Java aktuell

Ausgabe 01/2025

Java 23
Details zum aktuellen Release

CSS
Inheritance, Kaskade und Layouting

Microservices
Qualität durch Alerting und Visualisierung

Java aktuell

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

D: 7,40 EUR | CH: 11,99 CHF | A: 7,60 EUR | Benelux: 8,50 EUR

GROßES COMMUNITY-PROGRAMM!

JavaLand

AM NÜRBURGRING

JAVALAND

1. - 3. APRIL 2025

JAVALAND.EU

#JAVALAND

Präsentiert von: iJUG Verbund Heise Medien DOAG Veranstalter: JavaLand

Liebe Leserinnen und Leser,

wir freuen uns, euch zur ersten Ausgabe des Jahres 2025 begrüßen zu dürfen. Auch wenn das neue Jahr zum Erscheinungstermin dieser Ausgabe bereits zwei Monate alt ist, möchten wir es nicht versäumen, euch nachträglich ein frohes, erfolgreiches und vor allem gesundes neues Jahr zu wünschen. Wir hoffen, ihr hattet einen entspannten Jahreswechsel und seid mit bester Motivation und Energie in das neue Jahr gestartet. Doch widmen wir uns nun dieser Ausgabe und gehen diese gemeinsam durch.

Wie ihr spätestens anhand des Covers erkannt habt, ist Java 23 da. Falk Sippach beleuchtet das aktuelle Release ab Seite 14 im Detail und nimmt alle Neuheiten mit uns unter die Lupe. Danach vergleicht Sebastian Tiemann die REST und gRPC miteinander, indem er Gemeinsamkeiten und Unterscheide der beliebten Paradigmen hervorhebt. In Christina Zenzes' Artikel auf Seite 34 werfen wir einen Blick auf CSS. Ihr Ziel ist es, dass vor allem Backend-Entwickler ein besseres Verständnis für die Stylesheet-Sprache entwickeln können. Einen besonderen Fokus legt sie dabei auf Inheritance, Kaskade und Layouting. Worauf es beim Branching wirklich ankommt und wieso Entwicklerinnen und Entwickler sich nicht davor scheuen sollten, häufig zu liefern, berichtet Georg Berky in seinem Beitrag. Im Anschluss daran stellt Ildikó Tárkányi eine Open-Source-Lösung mit Visualisierung und Alerting vor, die die Softwarequalität von Microservice-Architekturen durch bessere Übersichtlichkeit verbessert.

Im zweiten und letzten Teil ihrer Artikelreihe ab Seite 58 widmen sich Peter Fichtner und Ralf Straßner dem Komponenten- und dem Testschnitt. Sie zeigen Möglichkeiten auf, Testansätze übersichtlich und kompakt zu dokumentieren und ziehen abschließend ein Fazit zur Artikelreihe. Die Freizeitplanung einer ganzen Familie kann manchmal eine große Herausforderung darstellen. Wie uns KI dabei unterstützen kann, Zeitpläne zu analysieren und gemeinsame, freie Termine zu finden, zeigt uns Alexander Culum zum Abschluss dieser Ausgabe.

Wir wünschen euch viel Spaß beim Lesen!

Lisa Damerow
Redaktionsleitung Java aktuell

INHALT

Alle Neuigkeiten rund um Java 23

CSS: Inheritance, Kaskade und Layouting

3 Editorial

6 Java-Tagebuch
Andreas Badelt

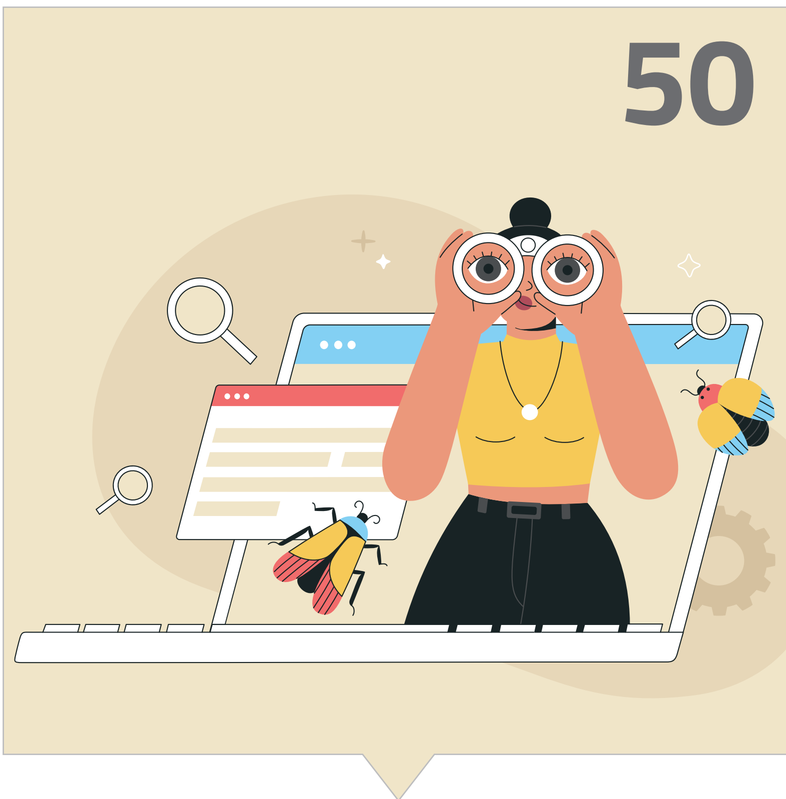
10 Die goldenen Regeln: Teil 5
Andreas Monschau

14 Java 23 – Was gibt es diesmal Neues?
Falk Sippach

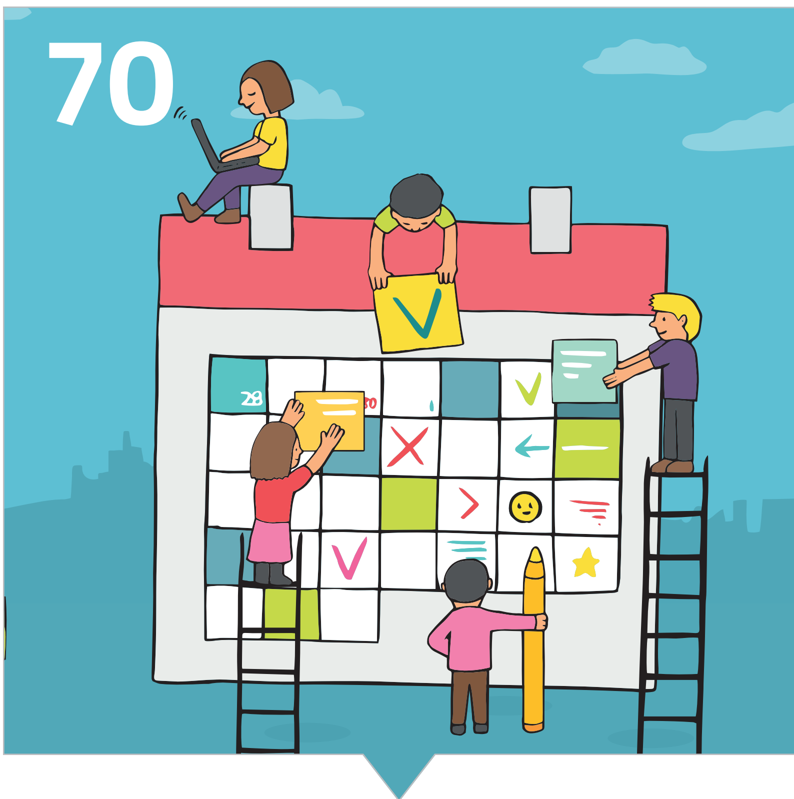
28 REST vs. gRPC
Sebastian Tiemann

34 CSS für Backend-Entwickler
Christina Zenzes

46 Falsch abgezweigt?
Was beim Branching wirklich wichtig ist
Georg Berky



Visualisierung und Alerting für verteilte Architekturen



Stressfreie Familienfreizeitplanung mithilfe von KI

50 Quality Metrics Unleashed: Softwarequalität im Griff mit Visualisierung und Alerting
Ildikó Tárkányi

58 Risikobewusst, modular… einfach passend testen! (Teil 2)
Peter Fichtner, Ralf Straßner

70 Familienplaner 2.0 – Wie KI und Function Calling den Alltag revolutionieren
Alexander Culum

78 Impressum/Inserenten

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

23. August 2024

MicroProfile 7.0 ist da

Mit Release 7 wird unter anderem *MicroProfile Metrics* ausgegliedert und als eigenständige Spezifikation weitergeführt; im Gegenzug rückt *MicroProfile Telemetry* 2.0, das nun Metriken, Logs und Tracing umfasst, in den *MicroProfile*-„Kern". Zudem wurde an den Jakarta-Abhängigkeiten gearbeitet: Statt diese mit zu „verpacken", wird eine Abhängigkeit zum *Jakarta EE 10 Core Profile* deklariert. Zu beachten ist, dass sowohl *MicroProfile Telemetry* 2.0 als auch *Open API* 4.0 nicht abwärtskompatible Änderungen beinhalten, daher auch die Versionssprünge. Auch der REST-Client hat mit 4.0 eine neue Major-Release-Version bekommen, aber nur weil er die neue Abhängigkeit zum *Jakarta EE 10 Core Profile* hat.

25. August 2024

Logging mit Struktur

Spring Boot 3.4 bietet jetzt direkt und ohne Deklaration weiterer Abhängigkeiten strukturiertes Logging an (also nicht von „irgendwelchen" Strings, sondern in der Regel *JSON* mit definierten Keys, um die Auswertung zu unterstützen). In den `application.properties` muss dazu lediglich das Format deklariert werden, also zum Beispiel `logging.structured.format.console=ecs` für das Elastic-Common-Schema. *Logstash* und eigene „Custom"-Formate werden auch unterstützt.

11. September 2024

Eclipse 2024-09

Die neueste Version der Eclipse IDE unterstützt kein Java 7 (und noch ältere Versionen) mehr. Ja, für die Wartung von Uralt-Software kann sich nun auch niemand mehr mit der neuesten IDE trösten – wozu auch immer das überhaupt gut sein soll, außer fürs Gefühl – wenn auch nicht fürs Sicherheitsgefühl (mit einer Java Runtime, die überhaupt keine Updates mehr erfährt). Ok, das stimmt nicht ganz: Es scheint ja noch Anbieter zu geben, die nicht so fies sind, siehe zum Beispiel die „Azul Zulu Builds of OpenJDK" ab Java 6. Aber das sind dann Nachbauten, die nur weitestgehend identisch sind. Ab dem nächsten Release der IDE im Dezember soll dafür Java 23 unterstützt werden, das in einigen Tagen verfügbar sein wird.

Hervorgehoben wird im Netz auch das neue „Sticky Scrolling"-Feature für große Files, das unter anderem schon aus *Visual Studio Code* bekannt ist. Damit lässt sich, basierend auf der Einrückung, wichtiger Kontext (zum Beispiel die aktuelle Klasse und Methode) oben im Text-Editor beim Scrollen automatisch „anpinnen".

17. September 2024

JDK 23 ist da…

Das OpenJDK 23 ist da. Oracles eigene JDK-Distribution erscheint immer gleichzeitig mit dem zugrundeliegenden OpenJDK. Auch andere Hersteller wie Azul (Zulu) haben ihre JDK-Distributionen schon freigegeben, andere werden in den nächsten Tagen folgen. Keine Hektik – es ist ja auch nur ein non-LTS Release, auf das viele sowieso nicht „aufspringen" werden.

Die neuen Features sind schon in vorherigen Ausgaben erwähnt worden *[1]* – es gibt aber eine kleine Enttäuschung für alle Fans der *String Templates Previews* (seit Java 21). Diese erfahren eine grundlegende Überarbeitung (also die Templates, nicht die Fans), und sind aus Java 23 entfernt worden; das wird auf der oben genannten Seite nicht erwähnt, da es ja „nur" eine Preview war (über den expliziten Schalter `--enable-preview` nutzbar). Grund der Überarbeitung war das bisherige Feedback unter anderem zu Syntax und „nested templates". Vor allem aber war es die Erkenntnis, dass es gar nicht nötig ist, syntaktisch einen Prozessor vor das Template zu stellen (in der Form `STR`."…"), damit die String-Templates möglichst besser als `String.format`(…) performen. Die geplante Prozessor-Abstraktion ist komplett verzichtbar – und ohne sie lassen sich die String-Templates viel leichter in die Sprache integrieren. Leider wird die Überarbeitung eine Weile dauern, sodass erst ein Folge-Release eine finale Version (oder eine weitere Preview?) der String-Templates bringen wird. Wer sich für die Details interessiert, kann diese in der E-Mail vom Java Language Architect Brian Goetz nachlesen *[2]*.

17. September 2024

… und auch GraalVM 23

Neben dem JDK 23 ist „traditionell" auch die von Oracle entwickelte GraalVM für das JDK 23 heute erschienen, die unter anderem mit einer Verbesserung für den Serial Garbage Collector aufwartet: Ein Mark & Compact GC für die „old generation", um den maximalen Speicherbedarf deutlich zu reduzieren (er wird in dieser Master-Arbeit der Uni Linz beschrieben *[3]*). Außerdem gibt es unter anderem einen neuen Optimierungs-Schalter „-Os", der für die kleinstmögliche Größe des Executables sorgen soll, sowie Verbesserungen an den Build-Reports: Einen HTML-Report, der eine Aufschlüsselung des *erreichbaren* Codes nach Größe und Methoden-Anzahl ermöglicht, sowie eine Ressourcen-Ansicht, die *alles*, was in ein Native Image kommt, nach Herkunft und Größe auflistet. Und auch nicht ganz irrelevant: *GraalPy* and *GraalWasm* werden jetzt als produktionsreif deklariert, sodass sich zum Beispiel aus Java-Code heraus direkt die unzähligen Python-Bibliotheken für Maschinelles Lernen nutzen lassen.

24. September 2024

EE Cargo Tracker

Der Cargo Tracker, seit langem die Referenzapplikation für Java EE und Jakarta EE (und ursprünglich aus Eric Evans bekanntem DDD-Buch), ist jetzt endlich Bestandteil der offiziellen Jakarta-EE-Dokumentation *[5]*.

25. September 2024

Ausblick auf Java 24

Die Termine für das nächste OpenJDK-Release stehen auch fest (beziehungsweise die Vorschläge, die ja in der Regel ziemlich genau eingehalten werden). Demnach soll das JDK 24 am 18. März 2025 freigegeben werden, Ende der eigentlichen Feature-Entwicklung ist bereits am 12. Dezember.

Einige „Java Enhancement Proposals" wurden schon für das Release vorgeschlagen: JEP 472 „Prepare to Restrict the Use of JNI", JEP 475 „Late Barrier Expansion for G1", JEP 484 „Class-File-API". Und mit JEP 489 die „Vector-API" in ihrer neunten(!) Inkubator-Version, aber ohne API- oder wesentliche Implementierungsänderungen im Vergleich zu JDK 23. Der Grund für die lange Serie ist einfach, dass die Vector-API auf notwendige Features aus dem Projekt *Valhalla*, insbesondere die *Value Classes/Objects*, wartet; sobald diese zur Verfügung stehen, soll die Vector-API darauf angepasst und als Preview präsentiert werden.

Übrigens, zu *Valhalla* und den *Value Classes* hat sich Brian Goetz auf dem Java Language Summit im August ausgelassen. Das Video dazu gibt's auf YouTube. Für die ganz Eiligen hier auch die wesentlichen Aspekte, in 10 Minuten von Nicolai Parlog auf den Punkt gebracht *[4]*. Da braucht man keinen Dieter Thomas Heck mehr (für die Jüngeren: Der beherrschte schon im alten Röhrenfernsehen die 1,5-fache Wiedergabegeschwindigkeit, bevor YouTube und WhatsApp überhaupt erfunden waren).

26. September 2024

Neue Quarkus-LTS-Version

Das nächste „Long Term Support" Release für Quarkus ist da. Aufgepasst: Long Term heißt hier 12 Monate, vielleicht wäre auch „Medium Term Support" angemessen. Aber Quarkus bringt auch im Monatsrhythmus neue Minor-Releases heraus, knüpft andererseits den LTS-Status nicht an ein Major-Release, und: Es handelt sich hier um das Community-Projekt, nicht um Red Hats kommerzielle Version, die längere Support-Zeiträume vorsieht. Die Feature-Liste ist, ähm, kurz. Genau genommen handelt es sich um Quarkus 3.14 (von August) plus einige Bugfixes und Dependency Updates. Die Features sind entsprechend bei 3.14 zu finden, zum Beispiel Let's Encrypt Support und ein gRPC-Plug-in für das Quarkus CLI *[6]*. Neue Features gibt es dann wieder mit 3.16 (bereits Ende Oktober), aber dann halt erst mal wieder ohne das „LTS".

1. Oktober 2024

„Tip & Tail Model of Library Development"

Interessante Lektüre: Unter dem „Informational JEP" 14 [*https://openjdk.org/jeps/14*] beschreiben mehrere Mitwirkende des OpenJDK (schon wieder Brian Goetz) „The Tip & Tail Model of Library Development". Im Wesentlichen ist es die Erklärung und Begründung des 2018 eingeführten OpenJDK Release Trains: mit regelmäßigen „Tip"-Versionen für neue Features sowie ausgewählten „Tail"-Versionen (sprich: Long-Term Support Releases), die langfristige Stabilität garantieren. Gleichzeitig ist es eine Empfehlung, die zugrundeliegenden Prinzipien für andere Projekte zu nutzen, die ebenfalls eine sehr heterogene Nutzerbasis haben.

2. Oktober 2024

Spring AI

Parallel zum Fortschritt des Spring-AI-Projekts (aktuelle Version 1.0.0 M2) hat die *Petclinic* (die Referenzapplikation für Spring) jetzt auch ihren AI Assistant bekommen *[7]*. Die Erweiterung zeigt einen typischen Chatbot basierend auf speziellen Prompts, Function Calls, die dem Language Model „angeboten" werden, RAG (Retrieval Augmented Generation) und die neuen Spring AI Advisors (die sich analog zur Aspektorientierung um Model-Aufrufe beziehungsweise Streams „legen" lassen). Nicht überliefert ist, ob die virtuellen Patienten und Ärzte der *Petclinic* ihren Datenschutz gefährdet sehen. Die Domäne des Jakarta EE Cargo Trackers wäre da vielleicht nicht ganz so sensibel für den Anfang; aber dahingehend gibt es wohl bislang keine Ambitionen.

2. Oktober 2024

Spring 7.0 und Spring Boot 2

Das Spring Framework 6.2 ist für November geplant, aber das nächste Major-Release wurde jetzt auch schon angekündigt: Release 7.0 soll im November 2025 fertig sein, mit Jakarta EE 11 ab-

gestimmt und mit dem nächsten Java-SE-LTS-Release 25 kompatibel sein. Gleichzeitig soll weiterhin Java SE 17 unterstützt werden, womit Features wie Virtual Threads oder die Class-File-API (Java 25) nicht direkt genutzt werden können. Als Kompromiss sollen aber laut Projektleiter Jürgen Höller Multi-Release-Jars dienen.

Vom „Tip" des *Spring Frameworks* zum „Tail" von *Spring Boot*: Vor wenigen Tagen wurde der (kommerzielle!) Support für *Spring Boot* 2.7 (das finale Minor-Release von 2.x) und die enthaltenen Projekte bis Ende 2026 verlängert.

16. Oktober 2024

Jakarta EE Developer Survey Report 2024

Die Eclipse Foundation hat das Ergebnis ihrer jährlichen Developer Surveys veröffentlicht. Beim Vergleich der Framework-Nutzung liegt wie gewohnt Spring (Boot) vorne, interessanterweise hat Jakarta EE aber fast aufgeschlossen (63 % zu 60 %, MicroProfile 32 % – Mehrfachnennungen möglich). Jakarta-Vorgänger Java EE 8 liegt im direkten Vergleich zwar weiter vorn, aber im letzten Jahr haben viele Firmen migriert. Jakarta EE 10 ist auch hier nicht mehr weit hinterher (40 % zu 34 %). Bei den zugrundeliegenden Java SE-Versionen hat sich auch etwas getan: Es nutzen immer noch 55 % der Befragten Java SE 8(!), aber inzwischen liegt Java 17 knapp vorne (56 %), und auch Java 21 wurde vielerorts schnell angenommen (30 %). Der gesamte Bericht ist im Eclipse Newsroom zu finden *[8]*.

20. November 2024

Weitere Kandidaten für das JDK 24

JDK 24 wächst und wächst – die Liste wird ein bisschen zu lang, um sie hier komplett durchzugehen. Ein interessanter Kandidat ist JEP 491 „Synchronize Virtual Threads without Pinning" mit dem Ziel, dass virtuelle Threads bei „synchronized" Statements und Methoden-Aufrufen ihre verknüpften Plattform-Threads freigeben. Damit wird eine wesentliche Quelle von Bottlenecks entfernt, ohne dass der entsprechende Code angepasst werden muss. Hinzu kommt noch eine verbesserte Diagnostik (*JDK Flight Recorder*) für Fälle, in denen die Freigabe der Plattform-Threads scheitert. Eine komplette Lösung wird es nicht sein, Probleme insbesondere bei Native Code-Aufrufen und I/O-Operationen werden bleiben, aber es ist eine deutliche Verbesserung zu erwarten. Und für die Anhänger alter Hardware: Der 32-bit x86-Port soll nun tatsächlich für das letzte unterstützte Betriebssystem – Linux – „deprecated" werden und dann in einem der nächsten Releases endgültig verschwinden (JEP 501). Die volle Liste ist im Web zu sehen *[9]*.

21. November 2024

Und was ist mit Jakarta EE 11?

Das Jakarta-EE-11-Release sollte eigentlich im Sommer kommen. Jetzt soll es rund um den JakartaOne-Livestream am 3. Dezember veröffentlicht werden. Das hat zumindest „Projektleiter" Ivar Grimstad kürzlich verkündet. Das *Jakarta EE 11 Core Profile* ist als erstes im Review. Mit *Wildfly* und *Open Liberty* gibt es auch gleich zwei Implementierungen, die dabei sind, sich für Kompatibilität mit dem *Core Profile* zertifizieren zu lassen. *Jakarta EE 11 Platform* und *Jakarta EE 11 Web Profile* folgen noch, sollen aber bis Anfang Dezember auch möglichst durch die Review sein.

Die Planungen für EE 12 sind aber bereits in vollem Gang. Zentrales Thema ist die „Migration" von EJBs zu CDI. Auch die Integration von MicroProfile und Jakarta EE spielt eine wichtige Rolle, vor allem mit Blick auf die Unterstützung von Large Language Models (LLMs). Es wird zum Beispiel überlegt, ob bestehende APIs wie *LangChain4J* direkt übernommen oder besser angepasst werden sollten. Außerdem wird über regelmäßige Releases diskutiert (der Fahrplan von EE 11 wurde ja eher von der Deutschen Bahn übernommen), um jeweils mit den Java-SE-Release-Versionen Schritt zu halten und die Plattform kontinuierlicher weiterzuentwickeln.

Quellen

[1] *https://openjdk.org/projects/jdk/23/*
[2] *https://mail.openjdk.org/pipermail/amber-spec-observers/2024-March/004225.html*
[3] *https://ssw.jku.at/Teaching/MasterTheses/Aistleitner/Thesis.pdf*
[4] *https://www.youtube.com/watch?v=Lf2Vr7Zjmj8*
[5] *https://jakarta.ee/learn/docs/cargotracker-documentation/current/index.html*
[6] *https://quarkus.io/blog/quarkus-3-14-1-released/*
[7] *https://spring.io/blog/2024/09/26/ai-meets-spring-petclinic-implementing-an-ai-assistant-with-spring-ai-part-i*
[8] *https://newsroom.eclipse.org/news/announcements/eclipse-foundation-releases-2024-jakarta-ee-developer-survey-report*
[9] *https://openjdk.org/projects/jdk/24/*

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").

gute software mit guten leuten_

Aus dieser Motivation hat CEO und Informatiker Jobst Eßmeyer cronn 2013 gegründet. Was unsere mehr als 150 cronner:innen verbindet: Wir schätzen Software Craftsmanship, guten Code, Zusammenhalt und Spaß bei der Arbeit.

Im erfahrenen Team kannst Du gemeinsam spannende Kundenprojekte umsetzen als:

- _Software Developer (m/w/d)
- _Software Architect (m/w/d)
- _Testengineer (m/w/d)
- _Analyst (m/w/d)

Bonn **Hamburg** **Białystok** **cronn.de/jobs**

Die goldenen Regeln: Teil 5

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!

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.

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

Regel 5: Kommuniziere möglichst schlecht

Stell' dir doch mal vor: Dein Neuling kommt in dein Projekt, dein Unternehmen, deine Organisation. Wie wir bereits gelernt haben, bist du nicht erreichbar, die Dokumentation ist unbrauchbar. So weit, so gut.

Aber dieser Neuling will sich dennoch Informationen beschaffen. Damit ihm dies nicht gelingt, ist es wichtig, sehr schnell Strukturen zu schaffen (und die richtigen Leute an der Hand zu haben), die dafür sorgen, dass es nicht sinnvoll möglich ist, an die gesuchte Information, an den richtigen Informationsfluss zu gelangen.

Stellen wir uns dazu doch folgende Gemengelange vor: Es existieren in diesem Beispiel vier Mitarbeiter in einem Projekt. Nennen wir sie Mitarbeiter A, Mitarbeiter B, Mitarbeiter C und Mitarbeiter D. Und sie alle kommunizieren miteinander, irgendwie. Wie sieht es nun im Einzelnen aus?

Mitarbeiter A

Nur Gemailtes ist Wahres. Andere Kommunikationsmittel interessieren ihn nicht. Informationen gibt er nur per E-Mail weiter, gerne auch einfach an alle, was dazu führt, dass es eine riesige Empfängerliste gibt, in der aber ab und zu auch jemand fehlen kann.

Mitarbeiter B

Mitarbeiter B kommuniziert auch mit allen. Er nutzt dafür allerdings eine Alternative – zum Beispiel Slack oder Teams. Es werden im besagten Tool Gruppen gegründet, um eine Kleinigkeit zu besprechen, oder insbesondere in Teams nutzt man noch wochenlange den Chat eines bereits längst vergangenen und vergessenen Termins.

Mitarbeiter C

Dieser Mitarbeiter ist etwas eigenbrötlerisch, denn er informiert nur Mitarbeiter B und niemand anderen. Und wie macht er das? Indem er ihn auf Confluence-Seiten verlinkt. Mitarbeiter B ist dann so nett und verteilt quasi als Durchlauferhitzer die Informationen an alle anderen. Hierbei geht entweder Wichtiges verloren oder die Informationen werden angereichert, denn schließlich gibt er nicht nur den Confluence-Link weiter, sondern noch weitere Informationen (weil es möglicherweise Zugriffsberechtigungen gibt, in denen zum Beispiel Mitarbeiter A nicht berücksichtigt ist).

Mitarbeiter D

Diesem Mitarbeiter ist es wichtig, dass wirklich alle Informationen weitergeleitet werden, und auch alle Mitarbeiter alle Informationen bekommen. Über alle Wege. Er nutzt E-Mail, Teams, Flurfunk, Confluence und Jira. Die Informationen sind meistens nur redundant, aber im schlimmsten Fall über alle Wege wirr aufgeteilt.

Dies ist nur ein Beispiel, aber wenn es mit möglichst schlechter Dokumentation kombiniert wird, ist es mit dieser Maßnahme absolut möglich, den Einstieg für Neulinge nahezu unmöglich zu gestalten.

Und der Bonus: Es wird nicht nur der Neuling verprellt, nein – mit den Beispielen der vier Mitarbeiter können Sie erfolgreich ihr Projekt an die Wand fahren!

Nächstes Mal wird es spannend – wir bewegen uns in das Spannungsfeld zwischen Entwicklung und Testing.

Andreas Monschau

Haeger Consulting

amonschau@haeger-consulting

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

DEINE VORTEILE

25 % Rabatt
auf JavaLand-Tickets

Java aktuell
Jahres-Abonnement

Java Community Process
Mitgliedschaft

JETZT MITGLIED WERDEN!
Ab 15 Euro im Jahr

www.ijug.eu

iJUG
Verbund

Java 23 – Was gibt es diesmal Neues?

Falk Sippach, embarc Software Consulting GmbH

JAVA

An den eigenen Kindern und an den sehr regelmäßigen Java-Versionen merkt man, wie alt man geworden ist. Aber irgendwie ist es ja auch ein bisschen wie Weihnachten, wenn im Frühjahr beziehungsweise Herbst die jeweils nächste Major-Version herauskommt. Im September 2024 ist das OpenJDK 23 erschienen und wie jedes halbe Jahr lohnt sich der Blick auf die Neuerungen und Änderungen. Es sind wieder sowohl kleine, für uns Entwickler aber sehr nützliche Funktionen dabei, und es werden natürlich auch die großen Themenblöcke weiterhin verfolgt.

Für einen ersten Überblick ist wie immer die Projektseite des OpenJDK *[1]* ein guter Startpunkt. Dort sind diesmal immerhin wieder 12 JEPs (JDK Enhancement Proposals) gelistet:

- 455: Primitive Types in Patterns, instanceof, and switch (Preview)
- 466: Class-File-API (Second Preview)
- 467: Markdown Documentation Comments
- 469: Vector-API (Eighth Incubator)
- 473: Stream Gatherers (Second Preview)
- 471: Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal
- 474: ZGC: Generational Mode by Default
- 476: Module Import Declarations (Preview)
- 477: Implicitly Declared Classes and Instance Main Methods (Third Preview)
- 480: Structured Concurrency (Third Preview)
- 481: Scoped Values (Third Preview)
- 482: Flexible Constructor Bodies (Second Preview)

Das sind exakt so viele JEPs wie schon bei Java 22. Übrigens wird Java 24 im März 2025 voraussichtlich einen neuen Rekord aufstellen und dann mit 24 JEPs aufwarten (Planungsstand von Ende November 2024).

Vereinfachungen beim Starten von Java-Anwendungen

Programmierneulingen fällt der Einstieg in Java häufig schwer. Seit einiger Zeit gibt es aber immer wieder kleine Verbesserungen, die die ersten Schritte erleichtern. Bereits seit Java 9 wird die *JShell* mitgeliefert. In dieser REPL *(Read Eval Print Loop)* lassen sich sehr schnell kleine Programmierbeispiele ausprobieren, ohne sich mit dem Thema Kompilieren und Build-Management befassen zu müssen.

In Java 11 wurden *Launch Single-File Source-Code Programs* eingeführt, wodurch einzelne Java-Dateien (mit Klassendeklaration und `main`-Methode) ohne separaten Kompilierungsschritt einfach von der Konsole aus gestartet werden können. Mit dem JEP 458 *(Launch Multi-File Source-Code Programs)* darf der Code seit dem OpenJDK 22 nun auch in beliebig vielen Java-Dateien strukturiert sein. Zusätzlich zu den Multi-Files kann auch Code aus JARs, die im Unterverzeichnis „libs" liegen, mittels `--class-path 'libs/*'` eingebunden werden. Liegt eine der Klassen in einem Package, zum Beispiel in „de.embarc", dann muss auch die Klasse im entsprechenden Unterverzeichnis „de/embarc" liegen und das Java-Kommando muss um den vollständigen Pfad erweitert werden: `java de/embarc/Klasse.java`.

In Java 21 kam mit dem JEP 445 *(Unnamed Classes and Instance main Methods)* ein Preview hinzu, um Java-Main-Anwendungen viel schlanker und ohne überflüssigen Boilerplate-Code zu definieren. Somit lässt sich die main-Methode viel kompakter schreiben (ohne `public, static, String[] args,` …). Außerdem muss sie nicht mal mehr in einer Klassen-Definition eingebettet sein. Gerade für Programmieranfänger, die in Java mit einem einfachen Hello-World-Programm starten, stellte die bisherige Vorgehensweise eine übermäßige Hürde dar. Sie müssen bereits zu Beginn gleich mehrere, in dem Moment nicht relevante Konzepte (Klassen, statische Methoden, String-Arrays und vieles mehr) verstehen. Im Zusammenhang mit dem oben erwähnten, im OpenJDK 11 als JEP 330 eingeführten, *Launch Single-File Source-Code Programs* sinkt der Aufwand weiter, da man in der *.java-Datei nur noch eine schlanke `main`-Methode benötigt und sogar auf eine Klassendefinition verzichten kann.

Für Java 22 wurde das Konzept nochmal überarbeitet und mit dem JEP 463 *(Implicitly Declared Classes and Instance Main Methods)* ein zweiter Preview vorgelegt. Dabei gibt es jetzt keine Unnamed Classes mehr, vielmehr wird von der Laufzeitumgebung ein Name gewählt. Bei ersten Tests der neuen Funktion wurde der Name der Java-Datei *(ImplicitMain.java* => Klasse `ImplicitMain`) verwendet. Das scheint für das OpenJDK der Default zu sein, kann aber bei anderen Distributionen abweichen. Diese implizit deklarierten Klassen verhalten sich wie normale Top-Level-Klassen und benötigen nun auch keine zusätzliche Unterstützung mehr beim Tooling, bei der Verwendung von Bibliotheken oder in der Laufzeitumgebung. Die einzige Bedingung ist, dass die Datei im Root-Package liegen muss.

Im OpenJDK 23 ist jetzt der dritte Preview erschienen (JEP 477). Es gibt nun zusätzlich eine Klasse `java.io.IO`, deren drei öffentliche, statische Methoden (`print()`, `println()` und `readln()`) automatisch in implizit deklarierten Klassen importiert werden. Und wie etwas später beschrieben, werden auch alle anderen exportierten Klassen des Moduls java.base importiert *(Module Import Declarations)*.

Ein einfaches Beispiel inklusive des Kommandozeilenbefehls zeigt *Listing 1*. Da es sich noch um ein Preview-Feature handelt, muss dies aktiviert und um die Angabe zur aktuellen Java-Version ergänzt werden.

```
// > java --source 23 --enable-preview Main.java
void main() {
  System.out.println("Hello, World!");
}
```

Listing 1: Instance Main Method

Die implizit deklarierte Klasse darf sogar noch weitere Attribute/Felder und Methoden enthalten *(siehe Listing 2)*. Dadurch fühlt sich Java

jetzt fast schon wie eine Skriptsprache an und es werden ganz neue Anwendungsfälle möglich. Wir können nun Shell-Skripte schreiben und profitieren dabei von dem mächtigen Funktionsumfang und der Typsicherheit Javas.

```
// > java --source 23 --enable-preview Main.java
final String greeting = "Hello";

void main() {
  System.out.println(greet("World"));
}

String greet(String name) {
  return STR."\{greeting}, \{name}!";
}
```

Listing 2: Zusätzliche Felder und Methoden mit der Instance Main Method

Da es theoretisch mehrere main-Methoden in einer implizit deklarierten Klasse geben kann, braucht es eine Priorisierung beim Aufruf der Start-Prozedur. Die Logik dahinter hat sich im zweiten Preview deutlich vereinfacht. Es wird nur noch unterschieden, ob es einen Parameter vom Typ String-Array gibt oder nicht. Wenn ja, dann hat diese Methode Vorrang. In *Listing 3* wird die oberste `main`-Methode aufgerufen. Der Compiler verbietet, dass es sowohl eine statische als auch eine Instanz-Methode mit der gleichen Methodensignatur gibt. Sichtbarkeitsattribute wie `public`, `protected` und so weiter werden an dieser Stelle ignoriert und der Compiler würde wiederum verhindern, dass es eine private und eine öffentliche Methode mit der gleichen Methoden-Signatur gibt.

