern.
4. Interaktion mit den durch Microcks bereitgestellten API-Mocks.
5. Stopp der Microcks-Instanz.

Je nach Einsatzzweck sind gewisse Schritte optional. Microcks lässt sich zum Beispiel als dedizierter Mock-Server inklusive Web-UI betreiben. Teams können diesen Server für kontinuierliches Integration-Testing mit neuen oder geänderten API-Definitionen nutzen. Die Schritte 1 und 5 sind dann für einzelne Testsitzungen optional. Gleiches gilt für Schritt 3, wenn die Verfeinerung von API-Mocks nicht nötig ist, weil eine API-Definition bereits alle für das Integration-Testing relevanten Informationen aufweist. Im Folgenden wird angenommen, dass Microcks für virtualisiertes Integration-Testing mittels Testcontainers zum Einsatz kommt und abgeleitete Mocks zu verfeinern sind. Hierzu sind alle zuvor genannten Schritte erforderlich.

Abbildung 1 zeigt schematisch die Beziehung zwischen zu testender Software und den von einer Microcks-Instanz in einem Testcontainer bereitgestellten API-Mocks.

Beim Start der containerisierten Microcks-Instanz (Schritt 1) wird ihr eine API-Definition in Form der OpenAPI-Datei *[8]* `api-spec.yaml` übergeben (Schritt 2). Als nächstes erhält sie zwei Secondary Artifacts zur Verfeinerung abgeleiteter API-Mocks (Schritt 3). Microcks unterscheidet zwei Arten deklarativer Secondary Artifacts, die jeweils einem speziellen YAML-Format folgen:

- **API Examples:** Anreicherung von API-Mocks um Request-Response-Paare. Grundsätzlich leitet Microcks diese Paare aus einer API-Definition ab, beispielsweise aus OpenAPI-Example Objects. Mit API Examples adressiert Microcks Situationen, in denen solche Informationen fehlen und die Erweiterung einer API-Definition nicht möglich ist. In *Abbildung 1* deklariert die Datei `api-smpl.yaml` Microcks API Examples.
- **API Metadata:** Anreicherung von API-Mocks, beispielsweise um Labels oder dediziertes Verhalten (Datei `api-meta.yaml` in *Abbildung 1*).

Aus der Kombination von API-Definition und Secondary Artifacts leitet Microcks API-Mocks ab, mit denen die zu testende Software interagieren kann (Schritt 4), bevor der Testcontainer nach Abschluss der Tests beendet wird (Schritt 5). *Listing 1* zeigt den Ausschnitt einer Java-Testklasse, in der ein Microcks-Testcontainer analog zu *Abbildung 1* konfiguriert wird.

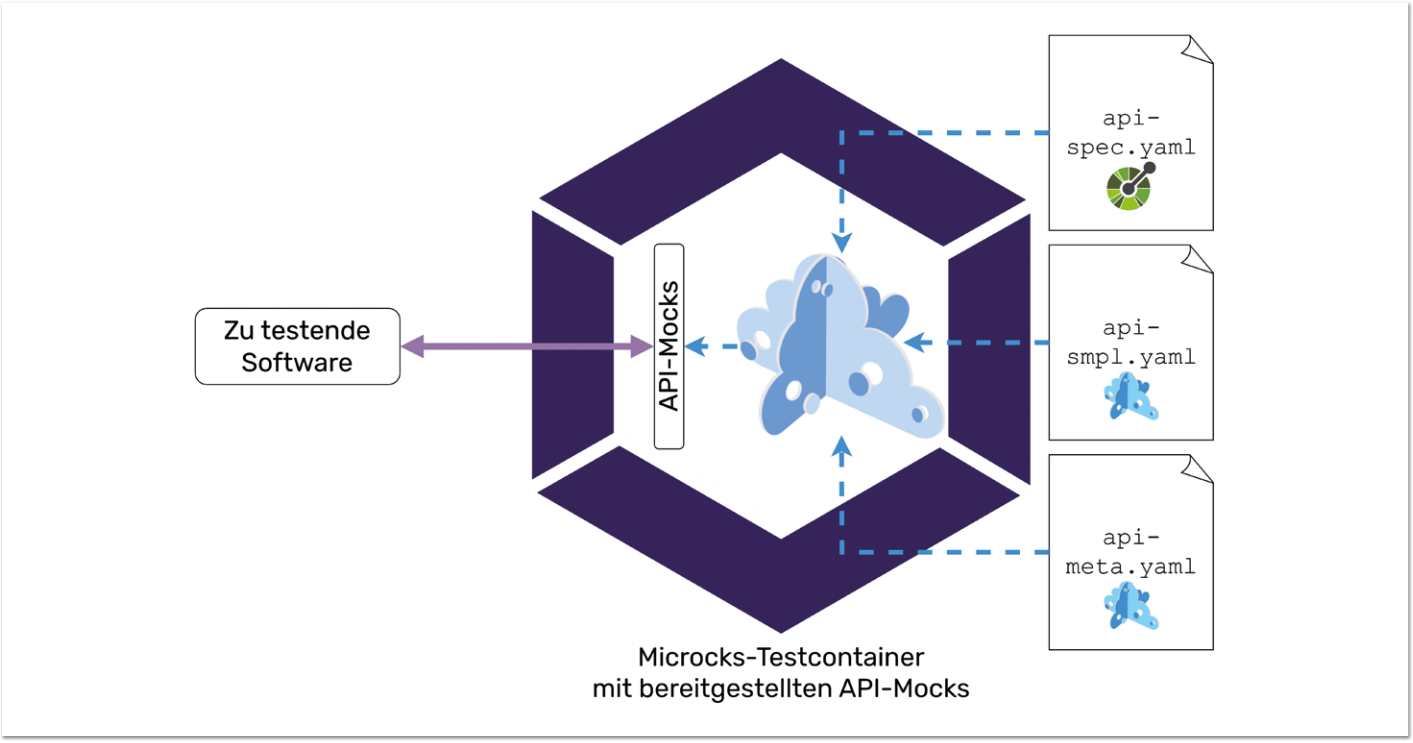


Abbildung 1: Beziehung zwischen zu testender Software und durch Testcontainers/Microcks bereitgestellter API-Mocks (© Dr. Florian Rademacher, Sheila Kolodziej)

```
@Testcontainers
class CrmTest {
  @Container
  static MicrocksContainer mContainer = new
     MicrocksContainer("quay.io/microcks/microcks-uber:1.11.2")
        .withNetwork(NETWORK)
        .withMainArtifacts("api-spec.yaml")
        .withSecondaryArtifacts("api-smpl.yaml",
           "api-meta.yaml");
}
```

