Java aktuell

www.ijug.eu

Ausgabe 04/2025

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

Java 25
18 neue JEPs im nächsten Release

Von grünen Systemen
Nachhaltigkeit systematisch denken

Nichts zu verbergen?
Überwachung, Privatsphäre und digitale Selbstbestimmung

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

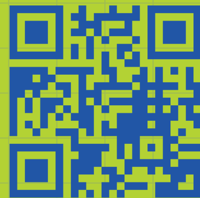
WWW.DEVLAND.EU #DEVLAND

12. + 13. MÄRZ 2026

DEVLAND

THE NEXT GENERATION OF DEVELOPERS

EUROPA-PARK IN RUST



ARCHITECTURE AVENUE
AI VALLEY
CLOUD CLIFFS
DATA DOCKS
CAREER COAST

nur 95 €
(ZZGL. MWST.)

STARTER PASS

Liebe Leserinnen und Leser,

Java bleibt in Bewegung – und wir gleich mit. In dieser Ausgabe tauchen wir nochmals in die Welt von Java 24 ein: Welche Features bringen echten Mehrwert? Welche Änderungen erleichtern uns den Alltag? Und wo lohnt es sich, genauer hinzuschauen? Außerdem werden für Java 25 bereits 18 neue JEPs angekündigt.

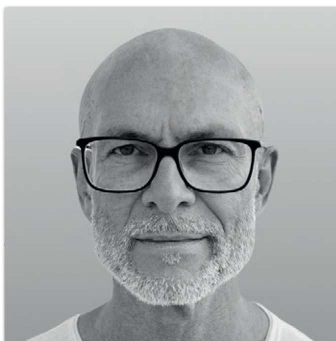
Das Java-Tagebuch liefert wieder spannende Einblicke in aktuelle Entwicklungen und Trends. Andreas Monschau setzt mit Teil 8 seiner Goldenen Regeln den Schlusspunkt einer Serie, die manchen von euch sicher zum Schmunzeln – und zum Nachdenken – gebracht hat. Marcel Vielsack wirft einen Blick auf Javas Ecken und Kanten und zeigt, warum auch kleine Eigenheiten große Auswirkungen haben können. Und wer schon immer verstehen wollte, warum so viele von hexagonaler Architektur schwärmen, bekommt von Danny Keller und Ferdinand Ade eine leicht verständliche und praxisnahe Einführung.

Auch Klassiker haben wir nicht vergessen: Dr. Fadil Kallat frischt das Thema Exception Handling mit modernen Ansätzen auf. Felix Tensing zeigt, wie sich Anforderungen mit dem Shift-Left-Prinzip sauber dokumentieren und testen lassen. Und mit SpiceDB sowie Forgejo gibt es gleich zwei Artikel, die euch Werkzeuge an die Hand geben, um Berechtigungen und Projekte eigenständig in den Griff zu bekommen.

Daneben nehmen wir euch mit auf eine Reise quer durch Kalenderwelten, denken gemeinsam über Nachhaltigkeit in der Softwareentwicklung nach und stellen uns der provokanten Frage: „Ich hab' nichts zu verbergen – oder doch?"

Marcos López, in Vertretung für Lisa Damerow

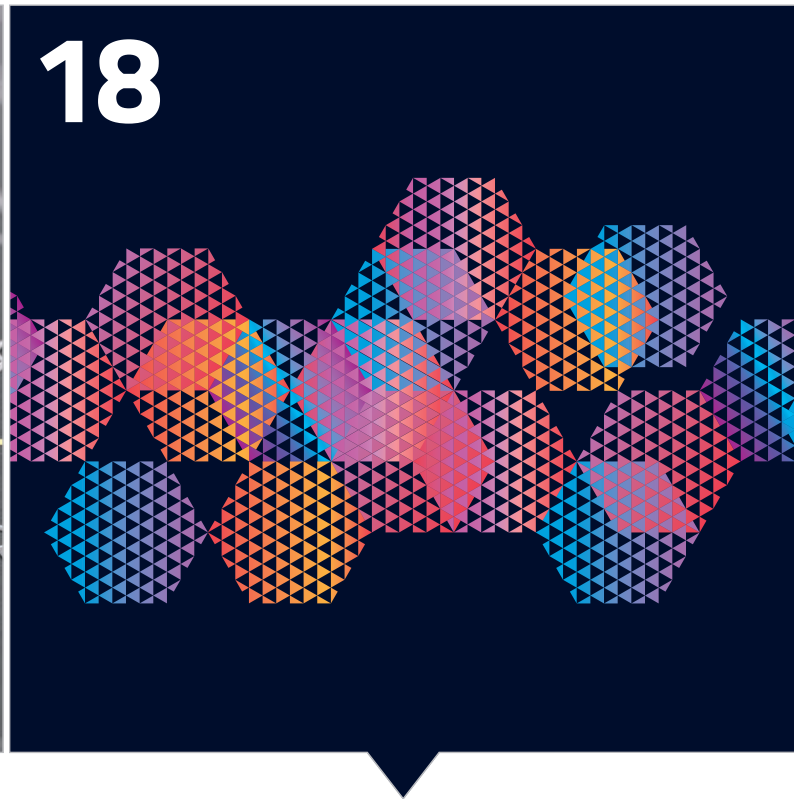
Wir wünschen euch viele Aha-Momente, neue Perspektiven und wie immer eine gute Portion Lesespaß.



Marcos López
Redaktion

INHALT

Die Eigenheiten des JDK



Vorteile hexagonaler Architektur

3 Editorial

6 Java-Tagebuch
Andreas Badelt

10 Die goldenen Regeln: Teil 8
Andreas Monschau

14 Javas Ecken und Kanten
Marcel Vielsack

18 Hexagonale Architektur einfach erklärt
Danny Keller, Ferdinand Ade

26 Exception Handling in Java: Klassiker mit modernem Schliff
Dr. Fadil Kallat

32 Shift-Left mit System: Fachliche Anforderungen dokumentieren und testen
Felix Tensing

Systematische Nachhaltigkeit in der Softwareentwicklung

Effektiver Wissensaustausch für erfolgreiche Teams

42 Transitive Berechtigungen mit der SpiceDB
Jelmen Guhlke

50 Softwareprojekte selbst hosten mit Forgejo
Marcus Fihlon

62 Eine Reise durch Zeit und Kalender für Softwareentwickler
Michael Krimgen

70 Greenify your System – Nachhaltigkeit systematisch denken
Sophia Resch

76 Ich weiß was, was du nicht weißt
Benjamin Garbers

82 Ich hab' nichts zu verbergen!
Philipp Houzar

90 Impressum/Inserenten

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

3. Juni 2025

Kotlin und Spring

Die von JetBrains als prägnantere Alternative zu Java entworfene JVM-Sprache Kotlin gibt es nun schon seit fast 15 Jahren. Doch außerhalb der Android-Welt, in der sie 2017 von Google zur bevorzugten Sprache deklariert wurde, ist sie nicht so richtig durchgestartet. Jetzt hat JetBrains eine Zusammenarbeit mit dem Spring-Plattform-Team angekündigt, um Kotlin auch im Server-Bereich besser zu positionieren. Neben technischen Verbesserungen („null safety", die „Bean Definition DSL" *[1]*) geht es auch darum, zentrale Lernmaterialien mit Kotlin-Beispielen zur Verfügung zu stellen *[2]*.

16. Juni 2025

„State of Java" 2025

Die Ergebnisse der „State of Java 2025"-Umfrage von Azul werden wohl bei Oracle nicht eingerahmt in der Empfangshalle hängen. Doch eventuell könnten sie in der Teeküche des Lizenz-Managements ihren Zweck erfüllen: Seit „Big Red" massiv an der Lizenzschraube für sein Oracle Java gedreht hat, ist die Zahl der Nutzer, die über einen Wechsel zu alternativen Java-Distributionen nachdenken, immer weiter gestiegen – jetzt auf 88 %. Gut, einer der Profiteure des Ganzen dürfte Azul selbst sein, daher haben sie auch allen Grund, diese Zahl nicht kleinzurechnen, sondern möglichst allen Kunden prominent unter die Nase zu reiben. Und warum denken diese seit zwei Jahren nach, statt es einfach zu machen? Das steht zwar nicht im Bericht, aber Gründe gibt es sicherlich: Die Mühlen in großen Konzernen und Behörden mahlen langsam und wer tauscht schon mal eben überall die Runtimes aus? Von den Lizenz- und Support-Themen mal ganz abgesehen... Es dürfte also noch ein bisschen dauern, bis wir die vollen Auswirkungen sehen (oder eben nicht viel).

Es gab aber auch noch ein paar andere interessante Zahlen: Zum Beispiel machen bei mehr als zwei Drittel der Unternehmen, die an der Umfrage teilgenommen haben, die Java Workloads über 50 % der „Compute"-Kosten aus. Ok, das ist jetzt nicht so überraschend, weil entsprechend viele davon (da sie überhaupt an der Umfrage teilgenommen haben) hauptsächlich auf Java setzen werden – ein „Java frisst Ressourcen" lässt sich daraus nicht ableiten. Trotzdem zeigt es, dass da viel Einsparpotential drinsteckt. Performance- und Startup-Optimierungen (schon mal CraC probiert?), native Binaries, Serverless... – Möglichkeiten gibt es genug.

Noch eine Zahl: 50 % dieser Unternehmen nutzen Java für „AI-Funktionalität", mehr als jede andere Sprache. Python und C/C++ liegen bei jeweils 41 %, dazwischen landet JavaScript mit 44 % (das wiederum finde ich sehr seltsam, aber was weiß ich schon...). Die mit Abstand am meisten verwendete Bibliothek ist dabei JavaML. Trotz des Java-Bias in den teilnehmenden Unternehmen sind diese Zahlen spannend. Vielleicht drücken sie einfach aus, dass immer mehr „AI"-Projekte von der Experimentierwiese ins Engineering kommen, und entsprechend mehr Wert auf Stabilität und vorhandenes Know-how gelegt wird.

17. Juni 2025

MicroProfile 7.1

Heute wurde MicroProfile 7.1 veröffentlicht. Die Mindestanforderung an das zugrundeliegende Java SE ist weiterhin Version 11.

Die wesentlichen Änderungen sind ein Update der Telemetry-Spezifikation auf Version 2.1, mit Verbesserungen für die Konsistenz von Thread-Count-Metriken, sowie ein Update der MicroProfile-OpenAPI-Spezifikation auf Version 4.1 (die erzeugten Dokumente basieren dann auf OpenAPI v3.1). Außerdem laufen die Kompatibilitätstests (TCK) jetzt auf Java 23.

Die erste kompatible Implementierung ist Open Liberty 25.0.0.7-beta.

26. Juni 2025

Jakarta EE 11: Volles Release

Nur wenige Tage nach dem (kleinen) MicroProfile-Release ist nun endlich das volle Jakarta-EE-11-Release veröffentlicht worden. Im Dezember 2024 war schon das reduzierte Core Profile und Ende März dann das Web Profile veröffentlicht worden. Mit dabei ist die neue Jakarta-Data-Spezifikation – und Unterstützung für Java SE

Records. Dafür wurden veraltete Spezifikationen entfernt, insbesondere die Managed Beans sowie Referenzen auf den abgekündigten SecurityManager. Außerdem wird CDI stärker und konsistenter über die Teilspezifikationen hinweg eingesetzt. Weitere Verbesserungen beziehen sich unter anderem auf die Modernisierung der Kompatibilitätstests (TCK) und sind damit eher intern relevant. Die Mindestanforderung an das zugrundeliegende Java SE ist hier jetzt Version 17, mit Java 21 werden auch Virtual Threads unterstützt.

Kompatible Implementierungen sind noch rar: Lediglich GlassFish implementiert Jakarta EE 11 vollständig (und damit logischerweise auch das Web Profile). Zumindest beim Core Profile besteht aber bereits Auswahl: mit Open Liberty, WildFly, Payara und der FUJITSU Software Enterprise Application Platform. Der nächste Stopp – Jakarta EE 12 – ist für 2026 in Planung.

27. Juni 2025

Jakarta und MicroProfile: Verschmelzung noch nicht in Sicht

Von der vorgeschlagenen „Hochzeit" von Microprofile und Jakarta unter der Haube der Eclipse Foundation gibt es nichts wirklich Neues. Nur ein paar organisatorische Diskussionen hinter den Kulissen. Nach den ersten enthusiastischen Stimmen hat sich unter anderem die Garden State JUG aus New Jersey kritisch zu Wort gemeldet. Um weiterhin Stimmrecht zu haben, dann innerhalb der Jakarta EE Working Group, müsste die JUG nach heutigem Stand jährlich – genau wie kommerzielle Unternehmen – einen Beitrag von 25.000 US-Dollar zahlen. Das dürfte für eine normale JUG völlig utopisch sein. Logischerweise lautet die Forderung daher: Wenn die Verschmelzung weiter verfolgt wird, soll eine Beteiligung für JUGs (finanziell) deutlich erleichtern werden, und außerdem die Anzahl der stimmberechtigten Repräsentanten der „Participant Members" von 1 auf 2 aufgestockt werden (als „Participant Members" müssen aktiv mitwirkende Organisationen wie JUGs keine Beiträge zahlen).

1. Juli 2025

Spring 7 im November

MicroProfile, Jakarta... da fehlt auf jeden Fall noch Spring. Der nächste große Wurf, das Spring Framework 7 und „obendrauf" Spring Boot 4, wird erst im November kommen. Die Eckdaten und auch die wesentlichen Features stehen aber schon fest: Für die aus Jakarta EE übernommenen Spezifikationen wird EE 11 die „Baseline" sein. Die Java SE Basis ist JDK 17 als Minimum, aber auch das bis dahin erschienene JDK 25 (LTS) soll unterstützt werden. Ebenso werden eine ganze Reihe anderer implementierter Spezifikationen oder eingebundener Bibliotheken und viele mehr auf neue Versionen gehoben, zum Beispiel WebSocket 2.2, GraalVM 24 mit dem neuen „exact reachability metadata" Format oder Kotlin 2.2.

Auf der Feature-Seite gibt es zum Beispiel eine direkte Unterstützung paralleler Versionen für MVC oder WebFlux APIs (inklusive Client-Seite und Test-Unterstützung wie von Spring gewohnt); oder den neuen BeanRegistrar, mit dem insbesondere komplexe Fälle programmatischer Bean-Registrierung deutlich einfacher werden.

Details sind auf der Seite des Spring Framework 7 Preview Releases zu finden *[3]*.

4. Juli

Java 25 mit 18 JEPs

Das JDK 25 ist auf Kurs für den geplanten Release-Termin am 16. September 2025. Insgesamt sind es satte 18 JEPs (Java Enhancement Proposals), die in das Release aufgenommen wurden: Von experimenteller Unterstützung für echtes CPU-Profiling mit dem Java Flight Recorder unter Linux (nicht das Gleiche wie das heute unterstützte Profiling der „Execution Time"), über die zehnte(!) Inkubator-Version der Vector API und die fünfte Preview von „Structured Concurrency" bis hin zu (produktionsreifen) „Scoped Values" und „Module Import Declarations". Die volle Liste gibt's wie immer auf der OpenJDK-Seite für das Release *[4]*.

12. Juli 2025

Java und AI

Was gibt's denn noch Neues im Bereich Java und AI? Hier sind ein paar spannende Frameworks entstanden, die im „State of Java" Report nicht mal erwähnt wurden. Neu ist beispielsweise das Agent-2Agent Java SDK *[5]*, eine Zusammenarbeit von Google und RedHat. Das A2A-Protokoll soll die Kommunikation von AI Agents untereinander standardisieren und erleichtern. Das Agent2Agent Java SDK dürfte mit diesem Hersteller-Hintergrund „die" Implementierung für Java werden.

Wer A2A sagt, muss auch MCP sagen: Beim von Anthropic initiierten Model Context Protocol geht es darum, Agents (beziehungsweise den LLMs dahinter) mitzugeben, welche Tools sie wie nutzen können (wobei ein Tool alles Mögliche sein kann, vom Zugriff auf ein Dateisystem über eine Web-Suche bis zur API eines ERP- oder CRM-Systems). Die offizielle Website *[6]* verweist auf mehrere Implementierungen, darunter auch für Java *[7]*. Und das Spring AI Projekt hat diese bereits unter anderem in seine „Starter" für MVC und WebFlux integriert.

Llama 3, das Open-Source-Sprachmodell von Meta, wird auch in Java unterstützt, hier ist insbesondere das neue GPULlama3.java Projekt zu erwähnen *[8]*, das verbesserte GPU-Unterstützung auf Basis der TornadoVM bietet.

Im OpenJDK-Projekt selbst wird auch an verbesserter GPU-Unterstützung gearbeitet: Das Heterogeneous Accelerator Toolkit (HAT) ist ein Modul von Projekt „Babylon" und nutzt auch Teile von Projekt „Panama" (insbesondere dessen Foreign Function & Memory API), um Java-Code effizient für GPUs nutzbar zu machen. Babylons Code Reflection analysiert Java Methoden und extrahiert die Funktionalität als „GPU Kernels", die HAT dann in ein Format zur parallelen Ausführung auf GPUs übersetzt (wobei zunächst von den konkreten „GPU Backends" abstrahiert wird, dafür gibt es herstellerspezifische Plug-ins). Und dank Panamas Memory-Segmenten kann Off Heap-Speicher ohne JNI basiertes Kopieren genutzt werden. HAT ist allerdings (bislang) kein integraler Bestandteil des JDK, sondern ein eigenständiges Toolkit im Babylon

Repo [9] [10].

Ein nicht so neues Projekt, aber dennoch ein Meilenstein ist die Veröffentlichung von Langchain4j 1.0 im Mai. Die Java-Portierung des bekannten LangChain-Frameworks, das die Entwicklung von LLM-gesteuerten Anwendungen durch modulare Komponenten wie Prompt Templates, Memory und Tool-Integration vereinfachen soll, steht damit erstmals in einer offiziell produktionsreifen Version zur Verfügung. Inzwischen gibt es schon den erste Release Candidate für Version 1.1 – es geht schnell, auch im oft behäbigen, auf Stabilität und Kontinuität fokussierenden Java Ökosystem.

Quellen

[1] *https://docs.spring.io/spring-framework/reference/languages/kotlin/bean-definition-dsl.html*
[2] *https://www.infoworld.com/article/4001245/kotlin-cozies-up-to-spring-framework.html*
[3] *https://github.com/spring-projects/spring-framework/wiki/Spring-Framework-7.0-Release-Notes*
[4] *https://openjdk.org/projects/jdk/25/*
[5] *https://github.com/a2aproject/a2a-java*
[6] *https://modelcontextprotocol.io*
[7] *https://github.com/modelcontextprotocol/java-sdk*
[8] *https://github.com/beehive-lab/GPULlama3.java*
[9] *https://cr.openjdk.org/~psandoz/conferences/2024-JVMLS/JAVA_BABYLON_HAT-JVMLS-24-08-05.pdf*
[10] *https://inside.java/2025/07/14/javaone-hat/*

Andreas Badelt
stellv. Leiter der DOAG Cloud Native Community
andreas.badelt@doag.org

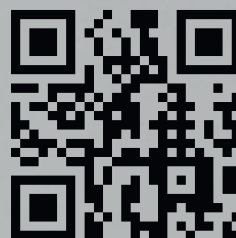
Andreas Badelt ist seit 2001 ehrenamtlich im DOAG e.V. aktiv und hat dort inzwischen seine Heimat in der Cloud Native Community gefunden, wobei ihn das Java-Ökosystem bis heute fasziniert. Beruflich hat er von Ende des vorigen Jahrtausends an als Entwickler und später auch Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet. Seit 2016 ist er als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

CLOUD NATIV FESTIV
im Heide Park Soltau

CloudLand
www.cloudland.org

SUPER-SAVER

Bis 31.10.2025



19. – 22.
MAI
2026

#CLOUDLAND2026

Die goldenen Regeln: Teil 8

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.

In den letzten zwei Jahren habe ich an dieser Stelle mehrere dieser Regeln vorgestellt, und ich hoffe, ich bin meinem pädagogischen Auftrag nachgekommen, denn tatsächlich ist es so, dass ihr diese Regeln **nicht** anwenden sollt. Diese Regeln entsprangen nicht meiner bunten Fantasie, ganz im Gegenteil: Ich habe sie selbst erlebt, sie wurden mir von Kollegen zugetragen, oder aber auch von Menschen, die meinen zugehörigen Talk gesehen haben.

Die Anwendung dieser Regeln sorgt im Grunde für drei Dinge:

- Im wahrsten Sinne des Wortes wird Neulingen der Einstieg in das Projekt, das Unternehmen, die Organisation oder auch das Verfahren massiv unerfreulich gestaltet.
- Die Neulinge werden für ihr ganzes weiteres Berufsleben entscheidend geprägt. Denke dran: Sie werden sich immer wieder an das erinnern, was du ihnen beigebracht hast, und werden dann beginnen, diese Muster zu wiederholen.
- Und natürlich setzt du den Erfolg deines Projekts aufs Spiel.

Somit weist die Anwendung dieser Regeln auch darauf hin, dass etwas Grundsätzliches falsch läuft, und sie sind daher auch ein zuverlässiger Indikator dafür, wie es um dein Projekt steht. Es deutet auf toxische Muster zwischen den Beteiligten hin und an dieser Stelle muss angesetzt werden. Betrachte dein Projekt ganz genau, analysiere, wo die Probleme liegen und versuche, sie zu lösen. Wenn du es nicht allein schaffst, suche dir Verbündete oder jemanden, der dir hilft.

Anti-Pattern, wie ich sie in dieser Artikelreihe beschrieben habe, sollten nicht mehr im Arbeitsalltag anzutreffen sein – und dazu kann jeder beitragen!

Als ich anfing, über dieses Thema zu sprechen, wurde ich oft gefragt, wie realistisch das alles denn sei. Ich las das auch schon das eine oder andere Mal in der Bewertung, wenn mein Talk bei einer Konferenz abgelehnt wurde (glücklicherweise wurde der Talk aber auch oft genug angenommen). Daher stellt sich wirklich die Frage: Wie häufig trifft man auf diese Regeln?

Mein Vortrag ist insofern interaktiv, dass ich die Zuschauer bitte, an einer Umfrage teilzunehmen, um mich damit an ihren Erfahrungen teilhaben zu lassen. Dabei stelle ich folgende Frage:

- Wie viele der vorgestellten Regeln hast du in der Vergangenheit in deinen Projekten erlebt?

Hinweis dazu: Der Talk hat genau zehn goldene Regeln behandelt und zur Vergleichbarkeit der Ergebnisse waren es stets die gleichen.

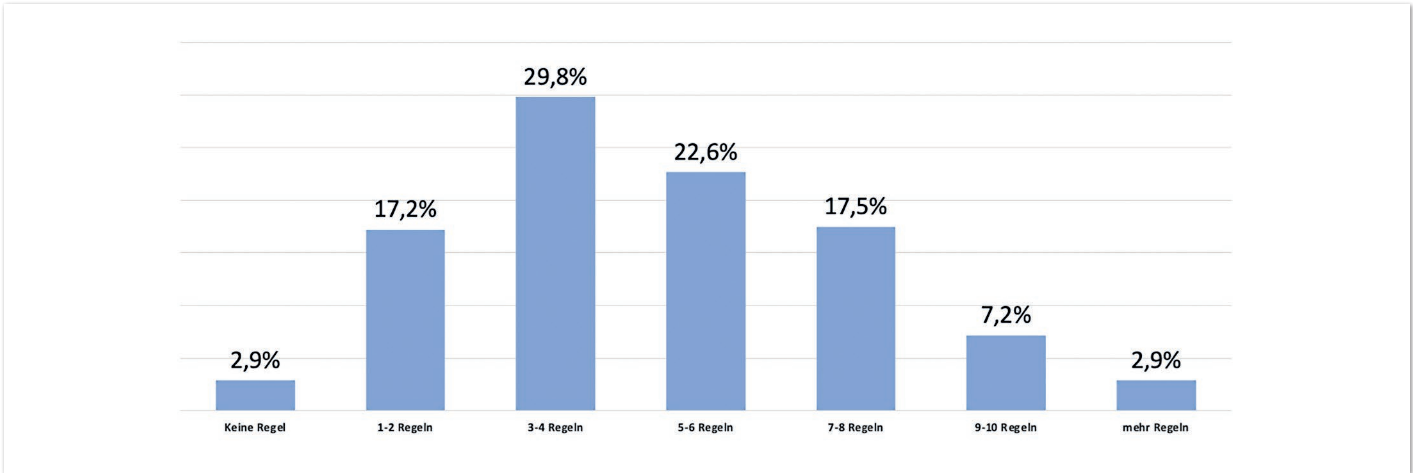


Abbildung 1: Wie viele der vorgestellten Regeln hast du in der Vergangenheit in deinen Projekten erlebt?

Bei folgenden Gelegenheiten habe ich (Stand Anfang Mai 2025) den Talk gehalten: Seacon 2023, Code Days 2024, JUG Hamburg, Magdeburger Developer Days 2024, JUG Bonn, Basel One 2024, IT-Tage 2024, JUG Schweiz, JUG Stuttgart und Dev Day Dresden 2025.

Abbildung 1 zeigt das Feedback auf die erste Frage.

Es ist statistisch eigentlich wenig überraschend, dass sich die Ergebnisse in dieser Art und Weise präsentieren. Interessant hierbei ist auf jeden Fall, dass es nur sehr wenige Teilnehmer gab, die noch keine Regeln in ihrer Anwendung erleiden/erdulden mussten, aber so ziemlich genau die Hälfte aller Teilnehmer drei bis sechs der Regeln erlebt haben, was einen irgendwie auch ein wenig traurig zu stimmen vermag, abseits von allen Statistiken.

Ich deute das Ergebnis so, dass diese Regeln tatsächlich der Realität entsprechen und man sie auch antreffen kann.

All good bad things…

Wie beschrieben: Hier und heute endet diese Artikelreihe. Ich hoffe, dass du, lieber Leser, auf der einen Seite gut unterhalten wurdest, aber gleichzeitig vielleicht auch an der einen oder anderen Stelle kurz innegehalten und über dein Arbeitsumfeld oder vielleicht auch sogar über deine eigenen Methoden reflektiert hast. Solltest du daher bis dato ein Verfechter dieser Regeln gewesen sein, würde es mich freuen, wenn du künftig auf sie verzichtest. Falls du aber inspiriert sein solltest, sie anwenden zu wollen, nun, dann sollten wir uns mal unterhalten… vielleicht auf einer Konferenz oder Java User Group, wenn ich den Talk nochmal halte?

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.

Andreas Monschau
Haeger Consulting
amonschau@haeger-consulting.de

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

Javas Ecken und Kanten

Marcel Vielsack, esentri AG

In diesem Artikel werden die Eigenheiten des JDKs diskutiert. Dieses Wissen kann sehr hilfreich bei der Fehlersuche oder auch bei der Entwicklung eigener Software sein. Wir werden anhand von Beispielen sehen, warum Vererbung nicht das Allheilmittel ist, und dass selbst in hochprofessioneller Softwareentwicklung wie im JDK dadurch Probleme entstehen können.

Überladen von Methoden

Wir alle tun es, oft ohne darüber nachzudenken: Wir überladen Methoden. Ein einfaches Mittel, um zum Beispiel in einer Kindklasse Funktionalität hinzuzufügen. Viel zu selten machen wir uns aber Gedanken über eventuelle unschöne Nebeneffekte. Werfen wir einen Blick auf *Listing 1* und *Listing 2*.

```
List<Integer> numbers = new ArrayList<>(List.of(1, 2, 3));
System.out.println(numbers); // => [1,2,3]
numbers.remove(1);
System.out.println(numbers); // => [1,3]
```

Listing 1: Aus einer Liste wird ein Element entfernt.

```
Collection<Integer> numbers = new ArrayList<>(List.of(1, 2, 3));
System.out.println(numbers); // => [1,2,3]
numbers.remove(1);
System.out.println(numbers); // => [2,3]
```

Listing 2: Aus einer Collection wird ein Element entfernt.

Diese unterscheiden sich lediglich darin, dass in *Listing 1* auf dem List-Interface gearbeitet wird und in *Listing 2* auf dem Collection-Interface. Die Ausgaben der beiden Codebeispiele unterscheiden sich allerdings. *Listing 1* enthält nach dem `remove()`-Aufruf die Elemente 1 und 3 und *Listing 2* enthält danach die Elemente 2 und 3. Warum ist das so? Im Collection-Interface existiert eine Methode, die ein Objekt aus der Collection entfernt, das gleich dem übergebenen ist. Die Gleichheit wird mittels `equals`-Vergleich bestimmt. Im List-Interface wurde die Methode überladen, um ein Element über den Index aus der Liste zu entfernen.

In unserem Fall kommt jetzt noch Autoboxing zum Tragen. Da wir eine Liste mit Integer-Objekten haben, wird im Fall der Collection aus dem „kleinen" `int` ein Integer-Objekt. Es wird das Objekt gelöscht, für das der `equals`-Vergleich `true` ergibt, daher ist die Ausgabe „2,3". Im Fall der Liste wird das Element mit dem Index „1" gelöscht, da es eine Methode gibt, die der Signatur mit kleinem `int` entspricht. Daher ist die Ausgabe „1,3".

Zugegeben, das ist schon ein etwas ausgefallener Sonderfall, er zeigt aber auch, wie schnell man einen Fall übersehen kann. Hätte man für das Entfernen via Index einen anderen Methodennamen gewählt, würde es nicht zu diesem Effekt kommen.

Vergleich von Objekten

Zum Vergleich von Java-Objekten existiert in jeder Java-Klasse die `equals`-Methode, die einen Vergleich von Objekten ermöglicht. Ein Blick in das JavaDoc von `equals` zeigt uns, welche Bedingungen für `equals` gelten sollen. Das sind folgende:

- reflexiv: Ein Vergleich mit sich selbst ergibt `true => a.equals(a) == true`
- symmetrisch: Wenn a zu b gleich ist, muss auch b zu a gleich sein `=> a.equals(b) == b.equals(a)`
- transitiv: Wenn a zu b und b zu c gleich ist, muss auch a zu c gleich sein `=> a.equals(b) == b.equals(c) == c.equals(a)`
- konstant: Mehrere Aufrufe ergeben immer das gleiche Ergebnis `=> a.equals(b) == a.equals(b)`

Dass diese Bedingungen nicht immer leicht einzuhalten sind, können wir an Beispielen aus dem JDK sehen.