Importieren von ganzen Modulen

Um in unseren Anwendungen andere Klassen verwenden zu können, müssen wir sie zuerst importieren. Das kann sowohl für einzelne Klassen (`import java.util.Date`) als auch für ganze Pakete (`import java.util.*`) erfolgen. Schon seit Java 1.0 werden automatisch alle Klassen des Pakets `java.lang` importiert. Deshalb können wir zum Beispiel die Klassen `String`, `Integer`, `Exception` oder `Thread` ohne separate import-Anweisung verwenden. Der JEP 476 ermöglicht nun auch den kompletten Import von Modulen. Genaugenommen werden alle Klassen importiert, die in dem von dem Modul exportierten Paketen liegen. Das basiert auf dem in Java 9 eingeführten Plattform Modul System (JPMS). Übrigens muss die Klasse, die Module importiert, nicht selbst Teil eines Moduls sein. *Listing 4* zeigt den Import des Moduls *java.base*.

```
import module java.base;
```

Listing 4: Modul-Import

Interessant wird es bei Namenskonflikten wie der Klasse Date (`java.util.Date` und `java.sql.Date`). Der Compiler stört sich zunächst nicht, wenn wir zwei Module importieren, die gleichnamige Klassen in unterschiedlichen Paketen enthalten. Bei der Verwendung der Klasse `Date` wird es aber dann zu einem Fehler kommen. Am einfachsten lässt sich das umgehen, indem die entsprechende Klasse zusätzlich importiert wird *(siehe Listing 5)*. Andernfalls wird der Compiler folgendes melden: *reference to Date is ambiguous*.

```
import module java.base;
import module java.sql;
import java.util.Date;

public class ModuleImports {
  public static void main(String[] args) {
     System.out.println(new Date());
  }
}
```

Listing 5: Modul-Import: Namenskonflike auflösen

Bei transitiven Importen von Modulen können auch alle Klassen der exportierten Pakete des transitiv importierten Moduls ohne expliziten Import verwendet werden. Beispielsweise benötigt das Modul *java.sql* transitiv *java.xml*. Dann können beim Import vom Modul *java.sql* auch Klassen wie `SAXParser` oder `DocumentBuilder` aus *java.xml* einfach direkt verwendet werden.

In zwei Fällen werden ähnlich dem automatischen Import des Pakets `java.lang` zukünftig auch einfach alle exportierten Klassen des ganzen Moduls `java.base` importiert. Denn sowohl bei der *JShell* als

```
// Gewinner, da mit String[] args
void main(String[] args) {
  System.out.println("instance main with String[] args");
}

// nicht erlaubt, da es schon eine nicht-statische main-Methode mit der gleichen Signatur gibt
/*
static void main(String[] args) {
  System.out.println("static main with String[] args");
}
*/

// wird beim Starten nicht aufgerufen
void main() {
  System.out.println("instance main without String[] args");
}
```

Listing 3: Aufrufreihenfolge von main-Methoden in unbenannten Klassen

auch bei den implizit deklarierten Klassen soll den Entwicklern die Arbeit vereinfacht werden. Bei der *JShell* kann man sich das sogar anzeigen lassen. Da es sich im Moment noch um ein Preview-Feature handelt, muss die *JShell* aber mit dem Parameter `--enable-preview` gestartet werden *(siehe Listing 6)*. Andernfalls werden alle Pakete angezeigt, die *JShell* bis Java 22 automatisch importiert hat.

```
> jshell --enable-preview
| Welcome to JShell -- Version 23-ea
| For an introduction type: /help intro
jshell> /imports
|     import java.base
```

Listing 6: Automatische Modul-Importe in JShell anzeigen

Flexible Konstruktor-Inhalte

Im OpenJDK 22 wurde diese Idee als *Statements before super()* eingeführt und nun umbenannt zu *Flexible Constructor Bodies* und als zweiter Preview veröffentlicht (JEP 482). Es ermöglicht in Konstruktoren das Einfügen von Codezeilen vor dem eigentlich in der ersten Zeile obligatorischen `super()`- oder `this()`-Aufruf. Der Grund für die bisherige Einschränkung ist die Sicherstellung durch den Compiler, dass zunächst potenzielle Oberklassen vollständig initialisiert sein müssen, bevor der Zustand der Sub-Klasse hergestellt oder abgefragt werden kann. Bei `this()` geht es zwar um eine Delegation an einen überladenen Konstruktor in der gleichen Klasse, aber der wird letztlich auch explizit oder implizit den `super`-Konstruktor der Oberklasse aufrufen.

Genau genommen gelten die strengen Anforderungen auch weiterhin. Die Statements, die man jetzt vor dem `super()`-Aufruf verwenden darf, dürfen nämlich nicht schreibend oder lesend auf die aktuelle Instanz zugreifen. Aber die Validierung oder die Transformation von Eingabeparametern sowie das Aufspalten eines Parameters in Einzelteile kann nun vor dem Aufruf von `super()` erfolgen *(siehe Listing 7)*.

Bisher wurde dafür typischerweise ein überladener Hilfs-Konstruktor oder eine private Methode als Workaround benötigt. Das hat den Lesefluss gestört und die Implementierung unnötig komplex gemacht. Im Falle von Berechnungen auf den Input-Parametern werden Werte zudem unnötigerweise mehrfach ermittelt. Im Beispiel in *Listing 7* müsste das Aufspalten in Vor- und Nachnamen zweimal erfolgen, um einmal das erste Element des `split()`-Aufrufs als ersten Parameter und dann das zweite Element aus der zweiten Berechnung als zweiten Parameter zu übergeben.

Alle Zeilen vor dem `super()`- oder `this()`-Aufruf werden Prolog genannt. Codezeilen nach beziehungsweise ohne `super()` oder `this()` werden Epilog genannt. Im Prolog darf nicht lesend auf Felder der Klasse oder der Oberklasse zugegriffen werden, da diese noch nicht initialisiert sein könnten. Außerdem dürfen keine Instanz-Methoden der Klasse aufgerufen und keine Instanzen von

```
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 7: Flexible Constructor Bodies

```
// Quelle
var number = Arrays.asList("abc1", "abc2", "abc3").stream()
  .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

nicht-statischen inneren Klassen erzeugt werden, da diese potenziell eine Referenz auf das noch nicht fertig initialisierte Elternobjekt enthalten können. Im Gegensatz dazu darf im Prolog des Konstruktors einer nicht-statischen inneren Klasse auf Felder und Methoden der äußeren Klassen zugegriffen werden.

Auch bei Records und Enums sind Codezeilen vor `this()` nach den oben genannten Regeln erlaubt. Da sie nicht von anderen Klassen ableiten können, verwenden wir in dem Umfeld keine Aufrufe von `super()`.

Es gab zur Veröffentlichung von OpenJDK 23 eine Änderung. Es dürfen jetzt im Prolog vor dem Aufruf von `super()` Instanzvariablen der Subklasse initialisiert werden. Das ist praktisch, wenn in der Subklasse eine Methode überschrieben wird, die in der Oberklasse deklariert und in deren Konstruktor aufgerufen wird und sowohl die Felder der Ober- als auch der Subklasse ausgeben will. Ohne die Möglichkeit die Felder der Subklasse zu initialisieren, würden diese noch den Default-Wert (0, false oder null) enthalten und somit ein falsches Ergebnis liefern.

Benutzerdefinierte Zwischen-Operationen bei Streams

Die Stream-API wurde in Java 8 eingeführt. Ein Stream wird erst bei Bedarf ausgewertet und kann potenziell eine unbeschränkte Anzahl von Werten enthalten. Diese können sequenziell oder parallel verarbeitet werden. Eine Stream-Pipeline besteht typischerweise aus drei Teilen: der Quelle, einer oder mehreren Intermediate Operationen und einer abschließenden Operation *(siehe Listing 8)*.

```
// 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]]
List<List<Integer>> windows = Stream.of(1, 2, 3, 4, 5, 6, 7)
  .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

```
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

```java
record WindowFixed<TR>(int windowSize)
  implements Gatherer<TR, ArrayList<TR>, List<TR>> {

  public WindowFixed {
     // Validate input
     if (windowSize < 1)
        throw new IllegalArgumentException("window size must be positive" );
  }

  @Override
  public Supplier<ArrayList<TR>> initializer() {
     // Create an ArrayList to hold the current open window
     return () -> new ArrayList<>(windowSize);
  }

  @Override
  public Integrator<ArrayList<TR>, TR, List<TR>> integrator() {
     // The integrator is invoked for each element consumed
     return Gatherer.Integrator.ofGreedy((window, element, downstream) -> {

        // Add the element to the current open window
        window.add(element);

        // Until we reach our desired window size,
        // return true to signal that more elements are desired
        if (window.size() < windowSize)
           return true;

        // When the window is full, close it by creating a copy
        var result = new ArrayList<TR>(window);

        // Clear the window so the next can be started
        window.clear();

        // Send the closed window downstream
        return downstream.push(result);

     });
  }

  // The combiner is omitted since this operation is intrinsically sequential,
  // and thus cannot be parallelized

  @Override
  public BiConsumer<ArrayList<TR>, Downstream<? super List<TR>>> finisher() {
     // The finisher runs when there are no more elements to pass from
     // the upstream
     return (window, downstream) -> {
        // If the downstream still accepts more elements and the current
        // open window is non-empty, then send a copy of it downstream
        if (!downstream.isRejecting() && !window.isEmpty()) {
           downstream.push(new ArrayList<TR>(window));
           window.clear();
        }
     };
  }
}

// [[1, 2], [3, 4], [5]]
System.out.println(
  Stream.of(1,2,3,4,5)
  .gather(new WindowFixed(2))
  .toList());
```

Listing 11: Verschiedene Stufen der Stream-Pipeline

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 kam immer wieder der Wunsch nach weiteren Methoden wie `window` oder `fold` auf. Doch anstatt nur genau die geforderten Operationen bereitzustellen, wurde eine API entwickelt. Diese ermöglicht es sowohl den JDK-Entwicklern als auch den normalen Anwendern, beliebige Intermediate Operations selbst zu implementieren. Dabei hat man sich an der Collectors-API orientiert, die genau das bereits seit Beginn der Stream-API für Terminal-Operationen bereitstellt.

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

- `fold`: zustandsbehafteter N-1-Gatherer, baut ein Aggregat inkrementell auf und gibt dieses Aggregat am Ende aus
- `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

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

Wenn man den Gatherer `windowFixed` nochmal implementieren möchte, würde es wie in *Listing 11* aussehen. Das Beispiel ist aus der Beschreibung des JEP 473 *[2]* übernommen.

Analysieren und Manipulieren von Bytecode

Die Class-File-API (JEP 466) 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 Klassen 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 sperrig und unflexibel. Es ist ein Workaround für die fehlende Unterstützung von Pattern Matching. Da Java Pattern Matching mittlerweile unterstützt, kann der notwendige Code in der neuen Class-File-API direkter und konsistenter ausgedrückt werden. *Listing 12* zeigt ein Beispiel, das auch im JEP 457 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 12: Parsen von Klassen mit Pattern Matching

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

Listing 13: Zu erzeugende Methode

```
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 14: Erzeugen der Methode aus Listing 13

Das Erzeugen von Klassen erfolgt im Gegensatz zum Visitor-Ansatz von ASM mit Buildern. Um beispielsweise eine Methode `foobar` *(siehe Listing 13)* zu erzeugen, kann das ebenfalls aus dem JEP 457 entnommene Code-Beispiel verwendet werden *(siehe Listing 14)*.

Es kann auch existierender Code verändert werden. *Listing 15* zeigt beispielhaft, wie bei einer bestehenden Klasse alle Methoden gelöscht werden, die mit `debug` beginnen.

```
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 15: Bestehende Klasse transformieren

Im Gegensatz zum OpenJDK 22 wurden in diesem zweiten Preview einige Verbesserungen vorgenommen. So wurden unter anderem:

- die API der Klasse `CodeBuilder` verschlankt
- Performance-Optimierungen in der Klasse Attributes vorgenommen (Zugriff auf `AttributeMapper` nun per statischen Methoden statt Feldern, um für schnellere Startzeiten eine verzögerte Initialisierung zu ermöglichen)
- Datentypen konsistenter modelliert (`Signature.TypeArg` ist nun ein algebraischer Datentyp)
- die Fehlerbehandlung beim Lesen von Einträgen in `ClassReader` beziehungsweise `ConstantPool` optimiert
- die Klasse `ClassSignature` verbessert (akkuratere Behandlung der generischen Signaturen von Super-Klassen und -Interfaces)
- unnötige Implementierungsmethoden von `ClassReader` entfernt
- sowie Namens-Inkonsistenzen in der Klasse `TypeKind` behoben

Vermutlich werden nur wenige Java-Entwickler direkt mit der Class-File-API arbeiten. Trotzdem schadet es nicht, sich mit der Funktionsweise unter der Haube zu beschäftigen. Es wird vor allem im JDK die Arbeit an neuen Features beschleunigen und auch die Weiterentwicklung des Tool-Supports vereinfachen. Wer mehr wissen möchte, findet alle Details im JEP 466 *[4]*.

Wiederkehrende Features

Die Vector-API ist nun schon das siebte 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 460 *[5]* 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. Accordingly, the Vector API will incubate over multiple releases until the necessary features of Project Valhalla become available as preview features."

Man wartet also auf die Reformen am Typsystem. Java hat aktuell ein zweigeteiltes Typsystem mit den primitiven und 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 Value Types (haben keine Identität) und Universal Generics (`List<int>`) ins JDK übernommen werden. Für Java 22 hat es der JEP 401 (Value Classes and Objects) leider noch nicht geschafft. Dementsprechend werden wir die Vector-API wohl auch noch einige Releases als Inkubator- beziehungsweise dann hoffentlich bald als Preview-Feature wiedersehen. Diesmal gab es keine Änderungen an der API und nur minimale Änderungen an der Implementierung im Vergleich zum JDK 22.

Neuer Baustein beim Pattern Matching

Ebenfalls schon lange in der Entwicklung ist das Pattern Matching. Hier wurden immer wieder Teile abgeschlossen, zuletzt in Java 22 die Unnamed Variables & Patterns (JEP 456). Neu hinzugekommen sind die Primitive Patterns. Damit können wir nun primitive Datentypen in Type Patterns im `switch` oder in `instanceof`-Vergleichen nutzen.

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 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 16)*.

Mit dem JEP 455 gibt es nun zwei Neuerungen. Einerseits dürfen beim Pattern Matching jetzt alle primitiven Datentypen verwendet werden, also auch `boolean`, `long`, `float` und `double`. Die Prüfung auf primitive Datentypen ist sowohl beim `instanceof` als auch im `switch` erlaubt. Bei der moderneren `switch`-Expression gilt zu be-

achten, dass sie *exhaustive* sein – also alle möglichen Fälle abdecken – muss. Im Falle von *Listing 16* wird das über den Fall `Integer i` abgefangen, alternativ hätte es auch ein `default`-Zweig getan.

```
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 16: Variablen mit primitiven Datentypen

Primitive Datentypen verhalten sich anders beim Pattern Matching, denn im Gegensatz zu Referenztypen unterstützen sie beispielsweise keine Vererbung. Wenn eine Variable `x` eines primitiven Typs (`boolean`, `byte`, `short`, `int`, `long`, `float`, `double` oder `char`) auf einen bestimmten primitiven Typ `y` geprüft wird, dann ergibt `x instanceof y` immer dann `true`, wenn der präzise Wert von `x` auch in einer Variablen von Typ `y` gespeichert werden kann. In *Listing 17* wird untersucht, ob der Wert des Parameters `value` vom Typ `byte` ist und ob er der `byte`-Variablen `b` zugewiesen werden kann. Alternativ wird ein Fehler angezeigt. Da der Wertebereich von `byte` von -128 bis +127 geht, ist der Parameter beim ersten Aufruf noch vom Typ `byte`, beim zweiten knapp nicht mehr.

```
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 17: Prüfung auf primitiven Datentyp

Wie bei Objekt-Typen dürfen sich auch bei primitiven Typen direkt beim `instanceof`-Check weitere Prüfungen mit `&&` anschließen. Der Code in *Listing 18* filtert beispielsweise die positiven byte-Werte (1 bis 127) heraus.

```
int value = -128;
if (value instanceof byte b && b > 0) {
  System.out.println("positive byte value: " + b);
} else {
  System.out.println("negative or not of type byte";
}
```

Listing 18: Guarded Patterns

Nach der Prüfung auf einen primitiven Typ und der Zuweisung findet möglicherweise eine Typkonvertierung statt. Dadurch lassen sich Ganzzahlen auch in Gleitkommazahlen (mit Nachkommastellen) oder Character (basierend auf dem Ascii-Code) umwandeln *(siehe Listing 19)*. Die Konvertierung in `boolean` ist allerdings nicht erlaubt, der Check wird vom Compiler abgelehnt. Ein `boolean` darf also nur mit einem `boolean` verglichen werden. Allerdings ergibt der Abgleich einer `boolean`-Variable mit dem Typ `boolean` sowieso immer `true`.

```
int value = 97;

if (value instanceof double d) {
  println(value + " instanceof double: " + d); // 97.0
}
if (value instanceof char c) {
  println(value + " instanceof char: " + c); // "a"
}
// if (value instanceof boolean b) { .. } => Compile error
```

Listing 19: Konvertierung von primitiven Datentypen

Aufgrund der unterschiedlichen Genauigkeit der primitiven Gleitkommatypen kann das zu interessanten Konstellationen führen, wenn ein bestimmter `int`-Wert als `double`, aber nicht als `float` darstellbar ist. Gleitkommazahlen ohne richtige Nachkommastelle (zum Beispiel 5,0) sind übrigens auch in Ganzzahlen konvertierbar, mit Nachkommastellen (zum Beispiel 5,5) matchen sie hingegen nur auf Gleitkommatypen.

Die primitiven Type Patterns können auch in einem `switch` verwendet werden. Auch hier sind Guarded Patterns (mit `when`) möglich. Zudem muss gegebenenfalls die Vollständigkeit der `case`-Zweige beachtet werden. Sind nicht alle möglichen Fälle abgedeckt, erwartet der Kompiler einen `default`-Zweig. Durch eine falsche Reihenfolge, könnten `case`-Zweige zudem ignoriert werden, weil dominierende darauffolgende dominierte Typen überdecken. Eine Zeile mit `case int` dürfte nicht vor einer Zeile mit `case byte` stehen, weil die zweite Zeile dann nie aufgerufen werden würde. Netterweise gibt der Kompiler für diesen Fall einen Fehler aus: „this case label is dominated by a preceding case label".

Weitere Neuerungen im Umfeld von Virtual Threads

Virtual Threads waren die größte Änderung im OpenJDK 21. Sie erlauben es, die konkurrierende Verarbeitung von parallel ausgeführten Aufgaben auch bei einer sehr großen Anzahl an Threads zu implementieren und dabei sogar gut verständlichen Code zu schreiben. Dieser lässt sich zudem wie sequenzieller Code mit herkömmlichen Mitteln debuggen. Die Virtual Threads verhalten sich dabei wie normale Threads, werden aber nicht 1:1 auf Betriebssystem-Threads abgebildet. Stattdessen gibt es einen Pool von Träger-Threads (*Carrier Threads*), denen dann virtuelle Threads vorübergehend zugewiesen werden. Sobald der virtuelle Thread auf eine blockierende Operation stößt, wird er vom Träger-Thread genommen, der dann einen anderen virtuellen Thread (einen neuen oder einen zuvor blockierten) übernehmen kann. Da Virtual Threads so leichtgewichtig sind, können sie jederzeit schnell erzeugt werden. Man muss also keine Threads wieder verwenden, sondern kann immer einfach neue instanziieren. An den Virtual Threads selbst hat sich nichts geändert, sie wurden bereits im JDK 21 finalisiert.

Im Umfeld von Virtual Threads wurde mit dem OpenJDK 19 die Struc-

tured Concurrency eingeführt. Jetzt ist sie als dritter Preview (JEP 480) ohne Änderungen wieder mit dabei. Man möchte weiteres Feedback einsammeln. Bei der Bearbeitung von mehreren parallelen Teilaufgaben erlaubt Structured Concurrency die Implementierung auf eine besonders les- und wartbare Art und Weise. Bisher wurden für die Aufteilung in parallel zu verarbeitende Aufgaben *Parallel Streams* oder der `ExecutorService` eingesetzt. Letzterer ist sehr mächtig, macht aber auch einfache Umsetzungen ziemlich kompliziert und fehleranfällig. Es ist zum Beispiel schwer zu erkennen, wenn eine der Teilaufgaben einen Fehler produziert, um dann sofort alle anderen sauber abzubrechen. Wenn zum Beispiel ein Task sehr lange läuft, erhält man erst spät Feedback, wenn andere Aufgaben in Probleme gelaufen sind. Auch das Debuggen ist nicht einfach, da in den Thread-Dumps die Tasks nicht den jeweiligen Threads aus dem Pool zugeordnet werden können.

Bei der Structured Concurrency ersetzen wir den `ExecutorService` durch einen `StructuredTaskScope`, bei dem man verschiedene Strategien (*Listing 20* verwendet `ShutdownOnFailure`) auswählen kann. Diese neue API macht nebenläufigen Code besser lesbar und kann zudem leichter mit Fehlersituationen umgehen.

```
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 20: Structured Concurrency

Mit `scope.join()` wird gewartet, bis alle Tasks erfolgreich erledigt sind. Schlägt einer fehl oder wird abgebrochen, werden auch die anderen beiden beendet. Mit `throwIfFailed()` kann der Fehler außerdem weitergegeben werden und das Ausgeben der Ergebnisse wird übersprungen.

Dieser neue Ansatz bringt einige Vorteile. Zum einen bilden Task und Sub-Tasks im Code eine abgeschlossene, zusammengehörige Einheit. Die Threads kommen nicht aus einem Thread-Pool mit schwergewichtigen Plattform-Threads, stattdessen wird jede Unteraufgabe in einem neuen virtuellen Thread ausgeführt. Bei Fehlern werden noch laufende Sub-Tasks abgebrochen. Zudem sind bei Fehlersituationen die Informationen besser, weil die Aufrufhierarchie sowohl in der Code-Struktur als auch im Stacktrace der Exception sichtbar ist.

Aufgrund des Preview-Status müssen für die Structured Concurrency beim Kompilieren und Ausführen aber weiterhin bestimmte Schalter aktiviert werden *(siehe Listing 21)*.

Scoped Values

Ebenfalls im Umfeld der virtuellen Threads wurde im JDK 20 auch eine Alternative zu den `ThreadLocal`-Variablen vorgestellt. Die Scoped Values befinden sich weiterhin im Preview Status (JEP 481). Es gab nur eine Änderung zur letzten Version: die Methode `getWhere()` wurde entfernt und dafür `callWhere()` erweitert. Der `Operation`-Parameter verwendet nun ein neues *Functional Interface*, wodurch der Compiler schlussfolgern kann, ob eine `CheckedException` geworfen wird. Ansonsten will man auch hier weitere Erfahrungen und Feedback sammeln.

Die Scoped Values erlauben das Speichern eines temporären Wertes für eine begrenzte Zeit, wobei nur der Thread, der den Wert geschrieben hat, ihn auch wieder lesen kann. Sie werden in der Regel als öffentliche statische (globale) Felder angelegt. Sie sind dann von beliebigen, tiefer im Aufruf-Stack befindlichen Methoden aus erreichbar. Verwenden mehrere Threads dasselbe `ScopedValue`-Feld, wird dieses je nach dem ausführenden Thread unterschiedliche Werte enthalten. Das Konzept funktioniert von der Idee wie `ThreadLocal`, bringt aber einige Vorteile mit.

Im Beispiel in *Listing 22* werden aus einem Web-Request Informationen zum angemeldeten Benutzer extrahiert. Auf diese Informationen muss in der weiteren Aufrufkette (im Service, im Repository, …) zugegriffen werden. Eine Variante wäre das Durchschleifen des User-Objektes als zusätzlichen Parameter in die aufzurufenden Methoden. Scoped Values verhindern diese redundante und unübersichtliche zusätzliche Parameterauflistung. Konkret wird mit `ScopedValue.where()` das User-Objekt gesetzt und dann der `run`-Methode eine Instanz von `Runnable` übergeben, für dessen Aufrufdauer der Scoped Value gültig sein soll. Alternativ kann mit der `call()`-Methode ein `Callable`-Objekt übergeben werden, um auch Rückgabewerte auszuwerten. Der Versuch, den Inhalt außerhalb des Scoped-Value-Kontextes auszuführen, führt zu einer Exception, weil bei diesem Aufruf dann kein User-Objekt hinterlegt ist.

Das Auslesen des Scoped Value erfolgt über den Aufruf von `get()`. Bei Abwesenheit eines Wertes kann man mit einem *Fallback* (`orElse()`) oder dem Werfen einer *Exception* (`orElseThrow()`) reagieren.

Auch hier müssen beim Kompilieren und Ausführen gewisse Schalter aktiviert werden (analog zu *Listing 21*).

Die Klasse `ScopedValue` ist immutable und bietet dementsprechend keine `set`-Methode an. Dadurch wird der Code besser wartbar, da es keine Zustandsänderungen am bestehenden Objekt geben kann. Muss für einen bestimmten Code-Abschnitt (Aufruf einer weiteren Methode in der Kette) ein anderer Wert sichtbar sein, kann ein Rebinding des Wertes erfolgen (zum Beispiel auf null setzen wie

```
$ javac --enable-preview --source 23 StructuredConcurrencyMain.java
$ java --enable-preview StructuredConcurrencyMain
```

Listing 21: Kompilieren und Ausführen von Preview-Funktionen

```
public final static ScopedValue<SomeUser> CURRENT_USER =
  ScopedValue.newInstance();

[..]

SomeController someController =
  new SomeController(
      new SomeService(new SomeRepository()));

someController.someControllerAction(
  HttpRequest.newBuilder()
      .uri(new URL( "http://example.com")
      .toURI()).build());

static class SomeController {

  final SomeService someService;

  SomeController(SomeService someService) {
      this.someService = someService;
  }

  public void someControllerAction(
      HttpRequest request) {

      SomeUser user = authenticate(request);
      ScopedValue.where(CURRENT_USER, user)
          .run(() -> someService.processService());
  }
}

static class SomeService {
  final SomeRepository someRepository;

  SomeService(SomeRepository someRepository) {
      this.someRepository = someRepository;
  }

  void processService() {
      System.out.println(CURRENT_USER
          .orElseThrow(() ->
              new RuntimeException("no valid user")));
      }
  }
}
```

Listing 22: Scoped Values

in *Listing 23*). Sobald der begrenzte Code-Abschnitt beendet ist, wird der ursprüngliche Wert wieder sichtbar.