Listing 1: Java-Konfiguration eines Microcks-Testcontainers gemäß Abbildung 1

`@Testcontainers` steuert den Lebenszyklus des Microcks-Testcontainers *[11]*, dessen Referenz die Variable `mContainer` speichert. Da sie mit `@Container` annotiert und `static` ist, startet und beendet Testcontainers den Microcks-Container vor beziehungsweise nach Ausführung aller Testmethoden. Für die Interaktion mit dem Container gemäß *Abbildung 1*, wird er einem Docker-Netzwerk zugeordnet (`withNetwork(`*`NETWORK`*`)`), auf das auch die zu testende Software Zugriff haben muss. Die Container-Konfiguration geschieht dann via `withMainArtifacts(...)` und `withSecondaryArtifacts(...)`. Die Methoden erhalten als String-Parameter die Dateinamen der API-Definition beziehungsweise der Microcks-spezifischen Secondary Artifacts aus *Abbildung 1*.

Das Git-Repository zum Artikel *[12]* enthält alle Listings in vollständiger Form und ein Java-Programm, das die Nutzung von Microcks für virtualisiertes Integration-Testing zeigt.

Microcks by Example

Dieser Abschnitt führt eine Beispiel-API ein, anhand derer folgende Abschnitte einige wesentliche Microcks-Features vorstellen. Ziel ist dabei, das Verhalten der API analog zu ihrer produktiven Implementierung durch ein fiktives Customer Relationship System (CRM)-System zu mocken, sodass Integration-Testing gegen das System mit geringen Kosten und hoher Verfügbarkeit der Testumgebung möglich wird.

Durch ihren CRM-Bezug stellt die Beispiel-API eine Reihe von HTTP-Operationen für die Verwaltung von Kund:innendaten bereit. *Listing 2* definiert ausschnitthaft die `login`-Operation des CRM-Systems in der OpenAPI-Datei *[8]* `api-spec.yaml` *(siehe Abbildung 1)*.

`login` ermöglicht Kund:innen die Anmeldung am CRM-System via HTTP-POST-Request und erwartet als Parameter ein Objekt vom Typ `LoginRequest`, das Kund:innen-E-Mail-Adresse und -Passwort kapselt. Bei erfolgreicher Anmeldung liefert `login` den HTTP-Statuscode 200 (OK) sowie ein `LoginResponse`-Objekt mit einem Login-Token, das für den Aufruf weiterer Operationen nutzbar ist.

Listing 3 zeigt den Ausschnitt der OpenAPI-Datei `api-spec.yaml`, der zur Definition der `existsCustomer`-Operation benötigt wird. Dies erlaubt externen Systemen die Existenzprüfung von Kund:innendaten.

`existsCustomer` wird mittels HTTP-GET-Request gegen den Endpunkt `/customers` aufgerufen und erwartet im variablen Pfadparameter `{email}` die Kund:innen-E-Mail-Adresse zur Prüfung der Existenz eines zugehörigen CRM-Datensatzes. Der HTTP-Statuscode 200 signalisiert dann die Existenz.

Ausgehend von `login` und `existsCustomer` illustrieren die folgenden Abschnitte wesentliche Microcks-Features für realitätsnahes API-Mocking.

```
paths:
  /login:
     post:
        requestBody:
           content:
              application/json:
                 schema:
                    $ref: "#/components/schemas/LoginRequest"
        responses:
           "200":
              content:
                 application/json:
                    schema:
                       $ref: "#/components/schemas/LoginResponse"
components:
  schemas:
     LoginRequest:
        required:
           - email
           - password
     LoginResponse:
        required:
           - login_token
```

Listing 2: Ausschnitt der OpenAPI-Datei `api-spec.yaml` zur Definition der beispielhaften `login`-Operation

```
/customers/{email}:
  get:
     responses:
        "200":
           description: OK
```

Listing 3: Ausschnitt der OpenAPI-Datei `api-spec.yaml` zur Definition der beispielhaften `existsCustomer`-Operation

API Examples: Mock-Anreicherung um Request-Response-Paare

Mit API Examples unterstützt Microcks die Anreicherung von Mocks um Request-Response-Paare, ohne Anpassung der zugrundeliegenden API-Definition. Damit lassen sich für die CRM-API beispielsweise Credentials für Integration-Tests simulieren. *Listing 4* zeigt ein entsprechendes Microcks API Example aus der Datei `api-smpl.yaml` *(siehe Abbildung 1)*.

Das API Example erweitert die `login`-Operation der CRM-API *(siehe Listing 2)*. Dazu referenziert es unterhalb des YAML-Mappings `operations` die Operation über ihren Mock-Namen. Bei OpenAPI setzt Microcks Mock-Namen aus HTTP-Methode und Endpunkt gemockter Operationen zusammen (in *Listing 4* also "`POST /login`" für die `login`-Operation). Jedes Mapping unterhalb eines Mock-Namens interpretiert Microcks als Request-Response-Paar. Der Request- und Response-Teil eines solchen Paares findet sich unterhalb der Mappings `request` beziehungsweise `response`, und dort vorrangig im body-Mapping. *Listing 4* spezifiziert demnach, dass Microcks für `login` ein Request-Response-Paar existing_customer anlegen soll, dessen Request-Teil im HTTP-Body ein JSON-Objekt erwartet, das der login-Definition *(Listing 2)* folgend Credentials für den fiktiven Kunden John Doe simuliert. Der Response-Teil von `existing_customer` legt fest, dass Microcks bei Mock-Aufruf mit dem gegebenen Request den HTTP-Statuscode 200 und ein Login-Token in Form eines JSON Web Token (JWT) *[13]* zurückgibt. `existing_customer` mockt also eine gültige Anmeldung durch die CRM-API.

```
kind: APIExamples
operations:
  "POST /login":
     existing_customer:
        request:
           body:
              email: "john.doe@example.com"
              password: "securePassword1!"
        response:
           status: "200"
           body:
              login_token: "eyJhbGciOiJIUzI1NiJ9..."
```