Werfen wir einen Blick auf die Klassen „Date" und „Timestamp". Die Klasse Timestamp erbt von Date. In *Listing 3* wird auf derselben Basis jeweils ein Timestamp- und ein Date-Objekt erzeugt. Vergleicht man die beiden nun miteinander, erhält man – abhängig davon, auf welchem Objekt man `equals` aufruft – ein anderes Ergebnis.

```
long now = System.currentTimeMillis();
Date date = new Date(now);
Timestamp timestamp = new Timestamp(now);

System.out.println(date.equals(timestamp)); // => true
System.out.println(timestamp.equals(date)); // => false
```

Listing 3: Vergleich eines Zeitpunktes mit Date und Timestamp

Offensichtlich ist hier der `equals`-Vertrag gebrochen. Der Grund liegt in der Implementierung der `equals`-Methode in Timestamp *(siehe Listing 4)*. Da für ein Date-Objekt der `instanceof`-Vergleich zu `false` evaluiert, wird der Vergleich hier immer mit `false` abgebrochen. In der `equals`-Methode von Date gibt es diesen `instanceof`-Check auch *(siehe Listing 5)*, doch hier ist dieser für ein Timestamp-Objekt `true`, weil Timestamp von Date erbt.

```
public boolean equals(java.lang.Object ts) {
  if (ts instanceof Timestamp) {
        return this.equals((Timestamp)ts);
   } else {
        return false;
   }
}
```

Listing 4: `equals`-Methode aus Timestamp

Ein weiteres Beispiel aus dem JDK, das Probleme mit dem `equals`-Vertrag hat, ist die URL-Klasse. Diese versucht während des Vergleichs, den Host in eine IP-Adresse aufzulösen *(siehe Listing 6)*.

```
public boolean equals(Object obj) {
        return obj instanceof Date && getTime() == ((Date) obj).getTime();
  }
```

Listing 5: `equals`-Methode aus Date

```
var url1 = URI.create("http://localhost").toURL();
var url2 = URI.create("http://127.0.0.1").toURL();
System.out.println(Objects.equals(url1, url2));
// => true, kann unter Umständen "false" ergeben
```

Listing 6: `equals`-Vergleich mit URLs

Nicht nur, dass diese Aktion recht zeitintensiv ist, es ist auch nicht garantiert, dass zwei Aufrufe das gleiche Ergebnis liefern – sei es durch einen Netzwerkausfall oder eine geänderte Konfiguration. Dadurch wird die Anforderung, dass der Vergleich konstant das gleiche Ergebnis liefern muss, gebrochen.

Spannender als der unmittelbare Grund ist die Frage: „Wie kann man so etwas verhindern?“ Wir setzen Vererbung oft ohne Bedacht ein, weil wir schon in der Ausbildung vermittelt bekommen haben, dass Vererbung DAS Werkzeug ist, um Funktionalität wiederzuverwenden. In der Praxis folgen oft lange Vererbungsketten. Das bedeutet eine hohe Komplexität und oft auch Funktionalität, die eine Klasse hat, aber gar nicht benötigt. Eine Komposition von Objekten erspart uns oft viele Nachteile, die wir uns durch eine Vererbung einhandeln. Joshua Bloch zeigt dies eindrucksvoll anhand mehrerer Beispiele in seinem Buch „Effective Java“ *[1]* – absolut lesenswert.

Auch `default`-Methoden in Interfaces sind ein toller Weg, um Funktionalität in die eigene Klasse zu integrieren, ohne Vererbung zu nutzen. Man kann sehr gezielt bestimmen, welche Funktionalität beziehungsweise Aufgabe eine Klasse noch erfüllen soll. Dieser Ansatz hilft dabei, dem Grundsatz „Separation of Concerns“ zu folgen.

Wer schon einmal vor der Herausforderung stand, dass Berechnungen sehr genau sein müssen oder die Zahlen, mit denen man arbeiten muss, sehr groß werden, hatte wahrscheinlich schon mit `BigDecimal` zu tun. Früher oder später will man die Zahlen auch miteinander vergleichen – sei es in Tests oder aus einem Use-Case heraus.

In *Listing 7* wird „0“ mit „0“ verglichen, trotzdem ist das Ergebnis `false`. Wie kann das sein? Wir haben doch alles richtig gemacht! Wir nutzen `equals`, um Objekte zu vergleichen – so wie man es uns beigebracht hat. Den Grund dafür finden wir, wenn wir die `equals`-Methode von `BigDecimal` genauer untersuchen. Diese vergleicht nicht nur den Wert, sondern zum Beispiel auch den Scale. Der Scale ist ein Teil der internen Darstellung der Zahl. 9,87*103 entspricht im Wert 98,7*102, unterscheidet sich aber in der internen Repräsentation. Aber wie kann man nun `BigDecimals` nur anhand ihres Zahlenwerts vergleichen? Im Gegensatz zu der `equals`-Methode betrachtet die `compareTo`-Methode in `BigDecimal` ausschließlich den Zahlenwert und liefert somit bei gleichem Zahlenwert „0“, also den Wert für Gleichheit *(siehe Listing 8)*.

Ein Blick hinter die Kulissen

Um besser zu verstehen, was hier genau geschieht, ist es wichtig, den Unterschied zwischen einem Vergleich mit == und `equals` zu

```
System.out.println(
Objects.equals(BigDecimal.ZERO,BigDecimal.valueOf(0.0d))
); // => false
```

Listing 7: Vergleich zweier BigDecimal-Objekte mit `equals`

```
System.out.println(
BigDecimal.ZERO.compareTo(BigDecimal.valueOf(0.0d)) == 0
); // => true
```

Listing 8: Vergleich zweier BigDecimal-Objekte mit compareTo

kennen. Der Vergleich mit == prüft auf Identität, also ob die beiden Referenzen, die ich vergleiche, auf ein und dasselbe Objekt zeigen. Bei der `equals`-Methode kommt es auf die Implementierung an (`java.lang.Object` implementiert den `equals`-Vergleich tatsächlich als Identitätsvergleich. Oft wird die `equals`-Methode genutzt, um zu prüfen, ob der Inhalt von Objekten gleich ist.

So weit so gut, aber warum gibt es dann Fälle, in denen der Vergleich zweier scheinbar nicht identischer Objekte mittels == Vergleich trotzdem `true` ergibt? In *Listing 9* wird 100 mit 100 verglichen und 1000 mit 1000.

```
Integer value1 = 100;
Integer value2 = 100;
System.out.println(value1 == value2); // => true
Integer value3 = 1000;
Integer value4 = 1000;
System.out.println(value3 == value4); // => true
```

Listing 9: Vergleich von Integer-Objekten mittels ==

Im ersten Fall ergibt der Vergleich `true`, im zweiten `false`. Warum ist das so? Schauen wir uns an, was hier genau passiert. Wir weisen einem Integer-Objekt ein Literal zu, das bedeutet, dass hier Autoboxing angewendet wird. Das Literal wird durch `Integer.valueOf(100)` ersetzt. Es wird also ein Integer-Objekt erstellt. Der Vergleich ergibt `true`, das bedeutet, dass `value1` und `value2` dasselbe Objekt sein müssen. Genauso ist es auch und der Grund hierfür ist, dass die Integer-Klasse intern einen Cache nutzt und standardmäßig im Zahlenbereich von -128 bis 127 die entsprechenden Objekte wiederverwendet. Für Zahlen außerhalb des Cachingbereichs wird jeweils ein neues Objekt erstellt, daher sind `value3` und `value4` nicht dasselbe Objekt. Die Größe des Cache kann über den JVM-Parameter `-XX:AutoBoxCacheMax` beeinflusst werden.

String-Manipulation mit `replace` und `replaceAll`

Oft kommen wir in die Situation, dass wir Strings manipulieren müssen. Ein Klassiker ist vermutlich, Trennzeichen zu tauschen. Das eine System möchte die CSV-Datei mit Komma separiert verarbeiten, das andere mit Semikolon. Es gibt unzählige Beispiele und doch stolpern wir immer wieder über zwei Methoden: `replace` und `replaceAll` der String-Klasse. Wir verwenden oftmals `replaceAll`, weil wir alle Vorkommen ersetzen wollen und `replace` das nicht könne. So oder so ähnlich habe ich die Begründung für den Einsatz von `replaceAll` schon mehrmals gehört. Allerdings ist sie nicht richtig. Sowohl `replace` als auch `replaceAll` ersetzen jeweils alle Vorkommen des übergebenen Suchstrings. `replaceAll` interpretiert den Suchstring allerdings als Regex. Das kann sehr hilfreich sein, aber auch zu ungewolltem Verhalten führen, wenn man sich dessen nicht bewusst ist. Ein Beispiel dafür sieht man in *Listing 10*.

```
var myString = "Das ist nur ein Test.";
System.out.println(myString.replace(".", "!"));
  // => Das ist nur ein Test!
System.out.println(myString.replaceAll(".", "!"));
  // => !!!!!!!!!!!!!!!!!!!!!
```

Listing 10: Vergleich von replace und replaceAll

Das Beispiel aus *Listing 10* treibt es auf die Spitze, zeigt aber auch, dass schnell ein regulärer Ausdruck in `replaceAll` gelangen kann.

Eine eher unbekannte Funktion von `replaceAll` ist, dass es auch Referenzen auf den match zulässt *(siehe Listing 11)*. Es werden alle Zahlen im Text in runde Klammern gesetzt. Die Ersetzung der jeweiligen Zahl beinhaltet diese und wird mit $0 referenziert.

```
String myStr = "Du hast 30 Gold und 500 Erfahrungspunkte gesammelt.";
String regex = "[0-9]+";
System.out.println(myStr.replaceAll(regex, "($0)"));
// => Du hast (30) Gold und (500) Erfahrungspunkte gesammelt.
```

Listing 11: replaceAll mit Referenz auf den match

Zusammenfassung

Wir haben am Beispiel einer überladenen Methode gesehen, welche Tücken Vererbung haben kann: Sei es durch das Überladen von Methoden mit weiterer Funktionalität oder wie schwer es sein kann, den `equals`-Vertrag korrekt zu erfüllen. Komposition und `default`-Methoden wurden als Alternative zur Vererbung vorgestellt. Wir haben uns auch den Vergleich von `BigDecimal`-Objekten angeschaut, der `equals`-Vergleich nutzt nicht nur den Zahlenwert, sondern noch weitere interne Felder wie `scale`. Um nur die Zahlenwerte zu vergleichen, muss die `compareTo`-Methode verwendet werden. Bei Integer-Objekten haben wir den `IntegerCache` kennengelernt, und dass dieser dafür sorgen kann, dass Identitätsvergleiche unerwarteterweise zu `true` evaluieren. Weiterhin haben wir den Unterschied zwischen `replace` und `replaceAll` in der String-Klasse betrachtet. Beide ersetzen alle Vorkommen, `replace` arbeitet mit einem reinen Zeichenvergleich, `replaceAll` mit einer Regex-Suche.

Quellen

[1] Joshua Bloch (2017): Effective Java. Addison-Wesley Professional, Boston.

Marcel Vielsack
marcel.vielsack@esentri.com

Seit über 15 Jahren beschäftige ich mich mit Java – im Masterstudium, der Produktentwicklung und als Berater. Java-Backend-Entwicklung ist mein Schwerpunkt, ich werfe aber auch sehr gerne den Blick über den Tellerrand. Aktuell arbeite ich für die esentri AG. Wir unterstützen Kunden bei einer digitalen und nachhaltigen Transformation.

Hexagonale Architektur einfach erklärt

Danny Keller, Ferdinand Ade, codecentric AG

Die klassische Schichtenarchitektur gilt bei vielen noch immer als Standardmodell für Softwarearchitekturen, doch in der Praxis stößt sie schnell an ihre Grenzen. Dieser Artikel zeigt auf, warum die hexagonale Architektur (Ports & Adapters) eine sinnvolle Alternative darstellt, wenn es darum geht, Geschäftslogik klar von technischen Details zu trennen. Anhand anschaulicher Analogien und praktischer Beispiele wird erklärt, wie diese Architektur hilft, Systeme besser testbar, wartbarer und flexibler zu gestalten und warum dieser Ansatz nicht nur für große Projekte relevant ist.

Bevor wir uns der hexagonalen Architektur widmen, sollten wir einen Blick auf die Probleme werfen, die entstehen, wenn Anwendungen nicht klar strukturiert und entkoppelt sind. In vielen klassischen Softwareprojekten sind fachliche Regeln eng mit technischen Details verflochten: SQL-Abfragen finden sich mitten in der Geschäftslogik, REST-spezifische Konzepte prägen das Domain-Modell, und Änderungen an der Infrastruktur ziehen tiefgreifende Änderungen im Kern der Anwendung nach sich. All diese Probleme finden sich auch in den Tests wieder, da hier ebenfalls technische und fachliche Belange oftmals auf einmal getestet werden und die Übersicht und Geschwindigkeit leidet.

Diese enge Kopplung führt zu gleich mehreren Herausforderungen:

- **Testbarkeit leidet**, weil die Fachlogik sich nicht isoliert testen lässt – ohne echte Datenbank oder komplexes Mocking geht oft gar nichts.
- **Technologische Entscheidungen werden irreversibel**, denn jede Änderung (zum Beispiel von PostgreSQL zu MongoDB) bedeutet meist einen großen Umbau der Anwendung.
- **Systemgrenzen sind unsichtbar**, was die Wartung erschwert und neue Entwickler viel Einarbeitungszeit kostet.
- **Das Wichtigste – die Geschäftslogik – ist ungeschützt**, weil sie sich nicht klar vom Rest des Systems abgegrenzt und überall verteilt ist.

Die hexagonale Architektur – auch bekannt als *Ports & Adapters* – setzt genau hier an. Sie schafft eine klare, explizite Struktur, in der die Domäne im Zentrum steht und gegen äußere Einflüsse abgeschirmt wird.

Um besser zu verstehen, welchen Paradigmenwechsel die hexagonale Architektur vorschlägt, lohnt sich zunächst ein Blick auf das, was in der Softwareentwicklung lange Zeit als bewährter Standard galt: **die klassische Drei-Schichten-Architektur**.

Drei-Schichten-Architektur

Die Drei-Schichten-Architektur, oft auch als Three-Layer Architecture *(siehe Abbildung 1)* bezeichnet, gehört zu den bekanntesten Strukturierungsmustern in der Softwareentwicklung. Sie teilt Anwendungen in drei klar abgegrenzte Schichten, die jeweils unterschiedliche Verantwortlichkeiten übernehmen und so für eine saubere Trennung der Zuständigkeiten (Separation of Concerns) sorgen.

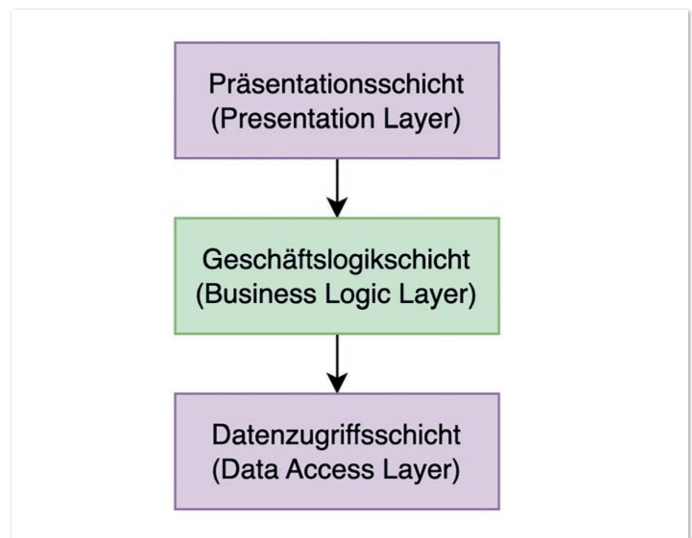


Abbildung 1: Klassische Schichtenarchitektur (© Danny Keller)

Presentation Layer

Die Präsentationsschicht (Presentation Layer) bildet die oberste Ebene dieser Architektur. Sie ist dafür zuständig, Informationen für die Benutzerinnen und Benutzer darzustellen und Eingaben entgegenzunehmen. In modernen Systemen wird diese Schicht häufig durch Webtechnologien wie HTML, JavaScript oder Frameworks wie React umgesetzt. Im Backend übernehmen REST-Controller diese Aufgabe und dienen als Schnittstelle zwischen Benutzeroberfläche und Geschäftslogik.

Business Logic Layer

Die Geschäftslogikschicht (Business Logic Layer), auch Application Layer oder Service Layer genannt, enthält die eigentliche Fachlogik der Anwendung. Hier werden fachliche Regeln umgesetzt, Abläufe koordiniert und Entscheidungen getroffen. Diese Schicht sorgt dafür, dass das System fachlich korrekt funktioniert und vermittelt zwischen der Präsentationsschicht und dem Datenzugriff.

Data Access Layer

Die Datenzugriffsschicht (Data Access Layer), auch Persistence Layer oder Repository Layer genannt, bildet die unterste Ebene. Sie kümmert sich um die Kommunikation mit Datenbanken oder externen Systemen und kapselt alle Details der Datenhaltung. So bleibt die Fachlogik von konkreten Implementierungen des Datenzugriffs entkoppelt und lässt sich einfacher warten und testen.

Grenzen und Herausforderungen der Drei-Schichten-Architektur

Trotz ihrer weiten Verbreitung und der klaren Trennung von Verantwortlichkeiten bringt die klassische Drei-Schichten-Architektur auch eine Reihe von Nachteilen mit sich, die insbesondere in komplexeren Systemlandschaften deutlich werden.

Ein zentrales Problem ist die **statische Richtung der Abhängigkeiten**: Die Präsentationsschicht kennt die Services der Geschäftslogik

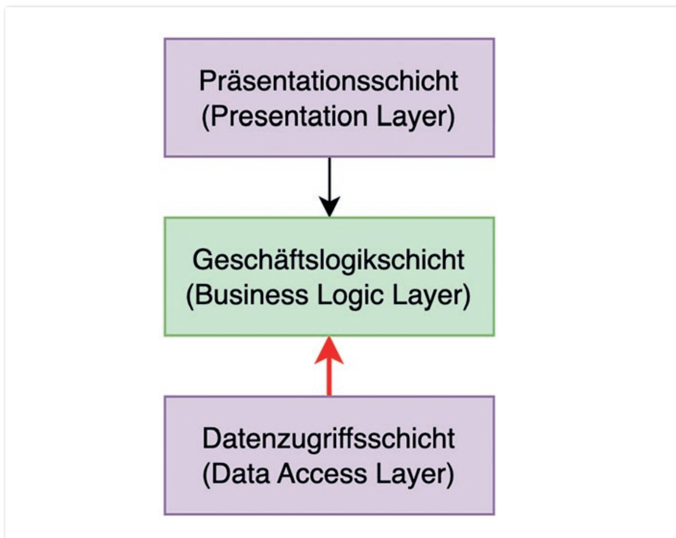


Abbildung 2: Schichtenarchitektur mit umgesetzten Dependency Inversion Principle (© Danny Keller)

und diese wiederum kennt die Implementierungen des Datenzugriffs. Diese „top-down"-gerichteten Abhängigkeiten führen dazu, dass technische Details wie konkrete Datenbankzugriffe tief in die Geschäftslogik hineinwirken. Fachlich motivierte Entscheidungen werden dadurch unnötig an technische Strukturen gekoppelt, was zu einer engen Kopplung (Tight Coupling) führt.

Daraus ergibt sich ein weiteres Problem: **Die Testbarkeit der Fachlogik leidet**, da sie sich kaum isoliert von technischen Komponenten wie Datenbanken oder Frameworks testen lässt. Ohne aufwendiges Mocking oder Integrationstests ist es kaum möglich, die eigentlichen Geschäftsregeln unabhängig zu prüfen.

Hinzu kommt, dass die Architektur die **Wiederverwendung der Domäne** in anderen Kontexten erschwert. Soll etwa dieselbe Geschäftslogik in einer alternativen Benutzeroberfläche oder in einem Batch-Prozess eingesetzt werden, ist dies oft nur mit erheblichem Aufwand möglich, da die Logik zu eng an das bestehende technische Umfeld gekoppelt ist.

Vom Schichtenmodell zur hexagonalen Architektur

Was wir also erreichen wollen, ist, dass die Geschäftslogikschicht nichts von den anderen Schichten weiß. Diese zentrale Schicht soll unabhängig davon sein, **wie** Daten gespeichert, verarbeitet oder transportiert werden. Ob die Daten in einer Datenbank liegen, im Dateisystem abgelegt werden oder an eine REST-API gesendet werden, darf für die Geschäftslogik keine Rolle spielen.

Um das zu erreichen, benötigt es einen Paradigmenwechsel: Weg von einer rein schichtbasierten Struktur hin zu einer **domänenzentrierten Architektur**, in der technische Abhängigkeiten umgekehrt werden *(siehe Abbildung 2)*. Das zentrale Prinzip dahinter ist das **Dependency Inversion Principle (DIP)**, eines der fünf SOLID-Prinzipien objektorientierter Softwareentwicklung.

Das DIP formuliert eine klare Forderung:

„High-level modules should not depend on low-level modules. Both should depend on abstractions." [1]

Übertragen auf die Softwarearchitektur bedeutet das: Die fachlich zentrale Geschäftslogik, also der „High-Level"-Teil, sollte nicht von konkreten technischen Implementierungen abhängig sein. Stattdessen definiert sie **Abstraktionen** in Form von Schnittstellen (Interfaces), die von der technischen Infrastruktur wie etwa Datenbankschichten, REST-Adaptern oder Messaging-Komponenten implementiert werden.

In der Praxis heißt das: Die Businesslogik kennt keine konkreten Klassen für den Datenbankzugriff oder für die Kommunikation mit anderen Systemen. Sie beschreibt lediglich, **was** getan werden soll – nicht **wie**. Die technische Umsetzung dieser Anforderungen erfolgt in separaten Modulen, die an die von der Geschäftslogik bereitgestellten Interfaces „andocken". Dadurch kehrt sich die klassische Abhängigkeitsrichtung um: Nicht mehr die Fachlogik ist von der Infrastruktur abhängig, sondern die Infrastruktur hängt von der Domäne ab.

Businesslogik definiert **WAS** getan werden muss, nicht **WIE**!

Abbildung 3: Alistair Island (© Danny Keller, Illustration von Fauzan Abusalam)

Analogie

Ihr denkt euch jetzt vielleicht: Abstraktionen, Interfaces, Dependency Inversion Principle – ich dachte, das hier ist *hexagonale Architektur einfach erklärt?* Genau deshalb lassen wir jetzt mal die Fachbegriffe hinter uns und werfen einen ganz anderen Blick auf die hexagonale Architektur. Eine Perspektive, die weniger technisch, dafür umso anschaulicher ist.

Alistair Island

Stellt euch eine Insel namens Alistair Island vor *(siehe Abbildung 3)*. Auf dieser Insel wird ein bestimmtes Gut hergestellt. Die Bewohnerinnen und Bewohner kümmern sich um alles, was damit zu tun hat: Sie besorgen die Rohmaterialien, stellen das Produkt her, verpacken es, versenden es an die Kundschaft und übernehmen außerdem die komplette Verwaltung.

Neben **Alistair Island** existieren weitere Inseln, die jeweils einem eigenen Zweck dienen und unabhängig voneinander arbeiten. Manche dieser Inseln stellen Materialien für unsere Insel her, andere liefern ausschließlich Informationen oder übernehmen bestimmte Dienstleistungen.

Das Besondere: Jede dieser Inseln spricht ihre eigene Sprache. Auch unsere Insel hat ihre eigene Art zu kommunizieren. Die Insel in unserer Analogie steht für die **Domäne**, also den Kern der Geschäftslogik *(siehe Abbildung 4)*. Sie ist das Zentrum, in dem die wichtigen Entscheidungen und Regeln stattfinden. Genauso wie auf der Insel alle Tätigkeiten rund um das hergestellte Gut zusammenlaufen, ist die Domäne der Ort, in dem die Geschäftslogik Geschäftsprozesse verwaltet und Daten verarbeitet.

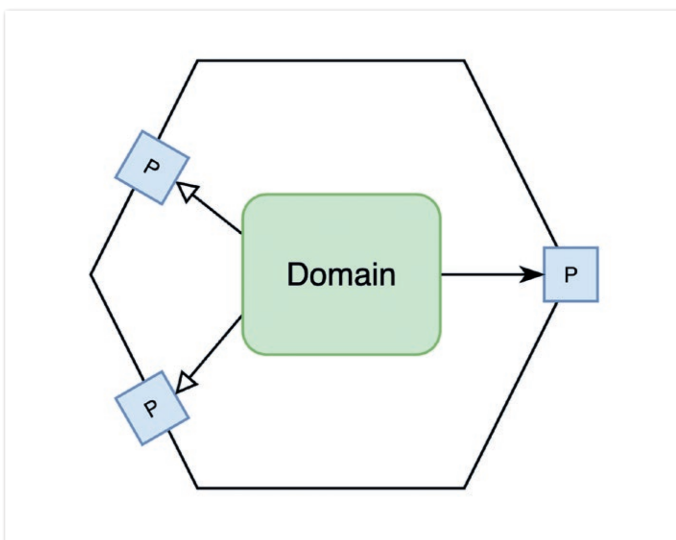


Abbildung 4: Diagramm der hexagonalen Architektur – Die Domäne (© Danny Keller)

Hafen

Obwohl alle Inseln isoliert und eigenständig für sich arbeiten, sind sie dennoch aufeinander angewiesen. Damit Handel zwischen den Inseln möglich ist, gibt es zahlreiche Handelsrouten. Jede Insel verfügt über verschiedene **Häfen**, die für das Versenden und Empfangen von Nachrichten, Materialien, Gütern und Informationen vorgesehen sind *(siehe Abbildung 5)*.

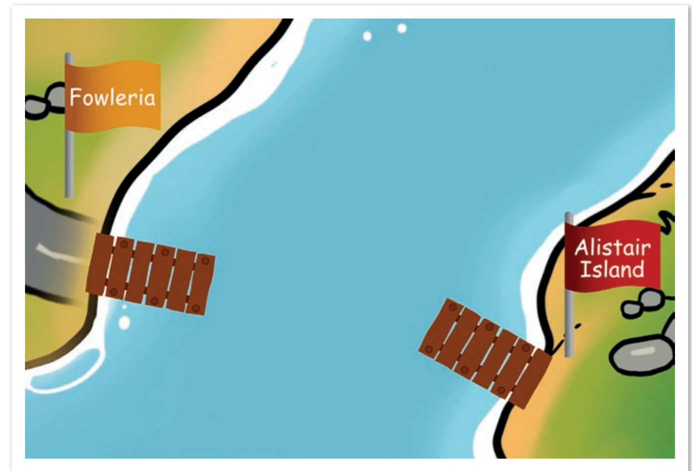


Abbildung 5: Häfen als Ports in der Insel-Analogie (© Danny Keller, Illustration von Fauzan Abusalam)

Das Besondere daran: Jeder Hafen ist genau darauf spezialisiert, **welche Art von Waren** empfangen oder versendet werden dürfen. Die Hafenmitarbeiter wissen jedoch **nicht**, von welcher Insel die Waren kommen oder wohin sie gehen. Sie wissen auch **nicht**, wie die Waren verpackt sein müssen oder in welcher Sprache die Beschriftung auf den Boxen geschrieben werden muss, damit andere Inseln sie verstehen. Ebenso wenig wissen sie, in welcher Sprache die empfangenen Waren ursprünglich verschickt wurden.

Die Häfen bekommen und versenden die Waren also **nur in der Form**, wie sie auf ihrer eigenen Insel bekannt und gebräuchlich ist. Dadurch bleibt die Insel für sich stabil und muss kein Wissen über die Sprachen oder Angewohnheiten der anderen Inseln besitzen.

In der hexagonalen Architektur entsprechen diese **Häfen** den sogenannten **Ports** *(siehe Abbildung 6)*. Ports sind klar definierte Schnittstellen, über die sich die Domäne, also der Kern der Geschäftslogik, von der Außenwelt abgrenzt. Dabei unterscheidet man zwischen **eingehenden** und **ausgehenden Ports**. Ein eingehender Port kann zum Beispiel eine Bestellung aus einem Webshop entgegennehmen, während ein ausgehender Port Zahlungsinformationen an einen externen Zahlungsdienstleister übermittelt.

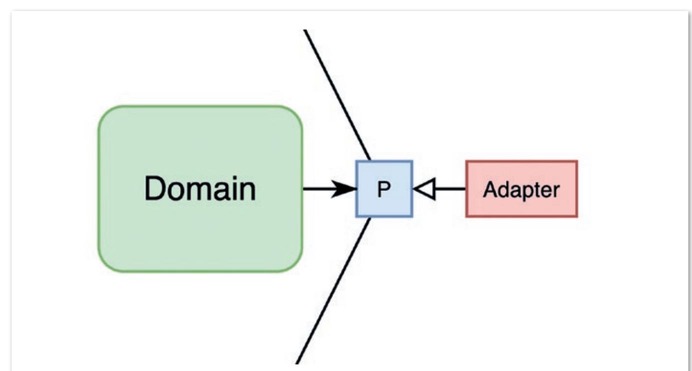


Abbildung 6: Diagramm der Hexagonalen Architektur – Der Port (© Danny Keller)

Wenn wir gleich zu den Schiffen kommen, wird uns diese Unterscheidung wieder begegnen. Dort wird noch klarer, wann wir von

eingehenden und wann von ausgehenden Ports sprechen und warum das so wichtig ist.

Schiffe

Nun müssen die Waren noch versendet, umgepackt und vor allem übersetzt werden. Diese Aufgaben übernehmen die Schiffe *(siehe Abbildung 7)*.

Abbildung 7: Schiffe als Adapter in der Insel-Analogie (© Danny Keller)

Es gibt verschiedene Arten von Schiffen: Große Frachtschiffe transportieren Lebensmittel und Vorräte, Passagierschiffe bringen Besucher zur Insel, kleinere Boote übermitteln spezielle Nachrichten und Aufträge.

Jedes Schiff hat einen **Heimathafen**, an dem es fest stationiert ist. Schiffe, die von unserer Insel aus gesendet werden, haben den Heimathafen an unserer Insel. Schiffe von anderen Inseln haben den Heimathafen auf der Insel, von der sie gesendet werden.

Entscheidend ist dabei **nicht**, in welche Richtung die Waren oder Nachrichten übermittelt werden, sondern **wer den Auftrag erteilt hat**. Aus Sicht unserer Insel gilt:

- Schiffe, die von anderen Inseln kommen, sind **eingehende Schiffe**.
- Schiffe, die von unserer Insel starten, sind **ausgehende Schiffe** – selbst dann, wenn sie unterwegs sind, um etwas für unsere Insel abzuholen.

Ein ausgehendes Schiff kann also die Absicht haben, Waren von anderen Inseln zu unserer zu transportieren. Dennoch gilt es als ausgehend, da die Initiative von unserer Insel ausgeht.

Da die Inseln unterschiedliche Sprachen sprechen, müssen die Schiffe mit **Übersetzern** ausgestattet sein. Diese sorgen dafür, dass Nachrichten und Informationen für die Inselbewohner verständlich werden. Auch die Güter müssen eventuell **umverpackt** werden: Vielleicht nutzt die andere Insel große Kräne zum Verladen, während bei uns alles per Hand geschieht. Eine **Qualitätsprüfung** hilft sicherzustellen, dass die gelieferten Waren den Anforderungen unserer Insel entsprechen. Falls nicht, gibt das Schiff direkt **Rückmeldung** an die liefernde Insel.

Das gilt auch in umgekehrter Richtung: Ausgehende Schiffe übersetzen, verpacken um und prüfen, ob die Waren für die Empfängerinsel passend sind.

Die Schiffe sind die einzigen, die sowohl die Sprache unserer Insel als auch die der anderen Inseln beherrschen. Sie wissen, wie man Güter und Informationen so aufbereitet, dass sie auf beiden Seiten verstanden werden. Allerdings ist jedes Schiff **nur für eine bestimmte Handelsroute** zuständig. Entsteht eine neue Handelsverbindung, braucht es ein neues Schiff.

Die **Schiffe** sind eine Analogie für die **Adapter** in der hexagonalen Architektur *(siehe Abbildung 8)*. Sie sorgen dafür, dass die Kommunikation zwischen der Außenwelt und der Domäne überhaupt stattfinden kann. Adapter sind spezialisierte Übersetzer, die die Sprache zwischen der Domäne und der Außenwelt übersetzen – in beide Richtungen.