```
ScopedValue.where(CURRENT_USER, null)
  .run(() -> someRepository.getSomeData());
```

Listing 23: Scoped Value Rebinding

Scoped Values arbeiten auch mit der Structured Concurrency zusammen. Die Sichtbarkeit wird an die über einen `StructuredTaskScope` erzeugten Kind-Prozesse vererbt. Somit können alle per `fork()` abgezweigten Kind-Threads ebenfalls auf die im Scoped Value befindlichen User-Informationen zugreifen.

Die Scoped Values stehen genau wie die `ThreadLocals` sowohl für Plattform- als auch virtuelle Threads zur Verfügung. Die Vorteile der Scoped Values sind vielfältig. Es erfolgt ein automatisches Aufräumen der Inhalte, sobald der Runnable-/Callable-Prozess beendet ist, wodurch Memory Leaks verhindert werden. Die Immutability der Scoped Values erhöht zudem die Verständlichkeit und Lesbarkeit. Bei der Structured Concurrency erzeugte Kind-Prozesse haben auch Zugriff auf den einen, unveränderbaren Wert. Die Informationen werden nicht wie bei `InheritedThreadLocals` kopiert, was zu einem höheren Speicherverbrauch führen kann.

Was sonst noch geschah?

Leider im negativen Sinne bemerkenswert war die vorläufige Entfernung der String Templates. Immerhin war es doch eines der interessantesten neuen Features der vergangenen Jahre. Es zeigt aber auch, dass der bestehende Prozess mit den Inkubator- und Preview-Phasen funktioniert. Die Macher des JDK haben sich aufgrund des Feedbacks zu den JEPs 430 und 459 entschieden, die Umsetzung der String-Interpolation in Java nochmal komplett zu überdenken.

Um bei uns Entwicklern keine falschen Erwartungen zu schüren und sich aber auch nicht zu sehr stressen zu lassen, wurde das Feature kurzerhand ganz entfernt. Aber mit dem Ziel, es nach einer wohl überlegten Überarbeitung bald wieder zurückkehren zu lassen. Wann das sein wird, ist momentan noch unklar. Javas Language Architect Brian Goetz sagt dazu immer süffisant: „Es ist fertig, wenn es fertig ist." Aber vielleicht bekommen wir die String Templates in abgewandelter Form bereits mit 25 zurück. Bis dahin spricht nichts dagegen, sie weiter zu testen, solange ihr noch auf Java 21 oder 22 unterwegs seid. Falls sie in Java 25 wieder aufgenommen werden und ihr sowieso nur vom letzten zum nächsten LTS-Release wechselt, steht dem Einsatz der String Templates nichts im Wege. Vermutlich wird es dann durch die Tools (IntelliJ IDEA Refactoring oder eine Open-Rewrite Rule) sogar automatisierte Code-Migrationen geben.

Des Weiteren wurden aufgrund der Verfügbarkeit von stabilen, sicheren und performanten Alternativen mit dem JEP 471 alle Methoden der Klasse `Unsafe` für den Zugriff auf On-Heap- und Off-Heap-Speicher in Java 23 als *deprecated for removal* gekennzeichnet und werden somit in einer zukünftigen Java-Version entfernt. Die Klasse `sun.misc.Unsafe` wurde bereits 2002 (Java 1.4) eingeführt. Sie bietet seit damals den direkten Zugriff auf Speicher – sowohl auf den Java Heap als auch auf den nativen (nicht vom Heap kontrollierten) Speicher. Aber wie der Name suggeriert, können die meisten dieser Methoden zu einem undefinierten Verhalten, Leistungseinbußen oder Systemabstürzen führen, wenn sie nicht korrekt eingesetzt werden. Eigentlich war diese Klasse auch nur für JDK-interne Zwecke gedacht. Aber durch das damals noch fehlende Modulsystem ließ sie sich nicht verstecken, zudem gab es keine Alternativen für die möglichst performante Implementierung bestimmter Operationen oder für den Zugriff auf größere Off-Heap-Speicherbereiche ab 2 GByte.

Heute existieren Alternativen: zum Beispiel seit Java 9 die `VarHandles`. Sie ermöglichen den direkten und optimierten Zugriff auf On-Heap-Speicher, können sogenannte *Memory Barriers* setzen und stellen auch atomare Operationen wie Compare-and-Swap bereit. Und mit dem OpenJDK 22 wurde die Foreign Function & Memory API finalisiert, eine Schnittstelle zum Aufruf von Funktionen in nativen Libraries und zur Verwaltung von nativem, also Off-Heap-Speicher.

Die Entfernung läuft in vier Phasen ab. Zunächst kommt es bei Java 23 bei der Verwendung zu Compiler-Warnungen, da die Methoden als *deprecated for removal* markiert sind. Dann soll mit Java 25 die Verwendung dieser Methoden zu Laufzeitwarnungen führen. Schon mit Java 26 werden die Methoden dann eine `Unsupported-`

`OperationException` werfen. Und schlussendlich werden die Methoden entfernt. Eine genaue Version wurde dafür aber noch nicht festgelegt.

Das Default-Verhalten kann für die jeweiligen Phasen durch die VM-Option `--sun-misc-unsafe-memory-access` überschrieben werden. Der JEP 471 *[6]* verrät im Abschnitt „sun.misc.Unsafe memory-access methods and their replacements" dazu weitere Details.

Neben diesen Deprecations wurden diesmal auch direkt Inhalte entfernt, nämlich die Methoden `Thread.suspend()/resume()`, `ThreadGroup.suspend()/resume()` sowie `ThreadGroup.stop()`. Bereits in Java 1.2 wurden diese für Deadlocks anfälligen Funktionen als Deprecated markiert und in Java 14 beziehungsweise 16 dann als *deprecated for removal* deklariert. Seit Java 19 beziehungsweise 20 werfen sie mittlerweile eine *Exception* zur Laufzeit und nun wurden sie im OpenJDK endgültig entfernt.

In Java 21 wurde beim alternativen Z Garbage Collectors (ZGC) der „Generational Mode" eingeführt. 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. Bisher musste dieser Modus aktiviert werden, jetzt ist es der Default. Natürlich lässt er sich aber auch deaktivieren. Weitere Details zu den VM-Optionen lassen sich im JEP 474 (ZGC: Generational Mode by Default) nachschlagen *[7]*.

Alle weiteren kleinere Neuerungen, für die es keine JEPs gibt, können in den Release-Notes *[8]* nachgelesen werden. Änderungen am JDK (Java-Klassenbibliothek) kann man sich zudem sehr schön über den Java Almanac *[9]* anschauen. In dieser Übersicht lassen sich beispielsweise auch alle aktuellen Änderungen an der Class-File-API nachvollziehen und die neue Klasse `java.io.IO` mit den Methoden `print()`, `println()` und `readln()` entdecken.

Fazit und Ausblick

Nach dem Release ist bekanntlich vor dem Release. Bereits im März 2025 wird das OpenJDK 24 *[10]* erscheinen. Da werden wir einige der hier angesprochenen Preview-Themen erneut wiedersehen, aber auch einige neue Themen kommen dazu. Wer sich vorab über andere zukünftige Themen informieren möchte, kann sich schon mal im JEP-Index unter „Draft and submitted JEPs" *[11]* umschauen.

Java ist und bleibt weiterhin sehr relevant. Im Jahr 2025 wird der 30. Geburtstag gefeiert. Oracle plant zu diesem Jubiläum im März 2025 ein Revival der JavaOne in der San Francisco Bay Area *[12]*. Vielleicht treffen wir uns dort und stoßen auf unser gutes, altes Java an.

Referenzen

[1] *https://openjdk.java.net/projects/jdk/23/*
[2] *https://openjdk.org/jeps/473*
[3] *https://asm.ow2.io/*
[4] *https://openjdk.org/jeps/466*
[5] *https://openjdk.org/jeps/460*
[6] *https://openjdk.org/jeps/471*
[7] *https://openjdk.org/jeps/474*
[8] *https://jdk.java.net/23/release-notes*
[9] *https://javaalmanac.io/jdk/23/apidiff/22/*
[10] *https://openjdk.java.net/projects/jdk/24/*
[11] *https://openjdk.org/jeps/0#Draft-and-submitted-JEPs*
[12] *https://inside.java/2024/03/19/announcing-javaone-2025/*

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

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

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/

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

REST vs. gRPC

Sebastian Tiemann, codecentric AG

APIs sind auch durch Datenprodukte und Tools wie ChatGPT und Co. nach wie vor unverzichtbar, wenn es um das Aufbereiten und Sammeln von Daten geht. Ein Grund dafür ist die synchrone Kommunikation zwischen Systemen, die über entsprechende Schnittstellen verbunden sind. REST als auch gRPC sind dabei zwei der am meisten verbreiteten Paradigmen. Daher möchte ich in diesem Artikel die Gemeinsamkeiten, aber auch die Unterschiede beleuchten und natürlich die Frage beantworten, ob das nicht ein Vergleich zwischen Äpfel und Birnen ist.

REST

Der Platzhirsch in der Schnittstellenkommunikation erfreut sich nach wie vor einer sehr großen Beliebtheit. Die Gründe dafür finden sich teilweise auch in den Grundlagen, daher sollten diese zunächst näher beleuchtet werden.

Der „Representational State Transfer" wurde im Jahr 2000 von Roy Thomas Fielding im Rahmen seiner Dissertation mit der Motivation veröffentlicht, einen Architekturstil zu schaffen, der den Anforderungen des modernen Web besser genügt. REST verwendet den HTTP-1.1-Standard als Basis für die Kommunikation. Die Übertragung erfolgt dabei immer in der Form einer abgeschickten Request, die von der Gegenseite mit einer Antwort quittiert wird. Die Datenübertragung erfolgt als einfacher Text. Dabei werden die bekannten HTTP-Methoden wie `GET`, `POST`, `PUT` und `DELETE` verwendet, um Daten über verschiedene Endpunkte zu verarbeiten. Diese Daten werden von REST als Ressourcen betrachtet, wobei jede Ressource über eine URI direkt identifizierbar ist. Handlungen, die mit einer Ressource vorgenommen werden können, sind über HTTP-Methoden wie `GET` oder `DELETE` abgebildet. Somit kann ich beispielsweise über `GET …/users/1` Informationen der Benutzerressource anfragen oder diese per `DELETE` löschen, sofern ich über die nötigen Berechtigungen verfüge.

Um über den Ausgang einer solchen Operation zu informieren, werden entsprechende HTTP-Fehlercodes verwendet. Dies sind dreistellige Zahlen, die je nach Anfangsstelle einen anderen Grundzustand beschreiben. Die 200er-Codes stehen dabei für Erfolgsmeldungen, während die 400er-Codes Fehler am gestellten Request oder der Ressource symbolisieren. Haben wir keine ausreichenden Berechtigungen, die Ressource „Benutzer 1" zu löschen, wird entweder ein 404-Fehlcode zurückgeliefert (Ressource konnten nicht gefunden werden) oder klassisch ein 403er-Code, was bedeutet, dass der Zugriff nicht gestattet ist.

Darüber hinaus müssen je nach Art der Anfragen und Antworten Daten im HTTP-Body angehängt werden, zum Beispiel in *JSON* oder *XML*. *[1]*

Insgesamt ist REST damit unabhängig von der Programmiersprache des Sender- und Empfängersystems und wir können sprachagnostisch arbeiten. Diese Eigenschaften haben sicherlich einen sehr

großen Anteil daran, dass REST trotz seines Alters immer noch sehr viel genutzt wird.

Darüber hinaus haben sich im Laufe der Jahre weitere Tools und Standards entwickelt, um das Arbeiten mit REST angenehmer zu gestalten. Allen voran ist hier die aus *Swagger* gewachsene *OpenAPI Specification* [2] zu nennen.

Diese ermöglicht als *Interface Definition Language* (IDL) die Beschreibung von REST-APIs. Auf dieser Basis können Consumer- und Producer-Codes in Form von Stubs generiert werden. Als simulierter Gegenpart unserer API sparen wir Zeit bei der Implementierung und beim Bugfixing, da wir Methoden und Tests direkt gegen die generierten Typen und Schnittstellen entwickeln müssen. Entsprechend sparen wir auch Zeit im produktiven Code, wenn sich Änderungen an den Schnittstellen ergeben. Nicht nur können wir mit Stubs für die Fehlerfreiheit dieser sorgen, der Compiler weist auch direkt auf sich geänderte Felder in Transferobjekten hin, die sich auch aus der *OpenAPI Specification* generieren lassen. Darüber hinaus ist es sinnvoll, Endpunkte und erwartete Response-Codes beispielsweise via Contract-Testing sicherzustellen und zu dokumentieren. Um unsere REST-API änderbar zu gestalten, empfiehlt es sich außerdem, über ein Konzept zur Endpunkt-Versionierung in URL, im Body oder im HTTP-Header nachzudenken. Auch das Weglassen einer Versionierung ist ein gängiger Weg, doch dieser sollte bewusst gewählt werden.

Die Kommunikation über REST versteht sich außerdem als zustandslos. Das bedeutet, dass alle Informationen, die für die Verarbeitung eines Requests nötig sind, in diesem enthalten sein müssen. Ein Beispiel dafür ist die Session-Information nach einer Anmeldung. Nicht die HTTP-Verbindung eines bestimmten Absenders oder vorangegangene Requests gestatten den Zugriff auf eine API, sondern ausschließlich die mitgesendete Session-Information.

Aufgrund des potenziell formlosen, textbasierten Nachrichtenaustauschs kann dieser natürlich auch abseits von *OpenAPI* mit beliebigen informellen Konstrukten erfolgen, da es hierfür keine Regeln gibt, jedoch gibt es das von Leonard Richardson erfundene *Maturity Model* [2, 3].

Dies unterscheidet zwischen mehreren Stufen der Reifegrade verschiedener Nutzungsweisen von REST (siehe Abbildung 1). Bei Level 0 werden weder die semantisch eigentlich vorgesehene HTTP-Methode verwendet noch inhaltlich die Request-Bodies formal strukturiert. Bei der Verwendung von Level 1 sind zwar schon verschiedene Ressourcen identifizierbar, HTTP-Methoden werden für die Endpunkte noch nicht weiter berücksichtigt. Das geschieht erst mit Level 2, dem vermutlich am weitesten verbreiteten Reifegrad. Ressourcen und HTTP-Methoden werden in Form einzelner Endpunkt verwendet. Level 3 bedient sich inhaltlich des HATEOAS-Ansatzes (*Hypermedia as the engine of application state*) und der richtig verwendeten HTTP-Verben. Dies gibt den von Fielding beschriebenen zu nutzenden Ansatz im Umgang mit REST wieder. Jede Ressource soll in der Antwort eine Art API bestehend aus möglichen Endpunkten mitliefern, die aus Methoden und Endpunkten für diese Ressource besteht.

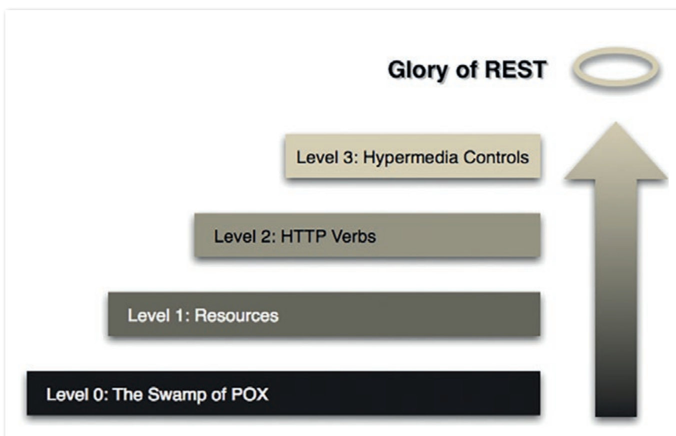


Abbildung 1: Glory of REST (© Martin Fowler)

gRPC …

… oder ausgeschrieben: „gRPC Remote Procedure Call", ist ein Framework, das von Google entwickelt wurde. Als Basis für die Kommunikation dient HTTP 2.0. Diese findet nicht im „dienstleistungsorientierten Stil" wie bei REST statt, sondern auf Prozedurebene. Im Kontext von Java wird beispielsweise statt über einen RestClient ein `GET` an `/users/1` zu senden, vereinfacht `GrpcService.getUser(1)` aufgerufen. Damit steht der Aufruf der Methode `getUser()` im Fokus und nicht der REST-Endpunkt mit seinem Fokus auf User-Ressourcen.

Entsprechend gibt es bei gRCP keine Unterscheidung zwischen verschiedenen HTTP-Verben. Stattdessen kapselt jeder RPC ein Status-Objekt, das die transportierte Nachricht und eine Liste von Status-Codes vorhält. Mögliche Codes sind beispielsweise „OK" im Falle der 0 oder „UNAUTHENTICATED" bei einer 16. gRPC ist ebenso *stateless* und sprachagnostisch wie REST. Ähnlich wie OpenAPI sorgt die IDL Protobuf für eine Strukturierung und Typisierung einzelner Felder der auszutauschenden Daten und kann für Typ-, Service- und Stubgenerierung genutzt werden. Diese proto-Dateien sind dagegen kompiliert und nicht wie ein OpenAPI-Document optional. Somit ist etwas Vergleichbares zum Richardson-Maturity-Model hierbei nicht notwendig, da es keine Reifegrade wie die in REST gibt. Für eine API-Versionierung bietet es sich an, den Protobuf-Package-Namen in Kombination mit einem dazugehörigen Service zu verwenden. Darüber hinaus wird in der Regel nicht mit dem oben genannten Request-Response-Modell gearbeitet, sondern mit verschiedenen Arten von Streams des HTTP-2.0-Standards.

Streaming

Durch die Verwendung von HTTP-2.0-Streaming kann gRPC nicht ohne Weiteres in Browsern analysiert werden, wie es mit REST möglich ist.

Dafür müsste ein direkter Zugriff auf die in HTTP 2.0 eingeführten Frames bestehen und genau das ist nicht möglich. *grpc-web* als JavaScript-Bibliothek bietet einen Workaround mittels eines Proxys an. Um tiefer auf die Ursachen und Details einzugehen, ist es wichtig, die Unterschiede zwischen HTTP 1.1 und HTTP 2.0 herauszustellen [4] [5] [6].

HTTP 1.1 vs. HTTP 2.0

Der Standard HTTP 2.0 stammt aus dem Jahr 2015, ist also noch gar nicht so alt. Ebenso ist das Problem, das man mit dieser neueren Version von HTTP lösen wollte, noch relativ jung.

Mit dem Aufkommen von mobilen Endgeräten wie Smartphones und Tablets entstand der Bedarf, Webseiten schneller zu laden zu können. Dies kann beispielsweise durch Reduktion der Größe der übertragenen Datenmenge erreicht werden, aber das eigentliche Problem, das man gelöst hat, betrifft den Übertragungsweg selbst. HTTP 1.1 überträgt Daten sequenziell in Form von Requests, die in einer Queue pro Connection eingereiht werden. Zwar hat man nach der Einführung von HTTP 1.0 schnell gemerkt, dass es nicht ausreicht, nur eine solche Queue bereitzustellen. Doch auch das Verwenden von mehreren Verbindungen war nicht ausreichend, ganz zu schweigen vom rechenintensiven Neuaufbau einer Verbindung. Entsprechend kann es passieren, dass die Abarbeitung der Elemente in der Queue längere Zeit benötigt, wenn die jeweils vorderen Requests nicht abgearbeitet werden können, und der Seitenaufbau sich so verzögert.

Dies wird oft als das „Head-Of-Line-Blocking"-Problem bezeichnet. Mithilfe der Einführung von uni- und bidirektionalem Streaming wird das Problem größtenteils umgangen.

Technisch gesprochen wird das, was in HTTP 1.1 als Request und Response bekannt ist, in mehrere sogenannten Frames aufgespalten und versendet. Jeder dieser Frames besitzt dabei eine ID, die sowohl Versender als auch Empfänger ermöglichen, die Pakete wieder vollständig zusammenzusetzen. Eine Reihenfolge der einzelnen Frames besteht nicht. Dadurch lässt sich auch erklären, warum das „Hineinschauen" in diese Kommunikation mit einem Browser nicht so einfach ist. In *Abbildung 2* werden diese Unterschiede zwischen HTTP 1.1 und HTTP 2.0 nochmal verdeutlicht.

Darüber hinaus überträgt HTTP 2.0 den Inhalt eines Frames nicht im Klartext, sondern binär. In Verbindung mit der Header-Komprimierung werden die Übertragungszeiten insgesamt verringert. Im Umkehrschluss bedeutet dies natürlich, dass auch alle beteiligten Komponenten, also zum Beispiel Browser, Client- oder Serverbibliotheken, mit HTTP 2.0 umgehen können müssen. Eine Liste von Frameworks, die HTTP 2.0 unterstützen, findet sich im GitHub-Repository der HTTPWG *[7]*.

Performance

Bei der Verwendung von HTTP 2.0 sind durch die Nutzung von Komprimierung und Streaming spürbare Performancegewinne zu erwarten. Durch die Verwendung von Protobuf kann ebenso Zeit bei der Serialisierung/Deserialisierung gespart werden, konkret behauptet Microsoft, es sei „bis zu 8-mal schneller". Auch die Verringerung der Request-Größe wirkt sich positiv auf die Performance aus. Diese wird im Kontext von gRPC verglichen mit REST mit „60-80 % kleiner" angegeben *[8]*.

Support im Java- und Kotlin-Ökosystem

Die vermutlich größte und zumindest laut eigener Aussage beliebteste Framework-Sammlung für Java ist Spring. Für REST finden wir eine breite Unterstützung für alle möglichen Aufgaben, die mit einer REST-Schnittstelle in Verbindung stehen. Ebenso existieren viele weitere helfende Frameworks, die uns das Leben leichter machen. So generieren wir mithilfe von Spring REST Docs eine Dokumentation aus unseren Tests *[9]* oder schreiben Contract-Tests für generierte Stubs mit Spring Cloud Contract *[10]*. Außerdem können wir uns mit Springdoc-OpenAPI unsere OpenAPI-Spec aus bestehendem Code erstellen lassen *[11]*. Auch gibt es ein Spring-Boot Starter-Projekt *[12]*, das uns ein breites Spektrum an Abhängigkeiten mitbringt, damit wir loslegen können. Diese Bibliotheken sind nur ein

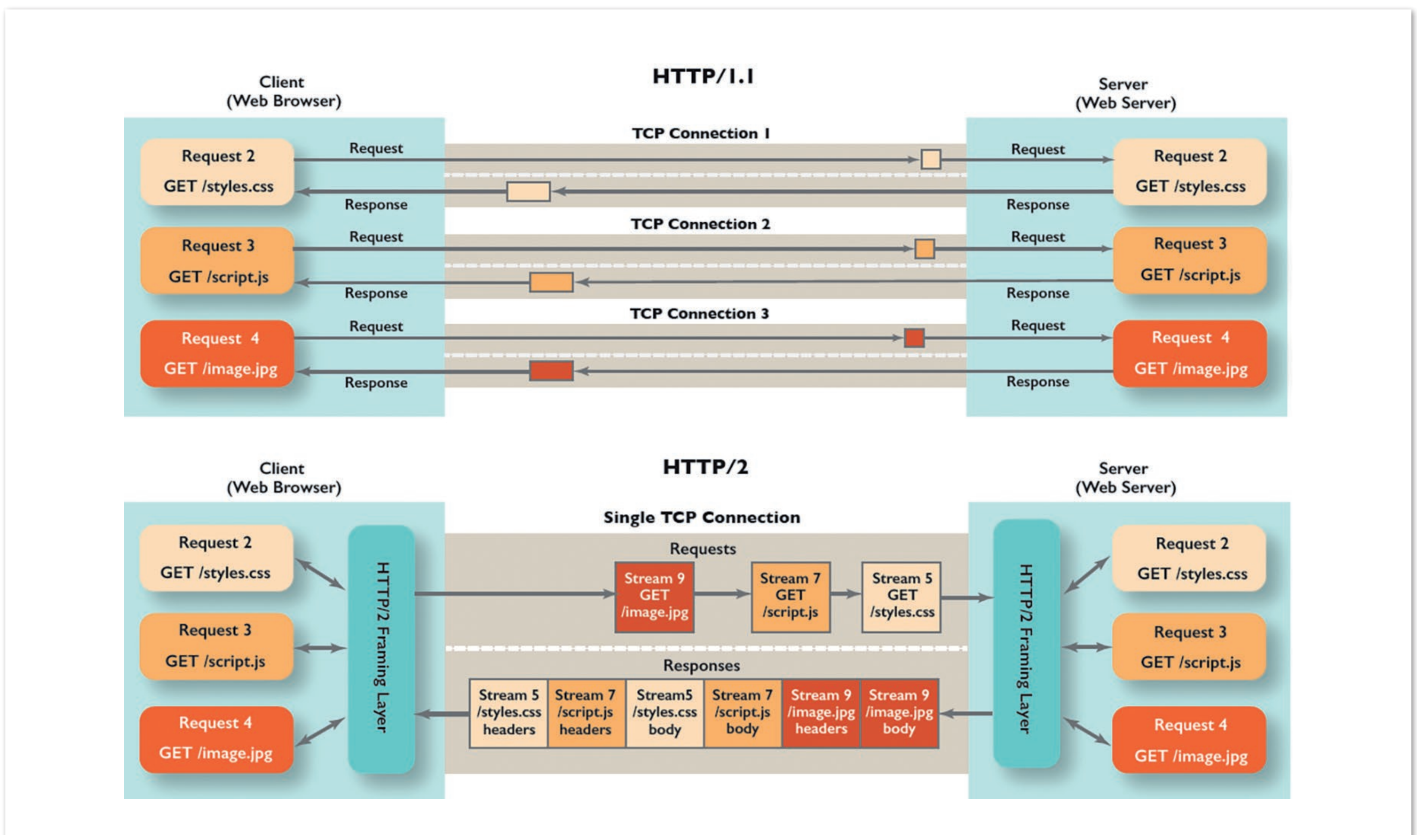


Abbildung 2: Unterschiede zwischen HTTP 1.1 und HTTP 2.0 (© Manning)

kleiner Ausschnitt. Insgesamt gibt es also ein riesiges Spektrum an Unterstützung für sehr viele Anwendungsfälle.

gRPC findet dagegen keinen derart breiten Support im Spring-Universum, aber dennoch gibt es Lösungen, wie beispielsweise *Spring Cloud Gateway* [13]. Auch das genannte *Spring Cloud Contract* [14] bietet die Möglichkeit, Schnittstellen-Tests für gRPC zu schreiben. Dieses Feature ist laut den Docs [15] aktuell noch im Beta-Status. Abseits von Spring finden sich dagegen Lösungen sowohl direkt von Google [16] als auch von Red Hat mittels *Quarkus* für Java und Kotlin [17].

Zusammenfassung

Ist gRPC der bessere synchrone Kommunikationsweg? Das kommt drauf an. Auf dem Papier bietet gRPC eine noch bessere Performance als REST, aber ist REST wirklich langsam? Zalando als großer Player im E-Commerce nutzt beispielsweise im eigenen Eventbroker Nakadi REST [18], auch wenn dieses Projekt seit Juni nur noch intern weiterentwickelt wird. Google hat bei der Entwicklung von gRPC und den dazugehörigen Komponenten sicherlich Anwendungsfälle mit noch höheren Anforderungen an die Performance bedienen müssen, weshalb selbst die kleinste Optimierung absolut notwendig gewesen ist. Dabei fielen der aktuell noch fehlende Browser-Support sowie die Unterschiede in der API-Struktur vermutlich nicht ins Gewicht. Auf der technischen Ebene nutzt OpenTelemetry gRPC und HTTP 1.1, um möglichst breit aufgestellt zu sein [19].

Ich persönlich habe die Erfahrung gemacht, dass weniger der Kommunikationsweg ein Problem ist, sonder eher die Implementierung, welche die API konsumiert oder mit Daten versorgt. Sich dabei auf einen gängigen Mitspieler wie Spring verlassen zu können, fühlt sich gut an. Dass aber gerade dieser Faktor nicht gegeben ist, zeigt mir, dass das Bedürfnis, gRPC im Java-Ökosystem verwenden zu wollen, nicht ganz so ausgeprägt ist. Oder, wie im Beispiel Nakadi demonstriert, der Standard gRPC noch nicht etabliert genug ist.

Quellen

[1] *https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf*
[2] *https://swagger.io/specification/*
[3] *https://www.crummy.com/writing/speaking/2008-QCon/act3.html*
[4] *https://learn.microsoft.com/de-ch/aspnet/core/grpc/comparison?view=aspnetcore-8.0*
[5] *https://grpc.io/blog/state-of-grpc-web/#f3*
[6] *https://grpc.io/blog/postman-grpcweb/*
[7] *https://github.com/httpwg*
[8] *https://learn.microsoft.com/de-de/dotnet/architecture/cloud-native/grpc*
[9] *https://docs.spring.io/spring-restdocs/docs/current/reference/htmlsingle/*
[10] *https://spring.io/projects/spring-cloud-contract*
[11] *https://springdoc.org/*
[12] *https://docs.spring.io/spring-boot/reference/web/index.html*
[13] *https://spring.io/blog/2021/12/08/spring-cloud-gateway-and-grpc*
[14] *https://docs.spring.io/spring-cloud-contract/reference/project-features-flows/grpc.html*
[15] *https://docs.spring.io/spring-cloud-contract/reference/project-features-flows/grpc.html*
[16] *https://grpc.io/docs/languages/java/quickstart/*
[17] *https://quarkus.io/guides/grpc-getting-started*
[18] *https://github.com/zalando/nakadi*
[19] *https://opentelemetry.io/docs/specs/otlp/*

Sebastian Tiemann

sebastian.tiemann@codecentric.de

Sebastian Tiemann ist leidenschaftlicher Softwareentwickler und unterstützt die Kunden der codecentric AG mit Rat und Tat als Senior API Consultant. Seine 15 Jahre Berufserfahrung helfen ihm vorrangig im Java- und Devops-Umfeld dabei, immer neue technische Wege zu finden, um gesteckte Ziele zu erreichen und mögliche Probleme zu lösen. Agile Methoden und wartbarer Code sind dabei die notwendige Basis. Außerdem organisiert er die Java User Group Dortmund und ist Konferenzsprecher.

Java aktuell

JAHRESABO

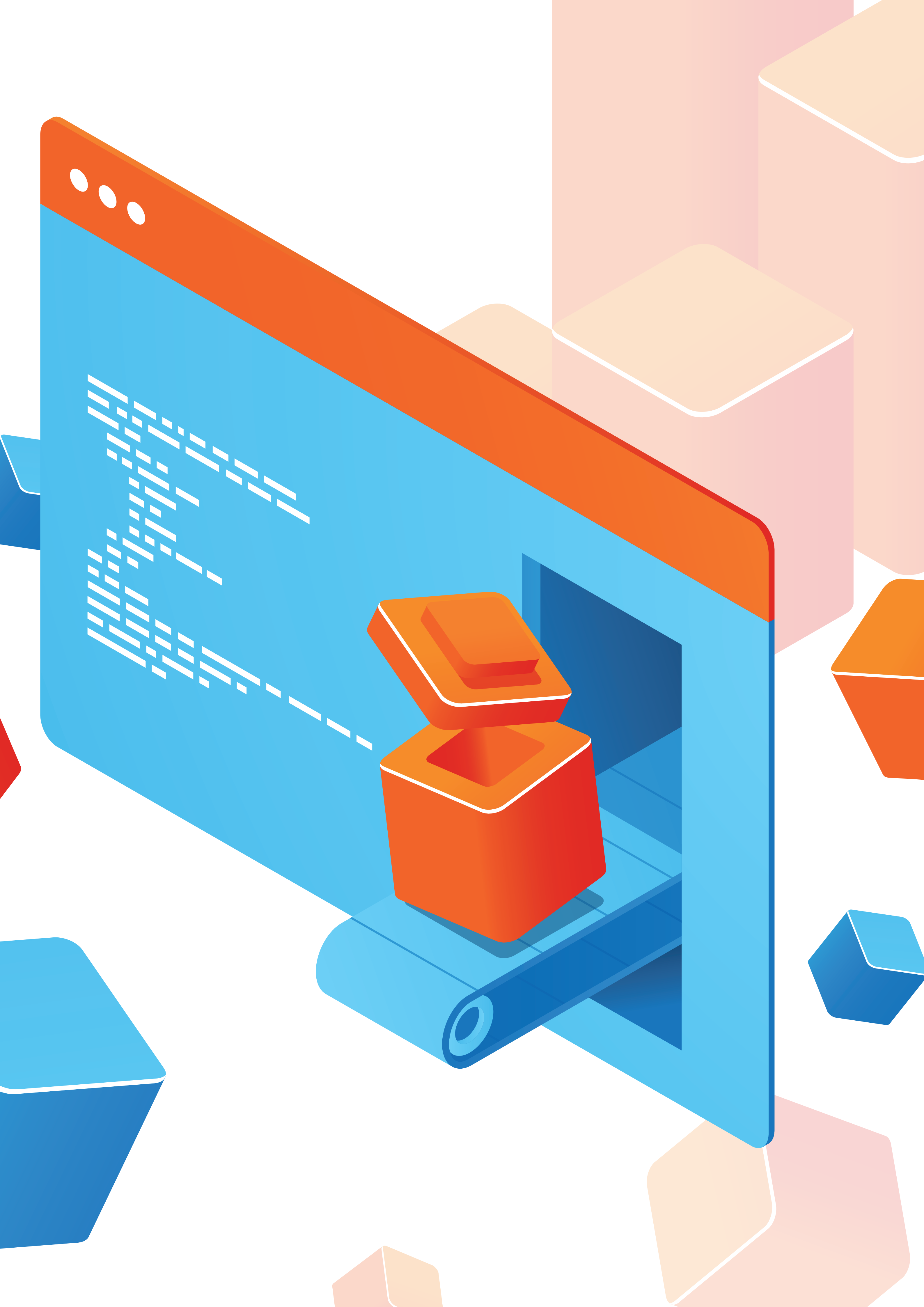
FÜR 29,00 €
BESTELLEN

Mehr Informationen zum Magazin und Abo unter:

www.ijug.eu/de/java-aktuell

CSS für Backend-Entwickler

Christina Zenzes, codecentric AG



CSS (Cascading Style Sheets) bildet zusammen mit HTML und JavaScript das Fundament moderner Websites und Web-Anwendungen. Während HTML die Struktur beschreibt und JavaScript für Interaktivität sowie dynamische Funktionen sorgt, kümmert sich CSS um die visuelle Gestaltung. Für viele Backend-Entwickler bleibt CSS jedoch ein Buch mit sieben Siegeln – dabei kann ein solides Verständnis der Grundlagen und eine gezielte Anwendung den Unterschied machen. In diesem Artikel werfen wir einen Blick auf drei der Schlüsselkonzepte von CSS: ***Inheritance****, die* ***Kaskade*** *und das* ***Layouting****. Ziel ist es, die wichtigsten Grundlagen zu vermitteln, um komplexere Probleme zu lösen und die Zusammenarbeit mit Frontend-Entwicklern zu vereinfachen.*

Um die verschiedenen Konzepte anschaulich zu erklären, wird in diesem Artikel durchgängig dasselbe Beispiel verwendet. Dabei handelt es sich um eine Website, die Benutzern Informationen zu einem Kurs bereitstellt und gleichzeitig die Anmeldung dazu ermöglicht. *Abbildung 1* zeigt eine Vorschau dieser Seite.

Da nicht alle HTML-Elemente für die Erklärungen relevant sind, werden in den Codebeispielen nur die wesentlichen Ausschnitte dargestellt. Das verkürzte HTML der Seite finden Sie in *Listing 1*, während das dazugehörige CSS in *Listing 2* aufgeführt ist.

Für diesen Artikel setze ich voraus, dass die Grundlagen von CSS bekannt sind. Die grundsätzliche Syntax und grundlegende Properties, beispielsweise für das Setzen von Farben oder Schriften, sollten bekannt sein. Hier steigen wir direkt ins Thema ein, denn immer wieder kommt es vor, dass Elemente Eigenschaften annehmen, die für diese nicht direkt definiert zu sein scheinen. Ein Grund hierfür kann die Vererbung innerhalb von CSS sein.

Inheritance

Jede CSS-Eigenschaft hat standardmäßig einen bestimmten Wert, der angewendet wird, wenn weder der Browser noch der Entwickler eigene Styles festgelegt haben. Diese Standardwerte lassen sich grob in zwei Kategorien unterteilen: vererbbare und nicht vererbbare Eigenschaften *[1]*.

Vererbbare Eigenschaften übernehmen den Wert des übergeordneten Elements, wenn keine spezifischen Styles für das aktuelle Element definiert sind. Ein Beispiel dafür ist die Eigenschaft *color* (Schriftfarbe). Wird beispielsweise die *color*-Eigenschaft des html-Elements auf *blue* gesetzt, erscheint die Schriftfarbe aller innerhalb dieses Elements liegenden Inhalte ebenfalls in Blau. Falls kein übergeordnetes Element existiert, wird der initiale Standardwert der Eigenschaft verwendet, der vom Browser festgelegt wird. Im Fall von *color* ist der Standardwert in der Regel *canvastext*. Weitere vererbbare Eigenschaften sind *font*-size (Schriftgröße), *font-family* (Schriftart) oder font-weight (Schriftgewicht).

> Die CSS-Farbe *canvastext* ist ein systemabhängiger Schlüsselwortwert, der die Standard-Schriftfarbe für Text auf Canvas-Elementen definiert. Sie gehört zu den sogenannten „System Colors" *[2]*, die sich an den visuellen Vorgaben des Betriebssystems oder Browsers orientieren. *canvastext* wird üblicherweise durch die Plattform- und Benutzereinstellungen bestimmt und sorgt dafür, dass Text auf Canvas-Elementen gut lesbar bleibt, zum Beispiel Schwarz auf weißem Hintergrund oder Weiß auf dunklem Hintergrund.

Nicht vererbbare Eigenschaften verhalten sich genau umgekehrt. Sie übernehmen nicht den Wert des übergeordneten Elements, sondern

```
<section>
  <h2>CSS for backend Devs</h2>
  <p>Lorem ipsum dolor sit amet, consetetur sadipscing elitr,…</p>
  <p>Duis autem vel eum iriure dolor in hendrerit in vulput …</p>
  <button><span class="icon"></span><span>Ähnliche Kurse</span></button>
</section>
<section>
  <h2>Anmelden zum Kurs</h2>
  <form>
     <label for="vorname">Vorname</label>
     <input id="vorname" />
     <label for="nachname">Nachname</label>
     <input id="nachname" />
     <label for="email">E-Mail-Adresse</label>
     <input id="email" />

     <button class="back">Zurück</button>
     <button class="primary" id="register">Anmelden</button>
  </form>
</section>
```

Listing 1: HTML-Code der Beispielanwendung

```
section:first-child {
  background-color: #3da35d;
  color: white;
}

form {
  display: grid;
  grid-template-columns: 1fr 1fr;
  gap: 1rem;
  grid-auto-flow: dense;
}

#vorname,
label:has(+ #vorname) {
  grid-column: 1 / 2;
}

#nachname,
label:has(+ #nachname) {
  grid-column: 2 / 3;
}

#email,
label:has(+ #email) {
  grid-column: 1 / 3;
}

form button#register {
  background-color: darkred;
}

form button#register:hover {
  background-color: rgb(185, 3, 3);
}

button {
  background-color: white;
  display: flex;
  gap: 0.5rem;
  align-items: center;
  justify-content: center;
}
form button {
  background-color: pink;
}

button.primary {
  background-color: #3da35d;
}
```

Listing 2: CSS der Beispielanwendung

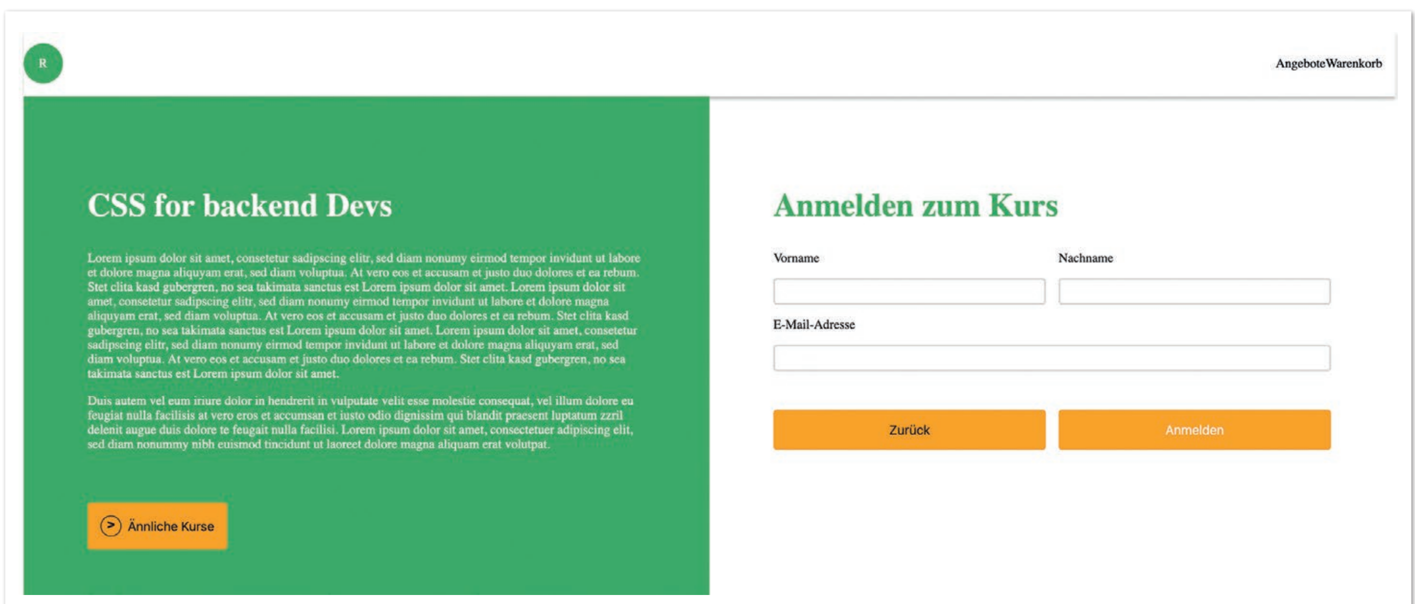


Abbildung 1: Beispielanwendung (© Christina Zenzes)

CSS for backend Devs

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Abbildung 2: Die Schriftfarbe wird über Vererbung weitergegeben. (© Christina Zenzes)

verwenden immer ihren initialen Standardwert. Ein Beispiel hierfür ist die Eigenschaft *padding*, die den Innenabstand zwischen dem Rahmen des Elements und seinem Inhalt festlegt. Das *padding* eines untergeordneten Elements wird nicht vom übergeordneten Element übernommen, da jedes Element seine eigenen Innenabstände besitzt oder auch keine haben kann. Der Standardwert von *padding* ist in der Regel *0*. Weitere nicht vererbbare Eigenschaften sind *border* (Rahmen), *margin* (Abstand) oder *position* (Positionierung).

Auf der Kurs-Seite *(siehe Abbildung 2)* lässt sich dieses Verhalten gut an der linken Seite beobachten, auf der die Kursinformationen angezeigt werden. Die Schriftfarbe ist bei allen Elementen weiß, obwohl sie nur auf dem übergeordneten Element einer *section* definiert wurde *(siehe Listing 3)*. Betrachtet man die untergeordneten Elemente genauer im Browser-Inspektor *(siehe Abbildung 3)*, wird deutlich, dass die Schriftfarbe von der *section* geerbt wird. Im Gegensatz dazu wird die Hintergrundfarbe (*background-color*) nicht vererbt. Diese Eigenschaft taucht in der Liste der untergeordneten Elemente nicht auf.

CSS-Kaskade

In CSS ist es häufig der Fall, dass verschiedene Styles auf einem Element miteinander konkurrieren. Um diese Konflikte zu lösen,

```
section:first-child {
  background-color: #3da35d;
  color: white;
}
```

Listing 3: Die Farbe wird vererbt, während die Hintergrundfarbe nicht vererbt wird.

verwendet der Browser einen Algorithmus namens CSS-Kaskade *[3]*. Die nachfolgenden Abschnitte erklären die einzelnen Schritte der Kaskade.

1. Relevanz

Der erste Schritt der CSS-Kaskade besteht darin, zu überprüfen, ob ein bestimmter Selektor auf ein Element zutrifft. Dies bedeutet, dass nur diejenigen Selektoren ausgewählt werden, die auf das entsprechende Element passen, wie zum Beispiel auf einen Button.

Ein kurzer Überblick über einige der Selektoren *[4]* und deren mögliche Kombinationen:

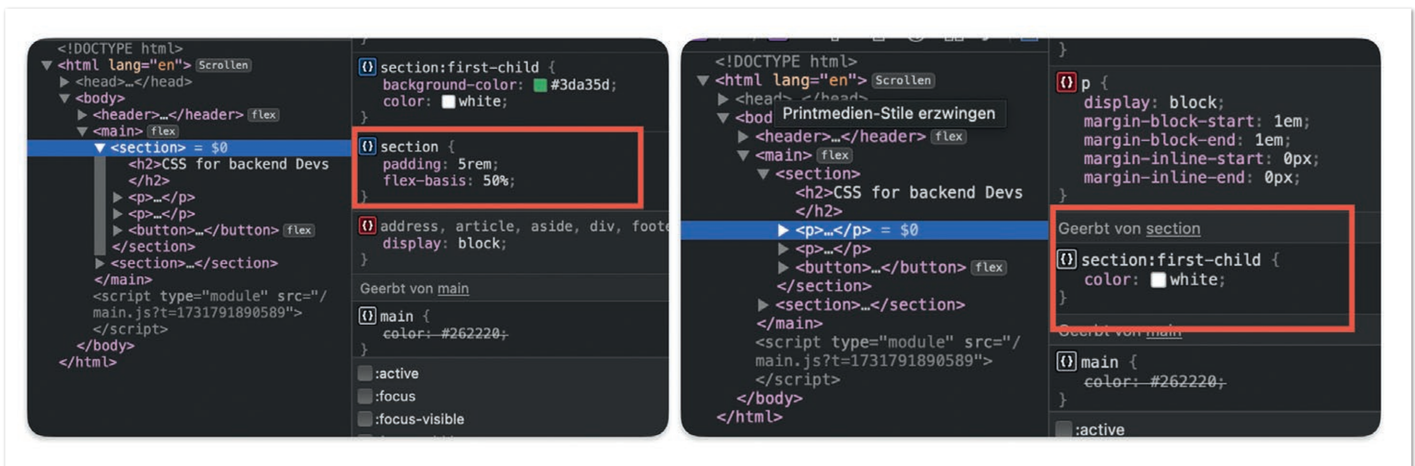


Abbildung 3: Im Inspector kann die Vererbung nachvollzogen werden. (© Christina Zenzes)

- **Tag-Selektoren:** Sie selektieren Elemente anhand des HTML-Tags. Zum Beispiel werden mit dem Selektor button alle *Button*-Elemente im HTML-Dokument erfasst und gestylt.
- **Klassen-Selektoren:** Sie selektieren Elemente anhand des HTML-*class*-Attributs. Klassen-Selektoren werden im CSS mit einem Punkt vor dem Namen gekennzeichnet. Die Stile werden angewendet, wenn der Selektor im *class*-Attribut eines Elements aufgelistet wird.
- **ID-Selektoren:** Sie selektieren Elemente anhand des ID-Attributes. ID-Selektoren werden im CSS mit einer Raute vor dem Namen gekennzeichnet.
- **Pseudo-Klassen:** Sie selektieren Elemente in einem bestimmten Zustand, zum Beispiel wenn der Benutzer mit der Maus darüberfährt. Pseudo-Klassen werden im CSS mit einem Doppelpunkt gekennzeichnet und treten normalerweise in Kombination mit anderen Selektoren auf.
- **Descendant-Kombinator:** Er definiert, welchen übergeordneten Selektor ein Element haben muss, damit der Selektor angewendet wird. Kombinierte Selektoren werden durch ein Leerzeichen getrennt.

Auf der Kursseite ist auf der rechten Seite ein Anmelde-Button sichtbar. Bei der Anwendung der CSS-Kaskade bleiben für den Button nur noch fünf Regeln übrig, die alle die Hintergrundfarbe des Buttons festlegen. Nun muss über die Kaskade bestimmt werden, welcher Wert letztendlich verwendet wird *(siehe Listing 4)*.