Listing 4: Microcks API Example für simulierte Test-Credentials der `login`-Operation der CRM-API

Parameter Constraints: Statische Beschränkung von Mock-Aufrufen

Durch Parameter Constraints schränkt Microcks Mock-Aufrufe auf Basis statischer Request-Eigenschaften ein. Für die `login`-Operation der CRM-API ist eine solche Eigenschaft ein erwarteter API-Key im HTTP-Header `Authorization`. *Listing 5* zeigt das Mocking eines ausschließlich für das Testing von `login` verwendeten API-Keys mit Hilfe einer Parameter Constraint als Microcks API Metadata in der Datei „api-meta.yaml" *(siehe Abbildung 1)*.

```
kind: APIMetadata
operations:
  "POST /login":
     parameterConstraints:
        - name: Authorization
          in: header
          mustMatchRegexp: "^Bearer\\s\\QeyJhbGci...\\E$"
```

Listing 5: Microcks API Metadata für eine Parameter Constraint, die den API-Key der `login`-Operation simuliert.

Analog zu API Examples *(Listing 4)* beginnt eine API Metadata-Deklaration mit dem `operations`-Mapping, das wiederum Mock-Namen kapselt. Das parameterConstraints-Mapping unter "`POST /login`" definiert dann eine Parameter Constraint für die `login`-Operation, die Microcks anweist, Aufrufe des `login`-Mocks nur dann zuzulassen, wenn der Wert des `Authorization`-Headers einem bestimmten regulären Ausdruck folgt (`mustMatchRegexp`). Konkret muss der Wert aus dem String „Bearer" und einem festgelegten JWT-API-Key *[13]* bestehen.

Dispatcher & Stafeful Mocks: Verhalten und Datenaustausch für Mocks

Durch Dispatching mappt Microcks Mock-Requests auf Mock-Responses. Microcks kann selbständig implizite Dispatcher aus API-Definitionen und Secondary Artifacts ableiten *[14]*. Zum Beispiel kann ein Dispatching automatisiert nach Pfadparametern wie `{email}` *(Listing 3)* erfolgen.

Durch explizite Dispatcher unterstützt Microcks die weitergehende Anreicherung von Mocks um Verhalten *[15]*. So erlaubt der deklarative JSON-BODY-Dispatcher die Auswahl von Mock-Responses anhand struktureller Eigenschaften von JSON-Requests. Deutlich mächtiger ist der SCRIPT-Dispatcher, durch den sich Mocks über Groovy-Skripte mit Logik versehen lassen. *Listing 6* illustriert dies für `existsCustomer` *(Listing 3)*.

Das `dispatcher`-Mapping bestimmt den Dispatcher-Typ, während das `dispatcherRules`-Mapping für SCRIPT-Dispatcher die jeweiligen Groovy-Skripte kapselt. Über das implizit von Microcks bereitgestellte `mockRequest`-Objekt kann der Dispatcher auf Request-Eigenschaften zugreifen. So besitzt das Objekt unter anderem die `getURIParameters()`-Methode, mit der in *Listing 6* der konkrete Wert des Pfadparameters `{email}` in der Variable `reqEmail` gespeichert wird. Nachfolgend prüft das Dispatcher-Skript, ob die E-Mail im impliziten store-Objekt vorhanden ist. Durch dieses Objekt unterstützt Microcks Stateful Mocking: `store` ist eine Map, die Strings zwischen unterschiedlichen Mock-Aufrufen speichert. Für die CRM-API ist damit das Integration Testing der Kund:innen-Registrierung möglich: Durch den Aufruf des Registrierungsendpunkts der produktiven API lassen sich neue Kund:innendaten anlegen. Mittels `store` kann der Mock des Registrierungsendpunkts *[12]* neue Kund:innen-Datensätze speichern, für deren Existenzprüfung der `existsCustomer`-Mock dann wie erwartet Erfolg vermeldet. Der SCRIPT-Dispatcher gibt dazu den Namen des entsprechenden Microcks API Examples für `existsCustomer` zurück ("`customer_exists`") *[12]*.

```
"GET /customers/{email}":
  dispatcher: SCRIPT
  dispatcherRules: |-
     def reqEmail = mockRequest.getURIParameters().get("email")
     if (store.get(reqEmail) != null)
        return "customer_exists"
```

Listing 6: Microcks API-Metadata für einen SCRIPT-Dispatcher, der das Verhalten der existsCustomer-Operation der CRM-API simuliert.

Abschluss

Microcks und Testcontainers können Aufwand und Fehleranfälligkeit mockbasierter Integration Tests durch Automatisierung und Virtualisierung deutlich reduzieren. Voraussetzung sind jedoch korrekte API-Definitionen. Microcks bietet neben den gezeigten Features unter anderem noch dynamische Mock-Inhalte und Contract Testing *[2]*.

Quellen

[1] Ham Vocke (2018): The Practical Test Pyramid. *https://martinfowler.com/articles/practical-test-pyramid.html*.

[2] The Microcks Team (2025): Microcks-Webseite. *https://microcks.io*.

[3] AtomicJar, Inc. (2025): Testcontainers-Webseite. *https://testcontainers.com*.

[4] Martin Fowler (2007): Mocks Aren't Stubs. *https://martinfowler.com/articles/mocksArentStubs.html*.

[5] WireMock, Inc. (2025): WireMock-Webseite. *https://wiremock.org*.

[6] The Microcks Team (2025): On Kubernetes with Operator. *https://microcks.io/documentation/guides/installation/kubernetes-operator*.

[7] The Microcks Team (2025): Testcontainers-Module für Microcks. *https://testcontainers.com/modules/microcks*.

[8] OpenAPI Initiative (2025): OpenAPI-Webseite. *https://openapis.org*.

[9] AsyncAPI Initiative (2025): AsyncAPI-Webseite. *https://asyncapi.com*.

[10] The Microcks Team (2025): Artifacts. *https://microcks.io/documentation/overview/main-concepts/#artifacts*.

[11] AtomicJar, Inc. (2025): Testcontainers container lifecycle management using JUnit 5. *https://testcontainers.com/guides/testcontainers-container-lifecycle*.