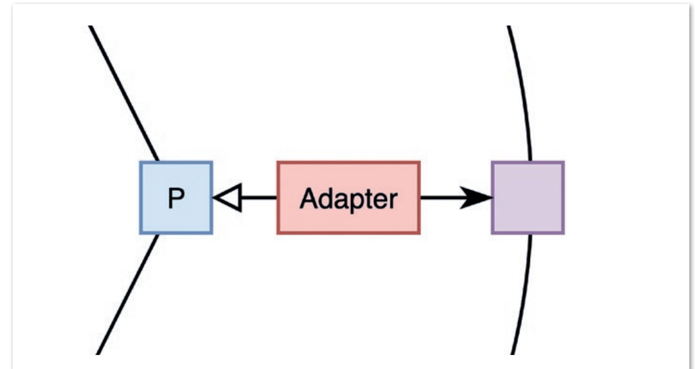


Abbildung 8: Diagramm der Hexagonalen Architektur – Der Adapter (© Danny Keller)

Adapter sind also maßgeblich dafür verantwortlich, dass Anforderungen von außen in eine Form gebracht werden, die im Inneren der Domäne verarbeitet werden kann. Umgekehrt wandeln sie die Ergebnisse und Antworten der Domäne in eine Form um, die von der Außenwelt verstanden und genutzt werden kann.

Wir unterscheiden dabei zwischen eingehenden und ausgehenden Adaptern:

- **Eingehende Adapter** nehmen Aufträge von externen Systemen oder Aktoren (andere Inseln) entgegen, übersetzen sie und stellen sie der Domäne bereit. Nach der Verarbeitung durch die Domäne übersetzen sie auch die Antworten zurück in ein für das externe System verständliches Format. Die eingehenden Adapter entsprechen den eingehenden Schiffen, die von den Häfen der anderen Inseln starten, um auf unsere Insel zu gelangen. Beispiel: Ein **eingehender Adapter** übersetzt eine REST-API-Anfrage in ein Domänenobjekt und anschließend das Ergebnis der Domäne zurück in eine JSON-Antwort.
- **Ausgehende Adapter** hingegen nehmen Aufträge der Domäne entgegen und kommunizieren mit externen Systemen oder Aktoren (andere Inseln) – sowohl um Daten zu senden als auch um Daten abzuholen. Sie übersetzen dabei in beide Richtungen zwischen der Domäne und der Außenwelt. Die ausgehenden Adapter sind die ausgehenden Schiffe, die an den Häfen unserer Insel bereitstehen, um im Auftrag der Domäne Informationen zu anderen Inseln zu transportieren oder von dort abzuholen. Beispiel: Ein **ausgehender Adapter** transformiert ein Domänenobjekt in ein Datenbank-Statement, führt die Abfrage aus und übersetzt das Ergebnis zurück in Domänenobjekte.

Ob ein Adapter eingehend oder ausgehend ist, entscheidet allein, **wer den Auftrag initial gegeben hat**. Dabei betrachten wir die Si-

tuation stets **aus Sicht der Domäne (unserer Insel)**. Entscheidend ist der **Kontrollfluss**, nicht der Datenfluss – also wer die Initiative ergreift und den Prozess startet.

Gesamtbild

In *Abbildung 9* seht ihr nun alle Bestandteile noch einmal in einer Übersicht zusammengefasst.

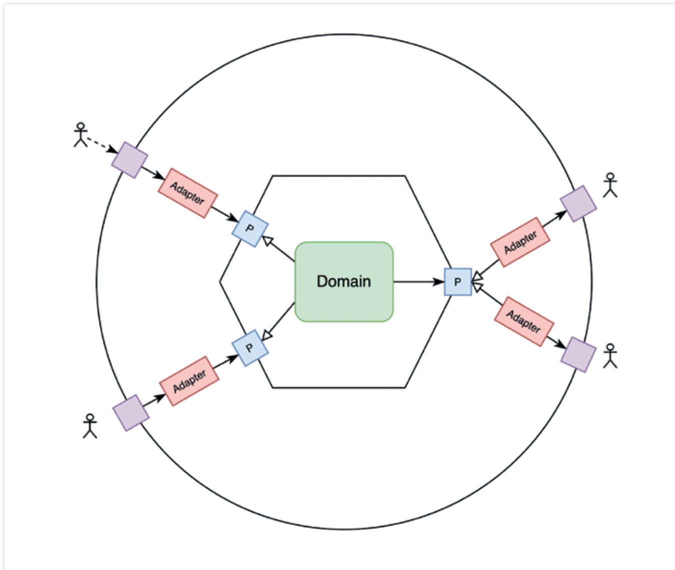


Abbildung 9: Gesamtschaubild der hexagonalen Architektur (© Danny Keller)

Wie ist die Domäne aufgebaut?

Obwohl wir bereits intensiv über Häfen, Handelsrouten und Schiffe gesprochen haben, haben wir bislang kaum einen Blick auf das Innenleben der Insel selbst geworfen. Dabei wären folgende Fragen durchaus naheliegend: Was passiert mit den ankommenden Nachrichten und Gütern? Wie verlaufen die Wege innerhalb der Stadt?

Es ist kein Zufall, dass diese Aspekte bisher unberührt blieben. Die konkrete Gestaltung der Domäne ist kein Bestandteil der hexagonalen Architektur. Vielmehr konzentriert sich dieses Architekturmuster ausschließlich auf die klare Trennung zwischen der Domäne und ihrer Umwelt. Im Zentrum steht die Gestaltung von Schnittstellen (Ports) und Adaptern, über die die Kommunikation mit der Außenwelt erfolgt.

Wie genau die internen Strukturen aufgebaut sind, wird durch andere Prinzipien bestimmt. Konzepte wie Domain-Driven Design (DDD) geben hier sinnvolle Hilfestellungen. Entscheidend ist: Die hexagonale Architektur schreibt der Domäne selbst nichts vor, sondern schafft lediglich den äußeren Schutzraum, damit sie unabhängig agieren kann.

Der wichtigste Leitsatz lässt sich so zusammenfassen: **Die Domäne muss von der Außenwelt entkoppelt sein**.

Vom Hafen zum Java Interface

Doch wie setzen wir das Ganze nun im Code um? Dazu hilft ein Blick auf ein typisches Architekturdiagramm *(siehe Abbildung 10)*.

In diesem Diagramm unterscheiden wir klar zwischen **eingehenden** und **ausgehenden Adaptern** – also unseren „Schiffen", die zwischen der Außenwelt und der Domäne unterwegs sind. Die Domäne selbst ist ebenfalls abgebildet, sie bildet unsere „Insel" und enthält die eigentlichen **Services** und **Ports**.

Anhand der Pfeile im Diagramm lässt sich gut erkennen, wer wen nutzt und wer wen implementiert. Besonders wichtig ist dabei der **ausgehende Port**, also das Interface, das von der Domäne definiert und von der Infrastruktur implementiert wird. Er ist zwingend notwendig, um die Abhängigkeitsrichtung umzukehren, ganz im Sinne des **Dependency Inversion Principle**. Der **eingehende Port** hingegen ist technisch nicht zwingend erforderlich: Ein eingehender Adapter kann theoretisch auch direkt einen Service der Domäne aufrufen.

Trotzdem ist es empfehlenswert, auch auf der eingehenden Seite einen Port zu definieren. Dadurch erreicht man eine **losere Kopplung** und verbessert insbesondere die **Testbarkeit** des Systems. Adapter und Services lassen sich so separat voneinander testen, und die Schnittstellen zwischen ihnen bleiben stabil und klar definiert.

Wie sieht das nun konkret im Code aus?

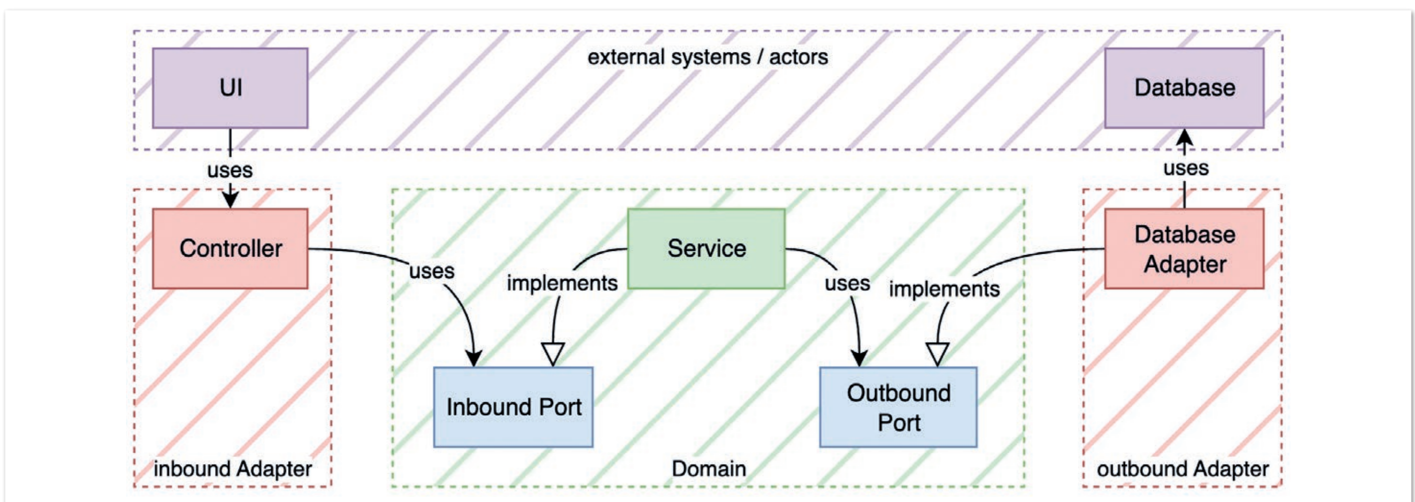


Abbildung 10: Implementierung der Hexagonalen Architektur im Überblick (© Danny Keller)

Eine Klasse, die ein Interface nutzt, also auf einen Port zugreift, deklariert dieses Interface einfach im Konstruktor *(siehe Listing 1)*.

```
// uses
@Service
class SomeService(private val forUsingPort: ForUsingPort) {
     // hier kommt die Logik des Services rein
}
```

Listing 1: Ein Service, der einen Port nutzt

Möchte eine Klasse ein Interface implementieren, also einen ausgehenden Port erfüllen, dann wird dieses Interface in der Klassendeklaration implementiert *(siehe Listing 2)*.

```
// implements
@Service
class SomeAdapter : ForImplementingPort {
     override fun functionOfPort() {
           // hier kommt die Logik des Adapters rein
     }
}
```

Listing 2: Ein Adapter, der einen Port implementiert

Und das ist im Grunde auch schon alles. Die Magie entsteht durch die klare Trennung von Rollen und Abhängigkeiten, nicht durch technische Komplexität. Mit dieser Struktur seid ihr in der Lage, eure Geschäftslogik sauber von technischen Details zu entkoppeln – ganz so, wie es die hexagonale Architektur vorsieht.

Fazit

Die hexagonale Architektur wirkt auf den ersten Blick komplex oder sogar überkonstruiert. Doch hinter dem Begriff verbirgt sich ein überraschend simples und klares Architekturprinzip.

Viele Teams starten mit der klassischen **Drei-Schichten-Architektur**, weil sie bekannt und leicht verständlich ist. Doch sobald Systeme wachsen und Anforderungen sich ändern, stößt dieses Modell schnell an seine Grenzen, vor allem wegen der **fest verdrahteten Abhängigkeiten** zwischen den Schichten. Fachlogik wird zu stark mit technischen Details vermischt, was die Testbarkeit, Wartbarkeit und Wiederverwendbarkeit deutlich einschränkt.

Genau hier setzt die hexagonale Architektur an. Sie hilft, diese strukturellen Schwächen zu beheben, indem sie drei zentrale Prinzipien betont:

- Die wichtigste Regel lautet: **Trenne die Domäne von der Außenwelt**. Geschäftslogik und Datenverarbeitung sollten unabhängig von Datenbanken, Frameworks oder externen Systemen gestaltet werden. So bleiben sie flexibel, leichter testbar und langfristig wartbar.
- **Technische Details haben in der Domäne nichts verloren.**
- Verwende **Ports**, um deine Domäne abzugrenzen und die Kommunikation kontrolliert über Adapter zu gestalten.

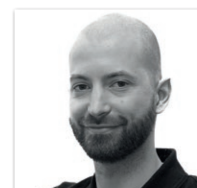
Hat man diese Prinzipien einmal verinnerlicht, lässt sich die hexagonale Architektur in nahezu jedem Projekt sinnvoll anwenden auch in kleineren Systemen, bei denen eine entkoppelte Domäne schnelle und isolierte Tests ermöglicht. Eine Geschäftslogik, die nicht direkt an die Infrastruktur gekoppelt ist, lässt sich viel einfacher pflegen, erweitern und testen. Die hexagonale Architektur schafft dafür einen klaren, strukturellen Rahmen.

Referenzen:

[1] *https://www.baeldung.com/java-dependency-inversion-principle*

Danny Keller
codecentric AG

Danny Keller ist seit 2021 als IT-Consultant und Developer bei der codecentric AG tätig. Seine fachlichen Schwerpunkte liegen in der Softwareentwicklung mit Kotlin, Softwarearchitektur und Domain Driven Design. In seiner Beratungstätigkeit unterstützt er Unternehmen dabei, komplexe Softwaresysteme zu konzipieren und nachhaltige, wartbare Lösungen zu entwickeln.



Ferdi Ade
codecentric AG

Ferdi Ade ist als Consultant & Developer und Trainer bei der codecentric AG unterwegs. Software Crafting und Domain Driven Design (DDD) sind ihm ein Anliegen und er arbeitet gerne eng mit anderen zusammen, um das zu bauen, was wirklich gebraucht wird. Außerdem ist er Co-Host beim Software Craft Leipzig Meetup und ist überzeugt, dass Hands-on-Sessions der beste Weg sind, um neue Fähigkeiten zu erlernen.

Exception Handling in Java: Klassiker mit modernem Schliff

Dr. Fadil Kallat, codecentric AG

Die Behandlung von Fehlern zählt zu den zentralen Aufgaben in der Softwareentwicklung. Während in frühen Programmiersprachen noch mit Rückgabecodes gearbeitet wurde, verfolgen moderne Hochsprachen, darunter auch Java, ein eigenes Konzept zur Modellierung von Fehlerzuständen: Exceptions.

Am grundlegenden Mechanismus hat sich seit den Anfängen wenig verändert. Dennoch gab es über die Jahre hinweg einzelne Erweiterungen und Verbesserungen im Umgang mit Exceptions, etwa beim Ressourcenmanagement oder im Bereich des Loggings. In diesem Artikel werfen wir einen Blick auf diese Entwicklungen und analysieren zugleich typische Fallstricke im Exception Handling. Anhand konkreter Anti-Patterns und Bad Smells zeigen wir auf, wie sich Ausnahmesituationen eleganter und robuster behandeln lassen und welche neueren Sprachmittel uns dabei unterstützen.

Grundlagen

In älteren Sprachen wurden Fehler häufig durch Rückgabecodes wie -1 oder `null` signalisiert, die aktiv geprüft werden mussten. Das erschwert die Lesbarkeit und vermischt Fehlerbehandlung mit Geschäftslogik. Java verfolgt einen anderen Ansatz, indem es Fehler über Exceptions signalisiert. Bei einem Ausnahmezustand wird der normale Programmfluss unterbrochen und eine definierte Fehlerbehandlung ausgelöst.

Dabei unterscheidet Java zwischen geprüften (checked) und ungeprüften (unchecked) Exceptions. Erstere stehen für Fehler außerhalb der Kontrolle der Anwendung, zum Beispiel eine `FileNotFoundException`. Diese müssen entweder behandelt oder deklariert werden, sodass ein Übersehen durch den Compiler kaum möglich ist. Ungeprüfte Exceptions wie `NullPointerException` signalisieren in der Regel Programmierfehler, von denen sich die Anwendung nicht erholen kann *[1]*.

Exception Handling – Status Quo

Studien zeigen, dass in vielen Java-Projekten funktionale Anforderungen Vorrang haben, während ein sauberes Exception Handling oft vernachlässigt wird. Die Folgen sind schlecht wartbarer Code, erschwerte Fehlersuche und potenzielle Sicherheitsrisiken *[2]*.

Begegnet man typischen Mustern wie Catch-All-Blöcken oder der Steuerung des Kontrollflusses über Exceptions, lohnt ein genauer Blick. Im Folgenden analysieren wir konkrete Beispiele und zeigen bessere Alternativen. Dabei unterscheiden wir zwischen Bad Smells, die auf strukturelle Schwächen hinweisen, und Anti-Patterns, die in der Umsetzung mehr schaden als nützen.

Bad Smells

In *Listing 1* sehen wir einen Klassiker der Bad Smells zum Thema Exception Handling. Hier wird im catch-Block die Klasse `Throwable` gefangen, die die Superklasse aller ungeprüften und geprüften Exceptions sowie Errors darstellt. In abgeschwächter Form wird so statt `Throwable` die Klasse `Exception` gefangen.

```
try {
  // …
} catch (Throwable e) {
  logger.error("Fehler aufgetreten", e);
}
```

Listing 1: Der Catch-All-Block fängt alle Fehler und verhindert gezielte Behandlung.

Das Problematische an diesem Bad Smell ist eine fehlende gezielte Ausnahmebehandlung. Zwar kann über `instanceof` geprüft werden, von welchem Typ der Fehler ist und dann eine Fehlerbehandlung stattfinden. Jedoch ist dieses Vorgehen nicht robust gegenüber Änderungen, denn sobald weitere Exceptions im korrespondierenden `try`-Block geworfen werden, muss der `catch`-Block um zusätzliche `if`-Abfragen ergänzt werden. Andernfalls könnte die Exception unter Umständen verloren gehen.

Eine verbesserte Vorgehensweise besteht darin, Ausnahmen gezielt in separaten catch-Blöcken zu behandeln. Wird im `try`-Block eine zusätzliche, geprüfte Exception geworfen, erzwingt der Compiler deren Behandlung, sofern nicht bereits eine passende Oberklasse abgefangen wird. Ist die Behandlung mehrerer Exceptions identisch, bietet das mit Java 7 eingeführte Multi-Catch-Konstrukt eine elegante Möglichkeit, um redundante Behandlungslogik zu vermeiden. *Listing 2* zeigt ein entsprechendes Beispiel.

```
try {
  // …
} catch (ErsteException | ZweiteException e) {
  behandleFehler(e);
}
```

Listing 2: Selektives Fangen bekannter Exceptions statt Catch-All

Listing 3 zeigt einen häufigen Bad Smell: Eine Exception wird zwar gefangen, aber weder behandelt noch protokolliert. Stattdessen findet sich oft der Hinweis, der Fehler könne „nicht auftreten". Jedoch lassen sich solche Annahmen technisch nicht absichern. Tritt die Ausnahme doch ein, fehlen Logeinträge, was die Fehlersuche erheblich erschwert. Folgefehler bleiben dadurch oft lange unentdeckt oder lassen sich nicht eindeutig zurückverfolgen.

Mögliche Gegenmaßnahmen bestehen in einer angemessenen Behandlung der Ausnahme, sofern diese technisch und fachlich sinnvoll ist. Ist das nicht der Fall, sollte die Ausnahme weitergereicht statt stumm abgefangen werden. Selbst bei als unwahrscheinlich eingeschätzten Fehlern ist zumindest eine Logmeldung ratsam, um die spätere Analyse nicht zu behindern.

```
try {
  // …
} catch (EineSpezifischeException e) {
  // nichts tun
}
```

Listing 3: Verschluckte Exception ohne Behandlung oder Logging

Anti-Pattern

Das erste Anti-Pattern zeigt sich exemplarisch in *Listing 4*. Dort wird ein Array ohne explizite Abbruchbedingung durchlaufen. Stattdessen wird eine `ArrayIndexOutOfBoundsException` geworfen, um den Schleifenabbruch auszulösen. Die Exception wird dabei zweckentfremdet und dient nicht der Fehlerbehandlung, sondern der Steuerung des Kontrollflusses. Das ist ein typisches Beispiel für den Missbrauch von Ausnahmen.

```
try {
  for (int i = 0;/ ** /; i++)
      verarbeite(array[i]);
} catch (ArrayIndexOutOfBoundsException e) {
  // Array vollstaendig iteriert
}
```

Listing 4: Array-Iteration durch Ausnahme gesteuert

In *Listing 5* wird im `try`-Block ein Konto geladen. Gehört der Kunde stattdessen zu einem Gruppenkonto, wird eine Exception geworfen, um diesen Sonderfall zu erkennen und eine alternative Methode aufzurufen. Auch hier wird die Ausnahme zur Steuerung fachlicher Abläufe verwendet und nicht zur Behandlung eines Fehlers.

```
try {
  holeKonto(kunde);
} catch (KundeHatEinGruppenKontoException e) {
  holeGruppenkonto(kunde);
}
```

Listing 5: Steuerung des Kontrollflusses durch Ausnahmebehandlung

Listing 4 und *Listing 5* haben gemeinsam, dass der Kontrollfluss durch das Auslösen von Exceptions gesteuert wird. Dieses Vorgehen ist aus mehreren Gründen problematisch: Zum einen leidet die Lesbarkeit. In *Listing 4* ist das Ende der Schleife nur durch das Verständnis der ausgelösten `ArrayIndexOutOfBoundsException` erkennbar. In *Listing 5* wird die fachliche Unterscheidung im catch-Block getroffen, was den Ablauf schwer nachvollziehbar macht.

Zum anderen widerspricht diese Technik dem Prinzip der geringsten Überraschung. Exceptions sollen unerwartete Zustände signalisieren und nicht zur Steuerung regulärer Abläufe verwendet werden. Darüber hinaus entsteht eine enge Kopplung an das Java-Modell der Fehlerbehandlung, was Portierungen in andere JVM-Sprachen erschwert.

Ein weiterer Nachteil betrifft die Performanz: Jede geworfene Exception erzeugt einen Stacktrace, was mit Laufzeitkosten verbunden ist. Wird dieses Mittel regelmäßig zur Steuerung eingesetzt, leidet die Effizienz der Anwendung.

Eine Alternative ist die Validierung vor dem Zugriff. So lässt sich in vielen Fällen ein regulärer Geschäftsfall erkennen, ohne eine Ausnahme zu provozieren. Allerdings ist das nicht immer zuverlässig, so beispielsweise bei Race Conditions, wenn sich der Zustand zwischen Prüfung und Zugriff verändert, wie beim gleichzeitigen Löschen einer Datei. In solchen Fällen bleibt das Exception Handling notwendig. Es sollte dann gezielt als Absicherung gegen nicht vermeidbare Fehler eingesetzt werden, jedoch nicht zur Steuerung des normalen Programmablaufs.

Eine Alternative zur Vermeidung von `null`-Prüfungen ist der Einsatz von Optional, das seit Java 8 zur Sprache gehört. Es macht die Abwesenheit eines Werts explizit und reduziert das Risiko von `NullPointerException`. Die funktionale API mit Methoden wie `.map()`, `.filter()` oder `.orElseThrow()` erlaubt eine kompakte Verarbeitung. In *Listing 6* wird das Gruppenkonto nur dann geladen, wenn kein reguläres Konto gefunden wurde.

```
Optional<Konto> kontoOpt = holeKonto(kunde);
return kontoOpt.orElse(holeGruppenkonto(kunde));
```

Listing 6: Fallback-Logik mit Optional bei fehlendem Wert

Eine weitere Möglichkeit, Exceptions als Steuerungsmechanismus zu vermeiden, bietet das Spezialfall-Muster *[3]*. Dabei werden Sonderfälle wie fehlende Daten nicht über Ausnahmen oder `null` modelliert, sondern durch spezialisierte Objekte mit abweichendem Verhalten. Diese implementieren die gleiche Schnittstelle wie die regulären Fälle, unterscheiden sich aber in der fachlichen Logik. So lässt sich die Verzweigungslogik vereinfachen und das Risiko für Fehler durch `null`-Zugriffe reduzieren. *Abbildung 1* zeigt eine beispielhafte Klassenstruktur nach diesem Muster.

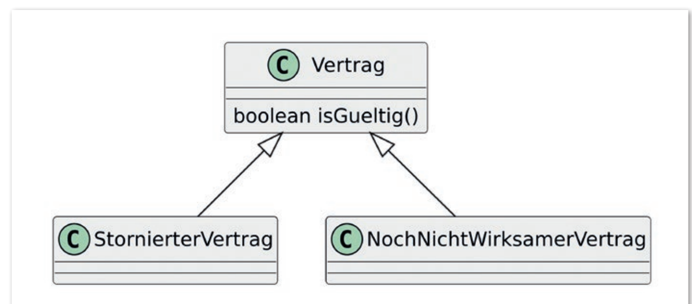


Abbildung 1: Klassenhierarchie nach dem Spezialfallmuster (© Dr. Fadil Kallat)

In der exemplarischen Umsetzung des Spezialfall-Musters stellt die Klasse „Vertrag“ den regulären Fall dar. Sie definiert etwa eine Methode zur Gültigkeitsprüfung (`isGueltig()`), deren Verhalten durch Unterklassen wie „NochNichtWirksamerVertrag“ oder „StornierterVertrag“ für Sonderfälle überschrieben wird. So lässt sich spezielles Verhalten ohne explizite Fallunterscheidung in der Verarbeitung modellieren, da stets mit der Oberklasse Vertrag gearbeitet werden kann.

Ein weiteres Anti-Pattern veranschaulicht *Listing 7*. Hier wird im `try`-Block eine Exception geworfen und im `finally`-Block eine weitere – meist eine RuntimeException. Das Problem: Die ursprünglich ausgelöste Ausnahme geht verloren, wodurch wichtige Informationen zur Fehlerursache nicht mehr im Log erscheinen. Dieses Verhalten erschwert die Analyse und fällt in das Muster der verschluckten Exceptions.

```
try {
  throw new IOException("Urspruenglicher Fehler");
} finally {
  cleanUp(); // wirft eine RuntimeException
}
```

Listing 7: Ursprüngliche Exception geht durch Exception im finally-Block verloren.

Wie lässt sich der Verlust der ursprünglichen Exception vermeiden? Eine Möglichkeit besteht darin, die erste Ausnahme zwischenzuspeichern und mit `addSuppressed` (seit Java 7) an eine zweite anzuhängen. So bleiben beide erhalten. In der Praxis führt dieses Vorgehen jedoch oft zu verschachtelten `try-catch-finally`-Blöcken, die schwer lesbar und kaum wartbar sind.

Deutlich eleganter ist `try-with-resources`, ebenfalls seit Java 7 verfügbar. Es übernimmt das automatische Schließen von Ressourcen und fügt auftretende Fehler beim Schließen als Suppressed Exceptions hinzu. Manuelles Verketten entfällt. Ab Java 9 können Ressourcen auch außerhalb der `try`-Klammer deklariert werden. *Listing 8* zeigt ein Beispiel mit `FileReader` und `BufferedReader`, bei dem kein explizites `close()` mehr nötig ist. Java übernimmt die Verwaltung im Hintergrund.

```
String leseErsteZeile(String pfad) throws IOException {
  FileReader fileReader = new FileReader(pfad);
  BufferedReader buffReader = new BufferedReader (fileReader);
  try (fileReader; buffReader) {
   return buffReader. readLine();
  }
}
```

Listing 8: Verwendung von try-with-resources, um eine Datei zu lesen.

Auch wenn `try-with-resources` bereits seit Java 7 Teil der Sprache ist, begegnen mir in vielen Projekten nach wie vor manuell geschlossene Ressourcen – mitsamt der bekannten Risiken. Ich führe das Konstrukt deshalb hier bewusst erneut auf, weil es eine einfache und zugleich wirkungsvolle Möglichkeit bietet, Code robuster und klarer zu gestalten.

Kommen wir zu einem dritten Anti-Pattern. Es basiert auf einem Vorschlag von ChatGPT (Modell GPT-4o), der als besonders elegante Lösung präsentiert wurde. In *Listing 9* sehen wir das neue Pattern Matching im switch, mit dem verschiedene Exception-Typen verarbeitet werden. Der Ansatz soll laut Modell die Lesbarkeit verbessern und die Unterscheidung der Fehlertypen vereinfachen.

```
try {
  //…
} catch (EineException e) {
  switch (e) {
     case MyException myEx -> // ...
     case OtherException otherEx -> //...
     default -> // …
  }
}
```

Listing 9: Verwendung vom neuen `switch-case`-Pattern-Matching zur Ausnahmebehandlung

Tatsächlich wirkt das Muster auf den ersten Blick strukturiert. Unterschiedliche Exceptions werden an zentraler Stelle typbasiert behandelt. Doch dieser Stil bringt, wie wir gleich sehen werden, mehrere Probleme mit sich.

Bei genauerem Hinsehen zeigt sich, dass dieser Ansatz erhebliche Schwächen hat. Er beruht auf `instanceof`-Prüfungen und birgt die Gefahr, dass bestimmte Exceptions nicht korrekt behandelt werden. Wird etwa eine neue Unterklasse einer bekannten Exception geworfen, landet sie möglicherweise im `default`-Zweig, ohne dass dies auffällt.

Hinzu kommt ein konzeptioneller Nachteil: Der Ansatz verletzt das Offen-Geschlossen-Prinzip. Jede neue oder entfallene Exception erfordert Änderungen am bestehenden Code, was die Wartbarkeit einschränkt.

Deshalb empfiehlt es sich, statt auf switch-Konstrukte oder `instanceof`-Kaskaden auf klassische `catch`-Blöcke zu setzen. Sie sind für den Umgang mit Ausnahmen konzipiert, führen zu besser lesbarem Code und entsprechen der Intention der Sprache.

Die in diesem Artikel vorgestellten Beispiele stellen eine bewusst ausgewählte Teilmenge häufiger Fehlermuster dar. Eine umfassende Übersicht über typische Bad Smells und Anti-Patterns im Exception Handling liefert Rocha et al., die in einer systematischen Auswertung von Literatur, Fachbeiträgen und statischer Codeanalyse eine entsprechende Kategorisierung vorgenommen haben *[4]*.

Logging von Exceptions

Ein zentraler Bestandteil einer sinnvollen Fehlerbehandlung ist das Logging. Dabei ist nicht nur wichtig, welcher Fehler auftritt, sondern auch, wo er auftritt. Daraus ergibt sich eine klare Regel: Exceptions sollten immer vollständig geloggt werden, das heißt nicht nur die Fehlermeldung, sondern auch inklusive Stacktrace. Moderne Logging-Frameworks wie SLF4J unterstützen dies direkt über Methoden, die Exceptions als Parameter entgegennehmen und korrekt ausgeben.

Auch die Java-Plattform selbst hat hier nachgebessert. In Java 21 wurde die Darstellung von Suppressed Exceptions klarer strukturiert, in Java 17 zeigt eine `NullPointerException` zusätzlich die betroffene Variable. Solche Detailverbesserungen erleichtern die Analyse erheblich.

Oft enthält ein Stacktrace jedoch auch irrelevante Informationen, etwa aus Drittanbieterbibliotheken. Genau hier hilft die mit Java 9

```
try {
     // ...
} catch (CustomException e) {
  logger.error("Fehler aufgetreten: ", e);
  String filteredStacktrace = StackWalker.getInstance()
.walk(frames -> frames
      .filter(f -> f.getClassName().startsWith("com.meineapp"))
      .map(StackFrame::toString)
      .collect(Collectors.joining("\n")));
  logger.error("Gefiltert:\n{}", filteredStacktrace);
}
```

Listing 10: Verwendung der StackWalker-API, um lediglich die Stacktrace-Elemente der eigenen Anwendung zu loggen.

eingeführte StackWalker-API. Sie ermöglicht es, Stacktraces gezielt zu durchlaufen sowie zu filtern und beispielsweise auf Klassen des eigenen Projekts zu beschränken. So bleibt der Log-Auszug übersichtlich und auf das Wesentliche fokussiert.