```
/* aus dem User-Stylesheet*/
button {
  background-color: orange;
}

/* aus dem User-Agent-Stylesheet*/
button {
  background-color: buttonface;
}

/* aus dem Author-Stylesheet*/
form button#register {
  background-color: darkred;
}

button {
  background-color: white;
}
form button {
  background-color: pink;
}

button.primary {
  background-color: #3da35d;
}
```

Listing 4: Die Hintergrundfarbe wird an mehreren Stellen definiert.

2. Origin und !important

Im nächsten Schritt ordnet der Algorithmus die Regeln nach dem Ursprungstyp der Regel und der Verwendung von *!important*. Es gibt insgesamt drei verschiedene Ursprungstypen von CSS-Regeln: User-Agent, Benutzer und Autor.

Das User-Agent-Stylesheet wird manchmal auch als Browser-Stylesheet bezeichnet. Die darin definierten Styles werden vom Browser selbst festgelegt. So legt der Browser in der Regel zum Beispiel das Aussehen von Überschriften oder das Unterstreichen von Links fest. Das Default-Styling des Browsers gewährleistet, dass eine Webseite immer lesbar ist.

Das Benutzer-Stylesheet kann vom Benutzer definiert werden. Der Benutzer kann beispielsweise über den Browser die Schriftart oder -größe ändern. Auch einige Browser-Plug-ins können die Styles eines Dokuments überschreiben.

Das Autor-Stylesheet könnte auch als Entwickler-Stylesheet bezeichnet werden. Es umfasst alle Stylesheets, die vom Entwickler geschrieben wurden.

Der Kaskade-Algorithmus ordnet die Styles nach den verschiedenen Ursprungstypen. Zuerst werden die Styles aus dem User-Agent-Stylesheet ausgewählt. Danach werden die Styles aus dem Benutzer-Stylesheet angewendet und überschreiben dabei die Styles aus dem User-Agent-Stylesheet. Letztendlich werden die Styles aus dem Autor-Stylesheet angewendet, die alle vorherigen Styles überschreiben, sofern diese definiert sind.

Im vorherigen Schritt, in *Listing 4*, wurde bereits gezeigt, dass einige Styles unterschiedliche Ursprungstypen aufweisen können. Wenn wir nun die beschriebenen Schritte anwenden, werden die Selektoren entsprechend sortiert und angewendet. Der Browser verwendet zunächst die im User-Agent-Stylesheet definierte Hintergrundfarbe. In diesem Beispiel wird dieser Style jedoch durch einen Style überschrieben, den der Benutzer im Benutzer-Stylesheet definiert hat (Anmerkung: Dies ist ein eher ungewöhnliches Beispiel, normalerweise werden hier eher Schriftgrößen oder Farben überschrieben). Schließlich nimmt der Browser alle Styles aus dem Autor-Stylesheet. Alle nicht überschriebenen Styles sind in *Listing 5* durchgestrichen. Im Autor-Stylesheet gibt es mehrere Regeln, die die Hintergrundfarbe festlegen. Der Browser verwendet diese Sty-

```
/* aus dem Author-Stylesheet*/
button {
form button#register {
  background-color: darkred;
}

button {
  background-color: white;
}
form button {
  background-color: pink;
}

button.primary {
  background-color: #3da35d;
}

/* aus dem User-Stylesheet*/
button {
  background-color: orange;
}

/* aus dem User-Agent-Stylesheet*/
button {
  background-color: buttonface;
}
```

Listing 5: Die durch Origin nicht mehr relevanten Properties sind durchgestrichen.

les und wendet anschließend den nächsten Schritt, die Berechnung der Spezifität, an.

Eine wichtige Ausnahme von dieser Regel ist das Keyword *!important*. Es erhöht entgegen der häufigen Annahme nicht die Spezifität eines Selektors, sondern ändert lediglich die Reihenfolge der Ursprungstypen. Die neue Reihenfolge mit *!important* sieht wie folgt aus:

1. Styles aus dem User-Agent-Stylesheet
2. Styles aus dem Benutzer-Stylesheet
3. Styles aus dem Autor-Stylesheet
4. Styles aus dem Autor-Stylesheet mit !important
5. Styles aus dem Benutzer-Stylesheet mit !important
6. Styles aus dem User-Agent-Stylesheet mit !important

Alle Styles aus dem Autor-Stylesheet mit *!important* überschreiben die Styles aus dem Autor-Stylesheet ohne *!important*. Diese wiederum werden von allen Styles mit *!important* aus dem Benutzer-Stylesheet überschrieben. Und wiederum werden diese von allen Styles mit *!important* aus dem User-Agent-Stylesheet überschrieben.

Wenn wir das Keyword *!important* wie in *Listing 6* bei der Hintergrundfarbe im User-Stylesheet hinzufügen, ändert sich die Reihenfolge der Anwendung der Styles. Zuerst werden alle Styles basierend auf den Ursprungstypen angewendet, gefolgt von allen Styles mit *!important* in umgekehrter Reihenfolge der Ursprungstypen.

```
/* aus dem User-Stylesheet*/
button {
  background-color: orange important!;
}

/* aus dem Author-Stylesheet*/
form button#register {
  background-color: darkred;
}
```

Listing 6: Auswirkung der Nutzung von !important

Das Schützen von Werten durch Überschreibungen macht Sinn, wenn beispielsweise eine Komponente oder ein Element ohne diese Styles nicht mehr funktioniert. Es kann auch verwendet werden, um das Überschreiben von Benutzer-Styles zu verhindern. Es kann vorkommen, dass ein Benutzer eine größere Schriftgröße benötigt, um die Seite besser lesen zu können. Wenn dieser Style jedoch durch das Autor-Stylesheet überschrieben wird, wird die Seite für den Benutzer unlesbar.

Aus diesen Gründen sollte das *!important*-Keyword nicht zu häufig in eigenen CSS-Dateien verwendet werden.

Das *!important*-Keyword sollte nur verwendet werden, um das Überschreiben von Styles zu verhindern. Es sollte nicht dazu verwendet werden, einen bestehenden Style zu überschreiben, wie es häufig empfohlen wird. In solchen Fällen sollte man stattdessen einen Selector mit höherer Spezifität verwenden.

3. Spezifität

Die Spezifität *[5]* eines Selektors bestimmt, welcher Selektor angewendet wird. Im Code-Beispiel aus *Listing 5* gibt es mehrere Regeln, die noch im Konflikt miteinander stehen. Allein mit Schritt zwei kann der Browser nicht entscheiden, welche Regel letztendlich angewendet werden soll.

Der Browser hat jedoch eine Lösung dafür. Er berechnet die Spezifität der einzelnen Selektoren. Der Selektor mit der höchsten Spezifität gewinnt und dessen Styles werden angewendet.

Die Berechnung der Spezifität erfolgt, indem der Browser die Anzahl der unterschiedlichen Selektoren in einem kombinierten Selektor zählt. Die Spezifität wird dabei in der Form „0-0-0" notiert. Die erste Zahl steht für die Anzahl der IDs in einem Selektor, die zweite Zahl steht für die Anzahl der Klassen und Pseudo-Klassen, und die letzte Zahl steht für die Anzahl der Tags und Pseudo-Elemente im Selektor. Kombinatoren wie der Descendant-Kombinator (*.parentClass .childClass*) oder wie die Next-Sibling-Combinators (*.previousElement > .nextElement*) werden bei der Berechnung der Spezifität nicht berücksichtigt.

Im Beispiel aus *Listing 5* konkurrieren vier Selektoren aus den Autor-Stylesheets miteinander. Bei der Berechnung der Spezifität ergeben sich folgende Werte:

- form button#register`: 1 ID-Selektor (#register) und 2 Tag-Selektoren (form und button) ergeben eine Spezifität von „1-0-2".
- button: 1 Tag-Selektor (button) hat eine Spezifität von „0-0-1".
- form button: 2 Tag-Selektoren (form und button) ergeben eine Spezifität von „0-0-2".
- button.primary: 1 Klassen-Selektor (.primary) und 1 Tag-Selektor (button) ergeben eine Spezifität von „0-1-1".

Anhand der Spezifität bestimmt der Browser nun, welcher Style tatsächlich angewendet wird. Zuerst überprüft der Browser, welcher Selektor den höchsten Wert für die ID-Selektoren hat. Wenn es ei-

```
/* aus dem Author-Stylesheet*/
form button#register { /* Spezifität: 1-0-2*/
  background-color: darkred;
}

button.primary { /* Spezifität: 0-1-1*/
  background-color: #3da35d;
}

form button { /* Spezifität: 0-0-2*/
  background-color: pink;
}


button { /* Spezifität: 0-0-1*/
  background-color: white;
}


/* aus dem User-Stylesheet*/
button {
  background-color: orange;
}

/* aus dem User-Agent-Stylesheet*/
button {
  background-color: buttonface;
}
```

Listing 7: Auswirkung der Spezifität

nen Gleichstand gibt, wird nach der Anzahl der Klassen und Pseudo-Klassen geschaut. Wenn es auch hier zu einem Gleichstand kommt, entscheidet die Anzahl der Tags und Pseudo-Elemente. In jedem Fall gewinnt der Selektor mit der höchsten Spezifität.

Im Beispiel gewinnt die Hintergrundfarbe mit dem Selektor form button#register. Dieser Selektor hat mit „1-0-2" die höchste Spezifität, da er eine ID beinhaltet. Nach der Anwendung der Spezifitätsregeln ergibt sich die Reihenfolge, wie sie in *Listing 7* dargestellt ist.

4. Reihenfolge in CSS

Es kann vorkommen, dass trotz der Anwendung aller Schritte immer noch mehrere Selektoren um die Priorität konkurrieren. In solchen Fällen wird der zuletzt definierte Style im CSS immer bevorzugt.

In *Listing 8* (nicht Teil der Kurs-Seite) sind zwei Selektoren zu sehen, die beide die Hintergrundfarbe eines Buttons festlegen. Beide kommen aus dem Autor-Stylesheet und haben die gleiche Spezifität. In diesem Fall wird der zuletzt definierte Selektor genommen, der dem Button eine grüne Hintergrundfarbe zuweist.

```
button { /* Spezifität: 0-0-1*/
  background-color: white; // Wird später überschrie-
ben
}

button { /* Spezifität: 0-0-1*/
  background-color: green;
}
```

Listing 8: Im Zweifel gilt die letzte Anweisung.

5. Andere Features, die die Kaskade beeinflussen

Es gibt einen weiteren Schritt in der CSS-Kaskade, der als *Scoping Proximity* bekannt ist. Dieses relativ neue Feature wird noch nicht von allen Browsern vollständig unterstützt *[6]*. Um den Rahmen dieses Artikels nicht zu sprengen, werde ich hier nur eine kurze Erklärung geben.

Durch das *Scoping (@scope)* *[7]* können CSS-Regeln definiert werden, die nur auf bestimmte Teile des HTML-Dokuments angewendet werden sollen. Dabei bestimmt der *Scoping Proximity*, welche Scopes schließlich in der CSS-Kaskade angewandt werden.

Ein weiteres neues Feature, das die CSS-Kaskade beeinflusst, sind CSS-Layers *[8]*. Mit CSS-Layers können Entwickler CSS-Styles so überschreiben, wie es mit Autor-, Benutzer- und Benutzer-Agent-Stylesheets möglich ist. Dabei ist kein spezifischer Selektor erforderlich, da jeder Style einen vorherigen Layer überschreibt. Durch die Verwendung von CSS-Layern wird ein effektives und unkompliziertes Überschreiben von Stilen ermöglicht, insbesondere in Bezug auf das Überschreiben von Framework-Stilen.

Muss ich die Kaskade immer im Kopf berechnen?

Nein, das ist in der Praxis nicht nötig. Mit einer gut strukturierten CSS-Datei lassen sich viele Probleme und Konflikte in der Kaskade vermeiden. Ein durchdachter Aufbau hilft, die Übersicht zu behalten und sorgt dafür, dass Stile leicht wartbar und erweiterbar bleiben. Ein typischer Aufbau einer CSS-Datei könnte folgendermaßen aussehen:

1. **CSS Reset oder Normalize:** Um Browser-Styles zu vereinheitlichen, werden häufig **CSS Resets** oder Normalize-Stylesheets verwendet. Ein CSS Reset entfernt alle Standardstile der Browser, während ein **Normalize** diese harmonisiert und optimiert, anstatt sie vollständig zurückzusetzen. Es gibt viele bewährte, fertige Lösungen, wie modern-normalize oder reset.css, die direkt verwendet oder angepasst werden können.
2. **Globale Styles:** Globale Styles definieren allgemeine Regeln, die auf der gesamten Webseite gelten, zum Beispiel Schriftarten, Farben oder grundlegende Layout-Einstellungen. Typische globale Elemente sind *body*, *html*, Überschriften (*h1-h6*) und Links (*a*).
3. **Helferklassen (Utility Classes):** Helferklassen sind kleine, wiederverwendbare CSS-Regeln, die spezifische Aufgaben erfüllen, wie *.text-center*, *.hidden* oder *.margin-top-small*. Sie ermöglichen es, Styles schnell und konsistent anzuwenden, ohne neue Selektoren schreiben zu müssen.
4. **Komponenten und Layouts (zum Beispiel mit BEM):** Für spezifische Bereiche der Webseite wie Navigationen, Buttons oder Karten, bietet sich ein strukturierter Ansatz wie **BEM** (*Block, Element, Modifier*) an.

Was ist BEM?

BEM *[9]* steht für Block, Element, Modifier und ist eine Methodik zur Benennung von CSS-Klassen, die Konflikte in der Kaskade reduziert und den Code wartbar hält:

- Block: eine eigenständige Komponente, etwa *.button*
- Element: ein Bestandteil des Blocks, etwa *.button__icon*
- Modifier: eine Variante oder ein Zustand, etwa *.button--disabled*

Ein Beispiel ist in *Listing 9* aufgeführt.

```
<button class="button button--primary">
  <span class="button__icon"></span>
  Anmelden
</button>

.button { background-color: #3da35d; color: white; }
.button--primary { background-color: darkblue; }
.button__icon { margin-right: 0.5rem; }
```

Listing 9: Ein Beispiel für die Verwendung von BEM in HTML und CSS.

Layouts mit CSS

Das Thema CSS-Layout umfasst wichtige Konzepte, darunter das CSS-Box-Model, den Normalflow, Flex-Layout und Grid-Layout. In diesem Artikel werden diese Konzepte ausführlich erklärt und ihre Anwendung zur Gestaltung und Positionierung von Elementen auf einer Website erläutert.

Box-Modell

CSS verwendet das Konzept der Boxen *[10]*, um jedes HTML-Element in einem Dokument zu repräsentieren. *Abbildung 4* veranschaulicht dieses Prinzip, indem ein roter Rahmen um jedes HTML-Element gezogen wird, um die Box darzustellen. Es ist möglich, Boxen wie HTML-Elemente ineinander zu verschachteln.

Jede Box besteht aus mehreren Unterboxen, die Content-Box,

Abbildung 4: Hervorhebung der Boxen in der Beispielanwendung (© Christina Zenzes)

Padding-Box, Border-Box und Margin-Box genannt werden. Diese Unterboxen sind ebenfalls ineinander verschachtelt. Die Verschachtelung der Boxen wird in *Abbildung 5* dargestellt.

Die verschiedenen Unterboxen erfüllen jeweils verschiedene Aufgaben:

- Die Content-Box stellt den eigentlichen Inhalt des HTML-Elements ohne jegliche Abstände dar.
- Die Padding-Box bestimmt den Innenabstand der Box und definiert somit den Abstand zwischen der Content-Box und der Border-Box.
- Die Border-Box legt den Rahmen der Box fest.
- Die Margin-Box bestimmt den Außenabstand des Elements von der Border-Box zu anderen HTML-Elementen.

Das Box-Modell kann auch auf den „Zurück"-Button der Kurs-Seite angewandt werden, wie in *Abbildung 6* gezeigt wird. In diesem Beispiel enthält die Content-Box lediglich den Text „Zurück". Der Abstand zwischen dem Text und dem schwarzen Rahmen wird durch die Padding-Box definiert. Die Border-Box wird durch den schwarzen Rahmen repräsentiert, und der Abstand zwischen dem Button und dem „Anmelden"-Button wird durch die Margin-Box festgelegt.

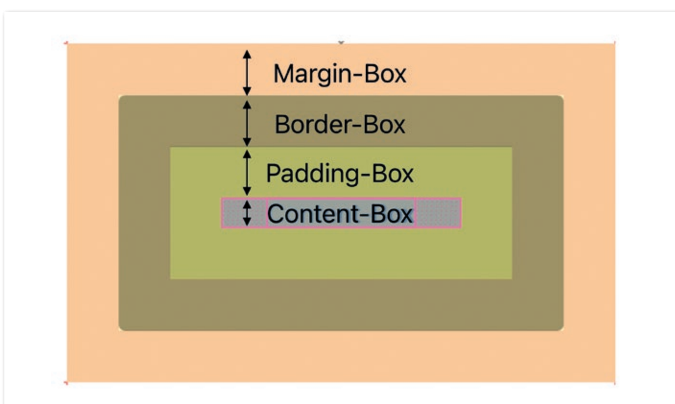


Abbildung 5: CSS-Box-Modell (© Christina Zenzes)

Es ist möglich, dass die Werte der Boxen auch 0 betragen können. Zum Beispiel hat der Anmelde-Button keine Border-Box mit einer Breite von 0.

Die verschiedenen Boxen können über CSS-Eigenschaften, wie beispielsweise *padding*, *margin* und *border*, gesteuert werden.

Normal-Flow

Der Normal-Flow (Normalfluss) *[11]* regelt die Anordnung von Elementen, wenn keine anderen Layout-Mechanismen wie Grid oder Flex ausgewählt wurden. In *Abbildung 7* wird die Kursseite im Normal-Flow dargestellt – ohne die Verwendung von Flex oder Grid-Layout.

Der Display-Typ eines Elements hat einen großen Einfluss auf die Anordnung der Elemente im Normal-Flow. Abhängig vom Display-Typ wird ein Element im Browser unterschiedlich dargestellt. Es gibt zwei grundlegende Typen, nämlich Block-Elemente und Inline-Elemente.

Die zentrale Eigenschaft eines Block-Elements besteht darin, dass es stets in einer neuen Zeile umgebrochen wird. Zudem berücksichtigen Block-Elemente die Werte der CSS-Eigenschaften *width*, *height*, *padding*, *margin* und *border*. Wenn für die *width* kein Wert festgelegt wurde, nimmt das Element automatisch die volle Breite ein.

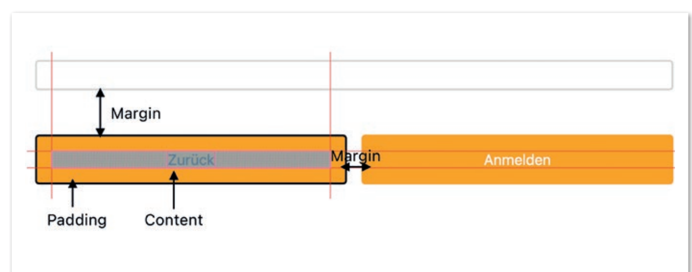


Abbildung 6: Das Box Modell am Beispiel des Zurück-Buttons (© Christina Zenzes)

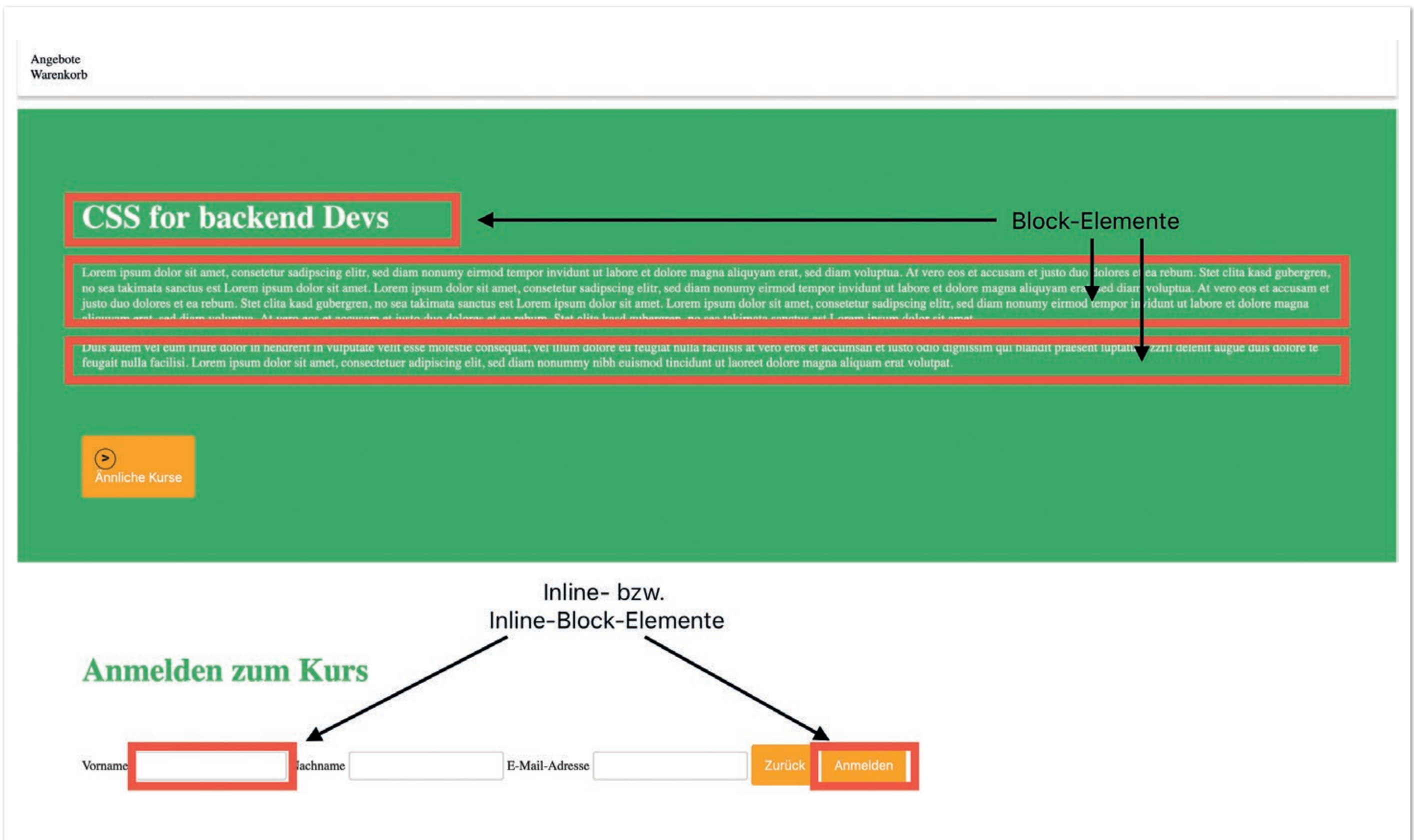


Abbildung 7: Die Anwendung im Normal-Flow (© Christina Zenzes)

Inline-Elemente hingegen werden nicht in eine neue Zeile umgebrochen, sondern reihen sich einfach neben dem vorherigen Element ein. Dabei werden *width*- und *weight*-Angaben ignoriert. Die *top*- und *bottom*-Werte von *padding* und *margin* verhalten sich anders. Im Gegensatz zu Block-Elementen beeinflussen sie nicht die *position* benachbarter Elemente.

Wie auf der Kursseite ohne Layouts zu sehen ist, werden die Überschriften und Texte auf dem grünen Hintergrund immer in eine neue Zeile umgebrochen. Dieses Umbruchverhalten entsteht, da Überschriften-Tags (*h1* bis *h5*) und Paragraphen standardmäßig den *display*-Typ *block* haben. Im Gegensatz dazu stehen die Eingabefelder und Buttons nebeneinander, da sie den *display*-Typ *inline* beziehungsweise *inline-block* haben.

inline-block ist eine Kombination aus den Eigenschaften von Block- und Inline-Elementen. In diesem Fall verhält sich das Element wie ein Block-Element, bricht jedoch nicht in eine neue Zeile um.

Der *display*-Typ eines Elements kann im CSS durch die Verwendung der Eigenschaft „Display" geändert werden. Diese Eigenschaft bietet verschiedene Werte an, darunter „*inline*" für Inline-Elemente, „*block*" für Block-Elemente und „*inline-block*" für Inline-Block-Elemente.

Flex-Layout

Das Flex-Layout *[12]* ermöglicht es, Elemente in einem eindimensionalen Layout anzuzeigen, wobei sie entlang einer Achse platziert werden. Standardmäßig werden die Elemente entlang der horizontalen Achse angeordnet, aber es ist auch möglich, sie entlang der vertikalen Achse anzuordnen.

In *Abbildung 8* sind verschiedene Beispiele für das Flex-Layout zu sehen. Drei dieser Beispiele zeigen Elemente, die entlang der vertikalen Achse platziert sind. Dabei kann die genaue Ausrichtung (zentriert, linksbündig oder rechtsbündig) ausgewählt werden. Die anderen drei Beispiele zeigen Elemente, die entlang der horizontalen Achse platziert sind.

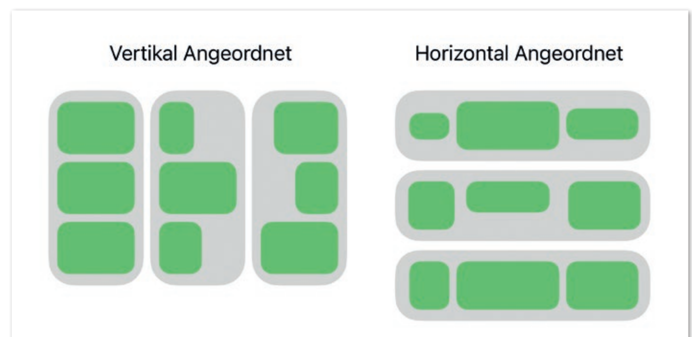


Abbildung 8: Beispiele für das Flex-Layout (© Christina Zenzes)

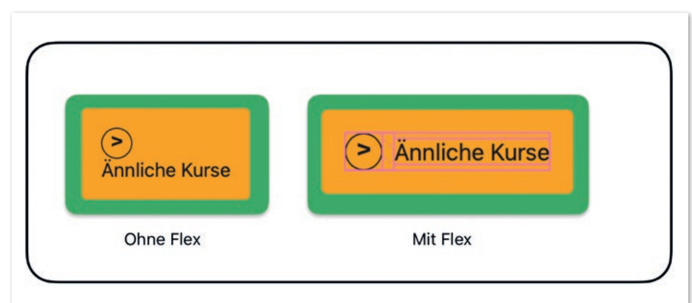


Abbildung 9: Ein Button mit und ohne Flex-Layout (© Christina Zenzes)

```
button {
  display: flex; /* Schaltet Flex-Layout ein*/
  gap: 0.5rem; /*definiert die Abstände zwischen Spalten und Zeile*/
  align-items: center; /*zentriert Element entlang der y-Achse*/
}
```

Listing 10: Ein Button mit Flex Layout

Auf der Kursseite wurde das Flex-Layout mehrfach angewendet. Zum Beispiel wurde es beim „Ähnliche Kurse"-Button verwendet, um Text und Icon vertikal zu zentrieren *(siehe Abbildung 9)*. Dabei wurde die Hauptachse horizontal angeordnet und die Elemente auf der Nebenachse, der vertikalen Achse, zentriert.

Dieses Ergebnis kann auch mit anderen Methoden erreicht werden, wie der Verwendung von Inline-Elementen und der Definition passender Paddings. Diese Methode ist jedoch oft komplexer und anfälliger für Fehler. Daher empfiehlt es sich in solchen Fällen, das Flex-Layout zu verwenden.

Das Flex-Layout kann mit der CSS-Eigenschaft *display:flex* für ein Element aktiviert werden. Dadurch werden alle Elemente automatisch in einer Zeile angeordnet. Wenn die Anordnung geändert werden soll, kann die Eigenschaft *flex-direction* auf *column* gesetzt werden, um die Elemente entlang der vertikalen Achse anzuordnen.

Mit der *gap*-Eigenschaft können Abstände zwischen den einzelnen Elementen festgelegt werden. Die Ausrichtung der Elemente entlang der Hauptachse (bei *flex-direction: row* die horizontale Achse) kann mit der Eigenschaft *justify-content* definiert werden, um sie linksbündig, rechtsbündig oder zentriert anzuordnen.

Die Ausrichtung der Elemente entlang der anderen Achse kann durch die Eigenschaft *align-items* umgesetzt werden.

In *Listing 10* wird das CSS für das Flex-Layout des Buttons gezeigt. Hier werden die Elemente einfach mit einem Abstand nebeneinander angezeigt, und mit *align-items* werden die Elemente vertikal zentriert.

Grid-Layout

Das Grid-Layout *[13]* legt fest, wie Elemente in einer zweidimensionalen Anordnung platziert werden. Ein solches Layout ermöglicht die Platzierung von Elementen entlang der vertikalen und horizontalen Achse.

In *Abbildung 10* ist ein einfaches Grid-Layout dargestellt. Beim Platzieren der Elemente entlang der vertikalen Achse dienen die Spalten als Referenzpunkte im Grid, ähnlich wie bei einer Tabelle. Es werden jedoch nicht die tatsächlichen Spalten gezählt, sondern die Linien, die eine Spalte umgeben.

Wenn ein Element wie das Label *Nachname* in die zweite Spalte platziert werden soll, befindet es sich zwischen Linie 2 und 3. Dabei ist zu beachten, dass die Zählung bei eins beginnt und es keine Null-Linie im Grid gibt. Es ist möglich, Elemente über mehrere Spalten oder Zeilen anzuordnen. Beispielsweise ist das E-Mail-Eingabefeld in der Abbildung zwischen Linie 1 und 3 platziert.

Abbildung 10: Das Formular im Grid-Layout (© Christina Zenzes)

Um das Grid-Layout für ein Element zu aktivieren, kann die CSS-Eigenschaft *display: grid* verwendet werden. Dadurch werden alle Elemente automatisch zuerst in Spalten (Columns) und dann in Zeilen (Rows) angeordnet. Falls die automatische Anordnung geändert werden soll, kann die Eigenschaft *flex-auto-flow* auf *column*, *row* oder *dense* gesetzt werden.

Durch die Verwendung der CSS-Eigenschaft *grid-template-columns* kann festgelegt werden, wie viele Spalten in einem Grid-Layout vorkommen sollen. Gleiches gilt für die Eigenschaft *grid-template-rows*, um die Anzahl der Zeilen festzulegen.

Wenn keine der beiden Eigenschaften gesetzt wurde, berechnet das Grid die Anzahl der Spalten und Zeilen basierend auf den enthaltenen Elementen.

Die Ausrichtung der Elemente und der Abstand zwischen ihnen können, ähnlich wie bei Flexbox, mithilfe der Eigenschaften *justify-*

```
form {
  display: grid;
  grid-template-columns: 1fr 1fr;
  gap: 1rem;
  grid-auto-flow: dense;
}