[12] Dr. Florian Rademacher (2025): Git-Repository zum Artikel. *https://github.com/frademacher/microcks-iam-example*.

[13] M. Jones et al. (2015): JSON Web Token (JWT). *https://datatracker.ietf.org/doc/html/rfc7519*, IETF.

[14] The Microcks Team (2025): Inferred dipatchers. *https://microcks.io/documentation/explanations/dispatching/#inferred-dispatchers*.

[15] The Microcks Team (2025): Advanced dispatchers and rules. *https://microcks.io/documentation/explanations/dispatching/#advanced-dispatchers-and-rules*.

Dr. Florian Rademacher

florian.rademacher@codecentric.de

Florian arbeitet seit 2023 als Senior Consultant bei der codecentric AG. Zuvor hat er als Wissenschaftler unter anderem zu Microservice-Architekturen geforscht. Zu Florians Kernkompetenzen zählen alle Tätigkeiten der Konzeption, Entwicklung, des Testens und Betreibens von Softwaresystemen mit monolithischen oder lose gekoppelten Architekturen. Florian ist zudem einer der Leiter des Services „Software Modernization", mit dem codecentric Kundensysteme ganzheitlich zukunftsfähig macht.

Sheila Kolodziej

sheila.kolodziej@codecentric.de

Sheila arbeitet nach Abschluss ihres Informatik-Masters an der TH Köln seit 2021 als Consultant bei der codecentric AG. Ihre Kernkompetenzen liegen in der Weiterentwicklung, Qualitätssicherung und Betreuung von Softwaresystemen – unabhängig davon, ob diese monolithisch oder in verteilten Architekturen organisiert sind.

Angular – The Silent Revolution, Teil 2

Stephan Rauh, Alexander Pahn, Julian Schmitt

www.ijug.eu

Im ersten Artikel dieser Serie haben wir euch ein paar spannende Änderungen in Angular vorgestellt. Jetzt geht es weiter. Angular wird von Grund auf renoviert – und das schließt auch und gerade heilige Kühe wie die Change Detection oder die Dependency Injection ein.

inject()

Kommen wir zu etwas sehr Überraschendem: Die Java-Community hat in den letzten Jahren die Constructor-Injection zum Königsweg erkoren und beginnt, Field-Injection zu verteufeln. Dafür gibt es gute Gründe und das Thema pflegt große Emotionen zu wecken. Und was macht Angular? Es findet ebenfalls gute Gründe, um den umgekehrten Weg zu beschreiten. Angular hatte von Anfang an auf Constructor-Injection *(siehe Listing 1)* gesetzt und löst diese jetzt durch Field-Injection ab *(siehe Listing 2)*.

```
@Component({ ... })
export class AppComponent {
  constructor(private chessboard: ChessboardService) {}
}
```

Listing 1: Traditionelle Constructor-Injection

```
@Component({ ... })
export class AppComponent {
  private chessboard = inject(ChessboardService);
}
```

Listing 2: Field-Injection mit inject()

Der neue Ansatz hat eine ganze Reihe von Vorteilen. Diese beginnen damit, dass wir bisher einen Decorator brauchten, um der Dependency Parameter mitzugeben. Decorators sind auf der Zielgeraden, in den JavaScript-Standard übernommen zu werden, aber noch kann sich die Spezifikation ändern. Wenn wir Decorators verwenden, haben wir immer ein Migrationsrisiko. Bei der `inject`-Funktion können wir Optionen einfach als JSON-Object mitgeben *(siehe Listing 3)*.

```
@Component({ ... })
export class AppComponent {
  private chessboard = inject(ChessboardService, { player:
'White'});
}
```

Listing 3: inject() mit Parametern

Bei Vererbung ist die Constructor-Injection außerdem lästig, was ein weiteres Argument für die Field-Injection darstellt. Das ist allerdings ein schwaches Argument: Angular-Komponenten unterstützen Ver-

erbung ohnehin nur unvollständig, und die meisten Leute sehen sie auch als Antipattern an. Falls ihr von einer Komponente ableiten wollt, überlegt, ob ihr das nicht auch mit einer Direktive lösen könnt und schaut euch die Direct-Composition-API an.

Ihr könnt die `inject()`-Funktion an sehr vielen Stellen verwenden – überall dort, wo ein Injection-Context vorhanden ist. Mit der Funktion `runInInjectionContext()` geht das sogar fast überall *[4]*. Damit könnt ihr beispielsweise Conditional Injects oder Lazy Injects umsetzen. Die Ressource wird erst dann erzeugt, wenn sie wirklich benötigt wird – mit dem Vorteil, dass sie womöglich nie erzeugt wird.

Jedoch solltet ihr diese Möglichkeit mit Bedacht verwenden. Der Nachteil ist, dass ihr sehr schnell den Überblick verliert, wo welche Dependency verwendet wird. Functional Inject ist ein scharfes Schwert, an dem ihr euch auch schneiden könnt!

Es gibt noch weitere Gründe, die für `functional inject()` sprechen: Es ist häufig lesbarer, ihr könnt leichter Kommentare anfügen und die Typinferenz ist oft besser. Ihr könnt die Details bei Rainer Hahnekamp *[1]* nachlesen. Dort erfahrt ihr auch, wie ihr eure Tests für `inject()` anpassen müsst.

Apropos anpassen: Die Migration ist denkbar einfach. Es gibt ein Schematics dafür: `ng generate @angular/core:inject`. Ob und wann Angular die Constructor-Injection abschafft, ist aktuell nicht abzusehen. Der einzige konkrete Hinweis ist, dass der Angular-Styleguide euch die Migration empfiehlt *[10]*.

Und was ist der eigentliche Grund für die Umstellung?

Wenn ihr ein wenig tiefer bohrt, findet ihr noch einen ganz anderen Grund, der für Functional Inject spricht. TypeScript ist eine Erweiterung von JavaScript und versucht gleichzeitig kompatibel zu bleiben sowie die zukünftige Entwicklung vorauszuahnen. Manchmal geht das schief. In diesem Fall ist die Syntax gleich, aber es gibt einen subtilen Unterschied bei der Semantik: Die Reihenfolge, in der die Klasse initialisiert wird, unterscheidet sich zwischen TypeScript und JavaScript. TypeScript führte ursprünglich den Konstruktor zuerst aus und initialisierte erst dann die Attribute der Klasse. Modernes JavaScript macht das genau andersherum. Das wurde den TypeScript-Entwicklern in Version 3.7 bewusst und sie haben das Flag `useDefineForClassFields` eingeführt, um beide Varianten zu unterstützen *[2]*. Seit TypeScript 4.9 ist `useDefineForClassFields` standardmäßig aktiviert.