Listing 10 zeigt ein Beispiel für den gezielten Einsatz der StackWalker-API. Hier werden nur die Stacktrace-Elemente geloggt, die der eigenen Anwendung zuzuordnen sind. Das reduziert unnötigen Output und erhöht die Übersichtlichkeit. Weiterführende Hinweise zur Nutzung der API finden sich in der offiziellen Dokumentation *[5]*.

Fazit

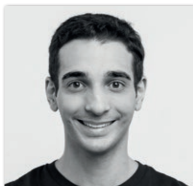
In diesem Artikel haben wir typische Fallstricke beim Exception Handling beleuchtet. Besonders hervorzuheben ist `try-with-resources`, das den Umgang mit Ressourcen vereinfacht, Exceptions zuverlässig behandelt und unnötigen Code reduziert. Ebenso wichtig ist der Grundsatz, Exceptions nicht zur Steuerung regulärer Abläufe einzusetzen. Wer das beachtet, schreibt nicht nur effizienteren Code, sondern erleichtert auch das Verständnis des Codes für Mitentwickelnde.

Ein weiteres Ziel war der Blick auf Neuerungen im Java-Exception-Handling. Diese fielen in den letzten Jahren eher zurückhaltend aus, etwa in Form verbesserter Ausgaben bei NullPointerException oder Suppressed Exceptions. Ergänzend kamen API-Anpassungen wie beim ExecutorService hinzu, die eine präzisere Fehlersteuerung in parallelen Abläufen ermöglichen.

Das alles zeigt: Das Exception Handling in Java ist stabil und bewährt. Größere konzeptionelle Änderungen waren bislang nicht erforderlich.

Quellen

[1] Christian Ullenboom (2024): *Java ist auch eine Insel*. Rheinwerk, Bonn.
[2] Felipe Ebert, Fernando Castor, Alexander Serebrenik (2015): *An exploratory study on exception handling bugs in Java programs*. Elsevier Science Inc., New York, USA.
[3] Martin Fowler (1999): *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, Boston, USA.
[4] Rocha, Jonathan, et al. (2018): *Towards a catalogue of Java exception handling bad smells and refactorings*. The Hillside Group, USA.
[5] Oracle (2023): StackWalker API Spezifikation. Oracle Corporation. Online verfügbar unter: *https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/StackWalker.html*



Dr. Fadil Kallat
codecentric AG
fadil.kallat@codecentric.de

Dr. Fadil Kallat ist IT-Consultant bei codecentric AG mit Fokus auf Full-Stack-Entwicklung und Softwarearchitektur. Er verfügt über langjährige Erfahrung in Java, modernen Webtechnologien und CI/CD-Prozessen. Neben seiner Promotion in Informatik hält er Fachvorträge und veröffentlicht regelmäßig zu IT-Themen.

Shift-Left mit System: Fachliche Anforderungen dokumentieren und testen

Felix Tensing, Nürnberger Versicherung

Frühe Tests versprechen bessere Qualität, schnellere Rückmeldung und dadurch geringere Kosten – das Prinzip „Shift-Left" ist aus modernen Entwicklungsprozessen kaum wegzudenken. Doch in der Praxis scheitert es oft an einem klaren Verständnis davon, was genau getestet werden soll. Dieser Artikel richtet sich an Entwickler:innen, Tester:innen, Product Owner – und alle, die ihre Anforderungen nicht nur besser dokumentieren, sondern auch automatisiert testen und langfristig nachvollziehen wollen. Er zeigt, wie sich fachliche Anforderungen so dokumentieren lassen, dass sie nicht nur verständlich, sondern auch testbar und versionierbar sind – und so eine nachhaltige Grundlage für Testautomatisierung und Zusammenarbeit schaffen.

„Shift-Left" beschreibt das Prinzip, Qualitätssicherung frühzeitig in den Entwicklungsprozess zu integrieren. Früh entdeckte Fehler lassen sich meist deutlich günstiger beheben. Werden sie dagegen erst spät gefunden – etwa in der Produktion – steigen Aufwand und Kosten erheblich. Dabei können Fehler nicht nur Bugs sein, sondern auch fachliche Ungereimtheiten oder eine falsche Implementierung aufgrund missverständlicher Anforderungsdefinition.

Doch Shift-Left bedeutet nicht nur frühzeitiges Testen, sondern auch eine erweiterte Verantwortung für die Entwicklungsteams. Sie müssen bereits während der Implementierung sicherstellen, dass Anforderungen korrekt umgesetzt und potenzielle Probleme rechtzeitig erkannt werden. Das setzt eine enge Verzahnung von Anforderungen, Dokumentation und Tests voraus. Im Artikel werden diese Zusammenhänge näher erläutert und ein Ansatz vorgestellt, der den Einsatz von Shift-Left fördert und gleichzeitig die Aufwände im Team gering hält.

Dokumentation ist nicht gleich Dokumentation

In der IT existieren viele unterschiedliche Arten von Dokumentationen – mit unterschiedlichen Zielgruppen, Formaten und Zwecken. Um Missverständnisse zu vermeiden, lohnt es sich, diese Begriffe kurz zu differenzieren. Die folgende Einordnung ist nicht vollständig, sondern konzentriert sich auf jene Formen von Dokumentation, die für das hier vorgestellte Vorgehen relevant sind – oder häufig zu Verwirrung führen.

Dokumentation im Quellcode umfasst sowohl JavaDoc als auch Kommentare. Diese dienen Entwickler:innen dazu, technische Entscheidungen zu erläutern und den Code verständlich zu machen. JavaDoc dokumentiert dabei insbesondere die öffentliche API von Klassen, Methoden und Schnittstellen – also wie Komponenten genutzt werden können. Solche Dokumentation ist eng an die Implementierung gebunden und ändert sich meist mit ihr. Für das fachliche Verständnis des Gesamtsystems ist sie jedoch nur bedingt geeignet.

Technische Systemdokumentation umfasst Artefakte wie Architekturmodelle, Umsystembeschreibungen, technische Vorgaben oder Architekturentscheidungsprotokolle (Architecture Decision Records, ADRs). Ein bekanntes Beispiel ist das Arc42-Template, das als strukturierter Leitfaden zur Dokumentation von Softwarearchitektur weit verbreitet ist. Diese Dokumentation liefert wichtige technische Orientierung für Entwicklung und Betrieb. Ihr Fokus liegt jedoch auf technischen Zusammenhängen und Infrastruktur, nicht auf der fachlichen Logik oder den Geschäftsregeln eines Systems.

Fachliche Dokumentation beschreibt, was ein System aus Sicht der Anwender:innen oder Fachabteilungen leistet – unabhängig von der technischen Umsetzung. Sie bildet die Grundlage für ein gemeinsames Verständnis der Geschäftsregeln, Prozesse und erwarteten Funktionen. Fachliche Dokumentation ist damit die Basis für die Testbarkeit der Anforderungen.

Wird in diesem Artikel von Dokumentation gesprochen, ist damit die fachliche Dokumentation gemeint, sofern nicht explizit eine andere Form von Dokumentation genannt wird.

Anforderung ist nicht gleich Feature

Anforderungen beschreiben, was ein System aus fachlicher Sicht leisten soll. Sie formulieren ein Ziel oder Bedürfnis, unabhängig von der konkreten technischen Umsetzung. Features hingegen sind konkrete Funktionseinheiten, mit denen Anforderungen umgesetzt werden – oder bereits umgesetzt wurden.

In der Praxis wird der Begriff „Feature" oft schon während der Planung oder Entwicklung verwendet, obwohl die Funktion noch nicht umgesetzt ist. Streng genommen handelt es sich dabei um einen *Feature-Wunsch* – also eine geplante Erweiterung. Im Alltag wird dieser Begriff jedoch häufig verkürzt und ebenfalls als „Feature" bezeichnet. Diese begriffliche Unschärfe ist ein typisches Beispiel für kommunikative Schwächen in Softwareprojekten – und ein häufiger Auslöser für Missverständnisse.

Die Beziehung zwischen Anforderungen und Features ist häufig nicht eindeutig: Ein Feature kann mehrere Anforderungen gleichzeitig erfüllen und eine einzelne Anforderung kann sich auf verschiedene Features verteilen. So führt etwa die Anforderung „Benutzer:innen können sich anmelden" zu Features wie „Login mit Benutzername und Passwort" sowie „Anmeldung über Single Sign-On (SSO)". Gleichzeitig erfüllt das Feature „SSO" auch technische oder sicherheitsbezogene Anforderungen, etwa zur zentralen Authentifizierung.

Was passiert eigentlich mit Anforderungen nach der Umsetzung?

Anforderungen bilden in vielen Projekten den Ausgangspunkt für die Umsetzung: Sie beschreiben gewünschte Funktionen, Ziele oder Rahmenbedingungen. In frühen Projektphasen dienen sie als Grundlage für Planung, Aufwandsschätzung und technische Konzeption.

Doch sobald die Umsetzung erfolgt ist und ein Feature entstanden ist, verliert die ursprüngliche Anforderung oft an Sichtbarkeit – oder verschwindet ganz aus dem Fokus.

In der Praxis werden Anforderungen selten versioniert oder aktualisiert. Bei Weiterentwicklungen kommen neue Anforderungen hinzu, die bestehende Anforderungen ändern oder erweitern – ohne dass die ursprünglichen Anforderungen systematisch angepasst wurden. So entsteht ein wachsender Strom an Änderungswünschen, während die alte Anforderungslage in den Hintergrund tritt oder schlicht veraltet.

Das führt dazu, dass Anforderungen in vielen Projekten nicht weiter gepflegt werden – und deshalb als dauerhafte Referenz oder Dokumentationsbasis nicht geeignet sind.

Im Sprachgebrauch ist mit „der Anforderung" oft nicht ein einzelner Wunsch gemeint, sondern ein ganzes Themenfeld – etwa „Login" oder „Zahlungsabwicklung". Dabei bleibt meist unklar, welche konkreten Funktionen, Regeln oder Ausnahmen damit gemeint sind. Diese Unschärfe erschwert die Aussage, ob eine Anforderung vollständig umgesetzt oder getestet wurde – und ist damit ein zentrales Problem bei der Anforderungsabdeckung (siehe Kasten: Anforderungsabdeckung erklärt).

Anforderungsabdeckung erklärt

Anforderungsabdeckung (Requirements Coverage) bezeichnet das Ausmaß, in dem definierte Anforderungen durch Tests überprüft werden. Ziel ist es, sicherzustellen, dass alle spezifizierten Anforderungen im Testprozess berücksichtigt werden. Eine klassische Definition findet sich bei Kaner, Falk & Nguyen (1999): „Requirements coverage is a measure of the extent to which the requirements are exercised by a test suite." *[1]*

In der Praxis zeigt sich jedoch, dass das reine Messen der Anforderungsabdeckung oft wenig Aussagekraft besitzt, da Anforderungen häufig unscharf formuliert sind oder sich im Verlauf der Softwareentwicklung verändern. Stattdessen ist die Featureabdeckung – also die Abdeckung der konkreten, implementierten Funktionalitäten – oft ein sinnvollerer Maßstab für Qualitätssicherung.

Allerdings ist der Begriff „Featureabdeckung" im üblichen Sprachgebrauch kaum etabliert, sodass in vielen Teams weiterhin von „Anforderungsabdeckung" gesprochen wird, obwohl eigentlich die Featureabdeckung gemeint ist.

Auch in diesem Artikel wird der Begriff Anforderungsabdeckung verwendet, wenn im Grunde die Abdeckung der Features gemeint ist.

Die Fachfunktion

Wie bereits gezeigt, sind Anforderungen vor allem ein Arbeitswerkzeug zur Umsetzung und nicht als Grundlage für dauerhafte Tests geeignet. Features – also die tatsächlich umgesetzten Funktionalitäten – sind die relevanten Objekte. Die Dokumentation der Features bildet die Basis für eine verlässliche und nachhaltige Anforderungsabdeckung. In diesem Artikel führe ich den Begriff Fachfunktion (siehe Kasten: Fachfunktion kurz und knapp) für eine strukturierte und nachvollziehbare Dokumentation der Features ein, die als verbindliche Grundlage für Tests und Qualitätssicherung dient. Dieser neue Begriff hilft, eine sprachliche Trennung zwischen Anforderungen und Features zu schaffen und macht deutlich, dass eine Fachfunktion ein Feature dokumentiert. In der Praxis hat sich gezeigt, dass diese Benennung dafür sorgt, dass allen Beteiligten klar ist, worum es geht, wenn von einer Fachfunktion die Rede ist.

Von der Anforderung zur Fachfunktion

Die Anforderung bildet die fachliche Grundlage, aus der in der Praxis meist mehrere User Stories oder Arbeitspakete abgeleitet werden. Während die Anforderung die Planungsebene darstellt, fokussieren sich Stories auf die konkrete Umsetzungsebene.

Entwickler:innen setzen diese Stories in Code um. Jede fertiggestellte Story führt dabei zu einer Änderung oder Ergänzung der Fachfunktionen. Da nur die Entwickler:innen genau wissen, welche Umsetzung tatsächlich erfolgt ist, dokumentieren sie in den Fachfunktionen den aktuellen Zustand der Software aus fachlicher Sicht. So entsteht eine Dokumentation, die stets den tatsächlichen Stand der Software widerspiegelt. Damit das funktioniert, ist es entscheidend, dass die Fachfunktion Teil des Codes ist – also nach dem Prinzip „Documentation as Code" (siehe Kasten: *Documentation as Code*) verwaltet wird. Nur so lässt sich sicherstellen, dass Dokumentation und Implementierung synchron bleiben.

Documentation as Code

„Documentation as Code" (oder kurz auch Docs-as-Code) bezeichnet einen Ansatz, bei dem Dokumentation nach denselben Prinzipien wie Quellcode behandelt wird. Sie liegt in textbasierter Form vor (zum Beispiel Markdown, AsciiDoc), wird versioniert in einem Versionskontrollsystem verwaltet, kann gemeinsam mit dem Code entwickelt und automatisiert geprüft, erzeugt oder verarbeitet werden.

Ziel ist es, Dokumentation zuverlässig, aktuell und integrativ in die Entwicklungsprozesse einzubinden – ohne Medienbrüche, manuelle Nachpflege oder separate Werkzeuge. Damit wird Dokumentation zum festen Bestandteil der Software – nachvollziehbar, überprüfbar und teamübergreifend nutzbar.

Da sie im gleichen Repositorium wie der Code liegt, kann die Dokumentation im Rahmen von Code-Reviews direkt mitgeprüft und versioniert nachvollzogen werden.

Der Ansatz eignet sich nicht nur für fachliche Dokumentation, sondern auch für Architektur-, Betriebs- oder Entwicklerdokumentation – überall dort, wo Nähe zum Code von Vorteil ist.
In vielen Teams und Organisationen gilt „Documentation as Code" heute als Best Practice.

Im Sinne des bereits erwähnten Shift-Left-Prinzips liegen fachliche Dokumentation und dementsprechend auch das Testen im Aufgabenbereich der Entwickler:innen. Damit das gut funktioniert, müssen die Aufwände dafür möglichst geringgehalten werden. Eine strukturierte Dokumentation hilft dabei, den Fokus auf die Inhalte zu legen, ohne durch unnötiges Layout oder administrative Aufgaben abzulenken.

Die Fachfunktion bietet einen solchen strukturierten Rahmen: Sie ermöglicht es, fachliche Funktionen klar und einheitlich zu beschreiben. Im Beispielprojekt (siehe Kasten: *Beispielprojekt*) des Autors wird YAML als Dateiformat genutzt. Denkbar sind aber auch andere Formate wie JSON oder auch XML. In *Listing 1* ist eine einfache Fachfunktion aufgeführt.

Beispielprojekt

Die hier gezeigten Codebeispiele sind frei verfügbar auf GitHub *[7]*.

Die Implementierung ist bewusst minimal gehalten, um die Verständlichkeit zu fördern. Weiterentwicklungen wie die Nutzung von Annotationen zur Testmarkierung wurden absichtlich nicht umgesetzt, um das Grundprinzip klar zu vermitteln.

Das Projekt kann als Vorlage zur Umsetzung in anderen Programmiersprachen dienen, solange das zugrundeliegende Prinzip erhalten bleibt. So hat der Autor beispielsweise auch eine Variante mit TypeScript, Vitest und Storybook für ein Next.js-Projekt realisiert.

Feedback, Rückfragen und Vorschläge sind willkommen – ebenso wie Merge-Requests. Nutzen Sie gern die GitHub-Issue-Funktion, um mit dem Autor und der Community in Kontakt zu treten.

Im Folgenden werden die einzelnen Bestandteile kurz erläutert:

name
Der Name benennt die Fachfunktion eindeutig. Er sollte möglichst präzise und fachlich sprechend sein.

kurzbeschreibung
Die Kurzbeschreibung gibt einen kompakten Überblick über die fachliche Aufgabe der Funktion. Sie ermöglicht ein schnelles Verständnis des Kontexts, ohne Details vorwegzunehmen.

akzeptanzkriterien
Die Akzeptanzkriterien beschreiben konkret, wann die Fachfunktion als korrekt umgesetzt gilt. Sie bilden die Grundlage für Tests und müssen entsprechend testbar formuliert sein. Ziel im Sinne von Shift-Left ist es, dass alle Kriterien auch automatisiert überprüft werden können. Dazu später mehr. Jedes Kriterium wird mit einer ID versehen, um gezielt darauf referenzieren zu können.

tags
Tags helfen, Fachfunktionen thematisch zu gruppieren oder mit zusätzlichen Attributen zu versehen. Sie können für Filterung, Auswertung oder einfach zur besseren Orientierung genutzt werden.

Diese einfache Struktur erlaubt es, Fachfunktionen konsistent und nachprüfbar zu dokumentieren – direkt im Code, versionierbar und testbar. Eine Erweiterung der Struktur an projektspezifische Anforderungen ist problemlos möglich – etwa um Angaben zu Rollen oder Berechtigungen zu ergänzen.

Zusätzlich zur strukturierten Dokumentation benötigt jede Fachfunktion eine ausführlichere fachliche Beschreibung. Sie dient dazu, Zusammenhänge zu erläutern, Hintergründe zu dokumentieren oder auf externe Quellen zu verweisen.

Im Beispielprojekt wird hierfür das Format AsciiDoc *[2]* verwendet. Die Beschreibung wird dabei als eigene Datei abgelegt, um eine bessere Unterstützung in der IDE zu ermöglichen – etwa durch Syntax-Highlighting, Vorschau und Validierung.

AsciiDoc bietet eine einfache, aber wirkungsvolle Möglichkeit zur ansprechenden und gut lesbaren Gestaltung technischer Texte – direkt im Editor. Es unterstützt unter anderem Überschriften, Formatierungen, Quellcode-Auszüge mit Syntax-Highlighting und Tabellen. Darüber hinaus lassen sich mit PlantUML *[3]* auch Diagramme und Abläufe direkt einbetten – vollständig textbasiert, versionierbar und ohne externe Tools. So können auch komplexe fachliche Zusammenhänge gut verständlich dokumentiert werden.

Listing 2 zeigt ein Beispiel der Beschreibung in AsciiDoc.

```yaml
---
name: Benutzer Login
kurzbeschreibung: Authentifizierung von Benutzer:innen anhand von Zugangsdaten.
akzeptanzkriterien:
 - 01: "Ein gültiger Benutzername und ein korrektes Passwort führen zur erfolgreichen Anmeldung."
 - 02: "Ein ungültiger Benutzername oder ein falsches Passwort führen zu einer Fehlermeldung."
 - 03: "Benutzer:innen ohne Aktivierung können sich nicht anmelden."
 - 04: "Die Anmeldung erzeugt ein gültiges JWT-Token mit Ablaufzeit."
tags:
 - Benutzername
 - Login
 - Passwort
```

Listing 1: FF-PROJ-0001.yaml, beispielhafte Fachfunktion zum Login

```
Der Login ist der Einstiegspunkt für alle geschützten Funktionen im System.
Benutzer:innen geben ihre Zugangsdaten ein und erhalten bei erfolgreicher Anmeldung ein Authentifizierungs-Token.


*Beispiel: Authentifizierungs-Token*
[source,json]
----
{
 "sub": "user123",
 "iat": 1616161616,
 "exp": 1616165216,
 "sessionId": "abc123xyz",
 "roles": ["user", "admin"]
}
----
```

Listing 2: FF-BSPP-0001.adoc, AsciiDoc Beispiel der Beschreibung einer Fachfunktion

Von der Dokumentation zur Testabdeckung

Im nächsten Schritt verbinden wir die Fachfunktion mit konkreten Tests – und schaffen damit die Grundlage zur Berechnung der Anforderungsabdeckung.

Dank der klaren Struktur verfügt jedes Akzeptanzkriterium über eine eindeutige ID. Wie gezeigt , setzt sich diese aus der Kennung der Fachfunktion und der Nummer des Akzeptanzkriteriums zusammen. Als Trenner wird ein Hashtag verwendet – etwa FF-BSPP-0001#02.

Automatisierte Tests können auf diese ID referenzieren. Im Beispielprojekt wird das Akzeptanzkriterium direkt im Namen des Tests mit aufgeführt *(siehe Listing 3)*. Diese Variante ist unkompliziert, gut verständlich und lässt sich ohne zusätzliche Implementierung umsetzen.

Natürlich kann ein Test auch mehrere Akzeptanzkriterien abdecken – insbesondere bei komplexeren Abläufen oder integrierten Testfällen. In diesem Fall lassen sich einfach mehrere IDs angeben.

Generell ist es wichtig, die Testnamen als aussagekräftige, gut verständliche Sätze zu formulieren. So lässt sich später aus der generierten Dokumentation unmittelbar erkennen, wie der Test auf die jeweiligen Akzeptanzkriterien einzahlt – ganz ohne den Testcode lesen zu müssen.

... und etwas Glue-Code

Mit der strukturierten Dokumentation der Fachfunktionen und der Verknüpfung der Tests mit den Akzeptanzkriterien sind die wichtigsten Grundlagen geschaffen, um eine Dokumentation mit Anforderungsabdeckung zu generieren.

Was für eine vollständige Lösung noch fehlt, sind ein Template zum Rendern, ein Skript zum Sammeln und Verarbeiten der vorhandenen Daten sowie einige Maven-Plugins zur Integration in den Build-Prozess.

Das Skript lässt sich vereinfacht wie folgt zusammenfassen:

- Scannen des Dokumentationsordners nach YAML-Dateien mit Fachfunktionen
- Ablegen der Informationen in einer internen Map
- Einlesen des Surefire-XML-Reports und Verknüpfen der Testergebnisse mit den Fachfunktionen in der Map
- Berechnung der prozentualen Anforderungsabdeckung: Ein Akzeptanzkriterium gilt als erfolgreich getestet, wenn mindestens

```
package de.fx.agiledocumentation

import spock.lang.Specification

class LoginTest extends Specification {

  def 'Login mit User Peter0815 und Passwort Geheim%*123 ist erfolgreich. [FF-PROJ-0001#01]'() {
        expect:
        true // Hier sollte die Logik für den Login-Test stehen, z.B. ein Aufruf einer Login-Methode
  }

  def 'Login mit User Hans4711 und Passwort FALSCHES_PASSWORT schlägt fehl. [FF-PROJ-0001#02]'() {
        expect:
        true // Hier sollte die Logik für den Login-Test stehen, z.B. ein Aufruf einer Login-Methode
  }

  def "Login mit User FALSCHER_USER und Passwort FALSCHES_PASSWORT schlägt fehl. [FF-PROJ-0001#02]"() {
        expect:
        false //Absichtlich fehlschlagender Test
  }

}
```

Listing 3: Referenzieren von Akzeptanzkriterien in Spock-Tests [4]

ein Test darauf referenziert und alle diese Tests erfolgreich ausgeführt wurden.
- Rendern einer AsciiDoc-Datei mithilfe eines Templates

Im Build-Prozess führt Maven vor dem Skript den Testrunner aus und nach dem Skript das AsciiDoc-Plugin, um aus der generierten AsciiDoc-Datei einen HTML-Report zu erzeugen. Dieser Report kann beispielsweise über ein weiteres Plugin nach Confluence exportiert werden. Alternativ ist auch die Umwandlung in ein PDF möglich, das als Artefakt im Artifactory abgelegt werden kann.

Das Ergebnis, der Report im HTML-Format, ist in *Abbildung 1* zu sehen.

Dokumentation: agile-documentation

1.0-SNAPSHOT, 2025-07-07

Diese Dokumentation dient ausschließlich Demonstrationszwecken.

Sie zeigt exemplarisch, wie Fachfunktionen beschrieben und mit Akzeptanzkriterien sowie automatisierten Tests verknüpft werden können. Die Inhalte basieren auf dem Beispielprojekt unter https://github.com/ziffit/agile-documentation].

In einem echten Projekt wären Fachfunktionen, Testabdeckung und Struktur entsprechend umfangreicher.

Inhalt

1. Fachfunktionen (FF)
1.1. FF-PROJ-0001 Benutzer Login
1.2. FF-PROJ-0002 Benutzerregistrierung
1.3. FF-PROJ-0003 Passwort zurücksetzen
2. Glossar

1. Fachfunktionen (FF)

Anforderungsabdeckung: 75,00% der Akzeptanzkriterien sind durch Tests abgedeckt.

Tag	Fachfunktion
Benutzername	FF-PROJ-0001 Benutzer Login FF-PROJ-0002 Benutzerregistrierung
Login	FF-PROJ-0001 Benutzer Login
Passwort	FF-PROJ-0001 Benutzer Login FF-PROJ-0002 Benutzerregistrierung FF-PROJ-0003 Passwort zurücksetzen

1.1. FF-PROJ-0001 Benutzer Login

Kurzbeschreibung: Authentifizierung von Benutzer:innen anhand von Zugangsdaten.

Der Login ist der Einstiegspunkt für alle geschützten Funktionen im System. Benutzer:innen geben ihre Zugangsdaten ein und erhalten bei erfolgreicher Anmeldung ein Authentifizierungs-Token.

Beispiel: Authentifizierungs-Token

```
{
  "sub": "user123",
  "iat": 1616161616,
  "exp": 1616165216,
  "sessionId": "abc123xyz",
  "roles": ["user", "admin"]
}
```

1.1.1. Akzeptanzkriterien

Coverage: 50,00 % (mindestens 1 Test pro Kriterium, alle Testausführungen erfolgreich)

FF-PROJ-0001#01: *Ein gültiger Benutzername und ein korrektes Passwort führen zur erfolgreichen Anmeldung.*

✓ Login mit User Peter0815 und Passwort Geheim%*123 ist erfolgreich. [FF-PROJ-0001#01]

FF-PROJ-0001#02: *Ein ungültiger Benutzername oder ein falsches Passwort führen zu einer Fehlermeldung.*

✓ Login mit User Hans4711 und Passwort FALSCHES_PASSWORT schlägt fehl. [FF-PROJ-0001#02]
✗ Login mit User FALSCHER_USER und Passwort FALSCHES_PASSWORT schlägt fehl. [FF-PROJ-0001#02]

FF-PROJ-0001#03: *Benutzer:innen ohne Aktivierung können sich nicht anmelden.*

keine Tests gefunden

FF-PROJ-0001#04: *Die Anmeldung erzeugt ein gültiges JWT-Token mit Ablaufzeit.*

✓ Dummytest für mehrere Fachfunktionen und Kriterien. [FF-PROJ-0001#04,FF-PROJ-0003#01,FF-PROJ-0003#02]

1.1.2. Ablauf Login

Benutzer
Login-Seite
Auth-Service
Benutzerdatenbank
Gibt Benutzername und Passwort ein
Übergibt Zugangsdaten
Prüft Benutzername und Passwort
Ergebnis (gültig/ungültig)
Antwort mit Token oder Fehlermeldung
Benutzer
Login-Seite
Auth-Service
Benutzerdatenbank

Abbildung 1: Ausschnitt des erzeugten HTML-Reports (© Felix Tensing)

Fachfunktion (FF) kurz und knapp

Merksatz:
„Eine Fachfunktion beschreibt, was die Software leistet – nicht, wie sie bedient wird."

Eine Fachfunktion:
- Hat eine eindeutige Kennung, z. B. FF-PROJ-0001 (FF: Fachfunktion, PROJ: Projekt- oder Artefakt-Kürzel, z. B. aus Jira, 0001: laufende Nummer)
- Behält ihre Kernfunktion über den gesamten Lebenszyklus weitgehend unverändert bei
- Ändert sich im Laufe der Weiterentwicklung
- Verfügt über testbare Akzeptanzkriterien
- Beschreibt nicht die Bedienung
- Enthält keine subjektiven Bewertungen oder Beschreibungen
- Erläutert fachliche Hintergründe
- Verweist auf zugehörige Fachfunktionen

Die Bedeutung der Detailtiefe bei der Formulierung von Fachfunktionen und Akzeptanzkriterien lässt sich folgendermaßen verdeutlichen: Gibt man einer Entwicklerin/einem Entwickler die Fachfunktionen und ausreichend Zeit, entsteht eine Software, die sich anders bedient, anders programmiert ist und in Details unterschiedlich funktioniert – aber alle geforderten Geschäftsprozesse abbildet.

Welche Tests eignen sich für die Anforderungsabdeckung?

Die Verknüpfung eines Tests mit einem Akzeptanzkriterium ist einfach, stellt aber zunächst nur eine semantische Verbindung dar. Doch welche Tests eignen sich zur Auswertung der Anforderungsabdeckung?

Prinzipiell kann jeder Test verwendet werden, der im Rahmen eines Testrunners ausgeführt wird und dessen Name sowie Ergebnis (Success/Failed/Skipped) in einem Report erscheint. Inhaltlich gibt es jedoch Unterschiede. Gehen wir die (vereinfachte) Testpyramide von unten nach oben durch.

Abbildung 2 zeigt eine Übersicht der Teststufen und deren Verwendung zur Anforderungsabdeckung.

Unit-Tests laufen schnell und eignen sich für viele Akzeptanzkriterien gut. Hier ist jedoch Vorsicht geboten: False-Positives (also Tests, die ein erfolgreiches Verhalten vortäuschen, obwohl es technisch nicht zutrifft) sind möglich. Angenommen, ein Akzeptanzkriterium besagt: „Der Login eines Benutzers

muss der Regex /^[a-zA-Z0-9]{5,20}$/ entsprechen." Die Prüfung des eingegebenen Usernamens erfolgt in einer Methode, die durch einen Unit-Test abgedeckt wird. Wird diese Methode allerdings im Code nicht verwendet, besteht die Gefahr, dass der Test grün ist und somit fälschlicherweise eine erfolgreiche Abdeckung signalisiert. In solchen Fällen helfen Code-Reviews oder Tests auf höherer Ebene, zum Beispiel Komponententests.