#vorname,
label:has(+ #vorname) {
  grid-column: 1 / 2;
}

#nachname,
label:has(+ #nachname) {
  grid-column: 2 / 3;
}
#email,
label:has(+ #email) {
  grid-column: 1 / 3;
}
```

Listing 11: Ein Button mit Flex-Layout

content, *align-items* und *gap* festgelegt werden.

Falls Elemente außerhalb der automatischen Platzierung positioniert werden sollen, können die Eigenschaften *grid-column* oder *grid-row* verwendet werden.

In *Listing 11* wird das CSS für das Grid-Layout des Formulars dargestellt. Dabei werden die einzelnen Elemente über die Eigenschaft *grid-column* direkt in die entsprechenden Spalten platziert.

Fazit

In diesem Artikel haben wir drei zentrale Konzepte von CSS erkundet, die eine solide Grundlage für das Arbeiten mit Stylesheets bilden. Wir haben gesehen, wie **Inheritance** (die Vererbung von Eigenschaften) das Styling beeinflusst und wie die **CSS-Kaskade** hilft, konkurrierende Regeln aufzulösen, damit der gewünschte Style zuverlässig angewendet wird.

Beim **Layouting** haben wir uns das **Box-Modell** und den **Normal-Flow** angesehen – die Basis, auf der Browser Elemente standardmäßig anordnen. Zusätzlich gab es einen Einblick in **Flexbox** und **Grid**, zwei leistungsstarke Layout-Methoden, die Ihnen mehr Kontrolle über die Gestaltung Ihrer Seiten geben.

Natürlich kratzt dieser Artikel nur an der Oberfläche dessen, was mit CSS möglich ist. Themen wie **Responsive Design** oder das Entwickeln nach dem Prinzip **Mobile First** bauen auf diesen Grundlagen auf und sind spannende nächste Schritte. Mit dem Wissen aus diesem Artikel können Sie nicht nur tiefer in diese fortgeschrittenen Themen einsteigen, sondern auch die Zusammenarbeit mit den Frontend-Entwicklern im Team verbessern und sie gezielt unterstützen.

Vielleicht haben Sie sogar festgestellt, dass CSS mit einem besseren Verständnis der Prinzipien weniger frustrierend sein kann und Ihnen neue Möglichkeiten eröffnet, kreativ und effektiv an Projekten mitzuwirken. Viel Spaß mit diesem neuen Werkzeug.

Quellen

[1] *https://developer.mozilla.org/en-US/docs/Web/CSS/Inheritance*
[2] *https://developer.mozilla.org/en-US/docs/Web/CSS/system-color*
[3] *https://developer.mozilla.org/en-US/docs/Web/CSS/Cascade*
[4] *https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Selectors*
[5] *https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity*
[6] *https://caniuse.com/css-cascade-scope*
[7] *https://developer.mozilla.org/en-US/docs/Web/CSS/@scope*
[8] *https://developer.mozilla.org/en-US/docs/Web/CSS/@layer*
[9] *https://getbem.com/*
[10] *https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/The_box_model*
[11] *https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Normal_Flow*
[12] *https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_flexible_box_layout/Basic_concepts_of_flexbox*
[13] *https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_grid_layout*

Christina Zenzes

christina.zenzes@codecentric.de

Christina Zenzes ist Senior Software Developer bei der codecentric AG. Ihr Schwerpunkt liegt auf der Frontend-Entwicklung und hier insbesondere auf der testgetriebenen Entwicklung. Dabei legt sie großen Wert auf Qualität und Wartbarkeit sowie die effektive Gestaltung von Oberflächen mit CSS, um ansprechende und funktionale Benutzererlebnisse zu schaffen.

Falsch abgezweigt? Was beim Branching wirklich wichtig ist

Georg Berky, Bryter GmbH

Unsere Industrie liebt falsche Dichotomien. Feature Branches oder Trunk-based Development sind zwei davon, die immer wieder ausführlich und kontextfrei diskutiert werden. In diesem Artikel soll es um mehr Freiheit gehen – sowohl um die Freiheit, häufig zu liefern, als auch um die Freiheit statt einer festgefahrenen Entscheidung eine der Alternativen zu wählen.

Ich habe mich neulich dabei erwischt, dass ich einen Branch noch nicht liefern wollte, weil sein Name „versprochen hat", dass er mehr Inhalt hat. Der Code war aber lieferbar und hätte dem Team und mir wertvolles Feedback gebracht, hätte ich auf den Merge-Button geklickt.

Das sollte anders sein, denn Branches sind eigentlich dazu da, uns Devs das Leben einfacher zu machen. Ich habe daraufhin therapeutisch angefangen, meine Branches „<Ticketnummer>-one", „<Ticketnummer>-two" und so weiter zu nennen, denn ich sollte entscheiden, ob ich liefern kann – nicht der Name meines Branches. Daher auch mein Toot auf Mastodon *[1] (siehe Abbildung 1)*.

Abbildung 1: Ich weigere mich, die Namen meiner Branches meine Lieferfähigkeit beeinträchtigen zu lassen.

Worum es nicht gehen soll

Dies soll keine religiöse Abhandlung über *Trunk-based Development* versus *Feature Branches* sein. Solche Diskussionen lenken uns von wichtigeren Themen ab:

Wenn ihr keine vertrauensvolle Beziehung zu euren Kollegen und Vorgesetzten habt, ist es egal, ob ihr *trunk-based* oder auf *Feature Branches* arbeitet oder ob ihr *Vim* oder *Emacs*, *Tabs* oder *Spaces* benutzt.

Wenn Deployments regelmäßig schief gehen, ist es schwierig, Zustimmung für noch größere Umbauten zu bekommen. Wenn im ganzen Team nur Einzelkämpfer arbeiten, ist es schwierig, die Codebase in einen Zustand zu bringen, der Deployments, Refactorings und alle anderen Umbauten weniger fehleranfällig macht, weil diese Umbauten für einzelne Teammitglieder zu komplex oder zu risikobehaftet sind.

Wenn es euch so vorkommt, als wären *Tabs* und *Spaces* die einzigen Entscheidungsfreiheiten, die ihr in eurer Organisation habt und die-

se deshalb ausgiebig diskutiert werden, habt ihr ein viel größeres Problem als *Tabs* oder *Spaces*.

Feature Branches

Unter *Feature Branches* verstehe ich in diesem Artikel *Branches*, deren *Merge* nach *Mainline* („main", „master", „trunk") ganz oder gar nicht erfolgt. Das ganze Feature wird auf einem Branch entwickelt. Am Ende der Entwicklung werden die Änderungen in *Mainline* integriert.

Diese Art zu entwickeln, wird oft im Open-Source-Umfeld eingesetzt, wo die Maintainer der Projekte und die Autoren der *Branches* sich nicht persönlich kennen. Alle Änderungen auf einem Branch zu sehen, gibt dem Maintainer mehr Sicherheit beim Erstellen der Review und macht es einfacher, das gesamte Feature als Einheit abzulehnen, oder es in die Codebase zu integrieren.

Integration

Integration ist, wenn Änderungen am Code auf die Realität treffen. Code wird in die lokale Entwicklungsumgebung integriert, sobald er geschrieben und vom Compiler verarbeitet wird. Danach wird der Produktionscode mit dem Testcode, der Testcode mit dem lokalen Build, das gebaute Artefakt mit der Deployment-Umgebung, die Deployment-Umgebung mit den Akzeptanztests, deren Clients mit unserem Server-Code und am Ende das laufende System mit den Anforderungen der Kunden integriert.

All diesen Schritten ist gemein, dass Schritt für Schritt zusätzliche Dimensionen der Interaktionen mit unserem laufenden Code stattfinden, von deren Ausgang wir Feedback bekommen: Entsprechen die Änderungen am Code dem, was von den Kunden oder anderen Stakeholdern gewünscht wurde, und läuft der Rest der Funktionalität dabei noch wie gehabt?

Branches müssen sich in dieses System des sich schrittweise erweiternden Feedbacks einfügen und tragen dazu bei, wie schnell und in welcher Qualität die Rückmeldung bei den Devs ankommt. Sie haben deswegen einen wichtigen Einfluss auf das Vertrauen in Code und das gebaute Artefakt.

Vertrauen in Branches

Um vertrauensbildend mit *Branches* zu arbeiten, muss man ihre Risiken kennen.

Ich sage in Kurzform gerne, dass *Branches* gefährlicher werden, je länger sie leben. Bei detaillierter Betrachtung ist es jedoch nicht allein die Länge eines *Branches*, sprich die Anzahl der Commits, die ihn gefährlich macht. Im konstruiert einfachsten Fall beträfen alle Commits eines *Branches* eine einzige Zeile einer einzigen Datei der Codebase. Ein solcher *Branch* ist nur dann gefährlich, wenn er einen *Merge Conflict* mit anderen *Branches* auslöst, der dieselbe Zeile derselben Datei betrifft oder die Änderung der einen Zeile einen logischen Fehler im Zusammenspiel mit einem anderen *Branch* auslöst. Eine dritte Möglichkeit wäre das Ändern des Konfigurations-Codes.

Deshalb ist nicht allein die „Länge", sondern auch die „Breite" eines *Branch* gefährlich, also die Anzahl der geänderten Stellen im Code, die er sich mit den anderen *Branches* der Codebase teilt. Da wir unsere Lieferartefakte von *Mainline* erstellen, kann ein *Branch* erst einen Wert erzeugen, wenn er durch einen Merge in *Mainline* integriert worden ist. Jeder *Branch* hat deswegen eine „integrative Distanz" zu *Mainline*, die durch den Merge geschlossen wird. Diese integrative Distanz besteht auch zu allen anderen gleichzeitig aktiven *Branches*, weil unser *Branch* nach dem Merge den Code und das Verhalten direkt oder indirekt beeinflussen kann.

Mit integrativer Distanz umgehen

Das Risiko, das von integrativer Distanz ausgeht, kann nur durch Merges behoben werden, weil wir erst dann Feedback bekommen, ob sich Änderungen auf verschiedenen *Branches* gegenseitig Probleme bereiten. Die Zahl der gleichzeitig aktiven *Branches* erhöht das Risiko, weil es bei `n` *Branches* `n!` mögliche Interaktionen zwischen *Branches* gibt. Bei fünf Einzelkämpfer-Devs mit je einem aktiven *Branch* sind es 120. Das Risiko kann auf zwei Arten minimiert werden: häufige Merges und gutes Design.

Häufige Merges werden möglich, wenn man Code zwar liefert, aber noch nicht für den Benutzer aktiviert. Der neue Code kann beispielsweise erst erreichbar sein, wenn ein *Feature Flag* aktiviert wird, das zunächst nur in Entwicklungs- und Testumgebungen sowie in den Tests des Builds umgelegt wird. Dies hat den Vorteil, dass wir schon Feedback bekommen, ob der Code im Zusammenspiel mit anderen Änderungen Probleme hervorruft: die integrative Distanz schrumpft.

Das spätere Entfernen von *Feature Flags* muss vom Team eingeplant werden, denn Feature Flags sind nichts anderes als höher integrierte Branches und können durch ungewollte Interaktion untereinander zu Problemen führen, wenn sie zu lange im System bleiben. Das Risiko ist im Vergleich zu Branches – vor allem bei einer kleinen Anzahl von Feature Flags – aber niedriger und man gewinnt durch sie schnelles Feedback.

Eine den Feature Flags ähnliche Strategie im Backend ist, die HTTP-Schicht des Features zuletzt anzubinden. Benötigen die Kollegen im Frontend schnelles Feedback zur Integration, einigt man sich am Anfang auf eine HTTP-Schnittstelle, Pfade und auf das Format der Payloads und liefert zunächst leere oder Dummy-Daten. Dadurch bekommt man gleich am Anfang Feedback, ob die Netzwerkanbindung funktioniert, ob die Payloads vom Frontend de-serialisiert werden können, ob Authentifizierung und Autorisierung richtig konfiguriert wurden und ob das Frontend die Payloads vom Backend verarbeiten kann.

Es besteht immer noch eine integrative Lücke bezüglich der Logik, aber die anderen Dimensionen der Integration wie Authentifizierung, Autorisierung und so weiter können früh geprüft und behandelt werden. Ist das Team nicht cross-funktional aufgestellt, wiegen diese Dimensionen oftmals schwerer, weil über Abteilungs- oder Teamgrenzen hinweg gearbeitet werden muss und die Behebung der Fehler so mehr Zeit beansprucht.

In der Backend-Entwicklung ist man dann mit einem solchen Ansatz frei, das Feature weiter in Teile zu zerlegen, die man in parallelen Arbeitsschritten implementieren kann. Hierbei kommen *Design Patterns* ins Spiel, die es ermöglichen, die Teile als „Plug-in" ins gesamte Feature zu entwickeln. Ein einfaches Beispiel wäre die Validierung von Eingabedaten aus dem Frontend: Statt den gesamten Input in einem Schritt zu validieren, gibt man ihn schrittweise an jeden registrierten *Validator*, der seinen Teil der Validierungslogik auf den Input

anwendet und auf eine vorher definierte Weise Fehler liefert.

Mit Spring oder anderen Frameworks zur *Dependency Injection* kann man sich alle *Beans*, die dasselbe Interface implementieren, in eine Liste dieses Typs injizieren lassen. Der „äußere" Teil der Validierungslogik besteht dann aus einer Schleife über der Liste, dem Aufruf der einzelnen Validatoren mit den Eingabedaten (oder einem Teil davon) und dem Aufsammeln und Kombinieren der Teilergebnisse zu einem Gesamtergebnis. Für den äußeren Teil bietet es sich an, im Team- oder Pair-Programming zu arbeiten, damit alle Betroffenen das „Framework" kennen, in das sie die Validatoren einbinden müssen. Die Validatoren selbst können dann je nach Wunsch allein oder auch im Pair entwickelt werden.

Auch im Legacy-Code lassen sich Änderungen liefern, ohne sie sofort aktivieren zu müssen. Mit *Sprouts* [2] entwickelt man das gesamte Feature getrennt von der Codebase und unter Unit-Tests. An einem *Seam* [3] lässt sich dann der so entwickelte Code ins Gesamtsystem integrieren. Details finden sich in „Working Effectively with Legacy Code" von Michael Feathers [4] – ein Buch, das auch nach 20 Jahren nichts von seiner Relevanz verloren hat.

Kontinuierlich vorwärts

Wenn ihr euch die Zeit nehmt, diese Branch- und Programmierweise auszuprobieren und schrittweise eure Pipeline optimiert, werdet ihr merken, dass durch schnelles Feedback die Angst vor dem Liefern weniger wird. Ein Merge kann damit zu einem No-Event werden, das stattfindet, wenn es gerade passt – auch mal zur Mittags- oder Kaffeepause.

Landet doch einmal ein Bug in der Produktivumgebung, ist das eine Gelegenheit, zu identifizieren, wo dieser in der Pipeline am schnellsten hätte gefunden werden können. Je schneller das Feedback bei euch ankommt, desto weniger Zeit kostet die Behebung, desto stressfreier wird euer Alltag und desto besser könnt ihr euch auf die Features konzentrieren.

Stellt euch jetzt vor, dass ihr dann nach einiger Zeit bei dieser sicheren Programmierweise, einem zuverlässigen Sicherheitsnetz durch Tests, einer stabilen Pipeline und Deployment bis ins Produktionssystem angekommen seid. Jeder Commit auf *Mainline* baut ein Artefakt, das nach Durchlaufen aller Prüfungen in der Produktion ankommt. In diesem Zustand ist der zusätzliche Build auf den *Branches* euer neues Bottleneck und ihr könnt euch mit *Trunk-based* Development diese doppelten Builds sparen. Wäre das mit dieser Pipeline und Arbeitsweise noch furchteinflößend?

Es geht für mich deswegen nicht um *Feature Branches* oder *Trunk-based Development*, sondern um die passende Arbeitsweise. Optimiert diese darauf, dass sie Vertrauen durch schnelles Feedback schafft.

Ich wünsche euch viel entspanntes Entwickeln durch häufige Merges und schnelles Feedback.

Quellen

[1] *https://chaos.social/@georgberky/112927137743841758*

[2] *https://understandlegacycode.com/blog/key-points-of-working-effectively-with-legacy-code/#1-the-sprout-technique*

[3] *https://understandlegacycode.com/blog/key-points-of-working-effectively-with-legacy-code/#identify-seams-to-break-your-code-dependencies*

[4] *Feathers, Michael C.: Working Effectively with Legacy Code, Pearson Education, EAN: 9780132931755*

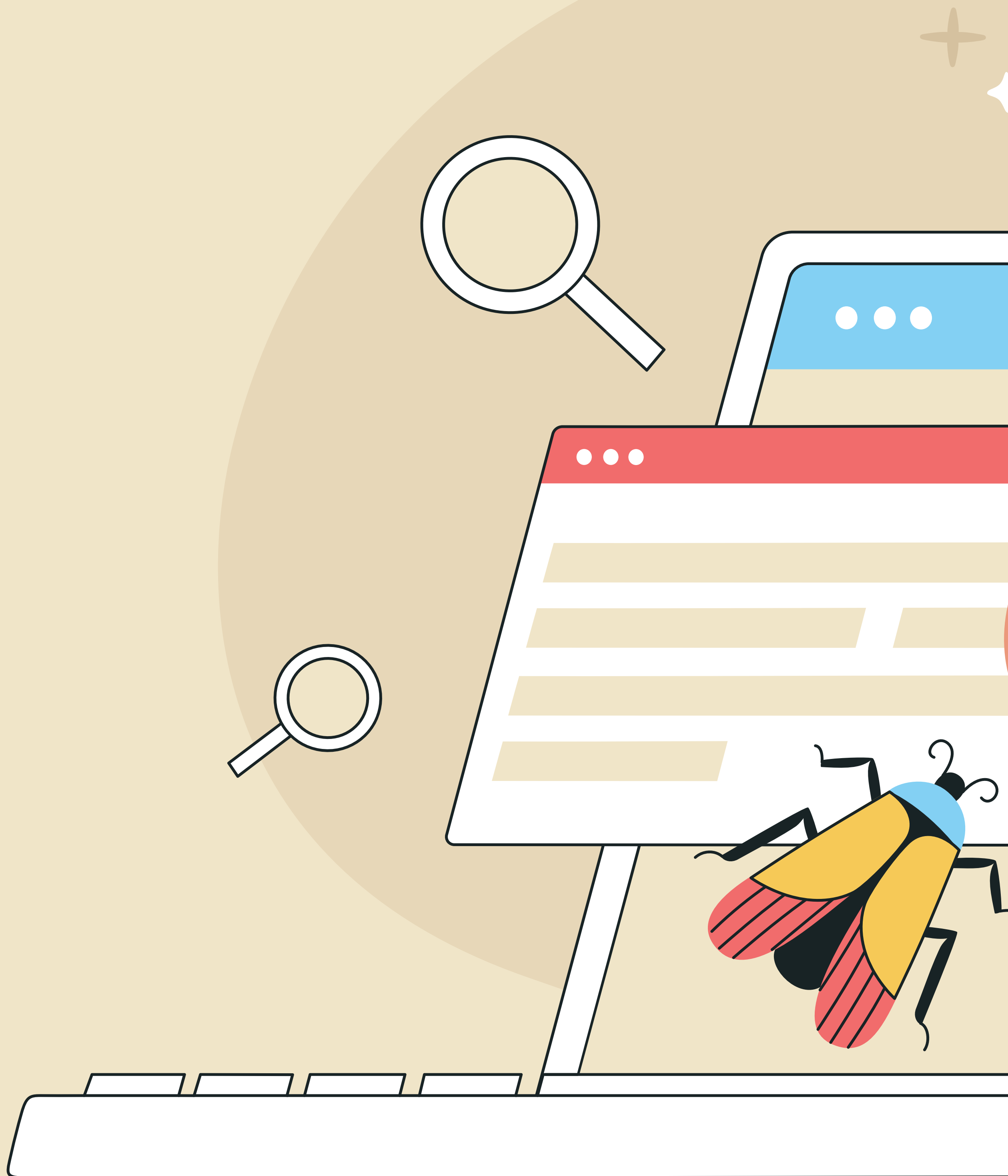
Georg Berky

hallo@berky.dev

Georgs Handwerk und Leidenschaft ist die Programmierung, meistens in JVM-Sprachen wie Java, Groovy, Kotlin oder Clojure. Zum Handwerk gehören für ihn auch Themen wie die Pflege von Legacy-Code, Automatisierung von Builds und Deployments oder Agilität im Team. Seit einigen Jahren ist er in der Software-Crafting-Community aktiv und Co-Organisator der Softwerkskammer im Ruhrgebiet. Wenn er mal nicht programmiert, spielt er Trompete, pflegt seine Bonsai oder praktiziert Aikido.

Quality Metrics Unleashed: Softwarequalität im Griff mit Visualisierung und Alerting

Ildikó Tárkányi, QAware GmbH



Die Sicherstellung der Softwarequalität in Microservice-Architekturen stellt eine erhebliche Herausforderung dar: Bewährte Ansätze, die in monolithischen Architekturen gut funktionieren, skalieren oft nicht für komplexe Systeme mit zahlreichen, verteilten Komponenten und stoßen an ihre Grenzen. Durch eine gezielte Kombination von Open-Source-Tools lässt sich jedoch eine Lösung entwickeln, die den Überblick über viele verteilte Repositories und Komponenten ermöglicht.

Ich schlage Dashboards zur übergreifenden Visualisierung der Softwarequalität an einer zentralen Stelle für alle Komponenten in Kombination mit einer Alerting-Lösung vor, um sicherzustellen, dass die Softwarequalität ständig sichtbar und präsent bleibt. Dieser Ansatz berücksichtigt verschiedene Metriken von der Codequalität über technische und fachliche Schulden bis hin zur Einhaltung von Architekturvorgaben, und macht Softwarequalität – mit Visualisierung und Alerting – in komplexen Microservice-Architekturen beherrschbar.

In der heutigen, stark digitalisierten und vernetzten Welt ist Software von grundlegender Bedeutung für die meisten Unternehmen. Ob im E-Commerce, der Gesundheitsbranche, im Finanzwesen oder in der öffentlichen Verwaltung – leistungsfähige und zuverlässige Softwarelösungen sind unverzichtbar. Doch Qualität ist keine Selbstverständlichkeit und erfordert eine kontinuierliche Prüfung und Optimierung.

Warum ist die Sicherstellung der Qualität in verteilten Architekturen so schwierig?

In modernen Softwarearchitekturen sind verteilte Systeme mittlerweile weit verbreitet, zum Beispiel durch Microservice-Architekturen. Verteilte Architekturen ermöglichen es, Anwendungen in kleine, spezialisierte Services zu unterteilen, die unabhängig voneinander entwickelt, implementiert und eingesetzt werden können. Während dies einerseits die Flexibilität und Skalierbarkeit erheblich steigert, entstehen gleichzeitig neue Herausforderungen für die Qualitätssicherung.

In den früher verbreiteten monolithischen Architekturen waren Softwareanwendungen kompakte Systeme, bestehend aus eng miteinander verbundenen Komponenten. Diese zentrale Struktur ermöglichte eine gezielte Fokussierung der Qualitätssicherungsmaßnahmen auf ein einziges System. Die Entwicklung der Anwendungen erfolgte häufig in genau einem Repository, und ihr Betrieb fand in einer einzigen Deployment-Unit statt *(siehe Abbildung 1)*. Bei sämtlichen Fragen zu Qualitätsaspekten war klar und eindeutig, wo sich diese messen ließen. Mit dem Aufkommen von verteilten Systemen hat sich dieser Ansatz jedoch grundlegend verändert.

In verteilten Systemen erstreckt sich die Qualitätssicherung über eine gesamte Landschaft miteinander verbundener Services, nicht nur über eine einzelne Komponente. Es genügt nicht mehr, den Zustand einer Anwendung isoliert zu betrachten; vielmehr müssen alle Services, ihre Interaktionen und Abhängigkeiten unter die Lupe genommen werden.

Die folgende Abbildung zeigt eine beispielhafte Servicearchitektur, in der jeder einzelne Service als eigene Deployment-Unit organisiert ist *(siehe Abbildung 2)*. Für jeden Service sind separate Repositories und Pipelines eingerichtet, die eine unabhängige Entwicklung, Bereitstellung und Verwaltung der Services ermöglichen.

Da die einzelnen Microservices über Netzwerkkommunikation miteinander interagieren, steigt die Komplexität durch potenzielle Netzwerkfehler, Latenzen und die Notwendigkeit, Verfügbarkeit und Zuverlässigkeit über mehrere Services hinweg zu koordinieren.

Zudem müssen Qualitätssicherungsmaßnahmen so gestaltet werden, dass sie mit der Dynamik und Flexibilität einer Microservice-Architektur Schritt halten können. Die Unabhängigkeit der Services führt dazu, dass Änderungen, Aktualisierungen und Wartungen regelmäßig und oft parallel an verschiedenen Teilen der Anwendung vorgenommen werden. Dies erschwert die Identifikation und Behebung von Fehlern erheblich.

Was meine ich überhaupt, wenn ich von Qualität spreche?

Softwarequalität lässt sich in verschiedene Dimensionen unterteilen, die unterschiedliche Anforderungen und Schwerpunkte abde-

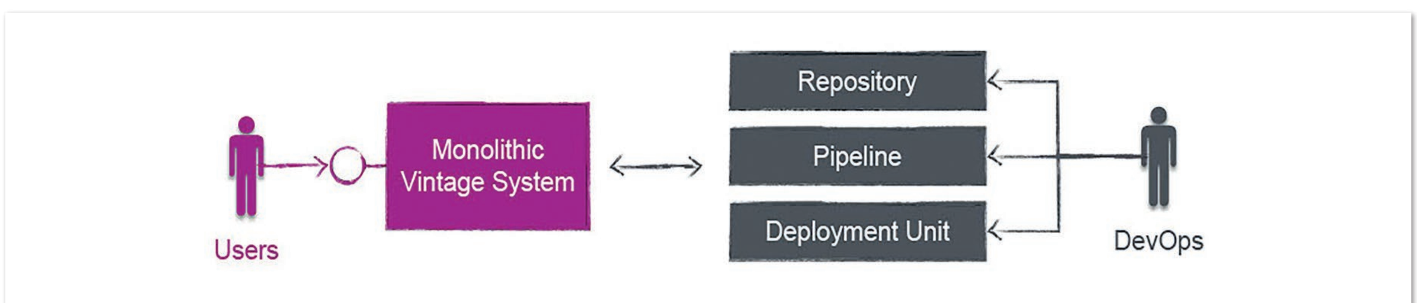


Abbildung 1: Monolithische Architektur (© Ildikó Tárkányi)

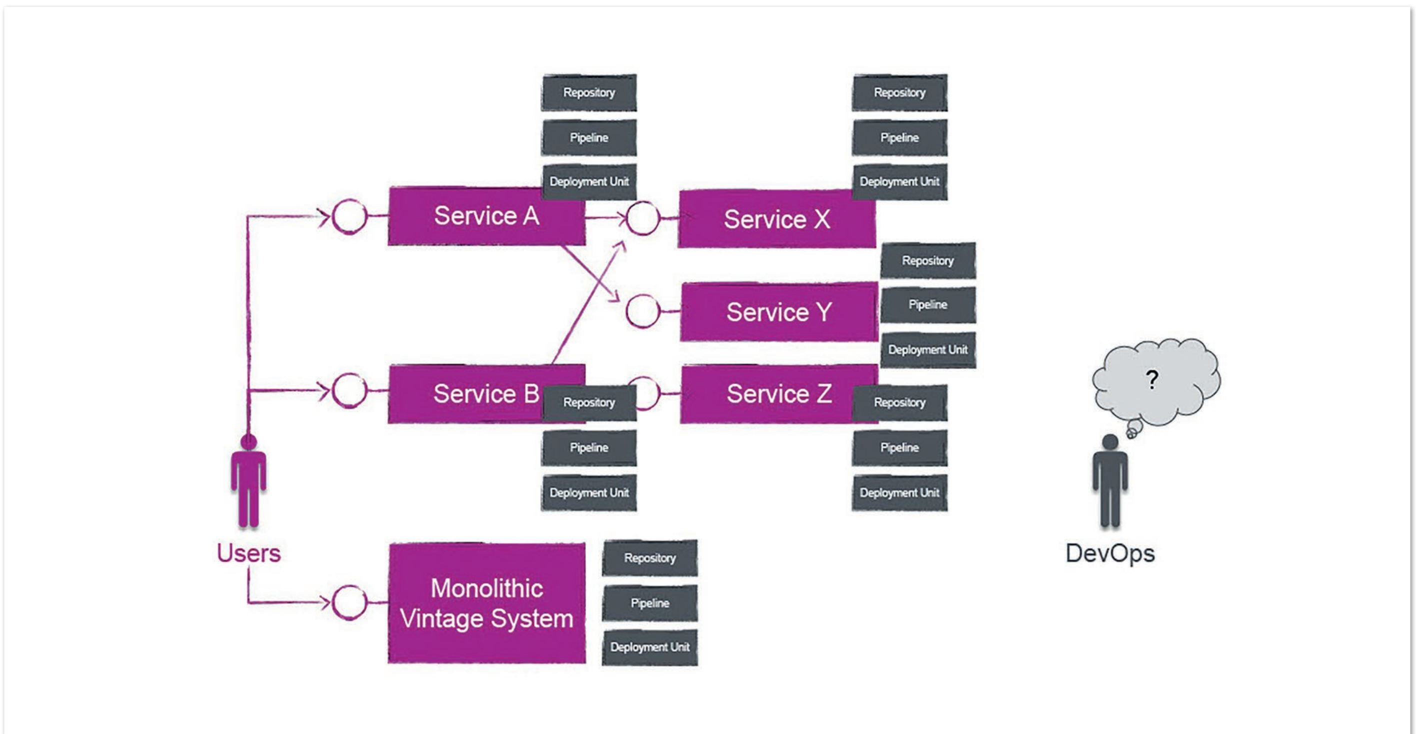


Abbildung 2: Verteilte Architektur (© Ildikó Tárkányi)

cken. Das arc42-Qualitätsmodell (Q42) *[1]* ist ein pragmatisches Modell, das acht konkrete Produkt- und Systemqualitätsaspekte ableitet, die ausreichen, um die meisten – wenn nicht sogar alle – der klassischen Qualitätsmerkmale abzudecken. Diese acht Aspekte sind *(siehe Abbildung 3)*:

1. Zuverlässigkeit
2. Flexibilität
3. Effizienz
4. Benutzerfreundlichkeit
5. Operabilität
6. Zweckmäßigkeit
7. Funktionssicherheit
8. Betriebssicherheit

Zu jedem dieser Aspekte lassen sich spezifische Metriken erfassen, die dabei helfen, technische und fachliche Anforderungen der Software objektiv zu bewerten. Jede Dimension von Produkt- und Systemqualität stellt spezifische Anforderungen an die Entwicklung und Implementierung der Software und erfordert somit auch individuelle Mess- und Überwachungsmethoden. Zum Beispiel:

- Flexibilität bedeutet, dass Änderungen und Erweiterungen einfach durchführbar sind, ohne den bestehenden Code erheblich zu beeinträchtigen. KPIs (Key Performance Indicators) wie der Aufwand für Änderungen (*Effort*) und die Anzahl neuer Fehler nach Anpassungen (*New Defects*) geben hier wertvolle Einsichten.
- Effizienz beschreibt die Performance der Software unter Berücksichtigung des Ressourcenverbrauchs (*Resource Utilizatio*n), etwa durch Antwort- und Ausführungszeiten (*Response and Run Time*).
- Benutzerfreundlichkeit betrifft die Frage, wie gut die Software die Bedürfnisse der Benutzer:innen erfüllt und wie leicht sie zu bedienen ist. Die wichtigsten KPIs, um die Benutzerfreundlich-

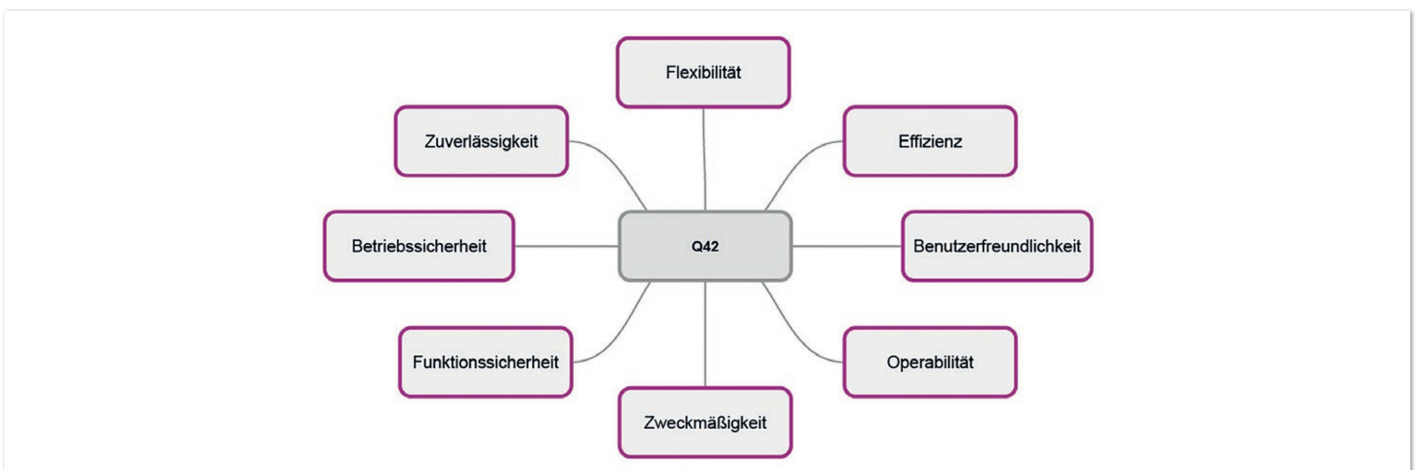


Abbildung 3: Die acht Produkt- und Systemqualitätsaspekte nach Q42 (© Ildikó Tárkányi)

keit zu messen, sind Zufriedenheitsbewertungen (*User Satisfaction Ratings*) und Bearbeitungszeiten (*Handling Times*).

- Operabilität beschreibt, wie effizient und zuverlässig die Software im laufenden Betrieb verwaltet und gewartet wird. Hierfür werden KPIs wie die Anzahl der gemeldeten Betriebsprobleme (*Operational Issues*), die Bereitstellungs-Erfolgsrate (*Deployment Success Rate*) oder die Änderungsfehlerquote (*Change Failure Rate*) herangezogen.
- Zweckmäßigkeit beschreibt, wie gut ein System die spezifischen Anforderungen der Stakeholder:innen erfüllt. Die wichtigsten KPIs zur Messung sind die Anforderungsabdeckung (*Requirement Coverage*), der Erfüllungsgrad der Anforderungen (*Business Requirement Fulfillment*) und Produktivitätsmetriken (*Productivity Metrics*).
- Funktionssicherheit bezieht sich auf die Fähigkeit eines Systems, die vorgesehenen Funktionen auch bei möglichen Störungen oder Fehlfunktionen zuverlässig auszuführen. Zur Messung der Funktionssicherheit bieten sich KPIs wie die Erkennungszeit von Anomalien oder Angriffen (*Detection Time*) und die Kompromittierungsrate (*Compromise Rate*) an.
- Betriebssicherheit bezieht sich auf die allgemeine Robustheit und Zuverlässigkeit des Systems im laufenden Betrieb. Sie bewertet, wie stabil und kontinuierlich das System auch unter widrigen Bedingungen betriebsfähig bleibt. Hier sind die Wiederherstellungszeit (*Recovery Time*) und die Zuverlässigkeitsrate (*Reliability Rate*) die wichtigsten KPIs.
- Zuverlässigkeit spiegelt sich in der Fähigkeit der Software wider, in den vorgesehenen Funktionen fehlerfrei zu arbeiten. Hier sind KPIs wie die Ausfallzeit (*Mean Time to Repair*) und die Verfügbarkeit (*Availability Rate*) entscheidend.

Wie gelingt die Sicherstellung der Qualität in verteilten Architekturen?

Meiner Ansicht nach bildet sich eine effektive Qualitätssicherung aus dem Zusammenspiel aus automatisiertem Testing, kontinuierlichem Monitoring und einem umfassenden Alerting-System, das Probleme frühzeitig erkennt und die Verantwortlichen proaktiv informiert. Die notwendigen Schritte dabei sind die Sammlung, Bearbeitung und Visualisierung von Metriken *(siehe Abbildung 4)*. In diesem Prozess ist es besonders wichtig, dass nicht nur eine solide Sammlung von Metriken vorhanden ist, sondern diese auch ansprechend und funktional visualisiert wird. So wird sichergestellt, dass relevante Informationen auf einen Blick erkennbar sind.

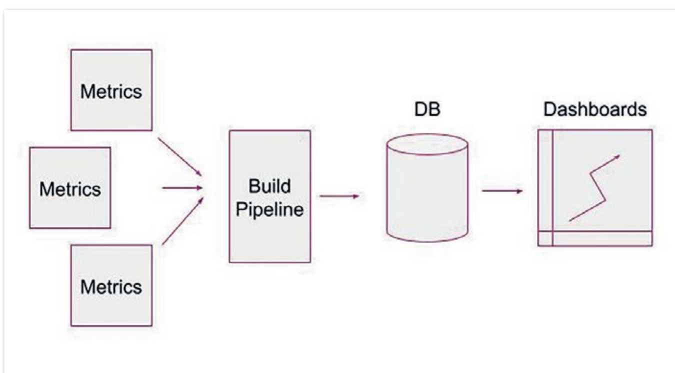


Abbildung 4: Notwendige Schritte für eine effektive Qualitätssicherung: Sammlung, Bearbeitung und Visualisierung von Metriken (© Ildikó Tárkányi)

Durch Visualisierungen können verschiedene Stakeholder:innen schnell auf kritische Werte und Trends aufmerksam werden und gegebenenfalls rechtzeitig und im besten Fall proaktiv eingreifen. Besonders im Kontext verteilter Architekturen ist eine durchdachte Darstellung notwendig, um komplexe Zusammenhänge anschaulich zu visualisieren.

Dashboards sind hierbei ein gängiges Mittel, um Metriken und deren Entwicklung über die Zeit hinweg übersichtlich darzustellen. Ein gut gestaltetes Dashboard ermöglicht es, eine Vielzahl von Qualitätsaspekte zu präsentieren und individuelle Filter für verschiedene Perspektiven zu setzen *(siehe Abbildung 5)*. Das ist essenziell, da unterschiedliche Stakeholder:innen – sei es Entwickler:innen, Manager:innen oder Kunden:innen – sich für unterschiedliche KPIs interessieren.

Visualisierung allein ist aber nicht ausreichend: Für Präsenz im Alltag braucht es Alerting. Dies stellt sicher, dass bei Erreichen bestimmter Schwellenwerte automatisch Warnmeldungen ausgegeben werden, sodass Administrator:innen, Netzwerkbetreiber:innen oder On-Call-Teams rechtzeitig informiert sind. In der Regel stellen diese Systembenachrichtigungen Informationen über den Vorfall bereit, darunter die Art des Vorfalls, die Ursache und den Ort. Sie können auch weitere Details enthalten, wie zum Beispiel die Schwere des Vorfalls oder empfohlene Maßnahmen für die Behebung des Problems, und sind dadurch ein elementarer Bestandteil der Qualitätssicherung.