Die Constructor-Injection von Angular verlässt sich darauf, dass der Constructor zuerst ausgeführt wird *[3]*. Also muss Angular das Flag deaktivieren, um die Constructor-Injection zu unterstützen. Das ist auf Sand gebaut: Früher oder später kann der Zeitpunkt kommen, an dem TypeScript nur noch die neue Variante unterstützt.

Functional Inject ist eine elegante Lösung, um dem Problem aus dem Weg zu gehen und gleichzeitig noch alle oben beschriebenen Vorteile zu bieten.

Server-Side-Rendering (SSR)

Es gibt einige kleine Optimierungen im Bereich SSR. Beim SSR werden eure Seiten auf dem Server gerendert und das fertige HTML an den Browser geschickt. Im Browser wird die Seite dann mittels JavaScript belebt (Hydration).

Klingt das umständlich? Ja – und das ist es auch. Dass es sich trotz höherer Komplexität lohnen kann, könnt ihr im Moment bei meiner (Stephans) Website ***(beyondjava.net)*** sehen. Ich hatte sie Anfang März sehr spontan von meinem AWS-Account zurück zu meinem klassischen Webhoster zurückgeholt. Dabei ist mir (hoffentlich nur vorübergehend) das SSR verlorengegangen. Jetzt dauert es deutlich länger, die Seite zu laden. Laut WebPageTest.org dauert es jetzt sechs Sekunden, ehe die Seite lesbar ist. Mit SSR waren es rund zwei Sekunden. Auch das ist eigentlich nicht gut: Die allgemeine Empfehlung ist, unter einer Sekunde zu bleiben. Ansonsten springen euch die potenziellen Kunden oder – in meinem Fall – Leserinnen und Leser ab.

Braucht ihr das? Vermutlich nicht – und deswegen fassen wir uns in diesem Kapitel auch kurz. Für eine hausinterne Anwendung ist Server-Side-Rendering normalerweise irrelevant. So etwas braucht ihr nur, wenn ihr eine Anwendung im Internet bereitstellt, die sich an Endkunden richtet, und auch nur, wenn euch Laufkundschaft wichtig ist. Regelmäßige Anwenderinnen und Anwender sind bereit, ein paar Sekunden am Anfang zu warten. Es geht beim Server-Side-Rendering nur um das initiale Laden der Anwendung. Wenn die Anwendung ein paar Stunden läuft, spielt das keine Rolle mehr.

Trotzdem steht Angular im Wettbewerb mit React/Next.js, Vue.js und Svelte – und mit diesen Frameworks und Libraries ist es einfach, eine Website zu bauen, die nicht erst lange lädt, sondern sofort da ist. Also investiert das Angular-Team seit Angular 16 in Server-Side-Rendering und sein leichtgewichtiges Geschwisterchen, die Static-Site-Generation. Bei letzterer werden alle Startseiten schon beim Kompilieren vorgerendert und können auf einem einfachen Webserver (zum Beispiel Apache oder nginx) abgelegt werden. Server-Side-Rendering hingegen erfordert einen node.js-Webserver. Dafür bietet es mehr Flexibilität und mehr Optimierungsmöglichkeiten.

Angular 16 führte die Full-Application-Hydration ein. „Hydration" ist dabei ein plastischer Begriff für das, was passiert: Erst wird die reine HTML-Datei angezeigt, und anschließend wird euer Angular-Code hinterhergeschickt und füllt das trockene HTML-Pflänzchen mit Leben. Die Trockenblume wird quasi „rehydriert".

In Angular 17 kam @defer hinzu, das euch erlaubt, auf einer Seite einzelne Komponenten verzögert zu rendern. Typischerweise sind das die Komponenten, die erst nach Scrollen sichtbar werden. In Angular 18 ist das Event Replay und in Angular 19 der Incremental Hydration Modus hinzugekommen.

Aus Benutzersicht ist vermutlich das Event Replay eine besonders nützliche Sache. Server-Side-Rendering bedeutet eigentlich, dass wir tricksen. Wir sorgen dafür, dass die Anwendung praktisch ohne Verzögerung zum Browser geschickt wird. Anwenderinnen und Anwender können also sofort mit der Arbeit beginnen. Und wenn sie sehr schnell sind, tun sie das, bevor das Anwendungspflänzchen vollständig bewässert ist, was bedeutet, dass die Benutzeraktionen verloren gehen. Event Replay puffert die Ereignisse und spielt sie nach der Hydration ab. Damit können eure Anwenderinnen und Anwender sofort mit der Arbeit loslegen, daher fällt unsere kleine Trickserei nicht weiter auf.

Wenn ihr euch für das Thema interessiert, erfahrt ihr im Video von Jessica Janiuk mehr [6].

Zoneless Angular

Der Verzicht auf zone.js ist schon seit Jahren ein Wunsch des Angular-Teams. Zone.js ist genial, aber nicht ganz ohne Probleme. Es kostet Performance, erschwert das Debugging durch lange Stacktraces, sorgt in seltenen Fällen selbst für Fehler und verwendet Monkey Patching.

Angular 19 bietet euch die Möglichkeit, mit einem experimentellen Feature auf zone.js zu verzichten und euch damit schon auf die Zukunft vorzubereiten [5]. Eventuell wird es schon in Angular 20 zu einem offiziellen Feature, aber das heißt nicht zwangsläufig, dass ihr es dann auch nutzen könnt: Alle Bibliotheken müssen auch mitspielen. Das kann einige Zeit dauern.

Soweit die Kurzfassung – aber wir hatten euch Tiefgang und Hintergründe versprochen. Fangen wir also mit dem Monkey Patching an.

Das ist eine Technik, die es in dynamischen Sprachen wie Python, Ruby, Smalltalk und JavaScript gibt. In JavaScript geht sie erstaunlich weit. Ihr könnt die Methoden einer Klasse zur Laufzeit jederzeit ändern. Und das gilt sogar für die globalen Funktionen von JavaScript. Angular nutzt das, um eine Vielzahl von Funktionen zu verändern. Dazu gehören unter anderem `setTimeout()`, `addEventListener()`, `fetch()`, `XMLHttpRequest` und alles, was mit `Promises` zu tun hat.