Komponententests prüfen nicht nur einzelne Klassen, sondern ganze Komponenten – oft mit gemockten Umsystemen. Über API-Calls oder ähnliche Schnittstellen wird das Akzeptanzkriterium getestet. So könnte geprüft werden, ob es möglich ist, einen Benutzer mit dem Namen „Hugo?#012" anzulegen. Das Mocken der Umsysteme ist wichtig, damit die Tests in der CI/CD-Pipeline reproduzierbar sind, beliebig oft parallel ausgeführt werden können und unabhängig von externen Systemen funktionieren. Ziel ist es, die Dokumentation selbst zu testen, nicht die Kompatibilität mit Umsystemen. Tests, die Umsysteme einbeziehen, sind zwar sinnvoll für das Gesamtprojekt, dürfen aber nicht Teil der Anforderungsabdeckung sein.

Aus diesem Grund sind **Systemtests** oder **Systemintegrationstests** zur Verifikation von Akzeptanzkriterien ungeeignet. Lässt sich ein Akzeptanzkriterium nur mit Systemintegrationstests prüfen, ist es sehr wahrscheinlich falsch formuliert, da es externe Abhängigkeiten hat.

Ob ein Test das Akzeptanzkriterium inhaltlich gut testet, lässt sich nicht automatisch bestimmen – hier sind Code-Reviews und reflektiertes Mitdenken unabdingbar. Ein „assert true" würde auch zu einem erfolgreichen Test und damit zu einer erfolgreichen Anforderungsabdeckung führen.

In komplexeren Projekten empfiehlt es sich, im Testkonzept klar festzulegen, welche Tests wann und wie zur Verifikation herangezogen werden.

Weitere Empfehlungen aus der Praxis

Um den vollen Mehrwert der Dokumentation zu nutzen, muss sie jederzeit und für alle Projektbeteiligten transparent und einfach zugänglich sein. Nur so können Entwickler:innen, Tester:innen, Product Owner und weitere Beteiligte effektiv zusammenarbeiten und auf derselben Wissensbasis agieren. Eine zentrale, aktuelle Dokumentation unterstützt schnelle Abstimmungen, reduziert Missverständnisse und verhindert Doppelarbeit. Deshalb empfiehlt es sich, die Publizierung der Dokumentation – sei es als HTML- oder PDF-Version – fest in die Build- oder Release-Pipeline zu integrieren und so einen automatisierten, stets aktuellen Zugriff sicherzustellen.

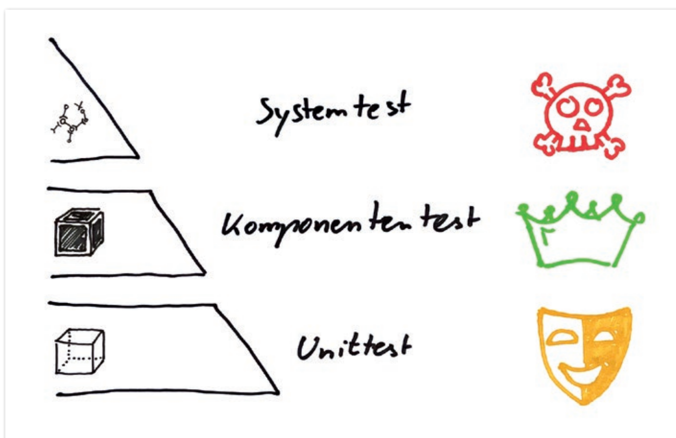


Abbildung 2: Einordnung von Testarten zur Anforderungsabdeckung (© Felix Tensing)

In der Praxis zeigt sich schnell, dass Fachfunktionen zu einer gemeinsamen Referenz im Team werden. Statt lange Erklärungen zu liefern, heißt es dann einfach: „Fachfunktion 15". Durch die eindeutige Kennung ist für alle klar, welche Funktionalität gemeint ist. Auch User Stories nehmen direkt Bezug auf Fachfunktionen und deren Akzeptanzkriterien – etwa mit Formulierungen wie: „FF-BSPP-0015#05 entfällt, dafür kommen die beiden Kriterien … hinzu." Dieser gemeinsame Bezugspunkt hilft, Missverständnisse zu vermeiden und macht die Kommunikation im Team effizienter.

Zwar liegt die Verantwortung für die finale Formulierung von Fachfunktionen und Akzeptanzkriterien bei den Entwickler:innen, doch sinnvoll strukturierte Vorgaben können diesen Prozess erheblich erleichtern. Besonders hilfreich sind Vorlagen, die sich am gewünschten Endergebnis orientieren oder sogar direkt aus der Story übernommen werden können.

Die letztgültige Beschreibung muss jedoch von denjenigen kommen, die die Umsetzung kennen – also von den Entwickler:innen selbst. Nur sie wissen genau, wie eine Anforderung tatsächlich implementiert wurde. Deshalb ist es entscheidend, dass sie die Beschreibung so präzise und fachlich verständlich wie möglich formulieren – als verbindliche Referenz für Tests, Dokumentation und spätere Weiterentwicklung.

Stellt sich in der täglichen Arbeit die Frage: „Ist das Verhalten meiner Anwendung ein Feature oder ein Bug?", sollte man als Erstes einen Blick in die Dokumentation werfen. Findet sich dort keine klare Antwort, wurde eine Dokumentationslücke entdeckt, die idealerweise zeitnah geschlossen werden sollte. Auch bei Bugs gilt: Zuerst die Dokumentation prüfen, dann die Tests und zuletzt die Implementierung untersuchen.

Eine häufige Frage ist der passende Dokumentations-Schnitt bei größeren Projekten – etwa bei Microservice-Architekturen oder typischen Client-Server-Systemen mit separatem Frontend und Backend. Hier sollte jedes Artefakt eigenständig dokumentiert werden. Erstens, weil die Testbarkeit von Fachfunktionen über Systemgrenzen hinweg sehr komplex wäre. Und zweitens, weil das Ziel ist, jeweils genau ein System zu beschreiben. Möchte man beispielsweise das Frontend durch eine App ersetzen, ist es essenziell, genau zu wissen, wie das Backend funktioniert. Eine Vermischung von Backend- und Frontend-Dokumentation widerspricht dieser Systematik.

Natürlich ist es möglich, zusätzlich eine übergreifende Dokumentation zu erstellen, die Fachfunktionen aus mehreren Systemen verlinkt und deren Zusammenhänge erläutert. Diese Art der Dokumentation ist klassisch und enthält in der Regel keine Akzeptanzkriterien oder Verknüpfungen zu Tests.

Typische Fallstricke

Auch wenn die Verknüpfung von Tests mit Fachfunktionen die Dokumentation robuster macht, gibt es typische Fehlerquellen, die man kennen sollte.

Ein inhaltlich unpassender oder ungenauer Test kann eine trügerische Sicherheit vermitteln. Wird etwa ein Test fälschlich mit dem Akzeptanzkriterium #04 statt #05 verknüpft – oder prüft nicht exakt das beschriebene Verhalten – erscheint das Kriterium als abgedeckt, obwohl es inhaltlich nicht korrekt getestet wurde. Noch schwerwiegender sind fehlende Akzeptanzkriterien: Sie tauchen in der Auswertung nicht auf und ihre Abwesenheit bleibt daher unbemerkt.

Ein guter Hinweis für die Praxis: Entwickler:innen sollten sich beim Schreiben von Code – insbesondere bei *if*- oder *switch*-Anweisungen – bewusst fragen, ob die jeweilige Logik durch ein konkretes Akzeptanzkriterium beschrieben ist. So lassen sich fehlende oder lückenhafte Kriterien frühzeitig erkennen.

Auch unpräzise formulierte Akzeptanzkriterien führen zu Problemen – insbesondere, wenn sie unbemerkt mehrere Anforderungen enthalten. Mehr dazu im Kasten: *Akzeptanzkriterien richtig formulieren*.

Akzeptanzkriterien richtig formulieren

Gute Akzeptanzkriterien sind präzise, atomar und testbar. Die folgenden Regeln helfen bei der Formulierung:

- **Atomar statt verschachtelt**
  Vermeide „und", „oder" sowie Aufzählungen mit Kommas. Sie deuten oft auf mehrere Aussagen hin – besser ist die Aufteilung in eigene Kriterien.
- **Keine vagen Begriffe**
  Formulierungen wie „intuitiv", „schnell" oder „komfortabel" sind subjektiv und nicht testbar. Stattdessen messbare Bedingungen verwenden (z. B. „Antwortzeit < 300ms").
- **Positiv und eindeutig formulieren**
  Negative Aussagen („darf nicht …") allein reichen nicht – was stattdessen passieren soll, muss klar benannt werden.
- **Fachlich statt technisch**
  Beschreibe, was aus Sicht der Nutzer:innen passiert, nicht wie es technisch umgesetzt ist.
- **Bedingungen trennen**
  Wenn ein Verhalten von Bedingungen abhängt („Wenn X, dann Y und Z"), sollten daraus getrennte Kriterien entstehen.

Ein häufiger Fehler beim Umgang mit Identifikatoren: Weder Fachfunktions- noch Kriteriumskennungen sollten erneut vergeben werden, wenn eine Funktion zuvor entfallen ist. Häufig werden diese Kennungen in E-Mails, Tickets oder Issues referenziert. Wird eine ehemals vergebene ID später neu vergeben, führt das zu Unklarheiten über den tatsächlichen Inhalt. Umgekehrt gilt: Ist eine ID in der aktuellen Dokumentation nicht mehr enthalten, wurde sie entfernt – ein völlig normaler Vorgang, der sich mit älteren Dokumentationsversionen nachprüfen lässt.

Schließlich ist auch von einer inhaltlichen Kodierung in der ID (zum Beispiel zur fachlichen Gruppierung) abzuraten. Fachfunktionen ändern sich mit der Zeit – ihre anfängliche Klassifikation kann dabei schnell veralten. Stattdessen empfiehlt sich eine einfache fortlaufende Nummerierung. Für eine inhaltliche Strukturierung bieten sich Tags an, wie sie auch in *Listing 2* und *Abbildung 1* gezeigt werden.

Die Integration von Dokumentation, Tests und Code im beschriebenen Ansatz bringt zahlreiche Vorteile für Teams und Projekte:

- **Effiziente Code-Reviews:** Dokumentation, Tests und Code bilden eine Einheit (siehe Abbildung 2) und werden gemeinsam geprüft, was die Qualitätssicherung deutlich verbessert.
- **Automatisierte Anforderungsabdeckung:** Das Team erhält eine verlässliche Selbstkontrolle und kann Schwachstellen frühzeitig erkennen. Der Testnachweis unterstützt insbesondere in regulierten Umgebungen häufigere und schnellere Releases.
- **Frühe Fehlererkennung:** Bereits das Lesen der entstandenen Dokumentation nach der Umsetzung zeigt mögliche Missverständnisse oder Fehlumsetzungen auf. Das Lesen der Dokumentation ist somit ein wichtiger erster Schritt im Review-Prozess.
- **Einheitliche Team-Sprache:** Fachfunktionen und Akzeptanzkriterien schaffen klare Vorgaben und reduzieren Missverständnisse. Dabei ist darauf zu achten, dass ein gemeinsamer Wortschatz verwendet wird.
- **Verbesserte Nachvollziehbarkeit:** Fachfunktions-IDs als Stichworte in Tickets helfen, alle zugehörigen Stories und Bugs schnell zu finden – eine Rückverfolgbarkeit, die auch in regulierten Umgebungen gefordert wird.
- **Best Practices von Documentation as Code:** Versionierbarkeit, textbasierte Änderungsverfolgung (Diffs) und die Einbindung von Code-Snippets unterstützen das Schreiben verständlicher Dokumentation mit geringem Aufwand.
- **Flexibilität:** Das Vorgehen lässt sich um eigene Felder in Fachfunktionen erweitern. Zudem ist eine Integration in Frameworks wie Storybook [5] oder Docusaurus [6] möglich.

Fazit

Die konsequente und strukturierte Dokumentation fachlicher Anforderungen als Fachfunktionen bildet die Grundlage für eine effektive Testabdeckung und eine nachhaltige Qualitätssicherung im Sinne des Shift-Left-Prinzips. Sie hilft Entwickler:innen, die Frage zu beantworten: Wie viel muss ich testen? Durch die enge Verzahnung *(siehe Abbildung 3)* von Dokumentation, Tests und Code entsteht ein transparentes, nachvollziehbares und stets aktuelles Abbild der Softwarefunktionalität.

Der Ansatz fördert die Kommunikation im Team, reduziert Missverständnisse und ermöglicht eine verlässliche Anforderungsabdeckung – selbst in komplexen Projekten und regulierten Umgebungen. Die Integration in bestehende Entwicklungsprozesse mit modernen Werkzeugen sorgt dafür, dass Dokumentation nicht als lästige Pflicht, sondern als wertvoller Teil der täglichen Arbeit erlebt wird. Den Mehrwert einer guten Dokumentation erkennen Teams meist bereits nach kurzer Zeit.

Das vorgestellte Beispielprojekt bietet einen guten Ausgangspunkt, um das Vorgehen in das eigene Projekt zu übernehmen und an die eigenen Bedürfnisse anzupassen. Es lädt zudem dazu ein, Erfahrungen und Erweiterungen mit der Community zu teilen – so profitiert nicht nur das eigene Team, sondern auch die gesamte Entwicklergemeinschaft.

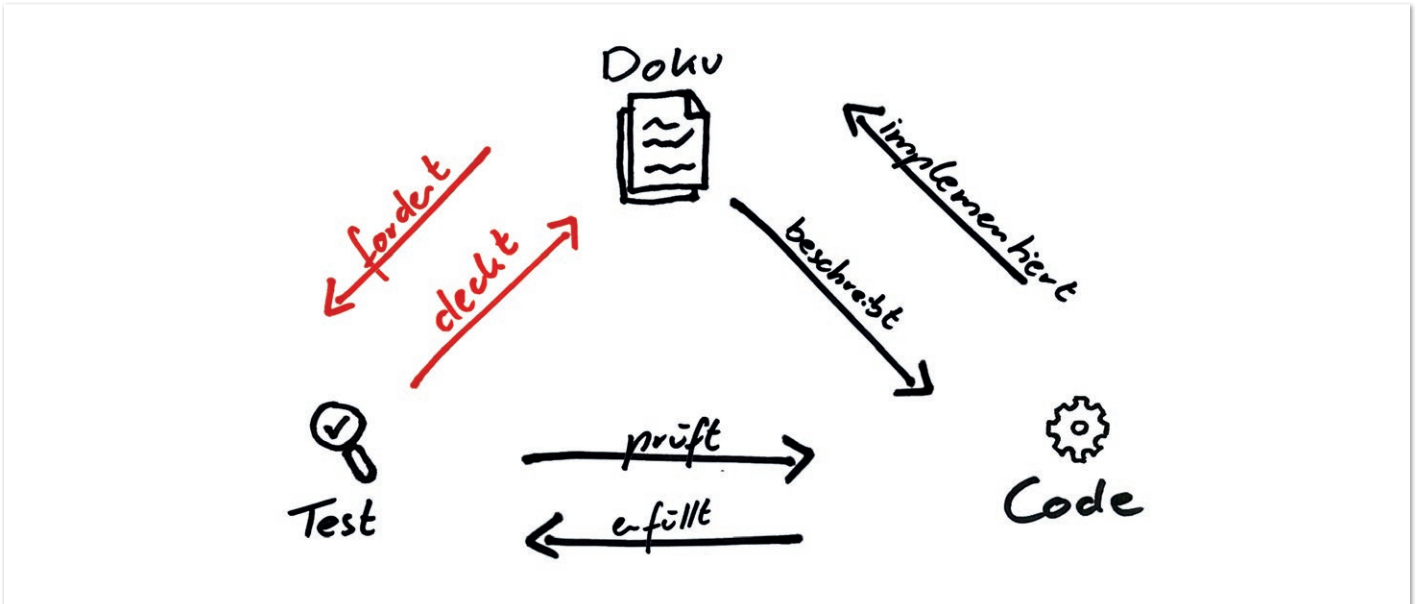


Abbildung 3: Zusammenhänge zwischen Code, Test und Dokumentation (© Felix Tensing)

Insgesamt zeigt sich: Wer frühzeitig Anforderungen klar, präzise und testbar dokumentiert, legt den Grundstein für stabile Software, effiziente Reviews und schnellere, qualitativ hochwertige Releases.

Quellen

[1] Kaner, Falk, Nguyen (1999): Testing Computer Software, 2nd Edition. Wiley.
[2] AsciiDoc-Dokumentation: *https://asciidoc.org*
[3] PlantUML-Dokumentation: *https://plantuml.com/de*
[4] Spock – the enterprise ready specification framework *https://spockframework.org*
[5] Storybook *https://storybook.js.org*
[6] Docusaurus *https://docusaurus.io*
[7] Beispielprojekt auf GitHub: *https://github.com/ziffit/agile-documentation*

Felix Tensing
Nürnberger Versicherung
felix.tensing@nuernberger.de

Felix Tensing ist Java Fullstack-Entwickler mit einer Leidenschaft für Testen, Dokumentation sowie agiles und crossfunktionales Arbeiten. Seit über 20 Jahren ist er in der IT tätig, vor allem im regulierten Umfeld. Wenn er nicht gerade als Speaker auf einer Konferenz auftritt, verbringt er seine Freizeit mit seinen Kindern, beim Angeln oder beim Umsetzen kreativer Projekte am 3D-Drucker – ganz ohne Dokumentation, Tests und doppelten Boden.

Transitive Berechtigungen mit der SpiceDB

Jelmen Guhlke

10101001
01010110
01010011

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

In den ersten beiden Artikeln dieser Serie wurden die Autorisierungs-Mechanismen von Jakarta EE *[1]*, insbesondere Jakarta Security *[2]* beleuchtet. Dabei sind jedoch Herausforderungen aufgefallen, die nicht ohne Weiteres mit den Jakarta-Bordmitteln gelöst werden können. Gerade wenn die Anwendung an Größe zunimmt, Zugriffsregeln komplexer gestaltet werden oder Geschäftslogiken über mehrere Services verteilt sind, lassen sich Code-Smells wie Code-Duplizierungen oder schlechte Wart- und Erweiterbarkeit kaum vermeiden. Der im letzten Artikel (Java aktuell 03/25) vorgestellte „Open Policy Agent (OPA)“ *[3]* versucht, durch die Trennung von Entscheidungsfindung/Prüfung und der Durchführung einer Richtlinie die Entscheidungslogik zentral zu bündeln, um somit die Handhabung dieser zu vereinfachen. Durch Jakartas Interceptors aus der Context-and-Dependencies-Spezifikation *[4]* lässt sich die Anbindung des Agents einfach in der Jakarta-EE-Anwendung umsetzen.

Die Herausforderungen von transitiven Beziehungen

Die Jakarta-Security-Spezifikation, aber auch die grundlegende Konzeptionierung des Open Policy Agents orientieren sich an den in der IT-Welt weit verbreiteten „Role-Based Access Control (RBAC)“- und „Attribute-Based Access Control (ABAC)“- Modellen. Dabei wird eine statische Beziehung zwischen einem Subjekt (zum Beispiel ein User oder ein System), einer Rolle, beziehungsweise Attribut, und einem Objekt aufgebaut. So kann zum Beispiel ein User mit der Rolle „Betriebsleitung“ auf die Stammdaten eines Betriebes zugreifen und diese manipulieren. Spannend wird es jedoch, wenn die Mitarbeitenden des Betriebs verwaltet werden sollen. In den seltensten Fällen wird für jeden Mitarbeiter und jede Mitarbeiterin – in diesem Fall das Objekt – eine statische Rollen-Beziehung mit einem vorgesetzten User modelliert. Dies würde sich schnell in kleinteiligem Micro-Management verlieren. Vielmehr kommt aus der fachlichen Domäne die abgeleitete Anforderung, dass ein User mit der Rolle „Betriebsleitung“ auf alle Mitarbeitenden des Betriebes zugreifen darf. Hier zeichnet sich schon eine erste transitive Beziehung ab *(siehe Abbildung 1)*.

Weil die User B, C und D dem Betrieb zugeordnet sind, darf der User A, der ebenfalls eine Beziehung über die Rolle zu dem Betrieb hat, auf diese User zugreifen. In der analogen Welt sind abgeleitete Berechtigungen omnipräsent. Da eine Person in einem bestimmten Bezirk lebt, darf diese an den Wahlen dort teilnehmen. Oder durch die Mitgliedschaft in einem Fitnessstudio hat eine andere Person Zugriff auf das Online-Angebot des Studios. Mit dem Paper *Access Control Requirements for Web 2.0 Security and Privacy* von Dr. Carrie Gates aus dem Jahr 2007 *[5]* wurde der Begriff *Relationship-Based Access Control (ReBAC)* als notwendige Weiterentwicklung aus den bestehenden Modellen wie RBAC beschrieben. Es wird als Wandel von „welche Rolle hat der Empfänger oder die Empfängerin der Daten“ zu „in welcher Beziehung stehen Empfänger oder Empfängerin zu den Daten“ beschrieben.

Mit Blick auf den Status Quo in der Jakarta-Security-Spezifikation ergeben sich aus den Anforderungen von ReBAC zwei zentrale Herausforderungen: Zum einen sind traditionelle relationale Datenbanksysteme nicht gut darin, Abhängigkeits-Graphen zu traversieren. Insbesondere wenn die Anzahl an Ebenen wächst, hat dies einen erheblichen negativen Einfluss auf die Performance. Zum anderen kann der häufige Zwang, SQL als Abfragesprache zu nutzen, zu kaum mehr lesbaren und komplexem Code führen. Bei der

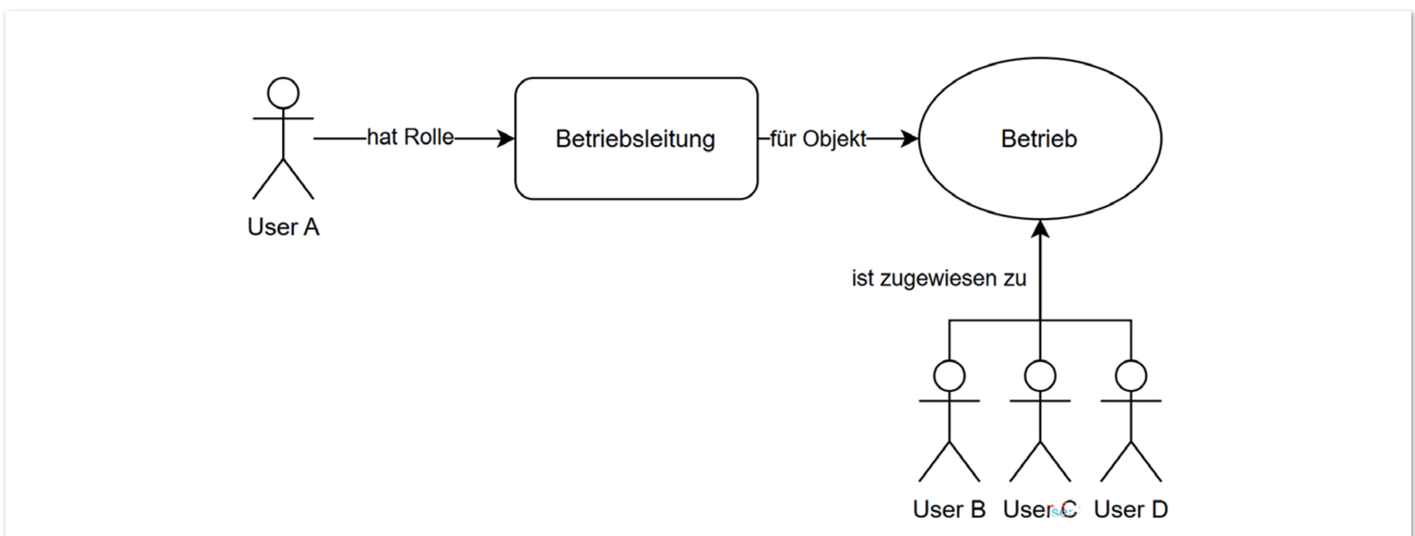


Abbildung 1: Eine einfache transitive Berechtigung (© Jelmen Guhlke)

```
services:
  spicedb:
     image: authzed/spicedb:latest
     volumes:
        - ./certs:/certs
     command: >
        serve
        --log-level=debug
        --grpc-tls-cert-path=/certs/spicedb-cert.pem
        --grpc-tls-key-path=/certs/spicedb-key.pem
        --grpc-preshared-key=mysecretkey
   --http-enabled
        --http-tls-cert-path=/certs/spicedb-cert.pem
        --http-tls-key-path=/certs/spicedb-key.pem
        --datastore-engine=memory
     ports:
        - "127.0.0.1:50051:50051" # gRPC - the default gateway
        - "127.0.0.1:8443:8443"       # HTTP - if activated
```

Listing 1: Eine docker-compose.yml-Konfiguration, um die SpiceDB lokal zu starten

Nutzung eines Policy Agents, wie dem OPA, fällt einem hingegen der eigentliche Vorteil der Zustandslosigkeit bei der Betrachtung der Anforderungen von ReBAC auf die Füße. Da der Agent selbst nicht dafür ausgelegt ist, Tausende bis Millionen von Datensätzen zu speichern, müssen die für die Prüfung relevanten Daten stets zunächst aus einem anderen persistenten Speicher geladen werden, um sie dem Agenten anschließend zur Berechnung zur Verfügung zu stellen.

Googles Zanzibar und AuthZeds SpiceDB

Dieses Problem hatte die Entwicklungsabteilung von Google vor einigen Jahren auch, weswegen nach einer zentralen und einheitlichen Lösung gesucht wurde. Mit dem Projekt Zanzibar *[6]* wurde diese gefunden. Seit über fünf Jahren führt Zanzibar unzählige Berechtigungsprüfungen von zahlreichen Google Services weltweit durch. Zanzibar selbst ist Closed Source und für die allermeisten Projekte auch vollkommen überdimensioniert.

```
/**
 * a simple object
 */
definition simpleobjecttype {}
```

Listing 2: Mindestens eine Objektdefinition sollte in einer Schema-Beschreibung vorhanden sein.

```
/**
 * user represents a system user
 */
definition user {}

/**
 * department represents a department in the system
 */
definition department {
  /**
        * manager relates a user
        * which is a manager in the department
   */
  relation manager: user
}
```

Listing 3: Eine Beziehung wird mithilfe des Schlüsselwortes `relation` angegeben.

2019 wurde jedoch ein Paper *[7]* veröffentlicht, das die theoretischen Grundlagen und Konzepte von Zanzibar der Öffentlichkeit zugänglich machte. Auf diesem aufbauend sind einige Open-Source-Projekte entstanden. Eines davon ist die SpiceDB *[8]* von AuthZed *[9]*. SpiceDB ist ein in Go geschriebenes Datenbanksystem, das explizit für ReBAC von großen Enterprise-Anwendungen ausgelegt ist. Dabei stellt es sowohl Schnittstellen mittels gRPC *[10]* als auch HTTP/JSON für die Verwaltung und Abfragen bereit. Wobei letztere beim Start von SpiceDB aktiviert werden muss. Zentrale architekturelle Konzepte sind an das Zanzibar-Paper angelehnt, zu denen zum Beispiel eine verteilte, parallelisierte Graph-Engine und das Consistency Model gehören. Als Implementierung der Persistenzschicht können je nach Anforderungen zwischen In-Memory, Googles Spanner System *[11]*, CockroachDB *[12]* oder PostgreSQL *[13]* und MySQL *[14]* gewählt werden. Um mit SpiceDB zu starten, muss das Datenbanksystem selbst deployt werden. Dies kann entweder direkt als Binary geschehen oder in einem Container wie mit Docker *(siehe Listing 1) [15]*.

Nützliche Informationen lassen sich dazu in den übersichtlichen Dokumentationen der SpiceDB finden *[16]*. Neben dem eigentlichen Datenbanksystem kann das featurereiche Command-Line Tool „zed" installiert werden, um mit der SpiceDB zu interagieren. Da als Schnittstellen aber die Standards HTTP/JSON und gRPC zum Einsatz kommen, kann auch mithilfe eines Clients der eigenen Wahl mit der SpiceDB kommuniziert werden.

Das Datenmodell der SpiceDB

Wie auch von relationalen Datenbanken bekannt, wird mittels eines Schemas *[17]* die Grundstruktur der Daten definiert. Dies geschieht in Form einer eigenen Schema-Sprache, die nach etwas Übung sehr einfach zu lesen ist. Eine Schema-Definition besteht aus mindestens einer Objekt-Beschreibung. Ein Objekt ist am ehesten mit einer traditionellen Datenbank-Tabelle vergleichbar. Ein simples Objekt kann einfach mittels einer Zeile beschrieben werden *(siehe Listing 2)*.

Um nun die schon mehrmals angesprochene Stärke der Beziehungen zu definieren, reicht die Angabe mittels des Keywords `relation`. So kann die Beziehung zwischen einem `user` und einem `department` wie in *Listing 3* beschrieben werden.

Jede Beziehung hat einen in dem jeweiligen Kontext eindeutigen

Namen. In dem Fall von *Listing 3* ist dies `manager`. Es wird empfohlen, Beziehungen immer als Nomen anzugeben: `reader`, `owner`, `manager` und so weiter. Definierte Relationen können wiederum von anderen Relationen als Referenz genutzt werden. So können vielschichtige Vererbungen definiert werden.

In *Listing 4* kann die Beziehung `employee` eines `departments` entweder direkt auf einen `user` zeigen oder auf die Menge aller `employees` eines bestimmten `departments`.

```
definition user {}

definition department {
   /**
     * employee can include both users
     * and the set of employees of other specific department.
     */
     relation employee: user | department#employee
}
```

Listing 4: Eine Beziehung kann auch als Typen auf eine Beziehung referenzieren.

Neben dem `relation`-Keyword gibt es noch `permission`. Eine `permission` definiert einen berechneten Satz von Subjekten, welche auf bestimmte Weise Zugriff auf ein Objekt haben. Ein `manager` darf zum Beispiel die Liste aller `employees` einsehen. Eine solche Erlaubnis benötigt für die Definition einen Namen und einen Ausdruck, der die Berechnung der Menge von Subjekten beschreibt *(siehe Listing 5)*.

```
definition user {}

definition department {
      relation manager: user
      relation employee: user | department#employee

      /**
        * view_employees determines whether a user
        * can view the employee list of the department
        */
   permission view_employees = manager
}
```

Listing 5: Eine Berechtigung wird durch das `permission`-Keyword angelegt.

Dabei können die Operatoren + (Vereinigung), & (Schnittpunkt), - (Ausschluss) und -> (abhängige Hierarchien) verwendet werden.

Interagieren mittels zed

Um nun die laufende SpiceDB mit den Daten zu versorgen, kann das schon erwähnte Command-Line Tool zed genutzt werden. So kann ein sogenannter `Context` angelegt werden *(siehe Listing 6)*, um nicht bei jedem Kommando immer wieder globale Parameter wie Host, Port oder Secret angeben zu müssen. Gerade für das lokale Testen und Ausprobieren kann dies sehr nützlich sein.

```
# create a context. Use --insecure if no TLS should be used
zed context set playground localhost:50051 mysecretkey --insecure

# use the created context
zed context use playground
```

Listing 6: Anlegen eines zed-Kontextes.

Eine Schema-Datei (mit der Endung `.zed`) kann mithilfe des `schema`-Kommandos in die Datenbank geschrieben werden *(siehe Listing 7)*.

```
# write schema
zed schema write schema.zed

# read current schema
zed schema read
```

Listing 7: Einlesen einer vorher erstellten Schema-Definition.