Qualitäts-Alerts informieren frühzeitig über Abweichungen von definierten Qualitätsstandards oder Leistungszielen, bevor sie den Betrieb beeinträchtigen. Beispiele für solche Alerts sind die Benachrichtigungen bei Fehler nach Anpassungen (*New Defects*), der Ressourcenverbrauch (*Resource Utilization*) oder die Erkennungszeit von Anomalien (*Detection Time*). Diese Qualitäts-Alerts sind essenziell, um proaktiv Schwachstellen zu identifizieren und sicherzustellen, dass die Software den gewünschten Anforderungen entspricht.

Wie können wir für die Sicherstellung der Qualität Open-Source-Bausteine einsetzen?

Wie das Diagramm in *Abbildung 6* zeigt, können wir mit Bausteinen wie *Gradle*, *InfluxDB*, *Grafana*, *Kubernetes* und *Flux* eine Lösung bereitstellen, die unsere Anforderungen an Visualisierung und Alerting erfüllt. Die dargestellte CI/CD-Pipeline bietet eine vollständige Sicherheitsprüfung und Qualitätssicherung, ist jedoch eine exemplarische Lösung und Kombination von Tools, die je nach Kontext und Anforderungen angepasst oder erweitert werden muss.

Der Quellcode wird in einem Repository verwaltet. Entwickler:innen pushen ihren Code in das Repository, wo – sobald Codeänderungen erfolgen – ein automatisierter Workflow startet.

In der Build-Pipeline übernimmt *Gradle* als Build-Management-Automatisierungs-Tool die Ausführung der notwendigen Tasks. Dies umfasst zwei zentrale Überprüfungen:

- Statische Codeanalyse: Diese Aufgabe führt eine statische Codeanalyse durch, um Codequalität und technische Schulden zu bewerten.
- Security-Check: Dieser Schritt prüft den Code auf Sicherheitsprobleme und Sicherheitsschwachstellen.

Abbildung 5: Beispiel-Dashboard, um Metriken und deren Entwicklung darzustellen. (© Thomas Zettlitz)

Die statische Codeanalyse und die Security-Checks generieren Berichte, die bei jedem Build aktualisiert werden und so eine kontinuierliche Übersicht über den Zustand des Codes bieten. Diese Berichte werden dann in einer *InfluxDB*-Datenbank gespeichert, die speziell für Zeitreihendaten geeignet ist. Da bei jedem Build neue Datenpunkte hinzukommen, ermöglicht die Speicherung als Zeitreihe eine effiziente Verwaltung dieser Daten. So lassen sich Veränderungen und Trends über verschiedene Builds hinweg leicht nachvollziehen und bewerten.

Grafana greift auf die Rohdaten in *InfluxDB* zu und erstellt aus diesen Informationen interaktive Dashboards. Diese Dashboards ermöglichen es, Codequalität und Sicherheit in Echtzeit zu überwachen, Trends im Verlauf zu erkennen und Alarme und Benachrichtigungen einzurichten.

Welche Best Practices und Empfehlungen gibt es für effektives Monitoring und Alerting?

Die Überwachung und Sicherung der Softwarequalität erfordern nicht nur geeignete Tools, sondern auch die effiziente Nutzung und Konfiguration dieser Tools. Wir haben die Erfahrung gemacht, dass folgende

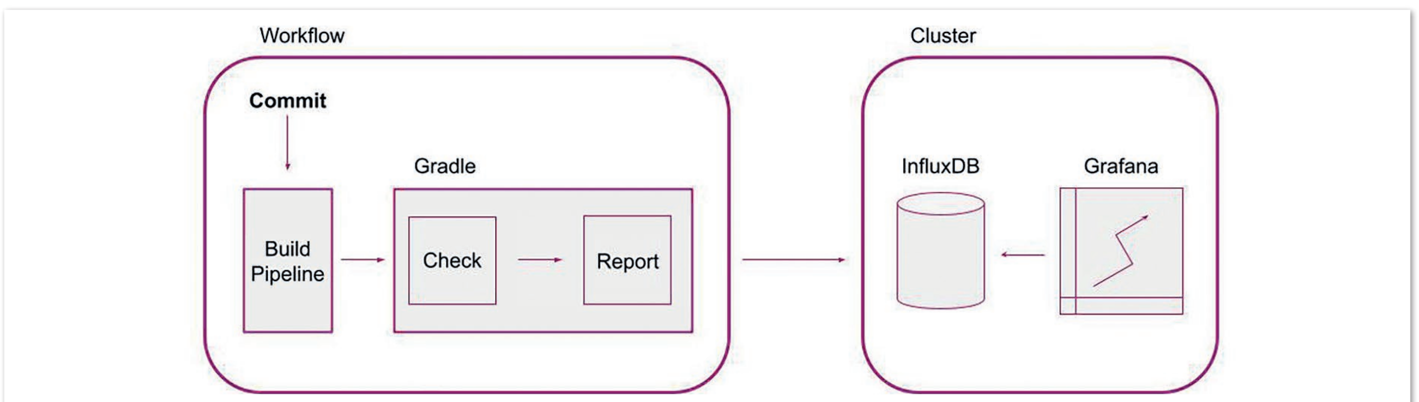


Abbildung 6: Sicherstellung der Qualität mit exemplarisch ausgewählten Open-Source-Bausteinen (© Ildikó Tárkányi)

Best Practices helfen, effektives Monitoring und Alerting umzusetzen:

- Es gibt keine „One Size Fits All"-Lösung: Jede Metrik, jede Visualisierung und jedes Alerting sollte auf die spezifischen Bedürfnisse der Stakeholder:innen angepasst sein.
- Die Erstellung von Dashboards ist zeitintensiv: Ein effektives Dashboard muss relevante Informationen aus einer Vielzahl von Datenquellen so zusammenstellen, dass sie für die Endnutzer:innen schnell und intuitiv erfassbar sind. Dies erfordert eine sorgfältige Planung und Abstimmung mit den Stakeholder:innen, um sicherzustellen, dass die relevantesten KPIs richtig ausgewählt und sinnvoll visualisiert werden.
- Dashboards ohne Alerting haben keinen Mehrwert: Dashboards zeigen zwar umfassende Informationen, doch ohne Alerts müssen Benutzer:innen das Dashboard aktiv im Auge behalten, um kritische Änderungen rechtzeitig zu bemerken. Ein Alerting-System löst automatisch Benachrichtigungen aus, sobald vordefinierte Schwellwerte überschritten werden oder ungewöhnliche Muster auftreten. So erhalten die Verantwortlichen sofort Informationen und können proaktiv handeln, ohne die Daten kontinuierlich manuell im Dashboard prüfen zu müssen.
- Schwellwerte und Warnmeldungen benötigen ständige Anpassung: Die Schwellwerte für Warnungen und Alarme erfordern eine kontinuierliche Überprüfung und Anpassung, um den sich verändernden Projektanforderungen gerecht zu werden. Ein Schwellwert, der heute sinnvoll ist, könnte in einem Jahr bereits zu restriktiv oder zu liberal sein.
- Werkzeuge müssen flexibel sein: Die eingesetzten Monitoring- und Alerting-Tools sollten eine hohe Flexibilität bieten, damit Stakeholder:innen verschiedene KPIs nach individuellen Anforderungen und Prioritäten konfigurieren können.

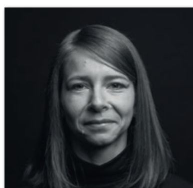
Sind Dashboards und Alerts der Weg zu einem stabilen und skalierbaren Qualitätsmanagement?

Gut gemachte Dashboards helfen definitiv! Qualitätssicherung und Metrik-Management sind unverzichtbare Bestandteile der modernen Softwareentwicklung und spielen eine zentrale Rolle im Betrieb und der Wartung verteilter Architekturen.

Die vorgestellte Open-Source-Lösung ist eine Möglichkeit, wie man durch ein strukturiertes Monitoring, maßgeschneiderte Dashboards und präzises Alerting eine hohe Softwarequalität aufrechterhalten kann. Die kontinuierliche Anpassung an neue Anforderungen stellt sicher, die Softwarequalität nicht nur zu messen, sondern gezielt zu steuern und zu verbessern.

Quellen

[1] o.V., *Introduction to the Q42 quality model, in*: *https://quality.arc42.org*, *https://quality.arc42.org/articles/arc42-quality-model*, letzter Zugriff: 26. Oktober 2024



Ildikó Tárkányi
ildiko.tarkanyi@qaware.de

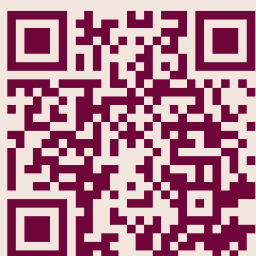
Ildikó Tárkányi ist Senior Software Entwicklerin, Projektleiterin und Agiler Lerncoach bei der QAware. Sie unterstützt die Konzeption, den Aufbau, das Testen und das Betreiben komplexer Softwaresysteme. Durch ihre Leidenschaft und ihr Engagement für hohe Qualitätsstandards hilft sie Teams, anspruchsvolle Probleme zu meistern.

EARLY BIRD BIS 01.04.25

APEX connect

EUROPA-PARK Rust

13. – 15. MAI 25

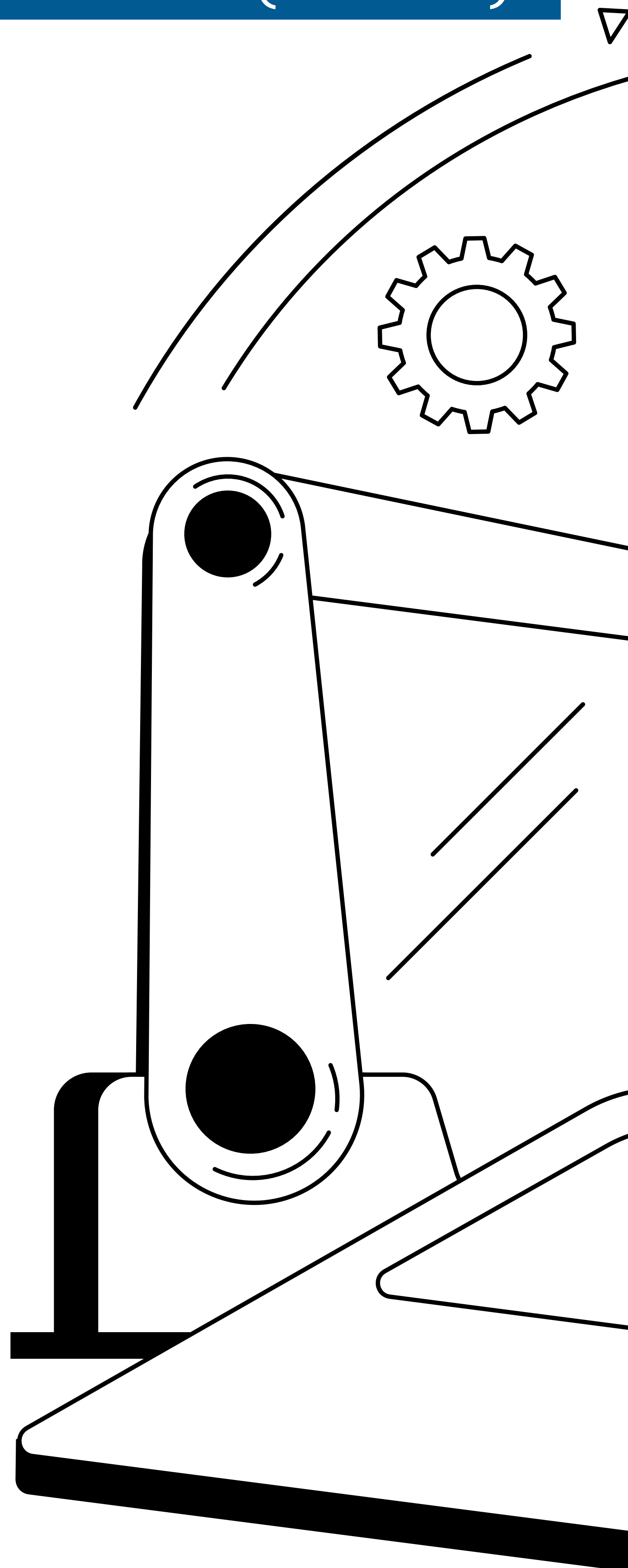


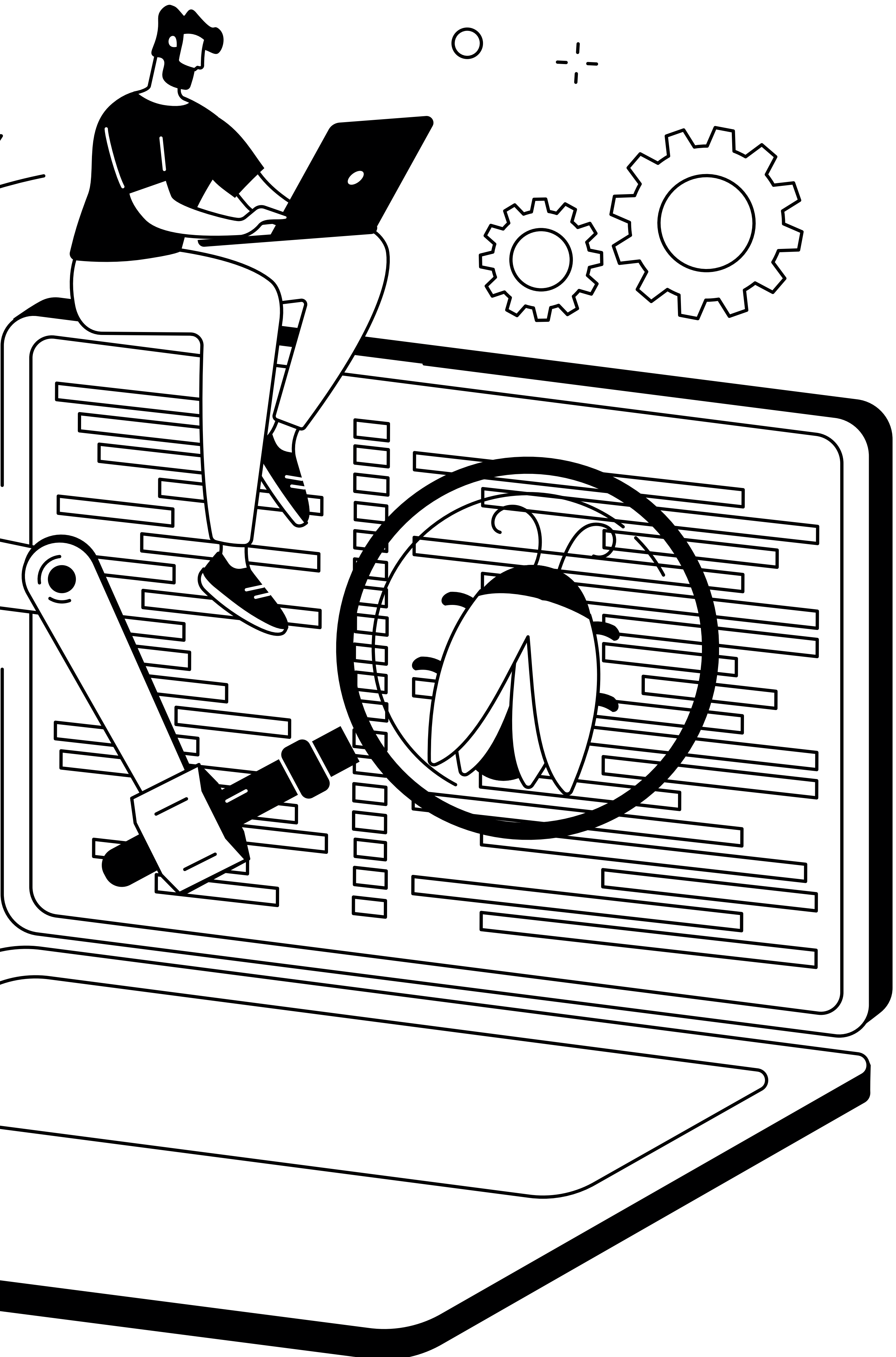
apex.doag.org

DOAG

Risikobewusst, modular… einfach passend testen! (Teil 2)

Peter Fichtner, Ralf Straßner, Atruvia AG





Im ersten Teil dieser Artikelreihe haben wir Möglichkeiten aufgezeigt, wie Teams durch den Fokus auf die produkt- und projektindividuellen Risiken erste Ideen für passende Testarten erhalten können, die einen großen Nutzen liefern. Gleichzeitig haben wir anhand der gängigen Testmodelle aufgezeigt, dass für eine gute Handlungsfähigkeit des Teams vor allem präzise automatisierte Tests notwendig sind. Um diese zu erreichen, widmen wir uns im zweiten Teil dem Komponenten-, vor allem aber dem Testschnitt und zeigen Möglichkeiten auf, den Testansatz kompakt und übersichtlich zu dokumentieren. Abschließend ziehen wir ein Fazit über beide Teile der Artikelserie.

Wenn ein Team seine produkt- und projektindividuellen Risiken kennt und erste Ideen für die dazu passenden Testarten hergeleitet hat, steht es vor der Herausforderung der technischen Umsetzung. Dabei geht es noch nicht um die konkrete Implementierung einzelner Tests, sondern erst einmal darum, den passenden Testschnitt zu finden, um von den ersten Ideen zu den passenden Testarten zu kommen. Hierbei spielt der Anwendungsschnitt eine zentrale Rolle. Wir werden nachfolgend an unserer Beispielanwendung aus dem ersten Teil, der Berechnung und Darstellung einer Bundesliga-Tabelle auf Grundlage der einzelnen Spieltagsergebnisse, den Zusammenhang zwischen Anwendungs- und Testschnitt veranschaulichen. Gleichzeitig sind die dabei getroffenen Aussagen allgemeingültig.

Wie sollten Anwendungen geschnitten sein?

Um den ermittelten Risiken besser mit Tests entgegenwirken zu können, ist es erforderlich, dass Anwendungen in Bestandteile mit klaren Zuständigkeiten aufgeteilt sind. Häufig steht schon vor Beginn der Umsetzung der zukünftige Architekturstil einer Anwendung fest, da zumindest bei nicht-trivialen Anwendungen weitere Gründe für diese Aufteilung sprechen. Auch ohne die individuellen Risiken zu kennen, wollen Teams durch Architekturstile wie Ports & Adapters [1] (auch hexagonale Architektur genannt) die Komplexität in der Umsetzung der Anwendung senken. Dazu trägt bei, dass die Domäne sowie jeder Adapter für sich eine klare und eindeutige Zuständigkeit besitzen. Hiervon können ebenfalls die Tests profitieren, da einzelne Bestandteile isoliert und somit zielgerichteter und leichter getestet werden können. Die Auswahl des Architekturstils wirkt sich also nicht nur auf die Struktur des Produktivcodes, sondern auch auf die Struktur der Tests aus.

Wir empfehlen, Anwendungen primär fachlich, anstatt technisch zu strukturieren, um eine lose Kopplung und eine hohe Kohäsion zu erreichen. An unserer Beispielanwendung veranschaulicht, sollte dies zu Modulen wie beispielsweise „ergebnis" und „mannschaft" statt „repositories" und „services" führen [2]. In Java lässt sich dies durch eine Aufteilung in entsprechende Packages erreichen. Ebenso möchten wir domänenzentriert Fachlichkeit umsetzen, ohne dass sich die Fachlichkeit mit technischen Aspekten wie Persistenz, Message-Broker-, HTTP-Kommunikation oder ähnlichem vermischt. Dies erreichen wir, indem wir innerhalb der jeweiligen Packages jede unterschiedliche Zuständigkeit in einer separaten Klasse abbilden.

Wie in *Abbildung 1* dargestellt, besteht die Grundstruktur unserer Beispielanwendung aus einem Frontend, in dem die Navigation durch die Anwendung implementiert ist und welches die berechnete Tabelle zur Visualisierung vom Backend abruft. Das Backend wiederum bedient sich unter anderem der Daten, die nicht in der Hoheit des Backends selbst liegen, sondern von anderen Systemen zur Erfüllung des Requests angefordert werden müssen. Diese Third-Party-Komponente liegt demzufolge nicht unter der Kontrolle des Teams.

Im Inneren des Backends ist die Fachlichkeit, hier nach Domain-Driven Design (DDD) als Domänenmodell, implementiert. Das Domänenmodell wird dabei so gestaltet, wie es für das Abbilden der Domänenlogik am besten geeignet ist. Da hier Ports & Adapters als Architekturstil gewählt wurde, unterteilt sich das Domänenmodell selbst nochmals in einen Domänenkern (*core*) und den außen am Rand liegenden, ebenfalls von der Domäne definierten Ports. Die in *Abbildung 1* links dargestellten Ports (*Primary Ports*) ermöglichen den Aufruf in die Domäne. Die Ports auf der rechten Seite (*Secondary Ports*) dienen der Domäne dazu, mit Infrastruktur interagieren zu können, ohne diese zu kennen oder gar von ihr abhängig zu sein. Der Domänenkern beinhaltet Domänenobjekte, konkret *Aggregates*, *Entites* und *Value Objects*, die keine 1:1-Abbildung der externen Mo-

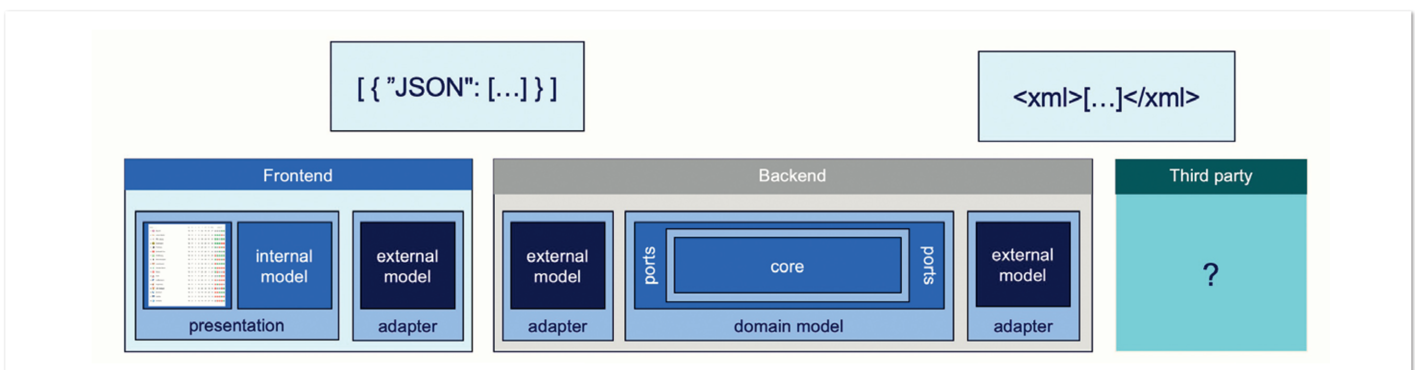


Abbildung 1: Übersicht über die Anwendung und ihre Bestandteile (© Peter Fichtner, Ralf Straßner)

delle darstellen.
Durch die Separierung des Infrastrukturcodes wird das Domänenmodell weder durch externe Modelle der Datenlieferanten (wie zum Beispiel Third Party) noch von der Struktur der eigenen Netzwerkschnittstellen (wie zum Beispiel JSON/HTTP) beeinflusst. In den externen Modellen befinden sich die Datentransferobjekte (DTOs). Neben Ports & Adapters forcieren dies ebenfalls die Architekturstile *Onion Architecture* [3] und *Clean Architecture* [4].

Somit ziehen sich strukturelle Änderungen der Datenlieferanten nicht durch die eigene Anwendung, sondern enden an der Stelle des Mappings der externen Modelle. Aber auch Umfang und Komplexität von fremden Domänenmodellen enden hier und können nicht das eigene Domänenmodell beeinflussen. Dies wird auch als *Anti-Corruption Layer* (ACL) [5] [6] bezeichnet.

Aus den genannten Gründen wurde auch in unserem Frontend zwischen externem und internem Modell unterschieden. Das interne Modell hält die Daten exakt so vor, wie sie zur Visualisierung benötigt werden. Interne Modelle sind häufig ähnlich, aber nicht identisch zum jeweiligen externen Model.

Wie sollten Tests geschnitten sein?

Würde die Absicherung unterschiedlicher Aspekte über End-to-End-Tests umgesetzt werden, so hätten diese Tests nicht nur lange Laufzeiten, sondern es würde diesen Tests an Präzision (Fehlerlokalität) fehlen. Da hier mehr Code durchlaufen wird, als es für den jeweiligen Testfall eigentlich notwendig ist, kann ein fehlschlagender Test viele Gründe haben. Derartige Tests sind also auch instabiler und verursachen bei Fehlschlägen meist größere Aufwände für die Analyse. Es ist also von großem Vorteil, eine Anwendung aus in sich getesteten Komponenten zusammenzusetzen. Selten dürfte es sinnvoll sein, erst in der zusammengesetzten oder gar deployten Anwendung festzustellen, dass Teilaspekte wie das Schreiben in die Datenbank oder die Kommunikation mit anderen Diensten nicht wie erwartet funktioniert. Weitere Beispiele sind, dass der eigene HTTP-Eingangsadapter nicht wie erwartet antwortet oder die eigene Fachlichkeit nicht korrekt implementiert ist. Welche Art von Anwendung gebaut und wie diese später deployt werden soll, ist hierbei nicht entscheidend. Dies gilt für *Lambda Functions* ebenso wie für Microservices oder für die Bestandteile eines Deployment-Monolithen.

Wie bereits erwähnt, führt die Aufteilung des Produktivcodes in Domäne und Infrastruktur zu einer besseren Testbarkeit. Das Fortführen dieser Aufteilung in den Tests ermöglicht auch die Bewertung funktionaler Korrektheit einzelner Komponenten unabhängig vom Verhalten anderer Komponenten. Somit erhält man zielgerichtete Tests, die schnelles und vor allem frühes Feedback liefern.

Im Folgenden beschreiben wir, wie Tests geschnitten werden sollten, und betrachten dabei, was in welchem Testschnitt getestet werden soll, aber auch was bewusst nicht. *Abbildung 2* dient zur Visualisierung dieser allgemeingültigen Testschnitte anhand unserer Beispielanwendung.

Testschnitt 1: Domänentests

Ist die Domänenlogik infrastrukturfrei, so lässt sich diese gut und einfach testen. Unterschiedliche Zustände lassen sich leichtgewichtig und unabhängig von der Struktur, dem Verhalten und der Verfügbarkeit von Fremdsystemen herstellen. Dies erfolgt direkt im Domänenmodell, was die *Arranges* der Tests vereinfacht. Trotzdem beobachten wir oft, dass Teams den Anwendungsschnitt hier ignorieren und einen größeren Testschnitt wählen. Das hat aber immer Nachteile.

Würden die Tests die Zustände in den Strukturen der Fremdsysteme (wie sie im *external model* abgebildet sind) erzeugen, müssten Tests der Domänenlogik bei strukturellen Änderungen der Fremdsysteme angepasst werden. Dies kann mit *TestFixtures* wie zum Beispiel *ObjectMother* [7] oder *TestDataBuilder* [8] zwar abgeschwächt, aber nicht verhindert werden.

Würden wir die unterschiedlichen Zustände in den Fremdsystemen selbst herstellen und somit Domänentests mit integrativem Charakter betreiben, so hätten diese Laufzeitabhängigkeiten und wür-

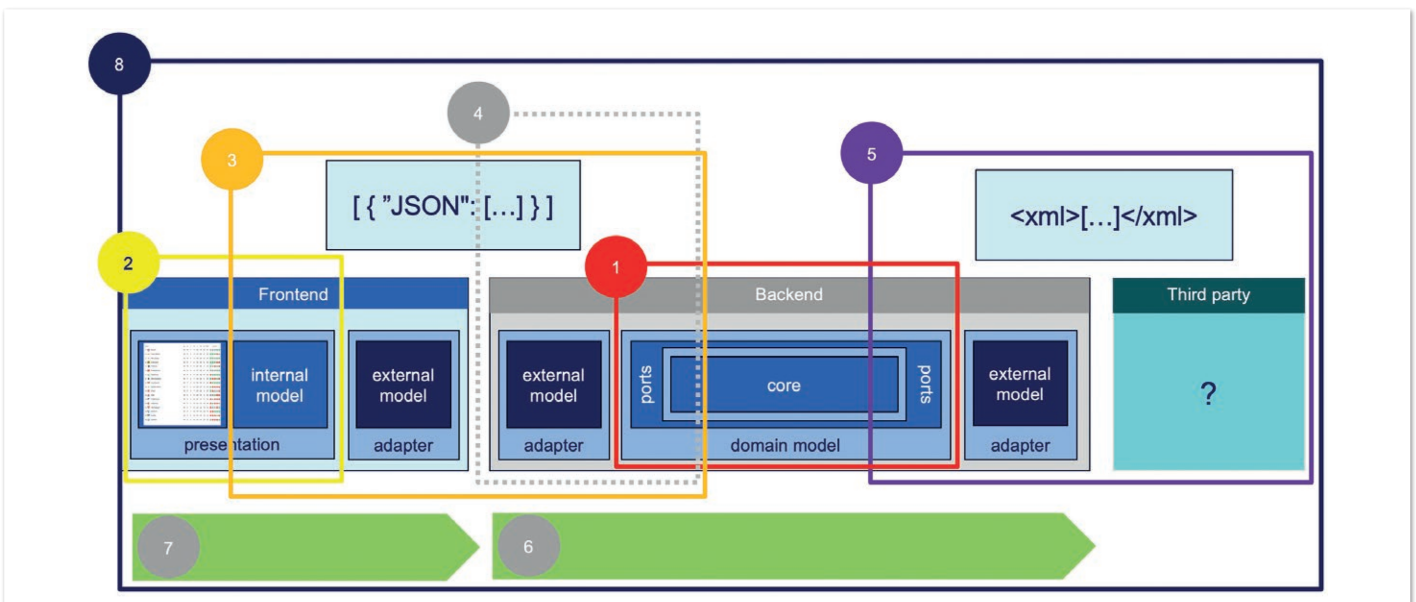


Abbildung 2: Testschnitte in der Beispielanwendung (© Peter Fichtner, Ralf Straßner)

den zusätzlich das Geheimnisprinzip *[9]* verletzen. Beispielsweise wäre das Eingreifen von außen in den Datenhaushalt (Persistenz) eines Fremdsystems nötig, wodurch eine technische Kopplung entsteht. Oder das Backend-Team benötigt das fremde Fachwissen, wie verschiedene Zustände in einem Fremdsystem herstellbar sind, und ist somit fachlich gekoppelt. In unserem konkreten Fall, bei dem das Fremdsystem (Third Party) gar nicht durch das Umsetzungsteam beeinflusst werden kann, scheiden diese Möglichkeiten sowieso aus.

Tests der Domänenlogik sind unabhängig von der Anzahl durchlaufener Methoden/Klassen sehr schnell. Lange Testlaufzeiten entstehen hauptsächlich durch das Erstellen von Verbindungen und durch Aufrufe zu Drittsystemen. Da dies hier entfällt, kann das *System under Test* (SUT) innerhalb des Testschnitts problemlos etwas weiter gefasst werden. Je größer das SUT wird, desto leichter lässt sich der fachliche Bezug des Tests erkennen. Allerdings sinkt dadurch die Fehlerlokalität, weil mehr Code für einen fehlschlagenden Test verantwortlich sein kann. Dieser Nachteil lässt sich durch häufige Testausführung auf kleineren Changesets abschwächen. Diese lassen sich durch häufigere, kleinere Commits erreichen, bei denen die Tests auch auf dem CI-Server durchlaufen werden. Ein zu klein gefasstes SUT birgt die Gefahr, dass Implementierungsdetails getestet und damit manifestiert werden.

Testschnitt 2: UI-Tests

Diese Tests haben das Ziel, die korrekte Darstellung zu prüfen. Durch die Trennung auf der Frontendseite von internem und externem Modell lassen auch sie sich leichtgewichtiger durchführen, denn hierbei finden keine Abrufe von Daten via HTTP statt, weder integrativ noch simuliert (umgangssprachlich: „gemockt"). Stattdessen können die Daten direkt in das interne Modell gelegt werden. Mögliche Änderungen der Struktur des Backends wirken sich so nicht auf die *Arranges* der Tests aus. Ob das Befüllen des internen Modells über einen HTTP-Aufruf und damit über das externe Modell korrekt erfolgt, ist hier irrelevant. Dies stellen die nachfolgend beschriebenen Tests in Testschnitt 3 sicher.

Testschnitt 3: Integrationstests (Frontend)

Diese Tests sollen im besten Fall nur die Integration zwischen Frontend und Backend sicherstellen und nicht durch Änderungen der Fachlogik des Backends beeinflusst werden. Hier soll sichergestellt werden, dass die Aufrufe zum Backend korrekt erfolgt sind und dessen Antworten korrekt in das interne Modell des Frontends übertragen wurden. Wie aber lässt sich bei den Frontend-Tests das Backend partiell, also ohne die eigentliche Fachlogik, hochfahren und wie lassen sich die unterschiedlichen Zustände von außen gesteuert im Backend herstellen, die für die unterschiedlichen Frontend-Tests benötigt werden?

Die Antwort auf diese Frage lautet *Consumer-Driven Contract Tests* (CDCT): Das Frontend fährt die eigentlichen Integrationstests gegen einen vom Frontend kontrollierten Stellvertreter des Backends. Das ermöglicht die Verifikation, dass die Integration in den unterschiedlichen Konstellationen erfolgreich ist, allerdings unter der Annahme, dass sich das echte Backend identisch zum Stellvertreter verhalten wird. Deswegen entsteht bei erfolgreicher Ausführung dieser Tests eine Beschreibung des konfigurierten Verhaltens des Stellvertreters, dies ist der *Contract*. *Contracts* sind also Nebenprodukte erfolgreicher Testläufe der Consumer – in unserem Beispiel ist dies das Frontend. *Contracts* können zu beliebigen Zeitpunkten vom Provider, also hier vom Backend, verifiziert werden, ohne dafür den/die Consumer selbst nutzen zu müssen. Dabei wird die reale HTTP-Schnittstelle des Backends hochgefahren und die im Contract definierten Zustände durch den Provider hergestellt. Anschließend wird das Backend mit den in den *Contracts* beschriebenen Interaktionen (HTTP-Requests) aufgerufen und die Antworten (HTTP-Responses) mit den jeweiligen Erwartungen des Consumers abgeglichen.

So kann (jederzeit und unabhängig) verifiziert werden, ob der echte Provider bei unterschiedlichen Zuständen und unterschiedlichen Requests auch wie von den Consumern erwartet antwortet.

Testschnitt 4: Primary-Adapter-Tests

Trotz CDCT sollte das Backend selbst Tests aufweisen, die die eigenen Erwartungen und Annahmen an das Verhalten der eigenen Schnittstelle beschreiben und überprüfen. Denn Fehler in der eigenen Eingangsschnittstelle sollten nicht erst durch Konsumenten beziehungsweise deren Tests entdeckt werden, weder durch „echtes" integratives Testen noch bei CDCT. Die eigenen Erwartungen können dabei strikter als die Erwartungen der Konsumenten sein.

So könnte bei den Primary-Adapter-Tests auf einen spezifischen HTTP-Status-Code wie 201 („created") geprüft werden, selbst wenn für alle Konsumenten jeder beliebige OK-Status-Code (2xx) ausreichend wäre. Jeder Primary-Adapter hat seinen eigenen zugehörigen Test, der sicherstellt, dass die technologiespezifischen Aufrufe korrekt vom Adapter entgegengenommen wurden, dass die Aufrufe in das Domänenmodell korrekt erfolgten und dass die Daten aus dem Domänenmodell als Antwort korrekt an den Aufrufer zurückgegeben wurden. Damit die Tests, wie auch im Testschnitt 3, von keiner beziehungsweise von möglichst wenig Domänenlogik abhängig sind, sollten in den *Arranges* der Tests Domänenobjekte genutzt werden, die möglichst nah am Primary-Adapter liegen.

Testschnitt 5: Integrationstests (Backend)

Wie auch bei Testschnitt 3 soll hier die Integration zweier Komponenten sichergestellt werden. Jedoch lässt sich CDCT für die Tests der Integration nur dann sinnvoll nutzen, wenn organisatorisch Einfluss auf beide Parteien genommen werden kann *[10]*. Wenn beispielsweise der Schnittstellengeber keine Contract-Verification durchführt, bieten Contracts wenig Mehrwert. Diese Situation besteht jedoch bei unserer Third-Party-Abhängigkeit, weshalb hier zu anderen Mitteln gegriffen werden muss.

Um die Integration sicherzustellen, müssen hier Aufrufe gegen den echten Service stattfinden. Zu beachten ist, dass hier häufig nicht mehr ähnlich strikte *Assertions* wie in den Testschnitten 3 und 4 verwendet werden können, da Responses aufgrund von fachlichen Änderungen oder variablen Daten nicht vorhersagbar sind (zum Beispiel Ergebnisse laufender, zukünftiger oder sogar bereits beendeter Spiele). Häufig ist es sinnvoll, zusätzlich mit aufgezeichneten Responses (*Captures*) zu arbeiten. Diese dienen allerdings nur der Optimierung des eigenen Arbeitsablaufes, nämlich für schnelleres und stabileres Feedback bei Änderungen am eigenen Code. Dieses Vorgehen ersetzt nicht die weiterhin notwendigen Integrationstests, kann diese jedoch verschlanken. Allerdings birgt dies die Gefahr, dass sich die echten Responses inkompatibel weiterentwickeln

können und man fälschlicherweise längere Zeit, konkret bis zum nächsten Test der Integration, grünen Tests mit veralteten *Captures* vertraut. Hinzu kommt der Pflegeaufwand für die *Captures* selbst.

Die Tests sollen sicherstellen, dass die Daten korrekt ins Domänenmodell gemappt wurden. Bei einem, wie hier, nicht-blutleeren Domänenmodell bedeutet dies mehr als nur das erfolgreiche Setzen von Werten/Attributen, es bedeutet nämlich, dass sich die Domänenobjekte im Domänenmodell auch in fachlich valide Zustände versetzen ließen. Daher sollten die *Assertions* den korrekten Zustand des Domänenmodells beurteilen, dies aber so nah wie möglich an der Stelle, an der in das Domänenmodell hinein gemappt wird (rechte Seite), sodass hier möglichst wenig Domänencode mitgetestet wird. Eine genauere Beschreibung, welche Teile des Domänenmodells dies betrifft und welche nicht, ist kontext- und umsetzungsspezifisch, sodass keine allgemeingültige Beschreibung oder gar grafische Darstellung möglich ist. Daher verzichten wir an dieser Stelle auf einen detaillierteren Blick. Als weiterführende Literatur empfehlen wir hierzu das taktische Design aus DDD *[5] [6]* mit den Building Blocks.

Testschnitt 6: Smoke-Tests (Backend)

Durch die Testschnitte 1 bis 5 sind sämtliche Anwendungsbestandteile mit Tests abgedeckt. Offen ist jedoch noch, ob die Anwendung aus diesen Bestandteilen gebaut werden kann und die Verifikation, ob das Zusammenfügen zu einem erfolgreichen, also releasefähigen, Ergebnis/Artefakt führt. Damit werden zum Beispiel folgende Aspekte getestet:

- Lässt sich das Artefakt, das in Produktion gehen soll, überhaupt bauen und zum Beispiel als Docker-Image packen?
- Funktioniert die Anwendung mit der für den Betrieb vorgesehenen Konfiguration/Konfigurationsdatei?
- Lässt sich das Artefakt in einen betriebsfähigen Zustand bringen, ist die Anwendung beispielsweise dann auf dem definierten Netzwerk-Port erreichbar?

Bei der Verifikation, ob die Anwendung betriebsbereit ist, besteht das Risiko von zu starken als auch von zu schwachen *Assertions*. In unserem Beispiel wäre die Assertion möglicherweise zu schwach, wenn lediglich geprüft würde, ob die Antwort ein JSON-Array beinhaltet. Hier bestünde die Gefahr, dass fehlerhaftes Verhalten unentdeckt bliebe (*False Negative*), wie zum Beispiel eine Fehlermeldung im JSON-Array statt der Tabelleneinträge. Das Abprüfen auf die Anzahl erwarteter Elemente im JSON-Array (18, da 18 Mannschaften in der Liga sind) ist eigentlich schon zu strikt, da der Smoke-Test durch fachliche Änderungen, wie die Änderung der Anzahl an Mannschaften oder den Ausschluss einer Mannschaft bei der Berechnung, fälschlicherweise rot wird (*False Positive*). Dennoch stellt dies einen möglichen Kompromiss dar, denn die Änderung dieser Eigenschaft ist eher unwahrscheinlich beziehungsweise selten und führt damit auch selten oder nie zu fälschlicherweise anschlagenden Tests. Die massive Verringerung der Gefahr von *False Negatives* wiegt das seltene Problem von *False Positives* bei weitem auf.

Testschnitt 7: Smoke-Tests (Frontend)

Die gleichen Herausforderungen wie beim Testschnitt 6 gelten auch für den Smoke-Test des Frontends. Dieser soll verifizieren, ob das gebaute/gebündelte Artefakt, das in Betrieb gehen soll, als solches funktioniert, ohne dabei bereits getestet Aspekte erneut zu testen.

Testschnitt 8: End-to-End-Tests

Zu guter Letzt sollten noch wenige End-to-End-Tests über alle Bestandteile der Anwendung durchgeführt werden, um folgende Fragen zu klären:

- Lässt sich die Gesamtanwendung in einem Umfeld betreiben?
- Können die einzelnen Bestandteile miteinander kommunizieren (sind beispielsweise die Firewall-Freischaltungen erfolgt)?
- Kann ein Gesamtprozess aus Kundensicht abschließend behandelt werden?

Herausfordernd bei diesen Tests ist, dass hier keine 1:1-Zuständigkeit existiert, also wer diese Tests definiert, verantwortet und wer im Falle von Fehlern/Problemen aktiv werden soll. Darum sollte dies organisatorisch geklärt und geregelt sein.