Für Angular ist das genial. Alle diese Funktionen werden oft verwendet, um die Attribute der Komponenten und Services zu verändern (also den State der Anwendung). Angular hängt sich in diese Funktionen ein und triggert automatisch die Change Detection. Das sorgt für ein sehr entspanntes Programmiermodell. Im Großen und Ganzen braucht ihr gar nicht zu wissen, was Change Detection ist. Sie funktioniert einfach. Jedes Mal, wenn ihr ein Attribut verändert, wird die Webanwendung aktualisiert. Mehr erfahrt ihr auf Stephans Blog [9].

Der Nachteil ist, dass zone.js wirklich sehr viele APIs erweitert und damit die Performance eurer Anwendung verschlechtert. Bei den meisten Angular-Anwendungen spielt das keine Rolle, aber es ist beispielsweise bei ngx-extended-pdf-viewer [8] deutlich spürbar. Seitdem ich (Stephan) einen Teil des Codes in einem `runOutsideAngular`-Block ausführe, läuft die Anwendung spürbar schneller.

Beim Debuggen ist zone.js auch ein Ärgernis. Eigentlich bieten Chrome und Co. sehr gute Werkzeuge, um Event Listener zu finden und sie zu debuggen. Bei Angular zeigen sie aber alle auf zone.js. Es ist mühsam, herauszufinden, welche Methode wirklich aufgerufen wird. Und schaut euch mal die Stack Traces an: Sobald ihr die Features von zone.js nutzt, wird der Stack Trace lang und unübersichtlich. In den meisten Angular-Anwendungen sollte das kein Problem sein. Aber sobald ihr Third-Party-Libraries verwendet, kann das anders aussehen. Ein extremes Beispiel ist ngx-extended-pdf-viewer, eine Library, die stark auf asynchrone Prozesse setzt, mit dem Ergebnis, dass die Stack Traces hauptsächlich zone.js-Methoden zeigen. Eure selbstgeschriebenen Methoden sind kaum noch zu sehen.

Monkey Patching und Security

Es kommt noch schlimmer: Monkey Patching ist ein schönes Werkzeug für Angreifer. Mögliche Angriffsvektoren sind unter anderem Browser-Plug-ins, Cross-Site-Scripting, manipulierte CDN-Scripte oder sogar Bookmarks mit JavaScript.

Streng genommen gehört das nicht zum Thema dieses Artikels, aber weil es so wichtig ist, wollen wir euch ein paar Tipps an die Hand geben, wie ihr euch schützt:

- Verwendet Content Security Policy. Damit könnt ihr verhindern, dass unerwünschte Skripte ausgeführt werden.
- Führt regelmäßig `npm audit` aus und behebt die dort gemeldeten Schwachstellen.
- Schützt wichtige APIs mit `Object.freeze` vor Manipulation.
- Wenn ihr Skripte von einem CDN ladet, verwendet ein SRI-Hash. Am besten, ihr lasst das Laden aber ganz bleiben: Die technischen Vorteile sind unbestritten, aber ihr verletzt damit mit großer Wahrscheinlichkeit die Vorschriften der DSGVO. Geht dem potenziellen Ärger einfach aus dem Weg, indem ihr die Dateien selbst hostet. Dann habt ihr auch die volle Kontrolle darüber.

Was verändert sich mit zoneless Angular?

Es gibt also zahlreiche Gründe, auf zone.js zu verzichten. Das ist natürlich eine enorme Herausforderung. Das Schöne an Angular ist, dass ihr Attribute in TypeScript einfach ändern könnt und die Änderungen werden automatisch im Browser angezeigt. Einfach so. Ihr müsst nichts dafür tun. Wenn zone.js wegfällt, geht das nicht mehr so einfach. Dann müsst ihr Angular aktiv Bescheid geben, dass sich etwas geändert hat, und damit verliert Angular einen großen Teil seines Charms.

Das ist nicht komplett neu. Ihr kennt bestimmt die `OnPush`-Strategie für die Change Detection. Wenn Performance im Vordergrund steht, waren Angular-Entwicklerinnen und -Entwickler schon immer bereit, Kompromisse einzugehen.

Wenn ihr zone.js deaktiviert, ist das ähnlich. Im Prinzip stellt ihr die komplette Anwendung auf `OnPush` um. Jede Methode, die Attribute einer Klasse verändert, muss `ChangeDetectorRef.markForCheck` aufrufen. Zum Glück gibt es ein paar Ausnahmen, bei denen die Change Detection wie bisher vollautomatisch funktioniert:

- Änderungen eines Signals (insbesondere der `input()`-Signals),
- die async-Pipe in den HTML-Templates und
- Event Listener – also alles in runden Klammern im HTML-Template, wie zum Beispiel das `(click)`-Event.

Damit ist ein großer Teil der Änderungen abgedeckt. Was nicht mehr funktioniert, sind die automatischen Change Detections bei `setTimeout()` und bei HTTP-Requests. Hier müsst ihr explizit `ChangeDetectorRef`.markForCheck oder `detectChanges()` aufrufen.

Angular 19 liefert einen ersten, experimentellen Support für zoneless Angular. Das ist ein großer Paradigmenwechsel. Es kann gut sein, dass es noch einige Versionen dauert, bis aus dem experimentellen Feature ein First-Class-Citizen wird. Angesichts der Bedeutung des Features empfehlen wir euch, euch jetzt schon damit vertraut zu machen.

LinkedSignal

Wir hatten im ersten Teil dieser Serie schon über Signale gesprochen. Es gibt noch viel mehr zum Thema Signals zu berichten. Allerdings sind einige Features noch als experimentell gekennzeichnet oder befinden sich noch in der Entwicklervorschau. Sie sollten also noch nicht produktiv eingesetzt werden. Dennoch lohnt sich ein Blick auf das, was kommen wird.

So zum Beispiel das `linkedSignal`. Seit Einführung von Signalen kam immer wieder dieselbe Frage auf: Wie kann ich einen Wert, der von einem Signal abhängig ist, zusätzlich manuell überschreiben? Also einen abgeleiteten Wert unter bestimmten Bedingungen aktiv überschreiben, aber wieder zurücksetzen, wenn sich der Quellwert ändert.