Beziehungen können dann mit dem dazugehörigen `relationship`-Kommando in dem System angelegt werden. Dabei erwartet *zed* als Parameter erst das Objekt, dann die Beziehung und dann das Subjekt der Beziehung. Referenziert werden bestimmte Objekte mit der Syntax `object_type:object_id`. *Listing 8* legt die Beziehung `user1` ist `manager` von `dep1` an. Nach der Anlage kann mit dem `permission check` validiert werden, ob das Subjekt `user:user1` die Berechtigung `view_employees` auf das Objekt `department:dep1` hat. Als Antwort wird ein kurzes `true` oder `false` zurückgegeben.

```
# create a new relationship
zed relationship create department:dep1 manager user:user1

# check permission on newly created relationship
zed permission check department:dep1 view_employees user:user1

true
```

Listing 8: Anlegen und Prüfen einer neuen Beziehung mittels zed.

Da die Zusammensetzungen von Berechtigungen und Beziehungen in realen Anwendungsfällen durchaus komplex ausfallen können, gibt es die sehr hilfreiche Flag `--explain` für den `permission check` Befehl. Mit der Angabe der Flag wird der Graph im Terminal dargestellt, der erklärt, wieso eine Entscheidung so ausgefallen ist, wie sie ist *(siehe Listing 9)*.

Anbindung an Jakarta EE

Das Kommandozeilen-Tool ist sehr ausgereift und erleichtert das schnelle Testen von Schema, Beziehungen und Berechtigungen erheblich. Doch für den produktiven Betrieb von Enterprise Anwendungen bietet sich die Nutzung selbstredend nicht an. AuthZed bietet aus dem Grund für verschiedene Programmiersprachen offi-

zielle SDKs an [18]. Daneben gibt es auch einige von der Community entwickelte Libraries. Per Default baut SpiceDB auf dem gRPC-Protokoll auf. Dies ermöglicht unter anderem eine wesentlich effizientere Kommunikation als über HTTP/JSON. Aus diesem Grund setzt auch das Java SDK auf dieser Technologie auf. Für Personen, die nie Kontakt zu gRPC hatten, mag der Aufbau im ersten Moment etwas sperrig erscheinen. An dieser Stelle sei auf die gut aufbereitete Dokumentation auf der Webseite von gRPC [10] verwiesen.

Ähnlich wie bei der Anbindung des Open Policy Agents aus dem vorherigen Artikel, bietet es sich an, die Integration der SpiceDB-Anbindung mithilfe von einem oder mehreren CDI(Context and Dependency Injection)-Interceptoren [4] umzusetzen. Dadurch lässt sich eine lose Kopplung zwischen Applikations-Code und der Autorisierungsimplementierung erreichen. Die eigentliche Verbindung zu der SpiceDB wird über einen `ManagedChannel` aus dem `io.grpc`-Paket hergestellt. Unter Nutzung dieses Channels kann dann ein `PermissionsServiceGrpc.PermissionsServiceBlockingStub` angelegt werden. Diese gRPC-Stub-Klasse aus dem `com.authzed.api.v1`-Paket stellt dann die eigentlichen APIs für die Interaktion mit der Datenbank zur Verfügung. *Listing 10* zeigt einen Verbindungsaufbau exemplarisch.

```
# check permission with explanation
zed permission check --explain department:dep1 view_employees user:user1
true
✓ department:dep1 manager (32.735µs)
└── user:user1
```

Listing 9: Durch die `-explain`-Flag „erklärt" die SpiceDB, wieso die Entscheidung getroffen wurde.

```
private final PermissionsServiceGrpc.PermissionsServiceBlockingStub permissionsService;

public SpiceDBClient() {
  ManagedChannel channel = ManagedChannelBuilder
     .forAddress(HOST, PORT)
  // .usePlaintext() is only suitable for local dev setups.
  // Use TLS in prod environments
     .usePlaintext()
     .build();

  permissionsService =
        PermissionsServiceGrpc.newBlockingStub(channel)
              .withCallCredentials(new BearerToken(SECRET));
}
```

Listing 10: Die Verbindung zu einer SpiceDB wird über das gRPC-Protokoll hergestellt.

```
public void writeRelationship(RelationshipDto relationshipDto) {
  Objects.requireNonNull(relationshipDto, "relationshipDto cannot be null");

  Relationship relation =
        Relationship.newBuilder()
              .setResource(
                    ObjectReference.newBuilder()
                          .setObjectType(relationshipDto.resourceType())
                          .setObjectId(relationshipDto.resourceId())
                          .build())
              .setRelation(relationshipDto.relationIdentifier())
              .setSubject(
                    SubjectReference.newBuilder()
                          .setObject(
                                ObjectReference.newBuilder()
                                      .setObjectType(relationshipDto.subjectType())
                                      .setObjectId(relationshipDto.subjectId())
                                      .build())
                          .build())
              .build();

  WriteRelationshipsRequest request =
        WriteRelationshipsRequest.newBuilder()
              .addUpdates(
                    RelationshipUpdate.newBuilder()
                          .setOperation(RelationshipUpdate.Operation.OPERATION_TOUCH)
                          .setRelationship(relation)
                          .build())
              .build();

  this.permissionsService.writeRelationships(request);
}
```

Listing 11: Das Schreiben einer Beziehung erfolgt über den vorher erstellten `PermissionsServiceGrpc`-Stub.

Um nun eine Beziehung in das bestehende Schema aus *Listing 5* zu schreiben, muss zuerst ein `Relationship`-Objekt erstellt werden, welches in ein `WriteRelationShipRequest` aufgeht *(siehe Listing 11)*. Der Aufbau für die Beschreibung einer Relationship ist dabei sehr ähnlich zu der von der Nutzung von zed bekannten Form. Bei Bedarf können mehrere `RelationshipUpdate` in einem Schreibprozess ausgeführt werden. Mit der Angabe der `RelationshipUpdate.Operation` kann definiert werden, ob die Beziehung angelegt (`OPERATION_CREATE`), gelöscht (`OPERATION_DELETE`) oder erst gelöscht und dann angelegt (`OPERATION_TOUCH`) werden soll. `OPERATION_CREATE` wirft einen Fehler, sofern die Beziehung schon im System existiert. Dafür ist ein einfaches Schreiben performanter, als die Beziehung erst zu löschen und dann neu zu schreiben.

Soll nun eine Abfrage zu einer bestimmten Berechtigung durchgeführt werden, geschieht dies nach dem gleichen Muster. Wie *Listing 12* aufzeigt, wird zuerst ein `CheckPermissionRequest`-Objekt erstellt, das dann ebenfalls über das `PermissionsServiceBlockingStub permissionsService` Stub-Objekt an die Datenbank gesendet wird. Erwähnenswert ist dabei die Angabe der `Consistency`. SpiceDB hat ein fein einstellbares und sehr umfangreiches Konsistenz-Modell. Dies ist nicht verwunderlich, da es im Zuge eines Autorisierungs-Systems mit der Fähigkeit ausgestattet ist, tausende gleichzeitige Anfragen zu bearbeiten. Die `Consistency` kann bei Bedarf bis auf die Ebene von Einzelanfragen definiert werden. Weitere Informationen dazu sind in der Dokumentation *[19]* zu finden.

Da die direkte Nutzung des SDKs durch die Verwendung des gRPC-Unterbaus etwas klobig wirkt, bietet es sich hier sehr gut an, die Logik der Anbindung an die SpiceDB wegzukapseln. Mithilfe einer anwendungsspezifischen DSL (Domain-Specific Language) können dann angenehm Beziehungen verwaltet und Berechtigungen abgefragt werden.

Fazit

Die SpiceDB von AuthZed stellt, wenn es um Relation-Based Access Control Management geht, eine sehr mächtige Plattform zur Verfügung. Die Möglichkeiten zur Angabe von nahezu unbegrenzt vielen Ebenen von Beziehungen und der trotzdem extrem schnellen Bearbeitung von Abfragen macht diese besonders für große Enterprise Anwendungen interessant. Durch die Zentralisierung der Berechtigungslogiken und der klaren Definitionssprache kann viel Komplexität aus dem Applikations-Code entfernt und vereinfacht werden. Ebenfalls lässt sich SpiceDB horizontal sehr gut skalieren. Jedoch ist dabei nicht der Overhead zu unterschätzen, ein bestehendes RBAC in ein ReBAC Model zu refactoren. Ebenso muss mit einer gut gewählten Applikations-Architektur sichergestellt werden, dass SpiceDB und sonstige Persistenz-Schichten synchron laufen.

Jakarta EE bietet ein äußerst solides Fundament für die Umsetzung moderner Zugriffssteuerungen. Besonders hervorzuheben ist die Möglichkeit, Bordmittel wie RBAC unkompliziert zu nutzen und bei Bedarf externe Systeme flexibel anzubinden. Dadurch lassen sich Sicherheitskonzepte realisieren, die nicht nur stabil, sondern auch skalierbar sind – und somit mit den Anforderungen der Anwendung mitwachsen können.

Wie immer gilt jedoch: Die endgültige Wahl von Technologien und

```
public boolean checkPermission(PermissionDto permissionDto) {
     Objects.requireNonNull(permissionDto, "permissionDto cannot be null");

  CheckPermissionRequest request =
        CheckPermissionRequest.newBuilder()
            .setConsistency(Consistency.newBuilder()
                .setMinimizeLatency(true)
                .build())
                .setResource(
                        ObjectReference.newBuilder()
                                .setObjectType(permissionDto.resourceType())
                                .setObjectId(permissionDto.resourceId())
                                .build())
                .setSubject(
                        SubjectReference.newBuilder()
                                .setObject(
                                        ObjectReference.newBuilder()
                                                .setObjectType(permissionDto.subjectType())
                                                .setObjectId(permissionDto.subjectId())
                                                .build())
                                .build())
                .setPermission(permissionDto.permissionIdentifier())
                .build();

  try {
     CheckPermissionResponse response = permissionsService
         .checkPermission(request);
     CheckPermissionResponse.Permissionship permissionship = response
         .getPermissionship();

     return permissionship == PERMISSIONSHIP_HAS_PERMISSION;
  } catch (Exception exception) {
     LOG.warn("Failed to check permission: " + exception.getMessage());
     throw exception;
  }
}
```

Listing 12: Die Abfrage einer Berechtigung erfolgt nach gleichem Muster wie das Anlegen einer Beziehung.

Architekturen sollte sich stets an den konkreten fachlichen Anforderungen orientieren.

Quellen

[1] Eclipse Foundation (2025): Jakarta EE. *https://jakarta.ee/*

[2] Eclipse Foundation (2025): Jakarta Security Version 4.0. *https://jakarta.ee/specifications/security/4.0/jakarta-security-spec-4.0*

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

[4] Eclipse Foundation (2024): Jakarta Contexts and Dependency Injection Version 4.1 – 8. Interceptor bindings. *https://jakarta.ee/specifications/cdi/4.1/jakarta-cdi-spec-4.1#interceptors*

[5] Dr. Carrie E. Gates (2007): CA Labs, CA, Islandia, NY 11749 Access Control Requirements for Web 2.0 Security and Privacy. *https://www.researchgate.net/publication/240787391_Access_Control_Requirements_for_Web_20_Security_and_Privacy*

[6] auth0 (2025): Zanzibar Academy. *https://www.zanzibar.academy/*

[7] Ruoming Pang, Ramon Caceres, Mike Burrows und andere (2019): Google, LLC; Humu, Inc.; Carbon, Inc. Zanzibar: Google's Consistent, Global Authorization System. *https://storage.googleapis.com/gweb-research2023-media/pubtools/5068.pdf*

[8] SpiceDB (2025): SpiceDB von Authzed. *https://authzed.com/spicedb*

[9] AuthZed (2025): Authzed. *https://authzed.com/*

[10] gRPC Authors (2025): gRPC. *https://grpc.io/*

[11] Google LLC (2025): Google Spanner. *https://cloud.google.com/spanner*

[12] Cockroach Labs (2025): CockroachDB. *https://www.cockroachlabs.com/product/overview/*

[13] PostgreSQL Global Development Group (2025): PostgreSQL. *https://www.postgresql.org/*

[14] Oracle Corporation (2025): MySQL. *https://www.mysql.com/*

[15] Docker Inc. (2025): Docker. *https://www.docker.com/*

[16] AuthZed (2025): SpiceDB Documentation. *https://authzed.com/docs/spicedb/getting-started/discovering-spicedb*

[17] AuthZed (2025): SpiceDB Schema Language. *https://authzed.com/docs/spicedb/concepts/schema*

[18] AuthZed (2025): SpiceDB Official Client Libraries. *https://authzed.com/docs/spicedb/getting-started/client-libraries*

[19] AuthZed (2025): SpiceDB Consistency. *https://authzed.com/docs/spicedb/concepts/consistency*

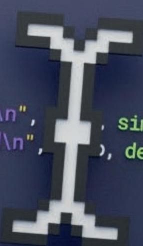
Alle angegebenen URLs wurde das letzte Mal am 28.07.2025 besucht.

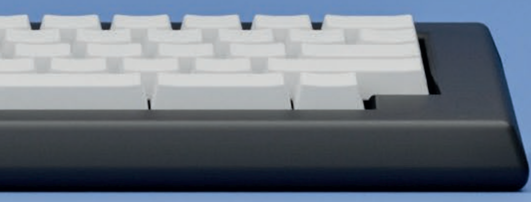
Jelmen Guhlke
mail@jguhlke.de

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

Softwareprojekte selbst hosten mit Forgejo

Marcus Fihlon



/Users/documents/process/develop/simple_test
counter(simple) = 442
simple(1071, 462)= 21
counter(develop) = 3
develop(1071, 462)= 21
Process finished with exit code 0


Forgejo ist eine Open-Source-Software, die es sehr einfach ermöglicht, Softwareprojekte selbst zu hosten. Es ist eine Abspaltung von Gitea und bietet ähnliche Funktionen wie GitHub oder GitLab. In diesem Artikel zeige ich dir, wie du Forgejo auf eigener Hardware selbst installieren und konfigurieren kannst.

Forgejo als echte Alternative

Vielleicht fragst du dich, warum du ausgerechnet Forgejo *[1]* einsetzen solltest, wo es doch mit GitLab *[2]* eine sehr bekannte und etablierte Plattform gibt. Der wichtigste Grund liegt in der Philosophie hinter dem Projekt. Forgejo ist vollständig Open Source und steht unter der GPLv3-Lizenz. Diese Lizenz garantiert, dass du die Software frei verwenden, studieren, verändern und weiterverbreiten darfst. Im Gegensatz zur stärker einschränkenden AGPL verpflichtet die GPLv3 nur dann zur Veröffentlichung eigener Änderungen, wenn du Forgejo weiterverbreitest; nicht jedoch, wenn du es lediglich als Dienst betreibst. Das macht Forgejo besonders attraktiv für selbstverwaltete Installationen und fördert gleichzeitig die Offenheit des Codes.

Im Gegensatz dazu verfolgt GitLab seit Jahren ein Open-Core-Modell: Die Community Edition (CE) ist zwar frei verfügbar, viele nützliche Funktionen wie erweiterte CI/CD-Pipelines, Sicherheitsfeatures oder Authentifizierungsmechanismen sind aber nur in der kostenpflichtigen Enterprise Edition (EE) enthalten. Forgejo geht hier einen anderen Weg und es gibt keine künstliche Trennung in kostenlose und Premium-Features. Alles, was entwickelt wird, steht allen zur Verfügung.

Ein weiterer Vorteil von Forgejo ist seine Leichtgewichtigkeit. Die Software ist in Go *[3]* geschrieben, einfach zu installieren und kommt mit geringen Systemanforderungen aus. Für kleine Teams, Einzelpersonen oder Bildungsprojekte ist das ideal. GitLab dagegen bringt eine ganze Reihe an Abhängigkeiten mit, darunter PostgreSQL *[4]*, Redis *[5]* und Sidekiq *[6]*, was nicht nur die Einrichtung, sondern auch den Betrieb deutlich aufwändiger macht.

Nicht zuletzt ist Forgejo ein echtes Community-Projekt. Es wurde als Abspaltung von Gitea *[7]* ins Leben gerufen, um die Mitbestimmung der Gemeinschaft zu stärken und die Entwicklung dezentral zu organisieren. Es gibt keine Firma im Hintergrund, die Entscheidungen trifft oder auf Profit schielt. Wer Forgejo nutzt, unterstützt damit ein freies, gemeinschaftlich getragenes Projekt – und genau das macht den entscheidenden Unterschied.

Passend dazu ist auch die Herkunft des Namens: Forgejo setzt sich aus dem englischen „forge“ (Code-Schmiede) und dem französischen „rejoindre“ (sich anschließen) zusammen. Das steht sinnbildlich für die Idee, gemeinsam an offenen Werkzeugen zu arbeiten. In einer Umgebung, der man sich gerne anschließt.

Ein spannender Ausblick: Aktuell arbeitet die Forgejo-Community an der Unterstützung des ActivityPub-Protokolls *[8]*, bekannt aus dem Fediverse (etwa von Mastodon *[9]*, PeerTube *[10]*, Pixelfed *[11]*, und vielen mehr). Ziel ist es, Aktionen wie Likes, Forks, Pull Requests, und mehr über Instanzgrenzen hinweg austauschbar zu machen. Damit könnte Forgejo künftig ein wichtiger Baustein für ein dezentrales, föderiertes Entwicklungsnetzwerk werden – ganz ohne zentrale Plattformen. Die Funktion steckt noch in der Entwicklung und ist in der stabilen Version noch nicht verfügbar. Sie zeigt aber schon jetzt, in welche Richtung sich Forgejo bewegt: offen, vernetzt und unabhängig.

Wenn dich das überzeugt hat und du Forgejo selbst ausprobieren möchtest, kannst du direkt loslegen. Die Installation ist nämlich erstaunlich einfach. Auf den nächsten Seiten zeige ich dir Schritt für Schritt, wie du Forgejo auf eigener Hardware installierst, konfigurierst und startklar machst.

Ausgangslage

Bevor wir in die eigentliche Installation einsteigen, werfen wir einen kurzen Blick auf die Anforderungen. Forgejo ist erfreulich genügsam und kommt mit deutlich weniger Ressourcen aus als viele vergleichbare Plattformen. Wer möchte, kann es sogar auf einem Raspberry Pi betreiben: Das ist ideal für den Heimgebrauch oder erste Tests.

In meinem Beispiel verwende ich allerdings einen kleinen Cloud-Server bei Hetzner *[12]*. Die hier gewählte Instanz ist das kleinste verfügbare Paket mit 2 vCPU-Kernen, 4 GB RAM und einem monatlichen Höchstpreis von gerade einmal 3,79 Euro (Stand: Mai 2025). Damit zeige ich, dass sich Forgejo auch mit sehr begrenzten Mitteln zuverlässig betreiben lässt. Ganz gleich, ob auf echter Hardware, in einer virtuellen Maschine oder in der Cloud.

Für die Installation setze ich auf Docker *[13]* und Docker Compose, was den Einstieg besonders einfach macht und auch die spätere Wartung vereinfacht. Dabei gehe ich davon aus, dass Docker bereits fertig installiert und funktionsfähig ist. Wer lieber Podman *[14]* verwendet, kann die folgenden Schritte sinngemäß übernehmen. Die Forgejo-Container laufen dort ebenso problemlos.

Um Forgejo später öffentlich erreichbar zu machen, setzen wir den Reverse Proxy Traefik *[15]* ein. Dieser übernimmt nicht nur das Routing von HTTPS-Anfragen an die richtigen Container, sondern kümmert sich auch automatisch um die Ausstellung und Verlängerung eines gültigen „Let's Encrypt“ *[16]* TLS-Zertifikats. In unserer Beispielinstallation nutzen wir die Domains git.example.eu für die Forgejo-Oberfläche und traefik.example.eu für das Dashboard von Traefik. Beide Domains zeigen auf die öffentliche IP-Adresse unseres Servers und werden in der Traefik-Konfiguration entsprechend eingebunden.

Diese beiden Domains sind nur Platzhalter – du musst sie in den folgenden Schritten durch deine eigenen Domains ersetzen.

Los geht's

Im ersten Schritt erstellen wir auf dem Server, auf dem Forgejo später laufen soll, ein neues Verzeichnis. Dort legen wir alle benötigten Dateien ab. Ich nenne das Verzeichnis forgejo, du kannst aber auch einen anderen Namen wählen. In diesem neuen Verzeichnis legen wir die neue Datei docker-compose.yml an. Diese Datei wird alle

Konfigurationen enthalten, die wir für Forgejo benötigen. Starten wir mit der Konfiguration eines neuen Netzwerkes für Forgejo.

Übernehme dazu die vier Zeilen aus *Listing 1* und achte dabei genau auf die Einrückungen! YAML ist sehr empfindlich gegenüber falsch gesetzten Leerzeichen. Bereits ein einzelnes fehlendes Leerzeichen kann dazu führen, dass Docker die Datei nicht mehr korrekt verarbeitet.

```
networks:
  forgejo:
     driver: bridge
     enable_ipv6: true
```

Listing 1: Konfiguration des Docker-Netzwerks

Das Netzwerk forgejo ist ein isoliertes, virtuelles Layer-3-Netzwerk mit einem NAT-Gateway. Es erlaubt den Containern die interne Kommunikation untereinander, ohne dass sie direkt über das Host-Netzwerk erreichbar sind. Mit der Option enable_ipv6: true aktivierst du die Unterstützung für IPv6-Adressen. Das ist vor allem dann sinnvoll, wenn dein Server über eine öffentliche IPv6-Adresse verfügt oder du Forgejo zukunftssicher betreiben möchtest, denn immer mehr Dienste und Netze setzen inzwischen auf IPv6.

Datenbank

Forgejo benötigt eine relationale Datenbank, um Benutzerdaten, Repositories, Issues und andere Inhalte zu speichern. Unterstützt werden SQLite *[17]*, MariaDB *[18]*, MySQL *[19]* sowie PostgreSQL *[4]*. Du kannst also je nach Einsatzzweck und Umgebung flexibel wählen.

Standardmäßig kommt SQLite zum Einsatz. Das ist eine schlanke, dateibasierte Lösung, die keine separate Datenbankinstanz benötigt. Für erste Tests oder kleine Einzelinstanzen eignet sich das hervorragend: Es genügt, Forgejo zu starten, und die SQLite-Datenbank wird automatisch als Datei im Datenverzeichnis angelegt.

Für produktive Umgebungen ist jedoch eine vollwertige Datenbank empfehlenswert. Vor allem, wenn mehrere Benutzer darauf zugreifen oder mit wachsendem Datenvolumen zu rechnen ist. In diesem Artikel verwende ich deshalb MariaDB, einen ursprünglich vollständig kompatiblen Fork von MySQL, der sich inzwischen jedoch eigenständig weiterentwickelt hat. Sie ist in der Open-Source-Welt weit verbreitet, gut dokumentiert und lässt sich problemlos als eigener Container einbinden.

Im nächsten Schritt richten wir die MariaDB-Datenbank ein und konfigurieren später Forgejo so, dass es diese als Backend verwendet. Das ermöglicht nicht nur mehr Stabilität und Performance, sondern auch eine einfachere Verwaltung und Datensicherung im laufenden Betrieb.

In der docker-compose.yml-Datei fügen wir nun den in *Listing 2* abgebildeten Service hinzu, um MariaDB zu konfigurieren.

```
services:
  mariadb:
     image: mariadb:11
     container_name: mariadb
     restart: unless-stopped
     networks:
        - forgejo
     ports:
        - "127.0.0.1:3306:3306"
     environment:
        - MARIADB_ALLOW_EMPTY_ROOT_PASSWORD=0
        - MARIADB_DATABASE=forgejo
        - MARIADB_USER=forgejo
        - MARIADB_PASSWORD=forgejo-db-passwort
     volumes:
        - /data/docker/mariadb:/var/lib/mysql
```

Listing 2: Konfiguration des MariaDB-Containers

Wir verwenden das offizielle MariaDB-Image in der Version 11. Der Container wird mit dem Namen `mariadb` gestartet und erhält die Option `restart: unless-stopped`, damit er automatisch neu gestartet wird, wenn er unerwartet beendet wird. Mit `networks: forgejo` wird der Container dem zuvor definierten Netzwerk `forgejo` zugewiesen. Das ist wichtig, damit die Container von MariaDB und Forgejo später untereinander kommunizieren können.

Wir geben den Port `3306` intern innerhalb des virtuellen `forgejo`-Netzwerks frei. Damit ist die Datenbank nur für die Container innerhalb des Netzwerks erreichbar und nicht von außen. Ein wichtiger Schritt, um die Sicherheit zu erhöhen und unbefugten Zugriff zu verhindern.

Damit beim ersten Start des MariaDB-Containers automatisch eine Datenbank und ein passender Benutzer erstellt werden, definieren wir einige Umgebungsvariablen. Diese greifen nur beim initialen Start, also so lange das Datenverzeichnis noch leer ist.

- **MARIADB_ALLOW_EMPTY_ROOT_PASSWORD=0**
  erzwingt, dass kein leeres Root-Passwort erlaubt ist. Da wir in unserem Beispiel kein Root-Passwort setzen, ist der Root-Zugriff deaktiviert und wird auch nicht benötigt.
- **MARIADB_DATABASE=forgejo**
  legt beim ersten Start eine neue Datenbank mit dem angegebenen Namen an.
- **MARIADB_USER=forgejo**
  erstellt einen Benutzer mit dem angegebenen Namen.
- **MARIADB_PASSWORD=forgejo-db-passwort**
  setzt das Passwort für den oben genannten Benutzer.

Dieser Benutzer erhält automatisch vollständige Zugriffsrechte auf die angelegte Datenbank. Forgejo kann sich später mit genau diesen Zugangsdaten verbinden, ohne dass wir manuell in die Datenbank eingreifen müssen.

Damit die Datenbankdaten auch nach einem Neustart des Containers erhalten bleiben, binden wir das interne Datenverzeichnis des MariaDB-Containers `/var/lib/mysql` dauerhaft an ein Verzeichnis auf dem Host-System, in unserem Beispiel ist das `/data/docker/mariadb`.

Auch wenn der Pfadname im Container auf `mysql` endet, ist das korrekt: MariaDB wurde ursprünglich aus MySQL heraus entwickelt und verwendet bis heute viele identische Strukturen, einschließlich des Standardpfads zur Datenbankablage.

Das gewählte Host-Verzeichnis kann beliebig angepasst werden.

Wichtig ist lediglich, dass es vor dem ersten Start existiert und leer ist. Beim Initialisieren legt MariaDB dort seine Datenbankstruktur an. Bestehen bereits Dateien im Zielverzeichnis, wird der Initialisierungsvorgang übersprungen.

Also erstellen wir jetzt zuerst das Datenverzeichnis und anschließend starten wir MariaDB über `docker compose`:

```
$ mkdir -p /data/docker/mariadb
$ docker compose up mariadb
```

Das kann etwas dauern, da zunächst das Image heruntergeladen und die Datenbank initialisiert wird. Wenn alles funktioniert, siehst du eine Ausgabe wie diese:

```
[Note] mariadbd: ready for connections.
Version: '11.7.2-MariaDB-ubu2404'  socket: '/run/mysqld/
mysqld.sock'  port: 3306  mariadb.org binary distribution
```

Das bedeutet, dass die Datenbank initialisiert und der Benutzer erfolgreich angelegt wurde und MariaDB nun bereit ist, Verbindungen entgegenzunehmen.

Du kannst den Container jetzt mit `CTRL+C` beenden. Dadurch wird die Datenbank sauber heruntergefahren und alle Daten werden zuverlässig gespeichert. Im Anschluss räumen wir unsere `docker-compose.yml` etwas auf und entfernen die folgenden drei Zeilen, die nur beim ersten Start benötigt wurden:

```
- MARIADB_DATABASE=forgejo
- MARIADB_USER=forgejo
- MARIADB_PASSWORD=forgejo-db-passwort
```

Die Zeile `MARIADB_ALLOW_EMPTY_ROOT_PASSWORD=0` bleibt bestehen, da sie auch bei zukünftigen Starts relevant ist.

Reverse Proxy mit TLS

Damit Forgejo später über das Internet erreichbar ist, benötigen wir einen Reverse Proxy. In diesem Beispiel setzen wir auf Traefik, einen modernen, leichtgewichtigen Proxy, der sich hervorragend in Docker-Umgebungen integrieren lässt. Traefik übernimmt nicht nur das Routing der Anfragen auf die richtigen Container, sondern kümmert sich auch um die automatische Ausstellung und Verlängerung von TLS-Zertifikaten über Let's Encrypt.

Wir fügen dazu die in *Listing 3* gezeigte Konfiguration in unsere `docker-compose.yml`-Datei im Abschnitt services: ein.

Hier eine kurze Erklärung der wichtigsten Punkte dieser Konfiguration:

- Der Container lauscht auf den Ports `80` (HTTP) und `443` (HTTPS) und ist damit von außen erreichbar.
- Über den Mount `/var/run/docker.sock` kann Traefik Informationen über andere Container auslesen und automatisch Routen generieren.
- Die Datei `traefik.yml` enthält die zentrale Konfiguration (dazu gleich mehr).
- Die Datei `acme.json` speichert die von Let's Encrypt ausgestellten TLS-Zertifikate.
- Unter der Domain `traefik.example.eu` wird das Traefik-Dashboard erreichbar gemacht und durch HTTP Basic Auth geschützt.

Die Datei `acme.json`, in der Traefik die von Let's Encrypt ausgestellten TLS-Zertifikate speichert, muss vor dem ersten Start von Traefik bereits existieren. Sie darf außerdem nur vom Container lesbar und schreibbar sein. Deshalb solltest du sie mit folgendem Befehl anlegen und korrekt absichern:

```
$ mkdir -p /data/docker/traefik
$ touch /data/docker/traefik/acme.json
$ chmod 600 /data/docker/traefik/acme.json
```

Wenn du an Stelle von Docker Podman verwendest, kann es notwendig sein, zusätzlich den Eigentümer des Verzeichnisses `/data/docker/traefik` und der Datei acme.json anzupassen, damit der Container die Datei überhaupt schreiben darf. In vielen Fällen genügt:

```
traefik:
   image: traefik:latest
   container_name: traefik
   restart: unless-stopped
   networks:
      - forgejo
   ports:
      - "80:80"
      - "443:443"
   volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - /data/docker/traefik/traefik.yml:/traefik.yml:ro
      - /data/docker/traefik/acme.json:/acme.json
   labels:
      - "traefik.enable=true"
      - "traefik.http.routers.dashboard.rule=Host(`traefik.example.eu`)"
      - "traefik.http.routers.dashboard.entrypoints=https"
      - "traefik.http.routers.dashboard.tls.certresolver=letsencrypt"
      - "traefik.http.routers.dashboard.service=api@internal"
      - "traefik.http.routers.dashboard.middlewares=auth"
      - "traefik.http.middlewares.auth.basicauth.users=benutzername:passworthash"
```

Listing 3: Konfiguration des Traefik-Containers

NEWSLETTER

Anmeldung

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

Abonniere ihn kostenfrei und bleibe immer auf dem Laufenden!

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

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

```
$ chown 1000:1000 /data/docker/traefik /data/docker/traefik/acme.json
```

Dabei steht `1000:1000` für den Standardbenutzer im Container. Du solltest dies gegebenenfalls an dein System anpassen. Grundsätzlich kannst du auch ein anderes Verzeichnis für die Traefik-Daten verwenden, sofern du es in der `docker-compose.yml` entsprechend anpasst.

Ohne diese Vorbereitung kann Traefik keine Zertifikate generieren oder speichern. Traefik tut sich manchmal mit Fehlermeldungen etwas schwer. Es kann daher vorkommen, dass du im Fehlerfall keine Fehlermeldung erhältst, sondern nur eine leere `acme.json`-Datei. Das ist ein Hinweis darauf, dass Traefik nicht auf die Datei zugreifen konnte oder aus anderen Gründen nicht in der Lage war, ein Zertifikat anzufordern. Überprüfe in diesem Fall die Berechtigungen und den Eigentümer des Verzeichnisses sowie die Traefik-Konfiguration.