Überlappungen der Testschnitte

Die Testschnitte sollten Überlappungen aufweisen. So treffen beispielweise die Tests der Fachlichkeit (Testschnitt 1) die Aussage, dass korrekt erstellte Tabellen unter der Voraussetzung bestimmter Spieltagsergebnisse entstehen. Dass die Spieltagsergebnisse überhaupt bezogen werden können, stellen die Tests in Testschnitt 5 sicher, jedoch nicht, ob die Spieltagsdaten, die aus Third Party bezogen werden, auch dem entsprechen, was der Testschnitt 1 in seinen Tests als Annahmen nutzt. Daher muss darauf geachtet werden, dass die Erwartungen auch dem realen Verhalten entsprechen. Dies kann manuell erfolgen, ist aber auch technisch mit *TestFixtures* umsetzbar. Hier referenzieren die *Asserts* des Tests in Testschnitt 5 eine *TestFixture*, die die Tests in Testschnitt 1 in den *Arranges* nutzen können, allerdings zu Kosten der strikten Trennung des eigentlich unabhängigen Testcodes.

Testschnitt für das Testziel „Fachliche Regeln"

Im ersten Teil unserer Artikelreihe haben wir das Testziel „Fachliche Regeln" formuliert, auf das wir an dieser Stelle wieder zurückgreifen wollen. In diesem Testziel sind die beiden identifizierten Risiken „Funktionales Risiko" und „Risiko von Fehlkommunikation" adressiert. Ein wichtiger Aspekt des Testziels ist es, ein frühes und schnelles Feedback in der Entwicklung zu erhalten. Der vorgenommene Schnitt der Tests ist hilfreich, um dieses Testziel beziehungsweise die dazugehörige Testart präzise und möglichst einfach umsetzen zu können. Wie bereits erläutert, können wir mit dem Testschnitt 1 unsere Domänenlogik testen, ohne die Anwendung hochfahren oder gar deployen zu müssen. Daher ist dieser Testschnitt ideal, um das „Funktionale Risiko" zu adressieren und die Korrektheit der fachlichen Regeln hier zu testen.

Um eine gute rollenübergreifende Zusammenarbeit und ein gemeinsames Verständnis zu erreichen, lassen sich Ansätze des *Behaviour-Driven Development* (BDD) nutzen. Wir hatten bereits im ersten Teil unserer Artikelserie Möglichkeiten aufgezeigt, wie wir BDD mit einem präzisen Testschnitt kombinieren können. Eine Möglichkeit wäre beispielsweise, dass der Fachexperte die fachlichen Szenarien über eine XLS-Datei beschreibt, die im Test eingelesen wird. Wir hatten diese Testart „präzise zugeschnittene User-Story-Tests" genannt und erachteten dies für unser Beispiel als passend, da die fachlichen Regeln die Kernfachlichkeit unserer Anwendung darstel-

len. Das heißt, sie sind für den Fachexperten nicht nur wichtig, sondern auch auf dem Detailgrad eines Domänentests nachvollziehbar und bilden nicht lediglich einen Teilausschnitt einer User Story ab. Daher sind die Tests aus Sicht der Agilen Testquadranten fachlich orientiert. Gleichwohl sei an dieser Stelle darauf hingewiesen, dass in komplexeren Anwendungen unter User Stories oftmals größere fachliche Prozesse mit häufiger Benutzerinteraktion verstanden werden.

In unserer Beispielanwendung ermöglicht die Kombination von BDD mit Domänentests zum einen, dem Fachexperten Gewissheit darüber zu geben, dass die von ihm beschriebenen fachlichen Regeln korrekt umgesetzt wurden. Zum anderen leiten die Tests die Entwickler:innen bereits während der Umsetzung und führen zu einer höheren Sicherheit bei dieser. Des Weiteren dienen die Tests als Schutz vor Regression der Anwendung. Das frühe und schnelle Feedback der leichtgewichtigen Tests ermöglicht eine häufige Ausführung und gewährleistet somit, dass das Team von den Tests stärker profitiert.

Somit ist die Testart für das Testziel „Fachliche Regeln" so konkret definiert, dass sie umgesetzt werden kann. Damit haben wir nun die Schritte aufgezeigt, die die Relevanz der Testarten sicherstellen und helfen, diese so zu gestalten, wie sie im individuellen Kontext benötigt werden. Das ist unser empfohlener Weg zu einem passenden Testansatz. Bei der Implementierung der Tests spielen Aspekte wie die Auswahl eines Testframeworks, gut gewählte Testszenarien und gutes Testdesign eine wichtige Rolle. Da diese Aspekte nicht im Fokus unseres Artikels stehen, betrachten wir stattdessen noch eine mögliche Art der Dokumentation des Testansatzes, um sicherzustellen, dass der Testansatz verständlich und übersichtlich bleibt.

Dokumentation des Testansatzes

Ein guter Testansatz, der auf präzisen Testschnitten aufbaut, ist in seiner Gesamtheit durchaus komplex. Um dennoch den Überblick zu behalten, auch über die vielen kleinen Entscheidungen, die für den Testansatz getroffen wurden, benötigen Teams eine entsprechende Dokumentation. Diese soll ihnen insbesondere dabei helfen, ein gemeinsames Verständnis zu schaffen. Dann sind sie in der Lage, den Testansatz weiterzuentwickeln und können diesen beispielsweise auf sich geänderte Risiken anpassen. Denn Risiken sind nicht statisch, sondern verändern sich über den Produktlebenszyklus. Ein anfangs hohes Marktrisiko verschwindet, wenn das Produkt erfolgreich am Markt etabliert wurde, aber im gewachsenen System könnte sich zum Beispiel ein Refactoring-Risiko ergeben. Wie muss dann der Testansatz angepasst werden, um darauf adäquat zu reagieren? Hierfür haben wir zwei kompakte Dokumentationsformate entwickelt: Den Teststeckbrief und den darauf aufbauenden Test Canvas.

Der Teststeckbrief

Ein Teststeckbrief dokumentiert eine einzelne Testart und hält die Entscheidungen fest, die beim Durchlaufen der von uns vorgestellten Schritte getroffenen wurden, um bestimmten Risiken zu begegnen. Darüber hinaus wird das *System under Test* noch genauer definiert, als es der reine Testschnitt auszusagen vermag. Der Teststeckbrief gliedert die für die Testart relevanten Informationen in vier Abschnitte: *Warum*, *Was*, *Was nicht* und *Wie*.

Zudem erhält jede Testart einen eindeutigen Namen. Das mag zunächst trivial klingen, beugt aber Missverständnissen vor, wenn das Team über bestimmte Tests spricht. In unseren Coachings stellen wir oft fest, dass Teammitglieder zum Beispiel über „die Unit-Tests" oder „die Cypress-Tests" sprechen, damit aber bei genauerer Betrachtung unterschiedliche Testarten zusammenwerfen, die nur einzelne Gemeinsamkeiten aufweisen, wie beispielsweise das verwendete Testframework.

Es ist daher wichtig, dass für eine Testart alle Abschnitte des Steckbriefs eindeutig beschrieben werden. Das heißt, jeder konkrete Test dieser Testart besitzt die identischen Ziele und Merkmale. Neben dem *System under Test* und den eingesetzten Technologien ist dies auch die Methodik, wie der Test erstellt wird. Denn ein Test kann beispielsweise nicht mehr auf das Testziel „das Team leiten" einzahlen, wenn bei der Methodik von Test-Driven Development zu Test-Last gewechselt wird. Können einzelne Abschnitte nicht mit eindeutigen Informationen gefüllt werden, beispielsweise weil eine Teilmenge der Tests andere Aspekte prüft als die sonstigen Tests, handelt es sich in Wirklichkeit um mehrere Testarten. Eine andere Fragestellung könnte sein: Warum prüfe ich die gleichen Aspekte mit zwei verschiedenen Technologien? Daher ist es wichtig, genau zu unterscheiden, um die Relevanz jeder Testart einzeln hinterfragen zu können.

Im Folgenden werden die einzelnen Abschnitte des Teststeckbriefes erläutert.

Abschnitt „Warum"

In diesem Abschnitt werden das Testziel und die adressierten Risiken festgehalten. Diese Verknüpfung mit der Testart zeigt auf, warum diese Testart benötigt wird. Dies ist auf Code-Ebene in der Regel nicht ersichtlich. Daher füllt der Teststeckbrief eine Informationslücke, vor der insbesondere neue Teammitglieder stehen. Der Teststeckbrief hilft ihnen, sich auf einer höheren Abstraktionsebene einen Überblick über die unterschiedlichen Tests und Testarten zu verschaffen und Zusammenhänge zu verstehen.

Eine Möglichkeit, die Code-Ebene ebenfalls zu verknüpfen, besteht darin, an den konkreten Testimplementierungen beispielsweise mittels Annotationen die entsprechende Testart zu benennen und es somit noch einfacher zu machen, den dazugehörigen Teststeckbrief zu identifizieren. Des Weiteren erleichtern es die Teststeckbriefe, auf Änderungen am Kontext und den daraus resultierenden Risiken reagieren zu können.

Abschnitte „Was" und „Was nicht"

Der Abschnitt „Was" setzt auf dem Testschnitt auf, beschreibt das *System under Test* aber noch konkreter. Diese Schärfung hilft bei der Umsetzung einer Testart und sollte idealerweise vorher stattfinden. Während in Teams oft eine grobe Vorstellung davon besteht, was eine Testart prüfen soll, so wird es doch mitunter schwieriger, exakt festzulegen, was die Testart eben nicht prüfen soll (Abschnitt „Was nicht").

In unseren Coachings hören wir oft Aussagen wie: „Wenn die Daten schon da sind, können wir sie doch gleich mitprüfen." Dies würde aber wieder zu den Effekten führen, die wir schon beim Testschnitt aufgezeigt haben: Die entsprechenden Tests werden unpräziser und können aufgrund vielfältiger Aspekte fehlschlagen, was wiederum die Suche nach der Fehlerursache erschwert. Zudem müssen zum Beispiel bei fachlichen Änderungen zahlreiche Tests angepasst werden. Letztendlich wird so die Reaktionsfähigkeit des Teams verlangsamt.

Es ist absolut sinnvoll, den Testschnitt im Циteststeckbrief zu referenzieren, jedoch nicht immer ausreichend. Hier zeigt sich auch der Unterschied zwischen Testschnitt und Testart: Es kann durchaus sinnvoll sein, zu einem bestimmten Testschnitt mehrere Testarten umzusetzen. Oftmals ist der Grund hierfür das gerade erwähnte präzisere Testen einzelner Aspekte, die aufgrund eines Testziels im Fokus stehen.

Abschnitt „Wie“

Dieser Abschnitt benennt die bei der Umsetzung verwendete Methodik und die eingesetzten Technologien. Beides sollte möglichst gut zu den Testzielen und zu dem zu prüfenden SUT passen. Des Weiteren helfen diese Informationen dabei, die Testart leichter den dazugehörigen implementierten Tests zuzuordnen. Falls es für eine Testart relevant ist, könnte man hier weitere Informationen zur konkreten Ausführung der Tests aufführen, wie die Teststufe oder das Testumfeld. Unsere Empfehlung ist es, dass so viele Testarten wie möglich lokal ausführbar sind und dass diese auch bevorzugt und regelmäßig lokal ausgeführt werden (siehe Shift-Left-Ansatz *[11]*).

Adaption des Teststeckbriefs

Die vier vorgestellten Abschnitte geben dem Teststeckbrief eine einfache, übersichtliche und aus unserer Sicht ausreichende Struktur. Oft kann der Verweis auf ein gutes Referenz-Beispiel, also einen konkreten Test aus dem Projekt, eine hilfreiche Ergänzung sein. Ein Team kann und sollte die Abschnitte zudem anders gestalten, wenn es dann mit dem Teststeckbrief besser arbeiten kann. Letztendlich ist es wichtig, dass die Struktur zum Team und seiner Vorgehensweise passt und innerhalb des Teams einheitlich angewandt wird. Gleiches gilt für die Darstellung des Teststeckbriefes. Das Team sollte prüfen, ob und wie es mit seinen gewohnten Tools den Teststeckbrief übersichtlich darstellen und komfortabel bearbeiten kann. Eine mögliche Darstellung zeigt *Abbildung 3* am Beispiel der Fachliche-Regeln-Tests.

Die Inhalte dieses Teststeckbriefes sind Beispiele zur Orientierung, wie ein solcher Teststeckbrief konkret gefüllt werden kann. Wir empfehlen, den Detailgrad ähnlich wie hier zu wählen und nur so viel zu beschreiben, wie für das gemeinsame Verständnis und zielführende Diskussionen benötigt wird.

Die Fachliche-Regeln-Tests könnten insbesondere in Bezug auf das „Wie“ auch anders gestaltet werden, solange das Testziel erfüllt wird und die Umsetzung für das Team mit seinen Skills einfach ist. Auch hier soll unser Beispiel nur aufzeigen, welche Informationen relevant sein könnten. Daher gehen wir an dieser Stelle nicht genauer auf die Details dieses konkreten Teststeckbriefes ein.

Test Canvas

Mit Hilfe von Teststeckbriefen kann also eine vollständige Dokumentation der einzelnen Testarten erreicht werden. Für einen noch besseren, visuellen Überblick über den gesamten Testansatz ist die naheliegende Fortführung, diese Steckbriefe mit den daraus referenzierten Informationen – den Testschnitten, den Risiken und dem zugrundeliegenden Kontext – auf einer Gesamtdarstellung zu vereinen: dem Test Canvas. Für unsere Beispielanwendung könnte der Test Canvas wie in *Abbildung 4* aussehen (die konkreten Inhalte sind hier bewusst ausgespart).

MITMACHEN UND AUTOR/IN WERDEN!

Ihr kennt euch in einem bestimmten Gebiet aus dem Java-Themenbereich bestens aus und möchtet als Autor/in euer Wissen mit der Community teilen?

Nehmt Kontakt zu uns auf und sendet euren Artikelvorschlag zur Abstimmung an redaktion@ijug.eu.

Wir freuen uns, von euch zu hören!

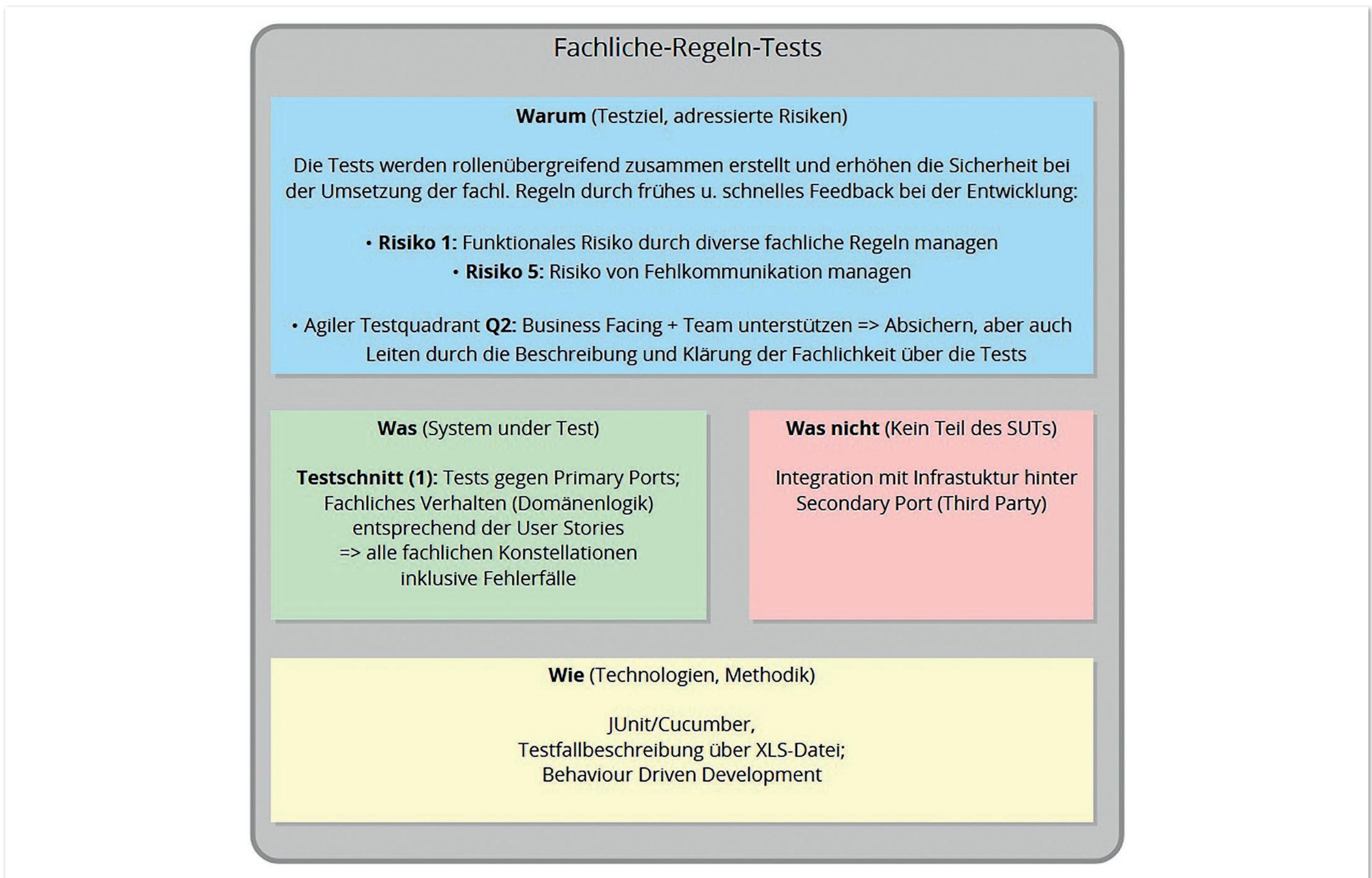


Abbildung 3: Beispiel für einen Teststeckbrief (© Peter Fichtner, Ralf Straßner)

Der Test Canvas zeigt die Zusammenhänge innerhalb des Testansatzes auf und hilft so, Wechselwirkungen zu verstehen. Es wird dadurch einfacher, Änderungen konsistent über den gesamten Testansatz durchzuziehen, also alle voneinander abhängigen Bestandteile anzupassen. Ändert sich zum Beispiel ein Risiko, so kann schnell ermittelt werden, welche Testarten hinterfragt werden müssen, und ob der dazugehörige Testschnitt noch passt. Der Test Canvas ist in erster Linie ein Arbeitsmittel. Es geht nicht darum, dort alles ausführlich „für die Nachwelt" zu dokumentieren, sondern vielmehr die essenziellen Informationen beim Gesamtblick auf das Testen (inklusive manueller Tests) kompakt zusammenzufassen, um das Verständnis im gesamten Team zu verbessern und Diskussionen zielgerichteter führen zu können.

Wie beim Konzept des Teststeckbriefs besteht auch beim Test Canvas kein Zwang zum formalen Erfüllen irgendeines Templates. Vielmehr soll das Team völlig frei im Test Canvas hilfreiche Informationen so anmerken, wie es für das Team praktikabel ist und dem Überblick nicht im Weg steht.

Beispielsweise können Informationen über einen Notizzettel (egal, ob auf einem physischen oder einem virtuellen Board) ergänzt werden, wie in *Abbildung 4* angedeutet. Relevante Informationen sind beispielsweise:

- Der Fokus liegt auf bestimmten Testarten (in Abbildung 4 als Stern dargestellt).
- Es gibt mehrere Testarten mit sich überschneidenden SUTs. Ist dies gewollt oder noch nicht dem Zielbild entsprechend?
- Es gibt Lücken, also ungetestete Teile des Systems. Sind diese, zum Beispiel aufgrund geringer Risiken, absichtlich vorhanden oder handelt es sich um technische Schulden?
- Eine Testart ist veraltet und wird gegebenenfalls durch eine andere Testart ersetzt.

Mit dem Test Canvas kann ein Vergleich zwischen dem Ist-Zustand, also der tatsächlichen Umsetzung, und dem Soll-Zustand des Testansatzes veranschaulicht werden. Das Team muss jedoch aufpassen, dass es den Test Canvas nicht überfrachtet. Hier bieten sich zwei alternative Varianten an: Weichen der Ist-Zustand und der gewünschte Soll-Zustand nur wenig voneinander ab, kann dies auf einem einzigen Test Canvas festgehalten werden. Ist der Soll-Zustand hingegen weit entfernt, bietet es sich an, diesen auf einem separaten Test Canvas eher im Sinne einer Vision festzuhalten und gegebenenfalls im Ist-Zustand zu kennzeichnen, welche Testarten bereits dem Soll-Zustand entsprechen.

Dokumentation für ein Legacy System

Wir empfehlen, den Test Canvas früh aufzubauen, im Idealfall direkt mit den ersten Überlegungen bezüglich Risiken und Testansatz oder spätestens parallel zur Umsetzung der ersten Tests in einem Projekt. Doch was kann ein Team tun, wenn es bereits seit Jahren ein Produkt (weiter-) entwickelt, ohne eine vergleichbare Dokumentation zu pflegen, und jetzt den Test Canvas einführen möchte? Ist es sinnvoll, den unter Umständen enormen Aufwand zu betreiben, den eine vollständige Dokumentation des Ist-Zustands mit sich bringen würde? Oder gibt es ein einfacheres Vorgehen, um zu einem Test Canvas zu kommen, der bereits eine hilfreiche Unterstützung darstellt?

Hierbei schlagen wir insbesondere in Bezug auf die Teststeckbriefe vor, klein anzufangen und schrittweise vorzugehen. Erfahrungsgemäß fällt für die nachträgliche Erstellung der Teststeckbriefe für alle bereits bestehenden Testarten der größte Aufwand an, weshalb sich Teams unter Umständen schwertun, hier zu einem für sie zufriedenstellenden vollständigen Stand zu kommen. Ein Ansatz ist es, zunächst den grundlegenden Kontext zu formulieren und die dazugehörigen Risiken zu erkennen und festzuhalten. Das Schreiben der Teststeckbriefe wird hingegen auf zwei bis drei bereits existierende Testarten reduziert, die das Team als die wichtigsten erachtet. Diese Teststeckbriefe werden dann mit dem Kontext und den Risiken verknüpft.

Dann sammelt das Team erst einmal Erfahrung mit der Verwendung des Test Canvas, indem es ihn beispielsweise bei Diskussionen zu einer Testart einsetzt und gegebenenfalls fehlende Informationen ergänzt. In der Folge wird die Dokumentation immer dann nachgezogen, wenn an noch nicht dokumentierten Testarten gearbeitet wird oder sich Änderungen am Kontext ergeben. So entwickelt das Team auch ein besseres Verständnis des Gesamtbilds und der Zusammenhänge und kann, wenn es dies als nötig und hilfreich erachtet wird, ein Zielbild (Soll-Zustand) erarbeiten, dem es sich auch wieder schrittweise annähert.

Fazit

Wir haben in diesem Artikel Schritte aufgezeigt, um einen ganzheitlichen Testansatz zu finden. Dieser legt den Fokus auf die Testarten, die die am höchsten priorisierten Risiken adressieren. Dadurch tragen die Testarten zu einem hohen Maß zum Projekterfolg bei. Dabei gibt es kein Patentrezept für die Lösung, sondern jeder Kontext ist individuell. Durch die schrittweise Herangehensweise wird aber sichergestellt, dass das Team sich die wichtigsten Fragen stellt, seine Entscheidungen bewusst trifft und seine Kapazität nicht für Tests mit einem niedrigen Nutzen vergeudet. Um den Aufwand für Tests zu reduzieren und in der Umsetzung zielgerichtet und präzise zu testen, haben wir zudem aufgezeigt, welche Aspekte beim Anwendungsschnitt relevant sind und wie wir den Testschnitt so wählen können, dass wir von einem guten Anwendungsschnitt profitieren, diesen fortführen und die Tests ideal auf das Testziel zuschneiden.

Die im Artikel erläuterten Testschnitte sind unser empfohlener Startpunkt. Ein größerer Schnitt vermischt Verantwortlichkeiten und bringt immer Nachteile mit sich. Dies sollte nur aus gutem Grund und bewusst getan werden, zum Beispiel weil der Anwendungsschnitt oder eingesetzte Technologien/Frameworks es nicht ohne großen Aufwand ermöglichen. Zudem werden viele Tests auch einen noch kleineren Schnitt aufweisen, der noch präziser Teilaspekte abdeckt. Hier sollte vermieden werden, Implementierungsdetails im Test zu zementieren, zum Beispiel zu kleinteilige Unit-Tests in der Domäne durchzuführen, obwohl hier keine störenden Abhängigkeiten vorhanden sind.

Bei der Erarbeitung und fortwährenden Weiterentwicklung eines Testansatzes werden viele Entscheidungen getroffen. Wir haben daher mit den Teststeckbriefen und dem Test Canvas Möglichkeiten der Dokumentation aufgezeigt, die dem Team helfen, den Überblick über ihren Testansatz zu wahren. Die Teststeckbriefe helfen dem Team auch dabei, ihre Testarten möglichst klar zu definieren, besonders in Hinblick auf das *System under Test*. Der Test Canvas wiederum visualisiert die Zusammenhänge zwischen dem Kontext, den dazugehörigen Risiken, dem Testschnitt und den Teststeckbriefen mit den dort formulierten Testzielen. Dadurch kann im Team leichter

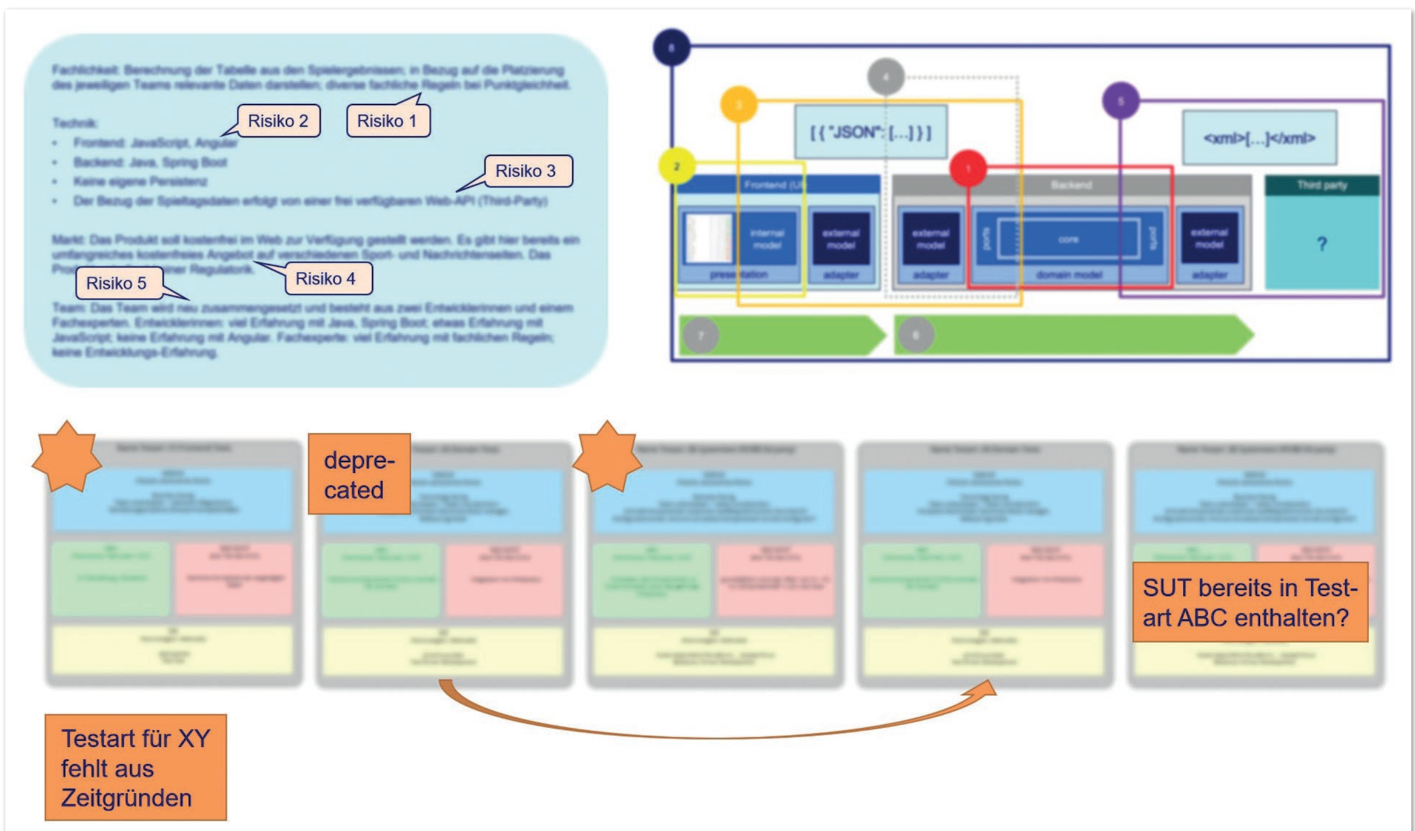


Abbildung 4: Möglicher Test Canvas für die Beispielanwendung (© Peter Fichtner, Ralf Straßner)

ein gemeinsames Verständnis des Testansatzes erreicht werden. Mit einem guten Testansatz lassen sich die Komplexität der Tests und die Aufwände für Anpassungen an Tests reduzieren. Durch Tests mit einem schnellen, frühen und präzisen Feedback steigert sich die Häufigkeit von Testausführungen und damit die gesamte Entwicklungsgeschwindigkeit.

Quellen

[1] Alistair Cockburn (2003): Hexagonal architecture (*https://alistair.cockburn.us/hexagonal-architecture/*)

[2] Devopedia (2022) „Cohesion vs Coupling" (*https://devopedia.org/cohesion-vs-coupling*)

[3] Jeffrey Palermo (2008) The Onion Architecture (*https://jeffreypalermo.com/tag/onion-architecture/*)

[4] Robert C. Martin (2017) Clean Architecture: A Craftsman's Guide to Software Structure and Design

[5] Eric Evans (2003) Domain-Driven Design: Tackling Complexity in the Heart of Software

[6] Vaughn Vernon (2013) Implementing Domain-Driven Design

[7] Martin Fowler (2006): Object Mother (*https://martinfowler.com/bliki/ObjectMother.html*)

[8] Nat Pryce (2007): Test Data Builders: an alternative to the Object Mother pattern (*http://www.natpryce.com/articles/000714.html*)

[9] Wikipedia zu „Datenkapselung" (*https://de.wikipedia.org/wiki/Datenkapselung_(Programmierung)*)

[10] pact foundation zu „organisatorischem Einfluss" (*https://docs.pact.io/getting_started/what_is_pact_good_for#what-is-pact-good-for*)

[11] Wikipedia zu „Shift-Left-Ansatz" (*https://de.wikipedia.org/wiki/Shift-Left-Ansatz*)

Peter Fichtner

peter.fichtner@atruvia.de

Peter Fichtner ist seit 1995 in der Atruvia AG als Softwareentwickler, Softwarearchitekt und seit 2013 als Technical Agile Coach tätig. Neben dem Ausarbeiten und Durchführen von Workshops und Schulungen zu Entwicklungsthemen beschäftigt er sich am liebsten im Rahmen von Samman Technical Coaching damit, Teams in ihrem Entwicklungsvorgehen auf die nächste Stufe zu bringen. Er programmiert seit 1999 in Java und mag es, wenn Dinge automatisiert sind und so Fehler frühzeitig gefunden werden können. Gerade deshalb liebt und lebt er Test Driven Development und Consumer-Driven Contract Tests.

Ralf Straßner

ralf.strassner@atruvia.de

Ralf Straßner ist seit 2011 bei der Atruvia AG und hat in dieser Zeit hauptsächlich als Full-Stack-Entwickler im Java- und JavaScript-Umfeld Erfahrung gesammelt. Seit 2013 ist er als Coach für agiles Softwareengineering (ASE) im Unternehmen aktiv und unterstützt Teams bei den Herausforderungen des Projektalltages, um Produkte mit einer guten inneren und äußeren Qualität umzusetzen. Dabei setzt er seit 2021 vor allem auf Samman Technical Coaching.

10 JAHRE JAVALAND

JavaLand

www.javaland.eu

JAVALAND 2024 VERPASST?

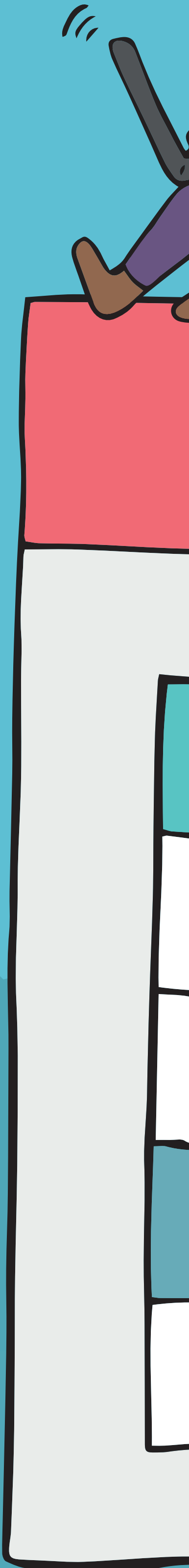
ALLE ON-DEMAND-ANGEBOTE IM TICKETSHOP

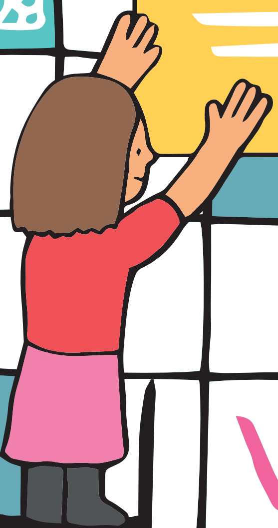
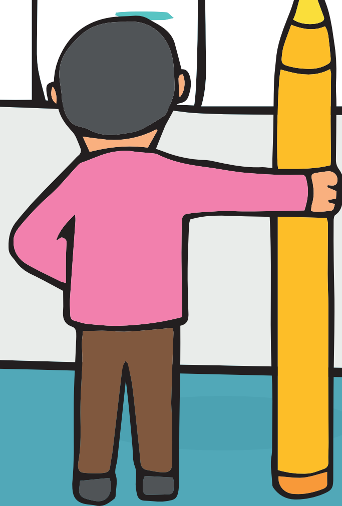
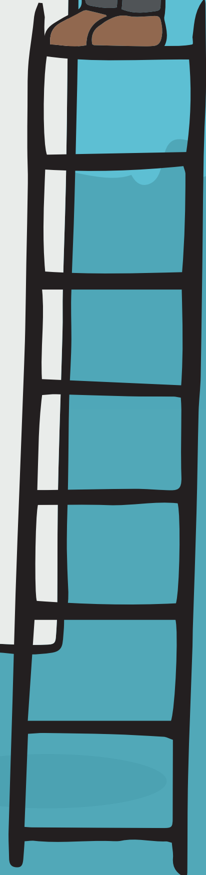
JETZT ON-DEMAND-TICKET BUCHEN UND VORTRAGSAUFZEICHNUNGEN ANSCHAUEN!

Präsentiert von: iJUG Verbund Heise Medien DOAG Veranstalter: JavaLand

Familienplaner 2.0 – Wie KI und Function Calling den Alltag revolutionieren

Alexander Culum



Im hektischen Alltag einer Familie den Überblick zu behalten, kann zur Herausforderung werden: Wann kommen die Kinder von der Schule nach Hause, wann stehen wichtige Termine wie Arztbesuche oder Elternabende an, wann kann die ganze Familie gemeinsam einkaufen gehen? Oft bedeutet das, mühsam den Kalender zu durchforsten und Stundenpläne durchzugehen. Was wäre, wenn eine KI dies automatisch erledigen könnte?

Stellen Sie sich vor, Sie könnten Ihren Alltag mit einer KI organisieren, die alles weiß: Wann Ihre Kinder zu Hause sind, wann der nächste Arzttermin ansteht und welche Lücken in Ihrem Zeitplan noch für ein wenig Freizeit offenstehen. All dies ohne manuelles Suchen im Kalender oder Stundenplan – einfach durch einen sprachlichen Befehl.

Dank der Integration von LangChain4j und OpenAI's Function Calling *[1]* sind solche Anwendungsfälle leicht zu realisieren. In diesem Artikel erfahren Sie, wie Sie mit LangChain4j *[2]* Tools kombinieren, um komplexe Anfragen automatisch zu beantworten. Wir zeigen praktische Anwendungsfälle, die Ihnen den Alltag erleichtern, und erklären, wie die Technologie im Hintergrund funktioniert.

Was ist Function Calling und wie funktioniert es?

Function Calling ist eine leistungsstarke Fähigkeit des OpenAI-API, die einem Large Language Model (LLM) wie GPT-4o ermöglicht, auf externe Tools und Funktionen zuzugreifen, um Anfragen zu beantworten, die Daten aus verschiedenen Quellen erfordern. Der Prozess läuft wie folgt ab:

- **Anfrage**: Der Benutzer stellt eine Frage oder Aufgabe, wie zum Beispiel: „Wann sind meine Kinder am Dienstag beide zu Hause?"
- **Toolauswahl durch das LLM**: Das LLM erhält Informationen über verfügbare Tools und entscheidet, welche Funktionen mit welchen Parametern aufgerufen werden müssen.
- **Ausführung durch den Client**: Der Client (LangChain4j) erhält die vom LLM formulierte *ToolExecutionRequest* und führt den eigentlichen Funktionsaufruf durch, wie zum Beispiel eine Abfrage an ein Google-Calendar-API.
- **Rückmeldung an das LLM**: Die Ergebnisse der ausgeführten Funktionen werden an das LLM zurückgesendet, das diese Daten zur Erstellung der endgültigen Antwort nutzt und an den Benutzer zurückgibt.

Dieser iterative Prozess sorgt dafür, dass die Fähigkeiten des Modells durch die Einbindung von Tools erweitert werden, sodass Aufgaben und Fragen, die normalerweise mehrere manuelle Schritte erfordern, automatisiert abgewickelt werden können.

Beispiel: Calculator Assistent (aus dem LangChain4j Repository)

In unserem ersten Beispiel soll die Frage beantwortet werden: Was ist die Quadratwurzel der Anzahl der Zeichen in der Wortfolge „Das alles ist viel einfacher mit Function Calling"?

Das Originalbeispiel befindet sich im LangChain4j Repository *[3]*. Unser abgewandeltes Beispiel findet sich in dem Artikel-Repository *[4]*.

Wer sich schon mit LLMs wie ChatGPT beschäftigt hat, weiß, dass solche Fragen schwierig für ein LLM zu beantworten sind:

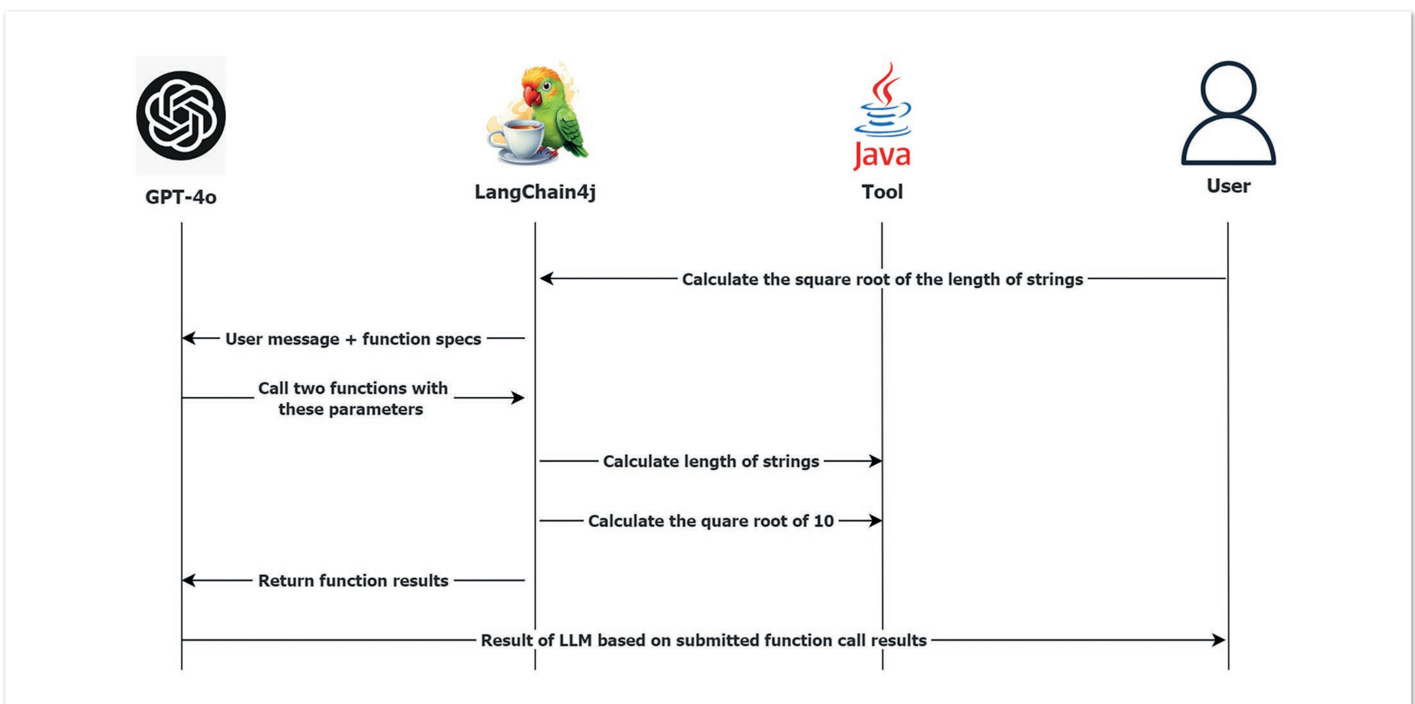


Abbildung 1: Aufrufsequenz für die Beantwortung der Frage: Was ist die Quadratwurzel der Anzahl der Zeichen in der Wortfolge „Das alles ist viel einfacher mit Function Calling"? (© Alexander Culum)