Wieso sollte man das wollen? Ist es nicht Sinn und Zweck eines Computed Signals, einen Wert abzuleiten? Auf den ersten Blick wirkt es wie ein Antipattern oder eine falsche Architektur. Tatsächlich gibt es aber mehrere Gründe, warum man genau dieses Verhalten benötigt, insbesondere dann, wenn man Gebrauch von Input Signals macht.

Ein Beispiel: In unserem rundenbasierten Spiel wird immer für die letzte Runde angezeigt, wer welche Aktion durchgeführt hat. Die Runden werden in einer Liste aufgeführt und der Spieler kann auf einen beliebigen Eintrag klicken, um die Details der entsprechenden Runde zu sehen. Eine Komponente soll dabei die Liste sowie die Details darstellen und somit auch für den State verantwortlich sein.

Man könnte nun auf folgende Idee kommen, um diese Anforderung abzubilden: Wir nutzen ein Input-Signal für die Liste der Runden und ein internes Signal für den Index der ausgewählten Runde. Bei einem Klick auf eine Runde in der Liste setzen wir den Index in unserem internen Index-Signal. Um auf Änderungen der Liste zu reagieren, erstellen wir einen `effect`, der dann unser Index-Signal entsprechend aktualisiert. Falls das verwirrend klingt, wird es mit Blick auf *Listing 4* schnell klar.

```
@Component({
  selector: 'app-runden',
  imports: [],
  template: `<!-- Template -->`,
})
export class RundenComponent {
  runden = input<Runde[]>();
  rundenIndex = signal(-1);

  constructor() {
     effect(() => {
        const runden = this.runden() ?? [];
        this.rundenIndex.set(runden.length -1);
     });
  }

  setRundenIndex(index: number){
     this.rundenIndex.set(index);
  }
}
```

Listing 4: Signal-Abhängigkeiten mittels effect()

Die Idee klingt auf den ersten Blick wunderbar und nach genau dem richtigen Anwendungsfall. Allerdings ist es so, dass `effect` gar nicht dafür vorgesehen ist, einen Einfluss auf andere Signals zu haben. Sogar das Angular-Team rät davon ab *[7]*. Das Problem ist, dass es mehrere Quellen gibt, die technisch gesehen erstmal unabhängig voneinander sind und wir versuchen dann manuell diese Beziehung herzustellen. Das bietet Potential für Fehler und Bugs.

Hier kommt nun endlich das `linkedSignal` zum Einsatz. Mit diesem können wir explizit diese Beziehung herstellen und erhalten einen „Hybrid" aus Signal und `computed`. Wir können es also manuell aktualisieren, aber es reagiert auch auf Änderungen seiner Abhängigkeiten. Im *Listing 4* seht ihr die Variante mit `effect()`, in *Listing 5* die Variante mit `linkedSignal` und in *Listing 6* findet ihr eine alternative Lösung, die ihr nutzen könnt, falls ihr auf einer Angular-Version arbeitet, die dieses neue Feature noch nicht hat. Die Idee dazu kommt von ebenfalls vom Angular-Team.

Signal Queries: ViewChild(ren), ContentChild(ren)

Seit Angular 19 sind die Signal Queries stabil und stellen eine Alternative für die bisherigen Decorator-basierten Queries dar. Aber was war das noch gleich? Und was ist der Unterschied zwischen einem `View`- und einem `ContentChild`?

View- und Content-Queries erlauben es euch, nach Elementen innerhalb eurer Komponente zu suchen. Dafür könnt ihr entweder klassische CSS-Selektoren, Template-Referenz-Variablen oder die Ziel-Komponentenklasse nutzen. Der Unterschied zwischen `View-` und `ContentChild` ist auf den ersten Blick subtil. Einfach ausgedrückt:

- Ein `ViewChild` ist immer etwas, das ihr bereits bei der Entwicklung eurer Komponente in eurem Template sehen könnt, also alles, was ihr selbst aktiv hineinpackt. Das können andere Komponenten sein oder eben normale HTML-Elemente.
- Ein `ContentChild` ist immer etwas, das über Content Projection in eure Komponente hineinprojiziert wird. Während ihr euch also zwar überlegt habt, was potenziell für Elemente vorhanden sein werden oder könnten, seht ihr im Template lediglich ein `<ng-content>`-Element. Was konkret im `<ng-content>` eingesetzt wird, entscheidet sich erst zur Laufzeit. Personen, die eure Komponente verwenden, bestimmen, was in das `<ng-content>` eingesetzt wird. Oder anders ausgedrückt: Wo Content draufsteht, ist auch Content drin, und der kann von anderen Personen geliefert werden.

Die Nutzung von Query Signals ist etwas leichter und kürzer als bei den Vorgängern. In *Listing 7 und 8* zeigen wir euch einen Vergleich anhand einer Tab-Panel-Komponente. Dort seht ihr auch, wie ihr Angular mitteilen könnt, dass ein Element immer verfügbar ist.

Wie bei allen Signalen, integrieren sich auch die Signal Queries wunderbar in das neue reaktive System. Man kann also in einem Computed Signal auf Änderungen des ViewChild reagieren und daraus einen anderen Wert ableiten.

Noch mehr Neues

In den letzten Jahren hat sich noch viel mehr getan. Es ist sogar so viel, dass wir nicht alles in ein bis zwei Artikeln angemessen be-

```
@Component({
  selector: 'app-runden',
  imports: [],
  template: `<!-- Template -->`,
})
export class RundenComponent {
  runden = input<Runde[]>();

 // Das kann bei einfacher Logik auch in derselben Art geschrieben werden wie ein computed Signal
  // Die hier gezeigte Variante bietet auch die Möglichkeit in einem Callback ein Objekt aus mehreren Signalen zu bauen
  rundenIndex = linkedSignal({
     source: this.runden,
     computation: (runden: Runde[] = []) => runden.length -1
  });


 setRundenIndex(index: number){
     this.rundenIndex.set(index);
  }
}
```