Auch das Traefik-Dashboard sollte nicht ungeschützt zugänglich sein. In unserem Beispiel verwenden wir dafür HTTP Basic Auth. Der Passwort-Hash wird in einer Traefik-Middleware definiert und muss im Format `benutzer:passworthash` angegeben werden. Du kannst den Hash mit dem folgenden Befehl generieren:

```
$ htpasswd -nB benutzername
```

Dadurch wirst du zur Eingabe eines Passworts aufgefordert. Der generierte Hash kann anschließend direkt in die Konfiguration übernommen werden. Das Tool `htpasswd` ist Bestandteil des Pakets apache2-utils, das auf Debian-basierten Systemen wie Ubuntu oder Linux Mint mit folgendem Befehl installiert werden kann:

```
$ apt install apache2-utils
```

Damit Traefik korrekt funktioniert, benötigen wir eine zentrale Konfigurationsdatei. Diese wird in unserem Beispiel als `traefik.yml` im Verzeichnis `/data/docker/traefik/` abgelegt und beim Start des Containers eingebunden. Sie steuert grundlegende Eigenschaften wie die aktivierten Protokolle, das Zertifikatsmanagement und das Dashboard.

Dazu erstellen wir die Datei `traefik.yml` mit dem Inhalt aus *Listing 4*.

```
entryPoints:
  http:
     address: ":80"
  https:
     address: ":443"

api:
  dashboard: true

providers:
  docker:
     endpoint: "unix:///var/run/docker.sock"
     exposedByDefault: false

certificatesResolvers:
  letsencrypt:
     acme:
        email: admin@example.eu
        storage: /acme.json
        httpChallenge:
           entryPoint: http
```

Listing 4: Unsere traefik.yml-Datei

Erklärung der wichtigsten Optionen:

- **entryPoints**
  definiert, auf welchen Ports Traefik lauscht. In unserem Fall auf Port 80 für HTTP und Port 443 für HTTPS.
- **api.dashboard: true**
  aktiviert das eingebaute Dashboard, das später unter der konfigurierten Domain erreichbar ist.
- **providers.docker**
  erlaubt es Traefik, automatisch Informationen über Docker-Container auszulesen. Wichtig ist hier exposedByDefault: false, damit nicht alle Container automatisch veröffentlicht werden, sondern nur solche mit dem Label traefik.enable=true.
- **certificatesResolvers**
  hier wird Let's Encrypt als Zertifikatsanbieter konfiguriert. Wir verwenden den httpChallenge, bei dem Traefik eine temporäre HTTP-Route bereitstellt, über die Let's Encrypt die Domain verifizieren kann.
- **email**
  die Kontaktadresse, die Let's Encrypt bei Zertifikatsanfragen hinterlegt. Bitte mit deiner echten E-Mail-Adresse ersetzen!
- **storage**
  der Pfad, unter dem die Zertifikate gespeichert werden. In unserem Fall /acme.json.

Erneut: Achte darauf, dass die Einrückungen in der `traefik.yml` exakt stimmen. YAML ist sehr empfindlich gegenüber Einrückungsfehlern.

Sobald diese Datei erstellt wurde und die zugehörige `acme.json` vorbereitet ist, kann Traefik gestartet werden. Der Proxy ist dann bereit, TLS-Zertifikate automatisch zu beziehen und Container anhand von Labels zu erkennen und zu veröffentlichen. Jetzt kannst du Traefik zum ersten Mal starten:

```
$ docker compose up traefik
```

Traefik läuft damit im Vordergrund, sodass du direkt im Terminal verfolgen kannst, ob alles wie erwartet funktioniert. Idealerweise gibt es von Traefik nach dem erstmaligen Herunterladen des Docker Images keine Ausgabe, da es sich bei laufendem Betrieb recht schweigsam verhält.

Rufe anschließend im Browser die von dir konfigurierte Domain für das Dashboard auf. In unserem Beispiel ist das *https://traefik.example.eu*. Du wirst aufgefordert, deinen Benutzernamen und dein Passwort einzugeben, das du zuvor mit `htpasswd` erstellt und in der `docker-compose.yml` hinterlegt hast.

Wenn das Traefik-Dashboard erscheint, ist alles korrekt eingerichtet. Zwar ist dort aktuell noch nicht viel zu sehen, aber du kannst bereits erkennen, dass Traefik läuft, HTTPS aktiv ist und das TLS-Zertifikat erfolgreich ausgestellt wurde. Dieses wurde automatisch in der Datei acme.json abgelegt.

Zum Vergleich findest du in *Abbildung 1* einen Screenshot des Dashboards nach dem ersten Start. Wenn alles wie erwartet funktioniert hat, kannst du den laufenden Traefik-Container mit CTRL+C wieder beenden.

Endlich: Forgejo einrichten

Jetzt, da Traefik läuft und die Grundlage für den sicheren Zugriff von außen geschaffen ist, kümmern wir uns um die Einrichtung von For-

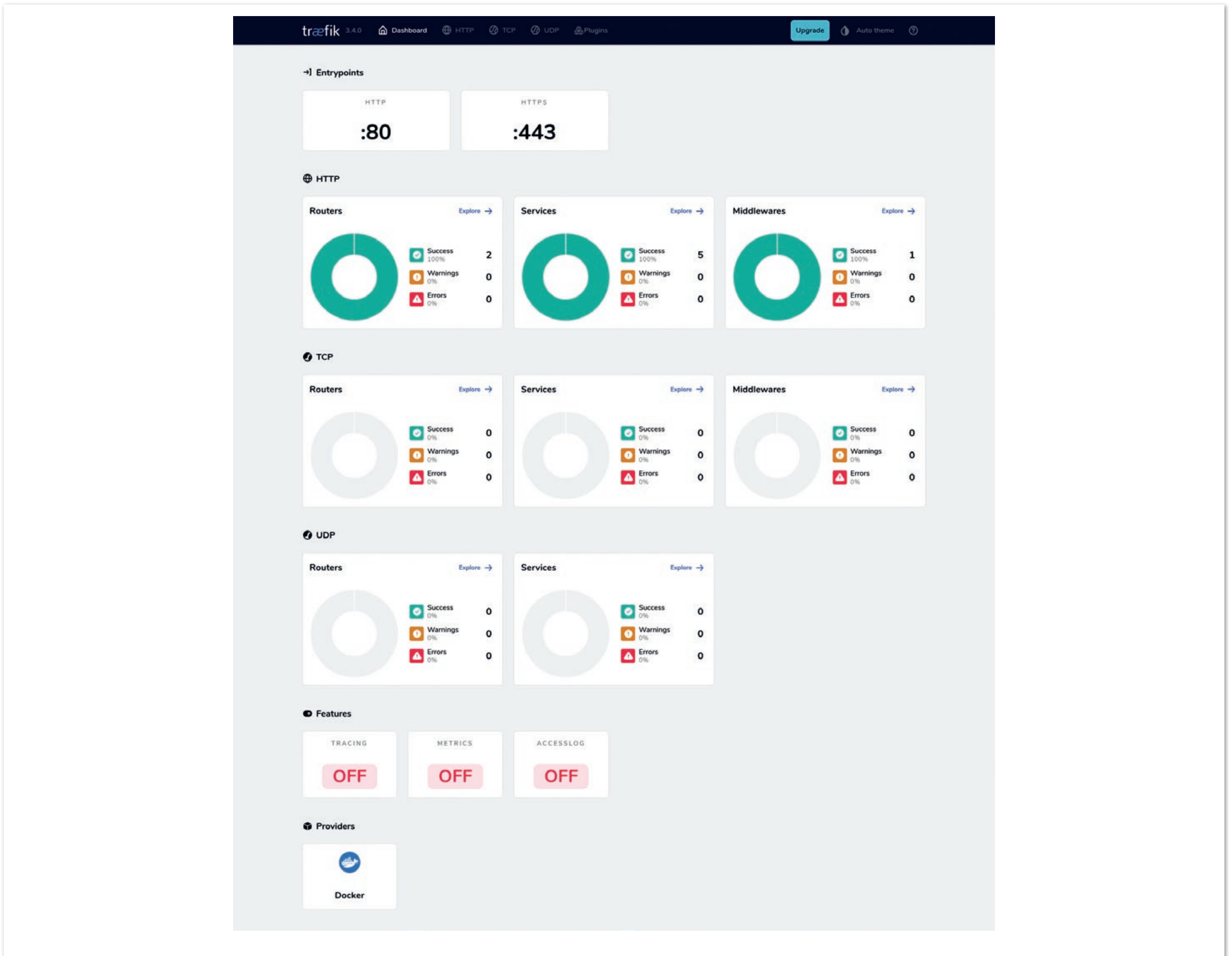


Abbildung 1: Das Traefik-Dashboard nach dem ersten Start

gejo selbst. Auch Forgejo betreiben wir als Docker-Container und integrieren es in das bereits eingerichtete `forgejo`-Netzwerk.

Ein wichtiger Punkt vorab: Forgejo bietet die Möglichkeit, über SSH Repositories zu klonen oder zu pushen. Der Standardport dafür ist 22. Dieser ist allerdings auf unserem Server bereits durch den SSH-Dienst des Host-Systems belegt. Daher verwenden wir für Forgejo stattdessen den Port 23. Dieser muss bei Bedarf auch in der Firewall geöffnet werden.

Wir übernehmen nun die in *Listing 5* vorbereitete Konfiguration in den Abschnitt `services:` unserer `docker-compose.yml`-Datei.

Erklärung der wichtigsten Punkte:

- **Ports:** Forgejo nutzt intern den Port `22` für SSH. Da dieser auf dem Host-System meist bereits durch den SSH-Dienst belegt ist, wird er hier auf den Host-Port `23` gemappt `(23:22)`. Wer SSH für Repositories nutzen möchte, muss später `ssh://user@git.example.eu:23/...` verwenden.
- **Volumes:** Das Volume `/data/docker/forgejo:/data` speichert alle Forgejo-Daten dauerhaft auf dem Host. Zusätzlich werden Zeitzone und Systemzeit eingebunden, damit Forgejo korrekt lokalisiert arbeitet.
- **Environment-Variablen:**
  - `USER_UID` und `USER_GID` stellen sicher, dass Forgejo im Container nicht als root Benutzer läuft.
  - `FORGEJO__server__DOMAIN` und `ROOT_URL` definieren die öffentliche URL, unter der Forgejo erreichbar ist. Das ist für Links in E-Mails und der Weboberfläche wichtig.
  - Die Datenbank-Parameter stellen die Verbindung zur zuvor eingerichteten MariaDB-Datenbank her.
- **Labels:** Diese Konfiguration macht Forgejo über Traefik erreichbar. Besonders wichtig ist hier:
  - `loadbalancer.server.port=3000`, da die Weboberfläche von Forgejo intern auf Port 3000 lauscht.
  - `traefik.enable=true` und die zugehörige Router-Definition sorgen dafür, dass Traefik automatisch HTTPS-Zugriff über die Domain `git.example.eu` bereitstellt.

Bevor du den Forgejo-Container startest, solltest du sicherstellen, dass das Verzeichnis für das Volume bereits existiert. In unserem Beispiel ist das `/data/docker/forgejo`. Du kannst es wie folgt anlegen:

```
$ mkdir -p /data/docker/forgejo
```

Du kannst auch ein anderes Verzeichnis verwenden. Wichtig ist nur,

```
forgejo:
   image: codeberg.org/forgejo/forgejo:11
   container_name: forgejo
   restart: unless-stopped
   networks:
      - forgejo
   ports:
      - "23:22"
   environment:
      - USER_UID=1000
      - USER_GID=1000
      - FORGEJO__server__DOMAIN=git.example.eu
      - FORGEJO__server__ROOT_URL=https://git.example.eu
      - FORGEJO__database__DB_TYPE=mysql
      - FORGEJO__database__HOST=mariadb:3306
      - FORGEJO__database__NAME=forgejo
      - FORGEJO__database__USER=forgejo
      - FORGEJO__database__PASSWD=forgejo-db-passwort
   volumes:
      - /data/docker/forgejo:/data
      - /etc/timezone:/etc/timezone:ro
      - /etc/localtime:/etc/localtime:ro
   depends_on:
      - mariadb
   labels:
      - "traefik.enable=true"
      - "traefik.http.routers.forgejo.rule=Host(`git.example.eu`)"
      - "traefik.http.routers.forgejo.entrypoints=https"
      - "traefik.http.routers.forgejo.tls.certresolver=letsencrypt"
      - "traefik.http.routers.forgejo.service=forgejo"
      - "traefik.http.services.forgejo.loadbalancer.server.port=3000"
```

Listing 5: Konfiguration des Forgejo-Containers

dass es leer ist und dauerhaft bestehen bleibt. Forgejo speichert darin unter anderem die Git-Repositories, Konfigurationsdateien, Benutzer-SSH-Schlüssel, Avatare, Anhänge und Logdateien. Die eigentlichen Benutzerinformationen wie Konten, Passwörter, Rechte sowie Issues, Pull Requests und Kommentare werden dagegen in der Datenbank abgelegt, die wir zuvor mit MariaDB eingerichtet haben.

Nun ist es endlich so weit: Wir starten Forgejo zusammen mit MariaDB und Traefik. Das geht ganz einfach mit dem folgenden Befehl:

```
$ docker compose up -d
```

Das `-d` sorgt dafür, dass die Container im Hintergrund gestartet werden. Dadurch sind die Ausgaben der Container nicht im Terminal sichtbar. Wir können den Status der Container mit folgendem Befehl überprüfen:

```
$ docker compose ps
```

Wenn alles gut gelaufen ist, sollten wir jetzt drei Container sehen: `mariadb`, `traefik` und `forgejo`. Um die Logfiles der im Hintergrund laufenden Container zu sehen, können wir den folgenden Befehl verwenden:

```
$ docker compose logs -f
```

Das `-f` sorgt dafür, dass die Logs in Echtzeit aktualisiert werden. Wir können die Ausgabe mit `CTRL+C` beenden.

Nun öffnen wir die konfigurierte Domain im Browser. In unserem Beispiel ist das *https://git.example.eu*. Wir werden auf das grafische Setup weitergeleitet, in dem wir die Grundkonfiguration von Forgejo vornehmen können. Da Forgejo im Docker-Container läuft, sind viele Einstellungen bereits korrekt vorausgefüllt. Ganz am Ende der Seite finden wir einige zusätzliche Optionen, die zunächst aufgeklappt und dann angepasst werden sollten:

1. E-Mail-Einstellungen: Damit Forgejo E-Mails versenden kann, etwa zur Aktivierung von Benutzerkonten oder für Benachrichtigungen.
2. Administratorkonto: Dieses Konto gibt dir Zugriff auf alle Funktionen von Forgejo.

Nach Abschluss des Setups können wir uns mit dem neu angelegten Administratorkonto anmelden und sollten eine funktionierende Forgejo-Instanz vorfinden, wie in *Abbildung 2* exemplarisch dargestellt.

Continuous Integration: Ein eigener Runner

Unsere Forgejo-Instanz ist nun für den täglichen Einsatz bereit. Wir können Repositories hosten, mit TLS absichern und über die Weboberfläche und per SSH verwalten. Um das volle Potenzial auszuschöpfen, richten wir jetzt einen eigenen Runner ein. Damit können wir auf dem Server automatisierte Abläufe wie Builds, Tests oder Deployments direkt auf unserer eigenen Infrastruktur ausführen und sind unabhängig von Drittanbietern.

Ein Runner ist ein separates Programm, das Aufgaben im Rahmen von CI/CD-Pipelines ausführt. Sobald wir in einem Repository eine Workflow-Datei im Verzeichnis `.forgejo/workflows/` hinterlegen, kann Forgejo diese automatisiert abarbeiten, um beispielsweise unseren Code zu testen oder ein Docker-Image zu bauen.

Forgejo bringt dafür eine eigene, leichtgewichtige CI-Lösung mit, die direkt in die Plattform integriert ist. Damit diese funktioniert, benötigen wir mindestens einen aktiven Runner, der die Jobs tatsächlich ausführt. Dabei ist es nicht notwendig, den Runner auf dem

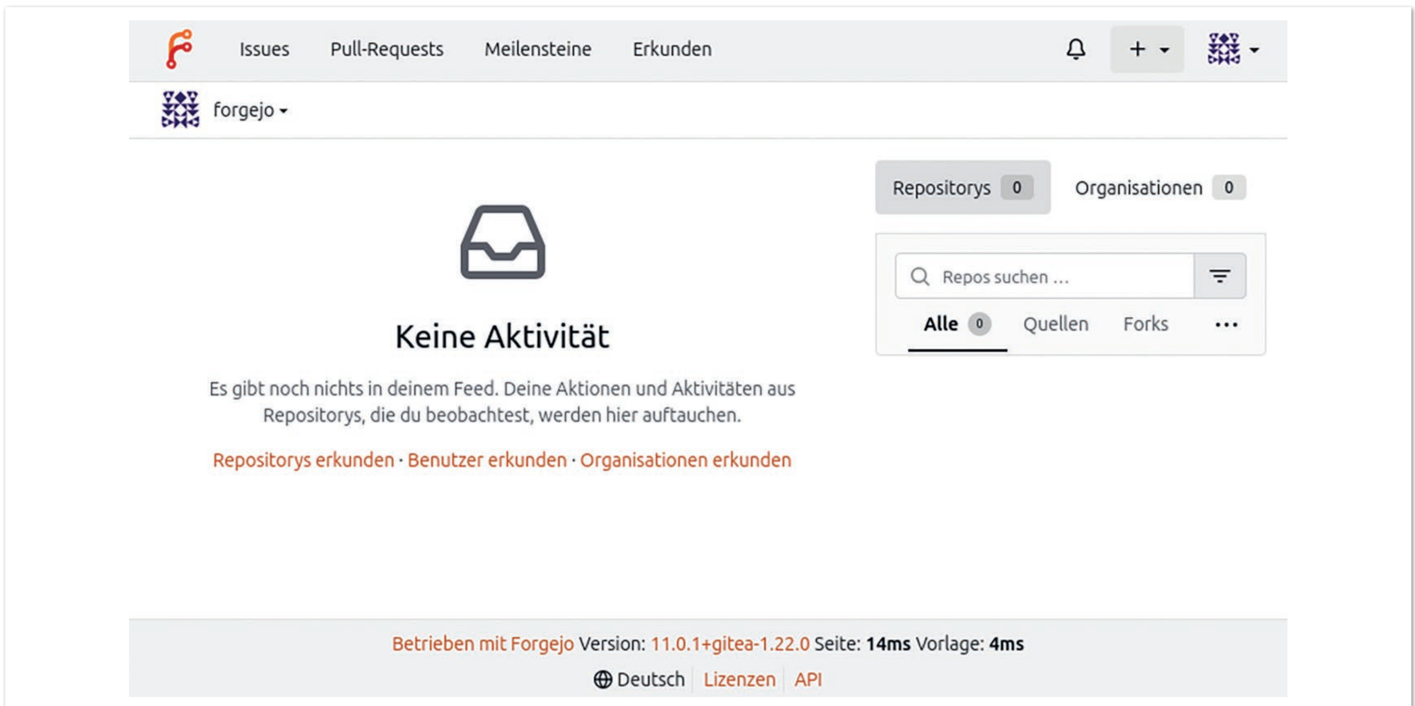


Abbildung 2: Die Forgejo-Oberfläche nach der Installation

gleichen Server wie Forgejo selbst zu betreiben. Er kann auch auf einem separaten System, in einer anderen Netzwerkumgebung, unter anderen Betriebssystemen oder sogar auf abweichender Hardwarearchitektur laufen.

Das ermöglicht uns, gezielt Build- und Testumgebungen für verschiedene Plattformen bereitzustellen; etwa für macOS, Windows, Linux oder auch für unterschiedliche Architekturen wie x86_64 und ARM. So lassen sich komplexe Multi-Plattform-Projekte mit Forgejo effizient und flexibel automatisieren.

Der Einfachheit halber installieren wir in unserem Beispiel jedoch nur einen einzigen Runner auf dem gleichen Host, auf dem auch Forgejo selbst läuft. Damit ist der Einstieg besonders unkompliziert und wir können alle notwendigen Komponenten gemeinsam verwalten und betreiben.

Um Forgejo-Jobs lokal ausführen zu können, müssen wir den Runner zunächst bei unserer Instanz registrieren. Dazu klicken wir im Webinterface oben rechts auf unser Profilbild und wählen aus dem Kontextmenü den Eintrag *Administration*. Anschließend können wir im Menü auf der linken Seite den Eintrag *Actions* aufklappen und dort den Unterpunkt *Runner* auswählen.

Da wir eine neue Installation vor uns haben, sollte die nun angezeigte Liste leer sein. Um unseren neuen Runner zu registrieren, klicken wir oben rechts auf *„Neuen Runner erstellen“* und kopieren aus dem dann erscheinenden Dialog nur den *Registration Token*. Diesen benötigen wir später, um den Runner mit unserer Forgejo-Instanz zu verbinden.

Nun ergänzen wir in unserer `docker-compose.yml` den neuen Service für den Runner, indem wir die Konfiguration aus *Listing 6* in den Abschnitt `services:` einfügen.

Bevor wir den Runner starten, müssen wir sicherstellen, dass das Datenverzeichnis existiert:

```
$ mkdir -p /data/docker/forgejo-runner/data
```

Dann erzeugen wir die initiale Konfigurationsdatei, die Forgejo zum Betrieb des Runners erwartet:

```
$ docker compose run --rm \
  forgejo-runner forgejo-runner generate-config \
  > /data/docker/forgejo-runner/data/config.yaml
```

Für die Registrierung führen wir den Runner zunächst manuell im interaktiven Setup-Modus aus. Dabei wird die Konfigurationsdatei `config.yaml` automatisch im Datenverzeichnis erstellt und mit den benötigten Werten befüllt:

```
$ docker compose run --rm -it \
  forgejo-runner forgejo-runner register
```

Wir werden nun am Terminal nach der Forgejo-Instanz-URL (in un-

```
forgejo-runner:
   image: code.forgejo.org/forgejo/runner:6
   container_name: forgejo-runner
   restart: unless-stopped
   volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /data/docker/forgejo-runner/data:/data
   environment:
      - DOCKER_HOST=unix:///var/run/docker.sock
   networks:
      - forgejo
   user: 0:0
   command: forgejo-runner -c /data/config.yaml daemon
```

Listing 6: Konfiguration des Forgejo-Runners

serem Beispiel *https://git.example.eu*) und dem Token gefragt, den wir im Webinterface erhalten haben. Danach folgen ein paar weitere Fragen zur Konfiguration des Runners. Für unser Beispiel können wir hier die Standardwerte übernehmen.

Sobald die Registrierung abgeschlossen ist, starten wir den Runner regulär über `docker compose`:

```
$ docker compose up -d forgejo-runner
```

Wenn wir nun die Seite mit den Runnern in der Weboberfläche von Forgejo neu laden, sollte der soeben registrierte Runner dort erscheinen. Ab diesem Moment ist er einsatzbereit. Sobald ein Repository eine gültige Workflow-Datei im Verzeichnis `.forgejo/workflows/` enthält, kann Forgejo die darin definierten Jobs an unseren Runner übergeben und automatisiert ausführen lassen.

Nächste Schritte

Unsere Forgejo-Instanz ist jetzt einsatzbereit. Sie ist sicher erreichbar sowie mit eigener Datenbank und CI-Runner ausgestattet. Um den Betrieb langfristig stabil und zuverlässig zu gestalten, gibt es jedoch noch einige sinnvolle Maßnahmen, die wir als Nächstes angehen sollten.
Damit keine Daten verloren gehen, sollten wir ein regelmäßiges Backup-Konzept etablieren. Wichtig ist dabei:

- **Datenbank sichern**
  Das MariaDB-Verzeichnis unter `/data/docker/mariadb` sollte nicht direkt gesichert werden, da laufende Datenbankdateien inkonsistent sein können. Stattdessen sollten wir regelmäßig einen Datenbank-Abzug erstellen, beispielsweise mit `mysqldump`, und diesen sichern.
- **Forgejo-Daten sichern**
  Das Verzeichnis `/data/docker/forgejo` enthält die Repositories, Konfigurationsdateien und andere persistente Daten. Dieses Verzeichnis sollte regelmäßig gesichert werden.
- **Runner-Konfiguration sichern**
  Auch das Verzeichnis `/data/docker/forgejo-runner` sollte bei Bedarf ins Backup aufgenommen werden, um Registrierungsdaten und Logs zu erhalten.

Damit unsere Dienste nach einem Neustart des Servers automatisch wieder starten, sollten wir `docker compose` als Systemdienst einrichten. Eine einfache Möglichkeit ist die Erstellung eines systemd Services, der die `docker compose up` Befehle beim Booten des Systems ausführt. Das sorgt dafür, dass alle Container automatisch gestartet werden, ohne dass wir manuell eingreifen müssen.

Optional kann es sich lohnen, die SSH-Ports zwischen dem Host-System und Forgejo zu tauschen. Standardmäßig lauscht der SSH-Dienst des Host-Systems auf Port 22, während der Forgejo-Container (wie in unserem Beispiel) über Port 23 erreichbar ist. Wer häufig über SSH mit Forgejo arbeitet (etwa zum Klonen oder Pushen von Repositories), muss dadurch immer die Portnummer explizit angeben.

Indem wir stattdessen den SSH-Dienst des Hosts auf Port 23 verschieben (zum Beispiel durch Anpassen von `/etc/ssh/sshd_config`) und Forgejo direkt auf Port 22 erreichbar machen, können wir uns diese zusätzliche Angabe sparen. SSH-Zugriffe auf Repositories funktionieren dann wie gewohnt:

```
$ git clone git@git.example.eu:gruppe/projekt.git
```

Anstatt:

```
git clone ssh://git@git.example.eu:23/gruppe/projekt.git
```

Dieser Schritt ist optional. Er kann jedoch den Arbeitsalltag deutlich vereinfachen, insbesondere wenn Forgejo von mehreren Personen genutzt wird oder SSH als primäres Zugriffsprotokoll dient.

Um die Sicherheit zusätzlich zu erhöhen – insbesondere, wenn Forgejo öffentlich zugänglich ist – empfiehlt es sich außerdem, ein Tool wie Fail2ban *[20]* einzurichten. Es erkennt und blockiert automatisiert wiederholte fehlgeschlagene Login-Versuche über SSH oder Web und schützt so zuverlässig vor Brute-Force-Angriffen.

Fazit

Mit Forgejo haben wir eine moderne, schlanke und vollständig freie Git-Plattform auf eigener Infrastruktur in Betrieb genommen, inklusive Weboberfläche, Datenbank, TLS-Absicherung und eigenem CI-Runner. Damit steht uns eine leistungsfähige Umgebung zur Verfügung, um Softwareprojekte selbstbestimmt und unabhängig zu verwalten.

Doch damit ist das Potenzial von Forgejo noch längst nicht ausgeschöpft. Neben dem klassischen Einsatz als zentrale Entwicklungsplattform eignet sich Forgejo auch hervorragend als Spiegel-Instanz für Repositories, die zum Beispiel auf GitHub oder anderen Plattformen liegen. So lassen sich Projekte dauerhaft sichern, Versionen nachvollziehbar archivieren und Abhängigkeiten von Drittanbietern reduzieren. Dabei kann wahlweise die eigene Instanz oder ein externer Anbieter als primäre Quelle (Upstream) dienen, Forgejo lässt sich flexibel konfigurieren.

Ein weiteres spannendes Feld ist die Föderierung: Die Forgejo-Community arbeitet aktiv an der Integration des ActivityPub-Protokolls, das bereits in vielen Fediverse-Diensten wie Mastodon oder Lemmy zum Einsatz kommt. Ziel ist es, dass sich Forgejo-Instanzen künftig über Instanzgrenzen hinweg vernetzen können, etwa um Stars, Forks oder Pull Requests auszutauschen. Das eröffnet völlig neue Möglichkeiten für die dezentrale Zusammenarbeit.

Ob als Plattform für eigene Projekte, als Mirror für kritische Repositories oder als Teil eines föderierten Entwickler-Netzwerks: Forgejo bietet uns alle Werkzeuge, um unsere Software nachhaltig, transparent und in unserer eigenen Verantwortung zu entwickeln und zu betreiben.

Der Code zum Mitnehmen

Alle in diesem Artikel gezeigten Konfigurationsdateien und Befehle findest du auch online auf meiner eigenen Forgejo-Instanz *[21]*. Dort kannst du sie bequem kopieren, anpassen und selbst ausprobieren. Das zugehörige Repository ist unter folgender Adresse erreichbar:

https://go.fihlon.swiss/2504-forgejo
Viel Spaß beim Ausprobieren und Selbst-Hosten!

Abbildung 3: QR-Code zum Repository

Referenzen

[1] Forgejo: https://forgejo.org
[2] GitLab: https://gitlab.com/
[3] Go: https://go.dev/
[4] PostgreSQL: https://www.postgresql.org
[5] Redis: https://redis.io
[6] Sidekiq: https://sidekiq.org
[7] Gitea: https://gitea.io
[8] ActivityPub-Protokoll: https://www.w3.org/TR/activitypub/
[9] Mastodon: https://joinmastodon.org/
[10] PeerTube: https://joinpeertube.org/
[11] Pixelfed: https://pixelfed.org/
[12] Hetzner Cloud: https://www.hetzner.com/cloud/
[13] Docker: https://www.docker.com/
[14] Podman: https://podman.io/
[15] Traefik: https://traefik.io
[16] Let's Encrypt: https://letsencrypt.org
[17] SQLite: https://www.sqlite.org
[18] MariaDB: https://mariadb.org
[19] MySQL: https://www.mysql.com/
[20] Fail2ban: https://www.fail2ban.org/
[21] Code zum Artikel: https://go.fihlon.swiss/2504-forgejo

Marcus Fihlon
marcus@fihlon.swiss

Marcus Fihlon arbeitet als Scrum Master und Agile Coach in der Schweiz. Er ist überzeugter Verfechter freier Software, nutzt seit über 30 Jahren Linux und programmiert fast genauso lange in Java. Als Vorstandsmitglied der Java User Group Switzerland und des iJUG engagiert er sich aktiv für die deutschsprachige Java-Community. Marcus organisiert Konferenzen und Community-Events für Software-Entwickler und gibt dabei selbst regelmäßig Vorträge und Workshops. In seiner Freizeit entwickelt er Open-Source-Software und erkundet auf seinem Fahrrad die Welt, gerne auch mal abseits befestigter Wege.

MITMACHEN UND AUTORIN ODER AUTOR WERDEN!

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

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

Wir freuen uns, von dir zu hören!

Weitere Informationen

Eine Reise durch Zeit und Kalender für Softwareentwickler

Michael Krimgen, Postcode Loterij

Die Arbeit mit Datum und Uhrzeit stellt für uns Softwareentwickler häufig eine Herausforderung dar. In diesem Artikel begeben wir uns auf eine kurze Reise durch die Geschichte des Kalenders und der Messung der Zeit.

Wir beginnen mit den historischen Entwicklungen rund um den Kalender, den Zeitzonen und der Sommerzeit. Anschließend werfen wir einen Blick auf aktuelle Entwicklungen und beleuchten abschließend einige interessante Aspekte der Arbeit mit Datum und Zeit in Java.