```
interface Assistant {

      @SystemMessage(
      """
      You are a helpful audio chat bot.
      Answer clearly and only add relevant information, use no fillers.
      Keep answers concise, except when asked to go into detail.
      Always use the declared functions to get the correct date and the correct day of the week.
      """)
      Result<String> chat(String userMessage);

}
```

Listing 1: Das Assistant-Interface mit der SystemMessage. LangChain4j erstellt die Implementierung zur Laufzeit.

```
static class Calculator {

      @Tool("Calculates the length of a string")
      int stringLength(String s) {
            System.out.println("Called stringLength with s='" + s + "'");
            return s.length();
      }

      @Tool("Calculates the square root of a number")
      double sqrt(int x) {
            System.out.println("Called sqrt with x=" + x);
            return Math.sqrt(x);
      }
}
```

Listing 2: Per Annotation können LangChain4j neben Methodennamen und Parametern weitere Informationen zur Aufgabe der Methode mitgeteilt werden.

- LLMs verarbeiten Eingaben zu Token und kennen keine einzelnen Buchstaben, deshalb ist die Antwort auf die einfache Frage „Wie viele Buchstaben „R“ sind in dem Wort Erdbeere?“ für eine LLM schwierig zu beantworten.
- LLMs basieren auf statistischer Textvervollständigung und speziell die ersten Versionen waren sehr schlecht beim Lösen selbst einfachster Mathematik- und Logikaufgaben.

Für Programmierer und Programmiererinnen sind diese Aufgaben hingegen sehr einfach zu lösen. Wir können also dem LLM Hilfestellung geben, indem wir auf zwei Funktionen hinweisen, die diese Aufgaben implementieren und die wir aufrufen können.

Das LLM versteht den Kontext und kann die Eingabeparameter für die Funktionsaufrufe ermitteln. Nachdem wir die Ergebnisse in einem weiteren Aufruf übermittelt haben, kann das LLM mit seinem Sprachverständnis die endgültige Antwort zusammenstellen und zurückgeben.

Abbildung 1 zeigt den gesamten Ablauf der Aufrufe der Komponenten zur Beantwortung der Frage. LangChain4j orchestriert diese Aufrufe und gibt dem LLM die Möglichkeit, die Teile entsprechend zusammenzusetzen.

LangChain4j erlaubt uns, ein Interface zu definieren, das wir mit Annotationen anreichern können, beispielsweise einer *SystemMessage*, die die Rolle des Systemnutzers (im Gegensatz zum User) per Prompt beschreibt. Die Rückgabe könnte auch nur ein String sein, bei *Result<String>* bekommen wir weitere Informationen zur Kommunikation, zum Beispiel die Aufrufsequenz mit Ein- und Ausgabewerten der Tools. Für unser Beispiel ist der ganze *Assistant* in *Listing 1* spezifiziert.

Wir definieren zwei Tools, das heißt zwei Funktionen, die dem LLM bekannt gemacht werden und die wir aufrufen können *(siehe Listing 2)*. Das LLM wird unsere Anfrage analysieren, die zur Verfügung stehenden Tools prüfen und uns Aufrufinstruktionen mit Funktionsaufrufen inklusive Parametern übergeben.

Schließlich müssen wir den LangChain4j-Client noch konfigurieren, unter anderem mit dem OpenAI-API-Key, dem zu nutzenden Modell (GPT-4o, GPT-4o-mini, etc.) und weiteren Eigenschaften (Logging etc.).

Listing 3 zeigt diese Konfiguration sowie den Aufruf des *Assistant* mit der *UserMessage*, also unserer Ausgangsfrage. Das gesamte Beispiel findet sich in dem GitHub-Repository zu dem Artikel *[4]*.

Für unser Beispiel sehen die *ToolExecutionRequests* wie in *Listing 4* dargestellt aus.

Hier kann man also gut erkennen, dass während der Verarbeitung der Anfrage auf die beiden zur Verfügung stehenden Funktionen zugegriffen wurde und das LLM diese Aufgaben nicht eigenständig erledigen konnte oder wollte.

KI für den Alltag – der intelligente Kalender

Wir wollen nun Tools nutzen, um dem LLM die Möglichkeit zu geben, auf drei externe Datenquellen zuzugreifen:

1. Google Calendar der Familie
2. Stundenplan des Sohnes
3. Stundenplan der Tochter

```java
@Override
public void run(String... args) throws Exception {
     ChatLanguageModel model = OpenAiChatModel.builder()
                 .apiKey(openAIKey)
                 .modelName(GPT_4_O_MINI)
                 .strictTools(true)
                 .logRequests(true)
                 .logResponses(true)
                 .build();

     Assistant assistant = AiServices.builder(Assistant.class)
                 .chatLanguageModel(model)
                 .tools(new Calculator())
                 .chatMemory(MessageWindowChatMemory.withMaxMessages(10))
                 .build();

     String question = """
           Was ist die Quadratwurzel der Anzahl der Zeichen
           in der Wortfolge "Das alles ist viel einfacher
           mit Function Calling"?
     """

     Result<String> answer = assistant.chat(question);

     System.out.println(answer.toolExecutions());
}
```

Listing 3: LangChain4j muss neben den API-Keys zu OpenAI auch unsere Funktionen kennen.

```
"request": {
 "id": "call_WQDOq6Nv4szoDf6E88Yixp02",
 "name": "stringLength",
 "arguments": "{\"s\":\Das alles ist viel einfacher mit Function Calling\"}
},
"result": "49"

"request": {
 "id": "call_MTDC2Fdh05g829tkukXlOCGv",
 "name": "sqrt",
 "arguments": "{\"x\":49}"
},
"result": "7.0"
```

Listing 4: Result-Objekt der LangChain4j-Interaktion. Anfragen und die jeweiligen Antworten der Funktionen können einfach evaluiert werden.

Abbildung 2 zeigt die Interaktionen zwischen dem LLM, LangChain4j und den externen Datenquellen. Wie in unserem ersten Beispiel orchestriert LangChain4j die Aufrufe, indem Anfragen des LLMs per Function Calling an die externen Datenquellen weitergeleitet werden.

Anforderung 1 lässt sich mit dem *Google-Calendar-API* einfach realisieren, wir haben es in dem Artikel-Repository *[4]* mit Beispielterminen „gemockt".

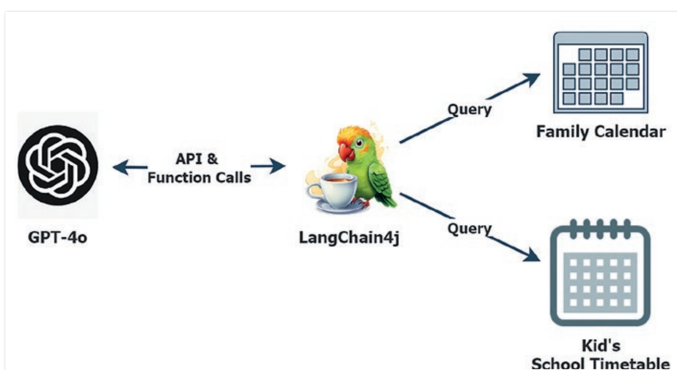


Abbildung 2: Übersicht der Interaktionen: Das LLM „spricht" über LangChain4j mit den externen Datenquellen. (© Alexander Culum)

Anforderungen 2 und 3 müssen wir manuell implementieren, *Abbildung 3* zeigt, wie mit ChatGPT eine entsprechende Datenstruktur und die Befüllung der echten Stundenplandaten in ein paar Minuten erledigt werden können.

Auf Basis der Ideen von ChatGPT entstand schnell die Datenstruktur aus *Listing 5*.

ChatGPT hat zunächst *DayOfWeek* nicht in den Entry gepackt, doch diese Entscheidung hat der Autor „überschrieben", also im Dialog die gewünschte Struktur angepasst und ChatGPT hat diese neue Struktur dann in weiteren Antworten entsprechend übernommen.

Die Datenstruktur konnte ChatGPT problemlos aus dem Bild erstellen, wie aus *Listing 6* ersichtlich ist.

Die API sollte für das LLM sinnvoll zu nutzen sein. Beschreibungen (mittels *@Tool*) sowie sinnvolle Methoden und Parameternamen können dem LLM helfen zu entscheiden, ob und wie die Methode zur Ermittlung der benötigten Daten aufgerufen werden soll. *Listing 7* zeigt alle Methoden, die dafür implementiert wurden.

In *Listing 8* wird schließlich die *SystemMessage* für den LangChain4j-

Erstelle eine Datenstruktur in Java für folgenden Stundenplan. Es sollen Aufrufe möglich sein wie:
* gebe alle Stunden des Tages mit den Zeiten (als LocalDates) zurück
* gebe alle Stunden der Woche mit Tag und Zeiten zurück

Abbildung 3: ChatGPT-Prompt zur Erstellung der Datenstruktur für die Stundenpläne (© Alexander Culum)

```
public class WeeklyRecurringEntry {

     private final DayOfWeek day;
     private final String lesson;
     private final LocalTime start;
     private final LocalTime end;
```

Listing 5: Vorschlag von ChatGPT für die Datenstruktur eines Stundenplans

Assistant übergeben. Nun fehlt nur noch die Konfiguration und der *Assistant* kann genutzt werden.

Für den Google Calendar nutzen wir eine sehr ähnliche Struktur zu unserer Stundenplan-Datenstruktur, in die wir die komplexeren Objekte des Google-Calendar-API-Events transformieren.

Das API für unseren *CalendarService* besteht nur aus der Methode *getNextCalendarEvents* mit der Annotation *@Tool('Returns next events from calendar')*. Diese Beschreibung, zusammen mit den jeweiligen Datentypen (String für *description*, LocalDate für *start* und end) genügt der LLM, um die Funktionsrückgaben korrekt zu interpretieren.

```
// Monday schedule
schedule.add(new WeeklyRecurringEntry("Deutsch", DayOfWeek.MONDAY, LocalTime.of(8, 15), LocalTime.of(9, 0)));
schedule.add(new WeeklyRecurringEntry("Deutsch", DayOfWeek.MONDAY, LocalTime.of(9, 0), LocalTime.of(9, 45)));
schedule.add(new WeeklyRecurringEntry("SU", DayOfWeek.MONDAY, LocalTime.of(10, 15), LocalTime.of(11, 0)));
```

Listing 6: Befüllung der Datenstruktur anhand eines Fotos des Stundenplans durch ChatGPT

```
@Tool("Return school schedule for the whole week")
public List<WeeklyRecurringEntry> getRecurringSchoolScheduling() {
     return schedule;
}

@Tool("Return school schedule for a specific day of the week")
public List<WeeklyRecurringEntry> getSchoolSchedulingForThisDay(DayOfWeek dayOfWeek) {
     return schedule.stream().filter(it -> it.getDay() == dayOfWeek).collect(Collectors.toList());
}

@Tool("Return current day of week")
public DayOfWeek getCurrentDayOfWeek() {
     return LocalDate.now().getDayOfWeek();
}
```

Listing 7: Bereitstellung der API für das Function Calling von LangChain4j

```
interface CalendarAssistant {

     @SystemMessage(
     """
     You are a helpful audio chat bot.
     Answer clearly and only add relevant information, use no fillers.
     Keep answers concise, except when asked to go into detail.
     Always use the declared functions to get the correct date and the correct day of the week.
     If asked for events, always think of querying all relevant date, both school schedules and the family calendar.
     """)
     String chat(String userMessage);

}
```

Listing 8: Chat-Interface mit SystemMessage für LangChain4j

	Montag	Dienstag	Mittwoch	Donnerstag	Freitag	Samstag	Sonntag
09:00						Zahnarzttermin	
10:00					Fußballspiel (Heimspiel)		
11:00							Familienbrunch bei Oma und Opa
12:00							
13:00							
14:00							
15:00		Reitstunde		Klavierunterricht			
16:00							
17:00	Fußballtraining		Tanzunterricht				
18:00							
19:00					Elternabend in der Schule		
20:00							
21:00							

Abbildung 4: Darstellung des Beispiel-Familienkalenders für eine Woche (© Alexander Culum)

Anwendungsbeispiele für die Stundenplan und Kalender-Funktionen

Wir nutzen für die Beispiele einen Google-Calendar-API-Mock, der Termine, wie in *Abbildung 4* dargestellt, in eine Datenstruktur einfügt und zurückgibt.

Außerdem bieten wir Methoden zu den Fragen „Was ist das aktuelle Datum?" und „Welcher Wochentag ist heute?" an, da wir nicht davon ausgehen können, dass das LLM diese Informationen für unsere Zeitzone kennt.

Wann sind meine Kinder am Dienstag zu Hause?

Sie möchten wissen, wann beide Kinder zu Hause sind, um eine gemeinsame Aktivität zu planen.

Die Anfrage („Wann sind meine Kinder am Dienstag zu Hause?") wird an das LLM gestellt.

Das LLM identifiziert die relevanten Datenquellen: den Stundenplan-Service der Kinder und den Google Calendar für weitere Aktivitäten.

Function Call: Das LLM formuliert einen Funktionsaufruf an den Stundenplan-Service und an Google Calendar, um die relevanten Zeitfenster zu prüfen.

Der Client führt die beiden Anfragen aus und übermittelt die Ergebnisse an das LLM, das die Daten kombiniert und folgende Antwort gibt: „Am Dienstag sind beide Kinder ab 15:00 Uhr zu Hause."

Wann ist Emilia Montag zu Hause?

Das LLM gibt folgende *ToolExecutionRequests* zurück:

- *getDateForNextDayOfWeek(MONDAY)*
- *getSchoolSchedulingForEmiliaForThisDay(MONDAY)*

Nach der Rückgabe der Toolaufrufergebnisse gibt das LLM die korrekte Antwort zurück: Emilia ist am Montag um 16:00 Uhr zu Hause, nachdem ihre Theater AG endet.

Können wir Dienstag um 15 Uhr gemeinsam einkaufen gehen?

Hier werden folgende Aufrufe benötigt:

- *getRecurringSchoolSchedulingEmilia()*
- *getRecurringSchoolSchedulingSimon()*
- *getNextCalendarEvents()*

Nach der Übergabe der Resultate gibt das LLM auch hier eine menschliche verständliche, korrekte Antwort zurück: Am Dienstag, den 15. Oktober 2024, hat Emilia von 13:50 bis 14:35 Uhr eine Betreuung und Simon hat von 12:00 bis 13:30 Uhr Unterricht.

Um 15 Uhr hat Emilia allerdings eine Reitstunde, die von 15:00 bis 16:00 Uhr dauert. Daher können wir um 15:00 Uhr nicht gemeinsam einkaufen gehen.

Können wir am Mittwoch gemeinsam einkaufen gehen? Wenn nicht, warum nicht?

Die Anfragen funktionieren nicht immer, häufig werden Falschannahmen halluziniert, wie zum Beispiel bei dieser Antwort: Am Mittwoch, den 16. Oktober 2024, können wir nicht gemeinsam einkaufen gehen, weil sowohl Emilia als auch Simon schulische Verpflichtungen haben:

Emilia hat Unterricht von 08:00 bis 14:35 Uhr.
Simon hat Unterricht von 08:15 bis 13:30 Uhr.
Zusätzlich hat Simon um 17:00 Uhr Tanzunterricht.

Es ist nirgends angegeben und auch falsch, dass der Tanzunterricht Simon zuzuordnen ist – hier hat das LLM also einfach halluziniert.

Wann ist mein nächster Zahnarzttermin?

Sie möchten wissen, wann der nächste Zahnarzttermin ansteht, ohne den Kalender manuell durchsuchen zu müssen.

Das LLM versteht die Anfrage und erkennt, dass der Google Calendar nach Terminen mit dem Stichwort „Zahnarzt" durchsucht werden sollte.

Function Call: Das LLM gibt einen *ToolExecutionRequest* für das Google-Calendar-API aus. Der Client führt diesen Aufruf aus und die Ergebnisse werden an das LLM zurückgesendet. Das LLM kann nun anhand der Beschreibungen der einzelnen Termine entscheiden, ob es sich um einen Zahnarzttermin handelt.

Das LLM formuliert die Antwort, zum Beispiel: „Dein nächster Zahnarzttermin ist am Donnerstag um 10:00 Uhr."

Weitere Anwendungsfälle für die Stundenplan- und Kalenderintegration

Wann haben die Kinder Zeit für ein gemeinsames Abendessen?
Das LLM nutzt die Schulstundenpläne der Kinder und Kalenderdaten, um das beste Zeitfenster für ein gemeinsames Abendessen zu finden.

Gibt es in den nächsten zwei Wochen freie Wochenenden für die Familie?
Das LLM analysiert alle Verpflichtungen im Google Calendar und in den Stundenplänen der Kinder, um freie Wochenenden zu identifizieren.

Welche beruflichen Termine überschneiden sich mit Schulaktivitäten der Kinder?
Das Modell vergleicht die beruflichen Meetings des Nutzers mit den Schulzeiten der Kinder und gibt eventuelle Konflikte zurück.

Wann kann ich die Kinder nächste Woche zur Schule bringen und gleichzeitig meinen Terminplan einhalten?
Hier greift das LLM auf den Kalender und die Stundenpläne zu und findet passende Zeitfenster.

An welchen Nachmittagen haben beide Kinder nächste Woche für Sportaktivitäten Zeit?
Die Schulzeiten der Kinder werden kombiniert, um freie Nachmittage zu finden, die für Sportaktivitäten genutzt werden können.

Gibt es anstehende schulische Veranstaltungen, die mit unseren Familienplänen kollidieren?
Das LLM kombiniert Daten aus dem Google Calendar mit den Schulterminen der Kinder und prüft mögliche Konflikte.

Wann ist der nächste Elternabend und überschneidet er sich mit meinen beruflichen Meetings?
Das Modell durchsucht den Kalender nach „Elternabend"-Terminen und prüft mögliche zeitliche Überschneidungen.

Wann ist ein optimaler Zeitpunkt für einen Familienausflug?
Das LLM sucht nach einem freien Wochenende ohne berufliche oder schulische Verpflichtungen und schlägt den besten Zeitpunkt vor.

Nutzung weiterer potenzieller APIs

Hier sind noch weitere Möglichkeiten für die Integration zusätzlicher APIs angegeben. Natürlich sind hier der Fantasie keine Grenzen gesetzt. Es muss dem Client nur möglich sein, die APIs nutzen zu können.

Fitnessplanung basierend auf Zeitplan und Wettervorhersage
Das LLM kombiniert die Wettervorhersage (über ein Wetter-API) mit dem persönlichen Zeitplan, um die beste Zeit zum Joggen vorzuschlagen.

Urlaubsvorschlag basierend auf Zeitplan und Budget
Die Kalenderdaten werden genutzt, um mögliche freie Zeiträume zu identifizieren, und ein Reisebuchungs-API liefert passende Angebote innerhalb des Budgets.

Verwaltung von Abonnements für Streaming-Dienste
Über ein Finanz-API werden aktuelle Streaming-Abonnements analysiert und Empfehlungen zur Kündigung wenig genutzter Dienste gegeben.

Gesundheitscheck-Erinnerungen
Das LLM greift auf Gesundheitsdaten zu und plant regelmäßige Gesundheitschecks und tägliche Medikamentenerinnerungen.

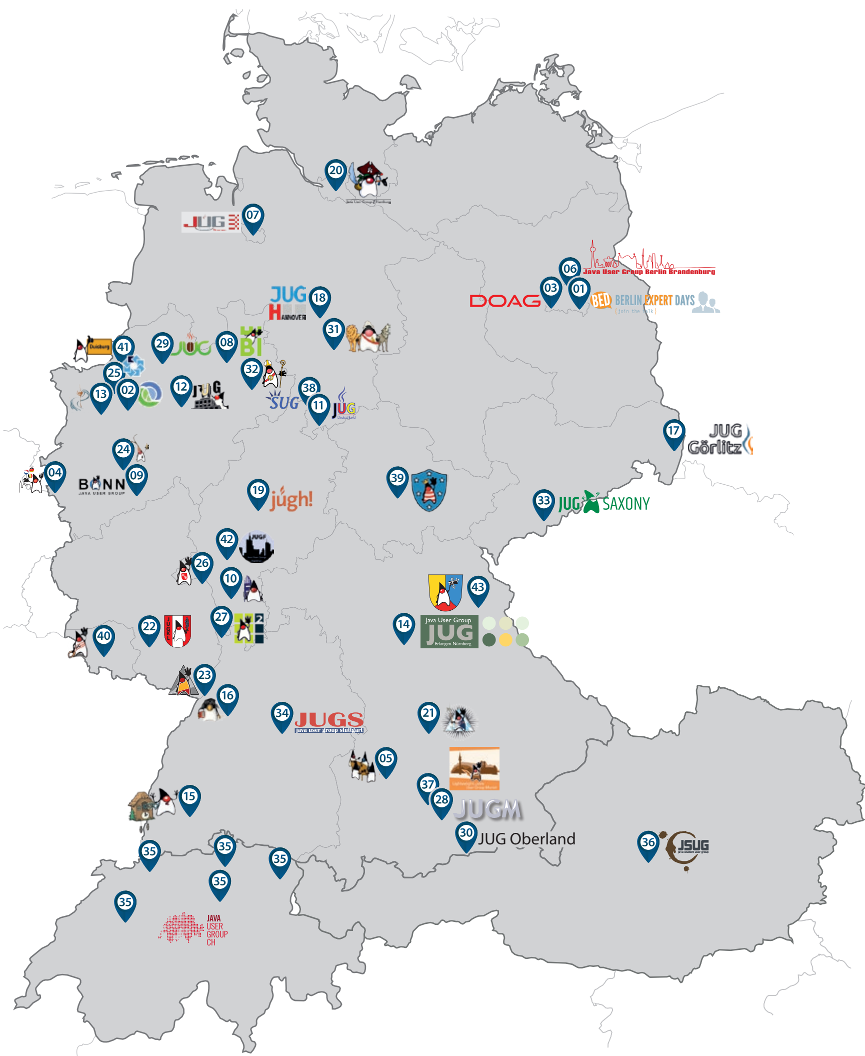
Mit diesen Funktionen wird das Planen und Organisieren des Alltags optimiert: LangChain4j und OpenAI's Function Calling ermöglichen es Ihnen, komplexe organisatorische Aufgaben automatisiert abzuwickeln, während die KI eigenständig entscheidet, welche Informationen benötigt werden und wie diese zu verarbeiten sind. Auf diese Weise werden Familienaktivitäten und persönliche Verpflichtungen nahtlos koordiniert – und das alles nur durch einen Sprachbefehl.

Quellen

[1] OpenAI Function Calling: *https://platform.openai.com/docs/guides/function-calling*
[2] LangChain4j Tools: *https://docs.langchain4j.dev/tutorials/tools/*
[3] LangChain4j Repository mit Beispielen: *https://github.com/langchain4j/langchain4j-examples/blob/main/other-examples/src/main/java/ServiceWithToolsExample.java*
[4] Artikel-Repository: *https://github.com/ice09/living-timetable*

Alexander Culum
alexander.culum@web.de

Alexander Culum ist seit vielen Jahren als Softwareentwickler und -architekt tätig und ist auf Java und Spring in der Enterprise-Entwicklung für Banken und Pharmaunternehmen spezialisiert. Neben seiner beruflichen Expertise engagiert er sich in der Java User Group Frankfurt (jugf.de) und unterstützt den Wissensaustausch in der Community. Außerdem bietet er die AG „KI4Kids" an der örtlichen Grundschule an, in der er Grundschülerinnen und Grundschülern die Welt der Künstlichen Intelligenz auf kreative Weise näherbringt.



Mitglieder des iJUG

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

www.ijug.eu

Impressum

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

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

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

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

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

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

Bildnachweis:
Titel: Bild © Mehemmed
https://stock.adobe.com
S. 10 + 11: Sloth McSloth
https://stock.adobe.com
S. 14 + 15: Bild © prachid
https://stock.adobe.com
S. 28 + 29: Bild © gustavo-fring
https://pexels.com
S. 34 + 35: Bild © Designed by fullvector
https://freepik.com
S. 46 + 47: Bild © The Little Hut
https://stock.adobe.com
S. 50 + 51: Bild © Alice
https://stock.adobe.com
S. 58 + 59: Bild © vectorjuice
https://freepik.com
S. 70 + 71: Bild © Designed by Freepik
https://freepik.com

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

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

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

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

Inserentenverzeichnis

cronn GmbH	S. 9
DOAG e. V.	S. 57, U 3
iJUG e.V.	S. 13, S. 27, S. 33, S. 65
JavaLand GmbH	U 2, S. 69, U 4

EARLY BIRD BIS 20.05.

#CLOUDLAND2025

DAS CLOUD NATIVE FESTIVAL

1. – 4. JULI 2025 • IM HEIDEPARK IN SOLTAU

CloudLand
WWW.CLOUDLAND.ORG

Das Event der Deutschsprachigen Cloud Native Community

GROßES COMMUNITY-PROGRAMM!

#JAVALAND

AM NÜRBURGRING

JAVALAND

1. - 3. APRIL 2025

Die JavaLand lädt Java-Fans aus der ganzen Welt ein, gemeinsam zu lernen, ihr Wissen rund um Java zu vertiefen, Know-how auszutauschen und Kontakte zu knüpfen.

Freut euch auf zwei Konferenztage und einen Schulungstag.

Jetzt zum günstigen Frühbuchertarif buchen: **www.javaland.eu**

Mitglieder des iJUG e.V. sparen weitere 25 %: **www.ijug.eu**

Präsentiert von: iJUG Verbund Heise Medien DOAG Veranstalter: JavaLand