Listing 5: Signal-Abhängigkeiten mittels linkedSignal()

```
@Component({
  selector: 'app-runden',
 imports: [],
 template: `<!-- Template -->`,
})
export class RundenComponent {
 runden = input<Runde[]>();
 state = computed(() => {
  const runden = this.runden() ?? [];
  return {
   runden: runden,
   // Es ist absolut erlaubt in einem computed Signal ein neues Signal zu erstellen
   rundenIndex: signal(runden.length -1)
  }
 });


 setRundenIndex(index: number){
  this.state().rundenIndex.set(index);
 }
}
```

Listing 6: Signal-Abhängigkeiten mittels computed()

```
@Component({
  selector: 'app-tab-panel',
  imports: [],
  template: `
     <div>
        <h2 #tabHeader>Tab</h2>
        <ng-content />
     </div>
  `,
})
export class TabPanelComponent {
  header = viewChild.required<ElementRef<HTMLHeadingElement>>('tabHeader');
  aktionen = contentChildren(TabButton);
  aktionsTexte = computed(() => this.aktionen().map(aktion => aktion.text));
}
```

Listing 7: Signal Queries

```
@Component({
  selector: 'app-tab-panel',
  imports: [],
  template: `
     <div>
        <h2 #tabHeader>Tab</h2>
        <ng-content />
     </div>
  `,
})
export class TabPanelComponent implements AfterContentInit {
  @ViewChild('tabHeader', {static: true, read: 'ElementRef'}) header: ElementRef<HTMLHeadingElement>;

 @ContentChildren(TabButton) aktionen: QueryList<TabButton>;
  aktionsTexte: string[];

  ngAfterContentInit() {
     this.aktionsTexte = this.aktionen.map(aktion => aktion.text)
  }
}
```

Listing 8: Decorator Queries

schreiben können. Daher wollen wir euch auf eine kleine Entdeckungsreise schicken, die ihr bei Interesse eigenständig vertiefen könnt. Dafür findet ihr nachfolgend ein paar Stichworte mit einer kurzen Beschreibung.

Signals

Neben den bereits erwähnten Signalen stehen jetzt auch einige andere bekannte Decorator in einer Signal-Variante zur Verfügung. Besonders interessant ist dabei das Output-Signal.

Ganz neu, mit Angular 19 aber noch experimentell, sind `resource` und `rxResource`. Ein Blick darauf lohnt sich aber bereits jetzt – zum Beispiel in der achtteiligen Serie der Angular Architects *[10]*.

Directive Composition API

Schon etwas länger dabei, aber sicherlich noch nicht so bekannt. Mit der Directive Composition API könnt ihr Directives direkt in euren eigenen Komponenten, im `@Component`-Decorator, anwenden. Diese werden dann automatisch an jeder neuen Instanz angehängt. Dies sollte allerdings nicht zu häufig benutzt werden, da es zu Performance-Problemen führen kann.

Schematics

Eine Migration zu neuen Funktionen ist häufig nicht trivial. Glücklicherweise liefert uns das Angular-Team einige Schematics, die uns einiges an Arbeit abnehmen. Schaut euch gerne diese hier an: Control-Flow, Signal-Input-Migration, Output-Migration und Cleanup unused imports. Ihr werdet sie lieben!

Resümee

Angular hat einen weiten Weg zurückgelegt. Wir sind uns sicher, dass es uns noch eine ganze Weile begleiten wird und sich dabei immer wieder aufs Neue verjüngen wird. Allein schon die Änderungen, die in den letzten drei Jahren passiert sind, würden für eine ganze Reihe von Folgeartikeln reichen. Andererseits – das wären nur noch mehr Details, die ihr auch woanders lesen könnt.

Unsere Botschaft ist: Die Revolution, die Angular gerade durchläuft, ist gleichzeitig eine sorgfältig vorbereitete Evolution. Ein paar der dahinterliegenden Überlegungen haben wir euch gezeigt. Das Angular-Team beobachtet, wie Angular in der freien Wildbahn verwendet wird, wo es noch Probleme und Verbesserungswünsche gibt und setzt diese konsequent um. Mit anderen Worten: Ihr könnt unbesorgt auf Angular setzen, es hat seine Zukunft noch vor sich!

Quellen

[1] Rainer Hahnekamp 2024: *https://dev.to/this-is-angular/how-do-i-test-code-using-inject-3f6d*

[2] *https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html#the-usedefineforclassfields-flag-and-the-declare-property-modifier*

[3] Johannes Hoppe und Ferdinand Malcher 2022: *https://angular.schule/blog/2022-11-use-define-for-class-fields*

[4] Netanel Basal 2022, *https://medium.com/netanelbasal/getting-to-know-the-runincontext-api-in-angular-f8996d7e00da*

[5] Dany Paredes 2024, *https://dev.to/danywalls/angular-19-and-zoneless-1of9*

[6] Jessica Janiuk 2025, *https://www.youtube.com/watch?v=v5KTJGEYLsM*

[7] Alex Rickabaugh auf Tech Stack Nation 2024, *https://www.youtube.com/watch?v=aKxcIQMWSNU&t=752s*

[8] ngx-extended-pdf-viewer, *https://www.pdfviewer.net*

[9] Stephan Rauh 2021, *https://www.beyondjava.net/zonejs*

[10] Angular-Projekt 2025, *https://github.com/angular/angular/blob/main/adev/src/content/best-practices/style-guide.md*

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.

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

iJUG Verbund
www.ijug.eu

Impressum

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

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

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

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

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

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

Bildnachweis:
Titel: Bild © Irène
https://stock.adobe.com
S. 10 + 11: Sloth McSloth
https://stock.adobe.com
S. 14: Bild © Grafigator
https://stock.adobe.com
S. 24 + 25: Bild © bou
https://stock.adobe.com
S. 28 + 29: Bild © yourapechkin
https://123rf.com
S. 34 + 35: Bild © igor.nazlo
https://stock.adobe.com
S. 40 + 41: Bild © EpochXstudio
https://stock.adobe.com
S. 48 + 49: Bild © Oleksandr Dibrova
https://stock.adobe.com
S. 52 + 53: Bild © Art_spiral
https://stock.adobe.com
S. 60 + 61: Bild © ArtemisDiana
https://stock.adobe.com
S. 66 + 67: Bild © Designed by bunny
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

Inserentenverzeichnis

Präsentiert von: iJUG Verbund heise medien DOAG Veranstalter: JavaLand