Das Ziel des vorliegenden Artikels ist es nicht, Lösungen für spezielle Probleme zu erläutern, sondern ein allgemeines Verständnis für die Arbeit mit Zeit und Datum zu schaffen, um den Softwareentwickler auf die Schwierigkeiten aufmerksam zu machen.

Warum ist die Arbeit mit Datum und Zeit kompliziert?

Viele unterschiedliche Datums- und Zeitformate, Zeitzonen sowie kulturelle und politische Aspekte machen Zeitangaben oft kompliziert. Im Deutschen sagen wir „halb vier", während es im Englischen „half past four" oder „half four" heißt. Vor allem die zweite Variante kann für Verwirrung sorgen.

Selbst innerhalb Deutschlands gibt es zwei Arten, die Uhrzeit anzugeben. So sagt man in einigen Regionen „Viertel vor drei" und „Viertel nach drei", in anderen Regionen jedoch „dreiviertel drei" und „viertel vier".

Jedes Jahr wird wieder aufs Neue gerätselt, ob die Uhr bei der Umstellung auf die Sommer- beziehungsweise Winterzeit nun eine Stunde vor- oder zurückgestellt wird.

Darüber hinaus existieren 37 Zeitzonen. Die Monate in unserem Kalender haben eine unterschiedliche Anzahl an Tagen, es gibt Schalttage und sogar Schaltsekunden. Darüber hinaus gibt es unzählige Möglichkeiten, Datums- und Zeitangaben zu speichern, etwa als Zeichenkette oder als Zahl (mit 32- oder 64-Bit).

Fast jeder kennt wahrscheinlich das Problem, ob mit „11/05/2025" nun der 11. Mai oder der 5. November 2025 gemeint ist.

Kein Wunder also, dass es unzählige Softwarefehler gibt, die auf Probleme bei der Zeit- und Datumsverarbeitung beruhen.

Softwarefehler aufgrund von Fehlern mit Datum und Zeit

Auf Wikipedia findet sich eine große Anzahl an Softwarefehlern, die auf Probleme in der Handhabung von Zeit und Datum zurückzuführen sind *[1]*. Hier eine kurze Auswahl:

Die meisten Leser haben wahrscheinlich vom sogenannten Jahr-2000-Problem gehört. Jahreszahlen vor dem Jahr 2000 wurden häufig nur als zweistellige Zahl gespeichert, wodurch Datumsangaben ab dem Jahr 2000 nicht von Jahresangaben ab 1900 zu unterscheiden waren (beispielsweise wurde das Jahr 2002 als „02" abgespeichert, was von vielen Systemen als „1902" verstanden wurde).

Im Jahre 2012 verursachte die Schaltsekunde Probleme an Flughäfen *[2]* und bei Linux-Distributionen *[3]*.

2024 schickte das Finanzamt in Schleswig-Holstein 1.700 identische Briefe an einen Einwohner, der in der Nacht der Umstellung von Sommer- auf Winterzeit ein Formular auf der Website der Behörde ausgefüllt hatte. Das Finanzamt entschuldigte sich daraufhin und gab als Grund einen Softwarefehler im Zusammenhang mit der Zeitumstellung an *[4]*.

Unser Kalender – der gregorianische Kalender

Im Folgenden wollen wir die Geschichte und Eigenarten unseres Kalenders kurz anhand einer Zeitleiste *(siehe Abbildung 1)* betrachten.

Im Jahre 45 vor Christus wurde der julianische Kalender von Julius Cäsar eingeführt. Das Jahr wurde auf 365,25 Tage festgelegt, wobei jedes vierte (durch vier teilbare) Jahr ein Schaltjahr wurde. Der Februar hatte dann 29 statt 28 Tage.

Da ein Sonnenjahr nicht genau 365,25 Tagen entspricht, ordnete Papst Gregor XIII. im Jahr 1582 eine Korrektur an, ließ 10 Tage aus-

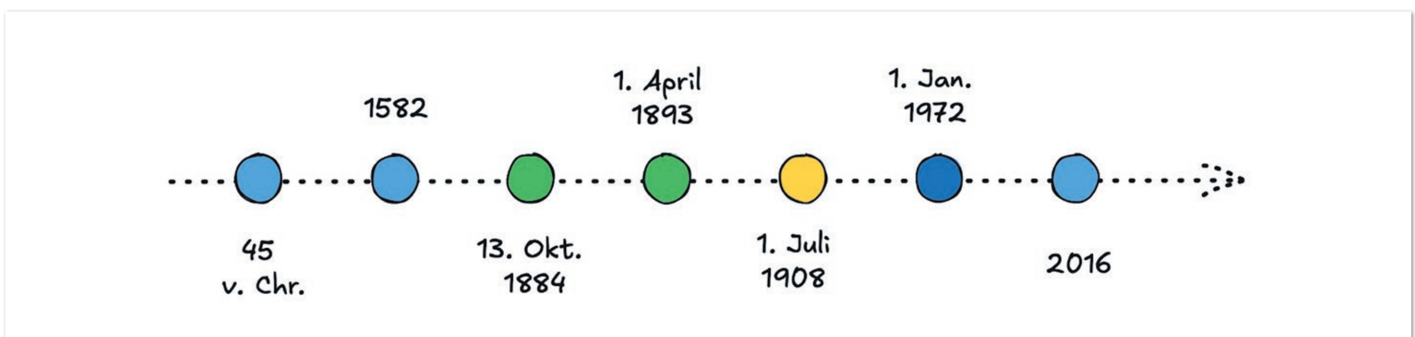


Abbildung 1: Zeitleiste unseres Kalenders (© Michael Krimgen)

Abbildung 2: Zeitzonen weltweit (© Wikipedia [6]).

fallen (auf den 4. Oktober 1582 folgte unmittelbar der 15. Oktober) und legte die Schalttage neu fest. Seither benutzen wir (in Europa und großen Teilen der Welt) den gregorianischen Kalender:

- Ein Jahr hat 365,2422 Tage.
- Ein Jahr ist ein Schaltjahr, wenn es durch vier, aber nicht durch 100 teilbar ist.
- Ausnahme: durch 400 teilbare Jahre sind ebenso Schaltjahre.

Zur Verdeutlichung hier ein paar Beispiele:

- 1980, 1984 und 2004 sind Schaltjahre (da durch vier teilbar).
- 1800, 1900 und 2100 sind keine Schaltjahre (da durch 100 teilbar).
- 2000 und 2400 sind Schaltjahre (da durch 400 teilbar).

Die ausgefallenen Tage mögen aus heutiger Sicht unbedeutend erscheinen, dennoch ist bei historischen Datumsangaben Vorsicht geboten. Oft wird der 23. April 1616 als Sterbedatum sowohl des spanischen Dichters Miguel Cervantes als auch des englischen Dichters William Shakespeares angegeben. Da Spanien 1616 schon den gregorianischen Kalender eingeführt hatte und England noch den julianischen Kalender verwendete, starb Shakespeare nicht am selben Tage wie Cervantes, sondern zehn Tage später.

Ähnliche Missverständnisse kann es bei historischen Zeitangaben geben, wenn unterschiedliche Termine für die Umstellung von Sommer- auf Winterzeit nicht korrekt berücksichtigt werden. Hierzu später mehr.

Universal Time Coordinated (UTC), Sommerzeit und Schaltsekunden

Nach einer kurzen Geschichte unseres Kalenders sehen wir uns nun ein paar wichtige Eckdaten zur Zeitmessung an.
Auf der Washingtoner Meridiankonferenz wurde am 13. Oktober 1884 die Greenwich Meantime (GMT) als Weltzeit eingeführt und am 1. April 1893 die Mitteleuropäische Zeit (MEZ) im Deutschen Reich. Hauptbeweggrund für die Vereinheitlichung der Zeit im ganzen Land war die Ausbreitung der Eisenbahn. Bis dato galten an unterschiedlichen Orten in den beiden Ländern verschiedene Zeiten, was die Erstellung eines Fahrplanes erschwerte. Am 1. Juli 1908 führte die Provinz Ontario als erstes die Sommerzeit ein.

Am 1. Januar 1972 folgte die Einführung der UTC (Universal Time Coordinated) sowie die Einführung der Schaltsekunde (in unregelmäßigen Abständen). Die letzte Schaltsekunde im Jahr 2016 war möglicherweise auch die letzte *[5]*.

Die Zeitzonen

Eine weitere Komplexität bei der Arbeit mit Zeitangaben stellen die vielen unterschiedlichen Zeitzonen dar. *Abbildung 2* zeigt eine Karte aller Zeitzonen.

Da eine Zeitzone im Durchschnitt 15 Grad umfasst, würde man 24 Zeitzonen vermuten. Tatsächlich gibt es zum aktuellen Stand 37 verschiedene Zeitzonen, die (vor allem aus politischen und praktischen Gründen) nicht überall vertikal verlaufen.

Die Zeitzonen verlaufen nicht nur nicht streng vertikal, sondern begrenzen auch nicht zwangsweise volle Stunden. Beispiele hierfür sind die Zeitzonen des Iran, Pakistans, Indiens und Myanmars *(siehe Abbildung 3)*. In Australien gibt es ebenfalls „halbe" Zeitzonen.

Eine weitere Besonderheit stellt China dar. Hier gibt es nur eine Zeitzone, die China Standard Time (CST) obwohl das Land sich eigentlich über zirka fünf Zeitzonen erstreckt *(siehe Abbildung 4)*.

Auch in Deutschland gab es früher einige Kuriositäten. Rund um den Bodensee gab es im 19. Jahrhundert fünf verschiedene Zeitzonen *[7]*. Reisende mussten bei Fahrten deshalb mehrmals die Uhr um-

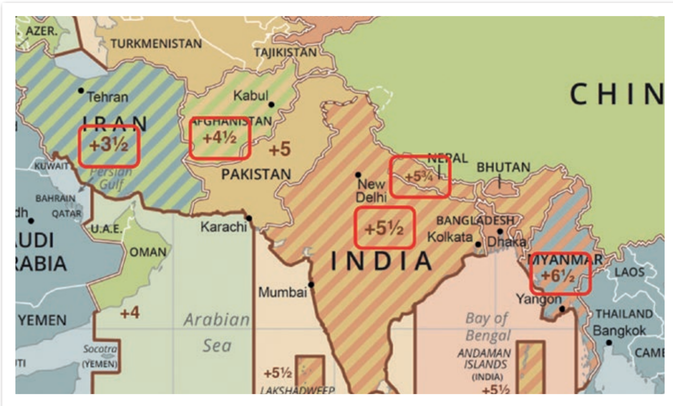


Abbildung 3: Zeitzonen – Beispiel für unterschiedliche Grenzen (© Wikipedia [6]).

Abbildung 4: Zeitzonen – Beispiel China (© Wikipedia [6]).

stellen *(siehe Abbildung 5)*.

Abbildung 5: Zeitzonen am Bodensee im 19. Jahrhundert (© Wikipedia [7])

Sommer- und Winterzeit

In Deutschland wurde die mitteleuropäische Sommerzeit im Jahr 1916 eingeführt. Ab 1919 wurde diese jedoch wieder verworfen. Im zweiten Weltkrieg wurden die Termine für die Zeitumstellung weitgehend willkürlich festgelegt. Nach dem Krieg gab es keine feste Regelung von den Alliierten. Ab 1950 galt dann wieder nur die Winterzeit.

Schließlich wurde 1980 wieder die Sommerzeit eingeführt und 1996 der Termin für die Zeitumstellung auf EU-Ebene vereinheitlicht. Immer wieder gibt es Diskussionen, die Zeitumstellung wieder abzuschaffen.

Wir werden sehen, was die Zukunft bringt. Eine diesbezügliche Änderung wird sicherlich einiges an Arbeit für die Softwareentwicklung mit sich bringen. *Abbildung 6* zeigt eine Zeitleiste mit den Eckdaten der Änderungen bezüglich der Sommerzeit in Deutschland.

Auch in anderen Ländern gab es im Lauf der Jahre Anpassungen bei der Zeitumstellungen. In Frankreich wurde die Sommerzeit 1916 eingeführt und 1945 wieder abgeschafft. 1976 dann wieder „vorrübergehend" eingeführt und 1996 der EU-Regelung angepasst *(siehe Abbildung 7)*.

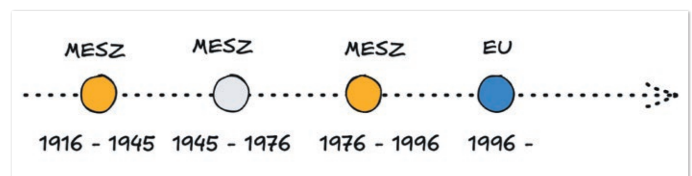


Abbildung 7: Sommerzeit in Frankreich (© Michael Krimgen)

Ein weiterer interessanter Fall ist Brasilien. Hier wurde die Sommerzeit 1931 eingeführt und 1988 in den nördlichen (also dem Äquator näheren) Bundesstaaten wieder abgeschafft. 2017 wurde der Termin der Zeitumstellung verschoben, da er mit dem Termin der Wahlen kollidierte. 2019 wurde die Sommerzeit schließlich im ganzen Land wieder abgeschafft *(siehe Abbildung 8)*.

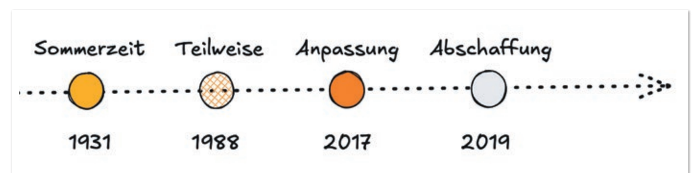


Abbildung 8: Sommerzeit in Brasilien (© Michael Krimgen)

Änderungen der Zeitzonen sind häufiger als man vermuten würde. Das lässt sich auf der Seite „Time Zone News" *[8]* nachlesen, die alle Veränderungen dokumentiert (*Abbildung 9* zeigt einige Beispiele).

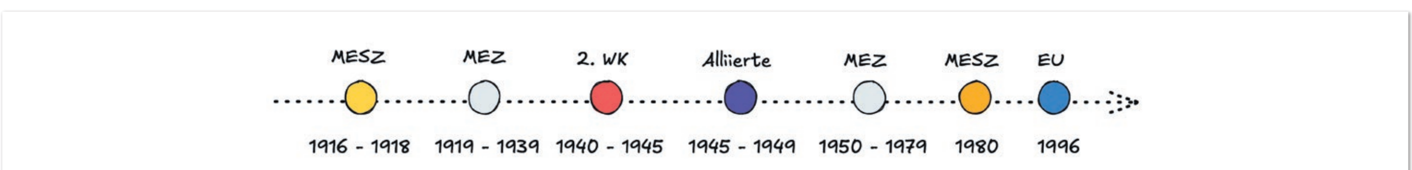


Abbildung 6: Sommerzeit in Deutschland (© Michael Krimgen)

2024-02-07 Palestine stays on standard time one week longer

2024-01-20 Kazakhstan goes from two time zones to one

2023-03-27 Lebanon reverses DST postponement

2023-03-23 Lebanon postpones DST on short notice

2023-03-22 Palestine postpones DST until after Ramadan

2023-03-19 Morocco will return to UTC +1 earlier than expected

2023-03-02 Egypt brings back DST

2022-11-27 Time zone change in Greenland

2022-10-28 Changes in Mexico time zones

Abbildung 9: Time Zone News (© Time Zone News [8]).

Datum und Zeit in Java

Dem Java-Entwickler stellt sich nun vermutlich die Frage, wie die Java-Time-API mit all diesen Anpassungen zurechtkommt. Beginnen wir mit einem Beispiel. *Listing 1* zeigt den Code, der die Zeit in Samoa, einer Insel im Pazifischen Ozean, an vier aufeinander folgenden Tagen Ende Dezember 2011 ausgibt. Im Vergleich geben wir auch noch die Zeit in UTC aus.

Wir sehen, dass auf den 29. Dezember 2011 unmittelbar der 31. Dezember folgt. Der Grund hierfür ist, dass Samoa Ende 2011 die Zeitzone gewechselt hat. Die Java-API kommt also auch mit dieser recht speziellen Situation zurecht.

Sehen wir uns nun einmal an, woher Java die Zeitzoneninformation bekommt. Ein Blick in die Klasse *TzdbZoneRulesProvider* gibt hierüber Aufschluss *(siehe Listing 3)*.

Die Datei *tzdb.dat* befindet sich in der Regel im `lib`-Verzeichnis der jeweiligen Java-Distribution. Diese Datenbank beinhaltet Information zu Änderungen der Zeitzonen ab 1970. Alle Einträge für Änderungen vor 1970 sind möglicherweise unvollständig, da es vielerorts keine einheitlichen Regeln gab. *Listing 4* zeigt einen Eintrag in der Datenbank.

Die Änderungen, die wir schon kennengelernt haben, sind leicht zu erkennen. Diese Datenbank liest Java nun ein und berechnet daraus unter Berücksichtigung der Zeitzonen die Zeit für vergangene und zukünftige Daten.

Besonderheiten der Java API

Abschließend sehen wir uns noch ein paar Eigenschaften der Java-Time-API an, die oftmals zu Verwirrungen führen können.

Im Beispiel in *Listing 5* subtrahieren wir zunächst zweimal einen Monat und zum Vergleich zwei Monate.

```
@Test
void Zeit_Samoa() {
    Instant utc = Instant.parse("2011-12-29T18:00:00.00Z");
    ZonedDateTime samoa = utc.atZone(ZoneId.of("Pacific/Apia"));
    System.out.println("UTC:         " + utc);
    System.out.println("Samoa      : " + samoa);
    System.out.println("Samoa (+1): " + samoa.plusDays(1));
    System.out.println("Samoa (+2): " + samoa.plusDays(2));
    System.out.println("Samoa (+3): " + samoa.plusDays(3));
}
```

Listing 1: Die Uhrzeit in Samoa Ende Dezember 2011.

```
UTC:        2011-12-29T18:00:00Z
Samoa     : 2011-12-29T08:00-10:00[Pacific/Apia]
Samoa (+1): 2011-12-31T08:00+14:00[Pacific/Apia]
Samoa (+2): 2011-12-31T08:00+14:00[Pacific/Apia]
Samoa (+3): 2012-01-01T08:00+14:00[Pacific/Apia]
```

Listing 2: Die Ausgabe des Codes in Listing 1.

```
public TzdbZoneRulesProvider() {
     try {
          String libDir = StaticProperty.javaHome() + File.separator + "lib";
          try (DataInputStream dis = new DataInputStream(
                      new BufferedInputStream(new FileInputStream(
                                  new File(libDir, "tzdb.dat"))))) {
                load(dis);
          }
     } catch (Exception ex) {
          throw new ZoneRulesException("Unable to load TZDB time-zone rules", ex);
     }
}
```

Listing 3: Der ZoneRulesProvider in Java

Europe

The following rules are for the European Union and for its
predecessor organization, the European Communities.
For brevity they are called "EU rules" elsewhere in this file.

Rule   NAME   FROM   TO -   IN ON AT SAVE   LETTER/S
Rule EU 1977   1980   - Apr Sun>=1 1:00u 1:00   S
Rule EU 1977   only   - Sep lastSun   1:00u 0   -
Rule EU 1978   only   - Oct 1   1:00u 0   -
Rule EU 1979   1995   - Sep lastSun   1:00u 0   -
Rule EU 1981   max - Mar lastSun   1:00u 1:00   S
Rule EU 1996   max - Oct lastSun   1:00u 0   -
```

*Listing 4: Der ZoneRulesProvider in Java.*

```
@Test
void Java_Time_Fluent_Minus_Two_Month() {
     LocalDate date = LocalDate.of(2025, 7, 31);
     System.out.println("- 1 Monat - 1 Monat = " + date.minusMonths(1).minusMonths(1));
     System.out.println("- 2 Monate = " + date.minusMonths(2));
}
```

Listing 5: Subtrahieren von Monaten.

Listing 6 zeigt das Ergebnis, das wahrscheinlich (je nach Anwendungsfall) nicht wie erwartet ausfällt.

Ein ähnliches Ergebnis erhalten wir, wenn wir die Addition eines Monates und eines Tages umgekehrt ausführen. *Listing 7* zeigt den Code und *Listing 8* die entsprechende Ausgabe.

```
- 1 Monat - 1 Monat = 2025-05-30
- 2 Monate = 2025-05-31
```

Listing 6: Ausgabe des Codes in Listing 5.

```
@Test
void Java_Time_Fluent_Month_Day() {
     LocalDate date = LocalDate.of(2025, 6, 30);
     System.out.println("+ 1 Monat + 1 Tag = " +  date.plusMonths(1).plusDays(1));
     System.out.println("+ 1 Tag + 1 Monat = " + date.plusDays(1).plusMonths(1));
}
```

Listing 7: Addition eines Monats und eines Tages und umgekehrt.

Listing 9 zeigt, dass Vorsicht bei Daten vor dem 15. Oktober 1583 geboten ist.

Die Ausgabe in *Listing 10* zeigt einen Tag, den es im gregorianischen Kalender nicht gegeben hat.

```
+ 1 Monat + 1 Tag = 2025-07-31
+ 1 Tag + 1 Monat = 2025-08-01
```

Listing 8: Ausgabe des Codes in Listing 7.

```
@Test
void Java_Time_Gregorian() {
     LocalDate date = LocalDate.of(1582, 10, 4);
     System.out.println("Den Tag hat es eigentlich nicht gegeben: " + date.plusDays(2));
}
```

Listing 9: Der nicht existierende Tag im gregorianischen Kalender.

Listing 11 zeigt, dass auch Timestamps mit Schaltsekunden korrekt geparst werden.

Listing 12 zeigt allerdings, dass Schaltsekunden nicht standardmä-

```
Den Tag hat es eigentlich nicht gegeben: 1582-10-06
```

Listing 10: Ausgabe des Codes in Listing 9.
```

```
@Test
void Schaltsekunden() {
     System.out.println(Instant.parse("1982-06-30T23:59:60.00Z"));
     System.out.println(Instant.parse("2024-06-30T23:59:60.00Z"));
}
```

Listing 11: Timestamp mit Schaltsekunden.

ßig ausgegeben werden. Beim Versuch einen Timestamp mit einer ungültigen Schaltsekunde einzulesen *(siehe Listing 13)* wirft die API eine Exception *(siehe Listing 14)*.

Zusammenfassung

In diesem Artikel haben wir eine kurze Geschichte unseres Kalenders kennengelernt. Wir haben gesehen, welche Schwierigkeiten bei der Arbeit mit Zeit und Datum auftreten können. Abschließend hoffe ich, dem Leser genug Informationen mit auf den Weg gegeben zu haben, um den ein oder anderen Bug zu vermeiden.

```
1982-06-30T23:59:59Z
2024-06-30T23:59:59Z
```

Listing 12: Ausgabe des Codes in Listing 11.

```
@Test
void Schaltsekunden_Nicht_Mitternacht() {
     Instant.parse("2025-04-01T06:59:60.00Z");
}
```

Listing 13: Timestamp mit ungültiger Schaltsekunde

Quellen

[1] *https://en.wikipedia.org/wiki/Time_formatting_and_storage_bugs*
[2] *https://www.theguardian.com/technology/2012/jul/02/leap-second-amadeus-qantas-reddit*
[3] *https://www.heise.de/news/Schaltsekunden-Bug-in-Linux-verschwendet-Strom-1631325.html*
[4] *https://www.ndr.de/nachrichten/schleswig-holstein/1700-Briefe-vom-Amt-Finanzministerin-entschuldigt-sich-,finanzamt212.html*
[5] *https://de.wikipedia.org/wiki/Schaltsekunde*
[6] *https://de.wikipedia.org/wiki/Zeitzone*
[7] *https://de.wikipedia.org/wiki/Fünf_Standardzeiten_am_Bodensee*
[8] *https://time.is/time_zone_news*
[9] Time database: *https://www.iana.org/time-zones*


```
java.time.format.DateTimeParseException: Text
'2025-04-01T06:59:60.00Z' could not be parsed at index 0
```

Listing 14: Ausgabe des Codes in Listing 11.

Michael Krimgen
software.michael@gmx.de

Michael arbeitet seit 2007 als Softwareentwickler, hauptsächlich mit Java. Seit fast drei Jahren ist er bei der Nationalen Postcode Lotterie (Postcode Loterij) in Amsterdam tätig. Seine Interessen liegen vor allem im Clean Coding und Core Java. Seit einigen Jahren spricht er zudem auf Konferenzen zum Thema Java. In seiner Freizeit klettert er gerne, läuft Marathon, fährt Fahrrad, liest und reist gerne und erweitert seine Sprachkenntnisse.

Greenify your System – Nachhaltigkeit systematisch denken

Sophia Resch, pentacor GmbH

Nachhaltige IT beginnt nicht erst beim nächsten Projekt auf der grünen Wiese – sondern im Hier und Jetzt. Dieser Artikel zeigt, wie Softwareentwicklung systematisch nachhaltiger gestaltet werden kann: durch gezielte Verbesserungen in Bestandssystemen, bewusste Architekturentscheidungen in neuen Anwendungen und strukturelle Verankerung im Unternehmen. Technisch fundiert, praxisnah und mit vielen konkreten Ansatzpunkten für den Alltag von Entwicklungs- und Architekturteams.

Der Klimawandel ist keine Zukunftsvision mehr, sondern spürbare Realität: mit Hitzerekorden, Wasserknappheit und globalem Artensterben. Gleichzeitig steigen die gesellschaftlichen Anforderungen an Unternehmen, Verantwortung zu übernehmen. Nachhaltigkeit wird vermehrt auch zum Wettbewerbsfaktor – getrieben durch neue regulatorische Anforderungen (wie die CSRD [1]), zunehmenden wirtschaftlichen Druck und sich ändernde Kundenansprüche.

Die IT ist dabei ein zentraler Hebel. Sie ist Teil des Problems, weil sie Energie verbraucht und Ressourcen bindet. Aber sie ist auch Teil der Lösung – weil sie Prozesse optimieren, Emissionen einsparen und neue, nachhaltige Lösungen ermöglichen kann. Entscheidend ist, wie wir Systeme bauen, betreiben und weiterentwickeln.

Bestandssysteme: Die unterschätzte Chance

Nachhaltigkeit beginnt dort, wo unsere Systeme bereits laufen – nicht erst beim nächsten Projekt auf der grünen Wiese. Bestehende Anwendungen bergen enorme Potenziale, gerade weil sie häufig über Jahre hinweg betrieben und erweitert werden. Wer hier ansetzt, kann mit überschaubarem Aufwand spürbare Wirkung erzielen. Dabei geht es nicht nur um die Reduktion von CO_2, sondern auch um Einsparungen bei Kosten, Komplexität und Wartungsaufwand.

Implementierung oder Architektur? Zwei sich ergänzende Hebel

Wer Bestandssysteme nachhaltiger gestalten will, steht oft vor der Frage: Wo fangen wir an? Prinzipiell sind zwei übergeordnete Ansätze denkbar – die Überarbeitung der Implementierung und die Überarbeitung der Architektur.

Implementierungsnahe Maßnahmen bieten einen pragmatischen Einstieg. Änderungen lassen sich meist auf Ebene einzelner Komponenten oder Funktionen durchführen, sind gut in bestehende Entwicklungsprozesse integrierbar und erfordern keine umfassende Umstrukturierung. Typische Beispiele können sein:

- gezielte Refactorings zur Reduktion redundanter Verarbeitung
- effizientere Datenabfragen und optimierte Caching-Strategien
- Nutzung leichtgewichtigerer Bibliotheken
- Entfernung ungenutzter Features
- Implementierung ressourcenschonenderer Algorithmen

Diese Maßnahmen folgen oft dem Prinzip des „Pfadfinderns“: Jede Änderung hinterlässt den Code ein Stück besser, als man ihn vorgefunden hat.

Architekturveränderungen greifen tiefer und betreffen strukturelle Eigenschaften des Systems. Hier ist das Potenzial für langfristige Einsparungen groß, der dafür benötigte Aufwand aber auch. Beispiele können sein:

- Umstellung auf dynamisches Autoscaling
- Einführung von Event-getriebener Verarbeitung statt periodischer Polling-Jobs
- Schneiden von Services anhand ihres Ressourcenaufwandes statt nach fachlichen Kriterien

Beide Ebenen greifen ineinander. Architekturänderungen wirken sich auf Implementierung und Betrieb aus – und umgekehrt. Der jeweils passende Ansatz ist meist eine Mischung aus beiden Herangehensweisen und hängt immer stark von der Systemlandschaft, den Abhängigkeiten und den betrieblichen Gegebenheiten ab.

Green Software Patterns gezielt einsetzen

Zur konkreten Umsetzung nachhaltiger Maßnahmen helfen die Green Software Patterns [2] – eine wachsende Sammlung erprobter Prinzipien, Methoden und Umsetzungsideen –, die durch die Green Software Foundation zur Verfügung gestellt werden [3].

Im Folgenden werden deshalb beispielhaft einige Gruppen von Green Software Patterns vorgestellt – ohne Anspruch auf Vollständigkeit, aber mit dem Ziel, einen Überblick zu geben und zum Nachdenken (und Nachbauen) anzuregen:

- Infrastruktur:
  - Autoscaling und Scale to Zero: Ressourcen nur dann bereitstellen, wenn sie benötigt werden.
  - Demand Shifting: Ausführung energieintensiver Tasks in Zeiten mit höherem Anteil erneuerbarer Energien.
  - CI/CD-Optimierung: Build- und Testprozesse gezielt ressourcenschonend gestalten.
- Datenverarbeitung:
  - Retention Policies: Daten nur so lange speichern, wie nötig.
  - Abfrageoptimierung: Reduktion von Overhead in Datenbankzugriffen.
  - Kompression: Verringerung des Datenvolumens bei Übertragung und Speicherung.
- Systemlogik:
  - Serverless: Kurzlebige Dienste bei Bedarf aktivieren, Ressourcen im Leerlauf vermeiden.
  - Push statt Polling: Netzwerk- und Rechenlast durch eventbasierte Kommunikation senken.
  - Circuit Breaker: Endlosschleifen und ineffiziente Fehlerwiederholungen vermeiden.
- Clientverhalten:
  - Abwärtskompatibilität: Unterstützung älterer Endgeräte verlängert deren Nutzungsdauer und hat möglicherweise eine große Tragweite.
  - Server-Side Rendering: Rechenlast auf den Server verlagern, auch um ältere Endgeräte länger zu unterstützen.

Diese Muster sind bewusst als Werkzeugkasten konzipiert – kein Dogma, sondern ein flexibles Set an Ideen, aus dem sich projekt- und systemspezifisch passende Ansätze ableiten lassen. Sie lassen sich zumeist inkrementell einführen – ideal für den kontinuierlichen Verbesserungsprozess bestehender Anwendungen.

Weiterhin ist zu beachten: Maßnahmen wie Demand-Shifting oder die Verschiebung von CI-Runnern können in vielen Fällen wirksam sein – aber sie müssen verantwortungsvoll eingesetzt werden: Wenn zu viele gleichzeitig dieselben „grünen Regionen" oder Zeitfenster nutzen, steigt der lokale Energiebedarf dort sprunghaft an – und der Vorteil verpufft. Nachhaltigkeit entsteht nicht durch Verlagerung allein, sondern auch durch bewusste Steuerung.

Nachhaltigkeit messbar machen

Nachhaltigkeit in der Softwareentwicklung lässt sich nur verbessern, wenn ihre Auswirkungen auch überwacht werden. Ein erster, pragmatischer Schritt ist dabei die Nutzung vorhandener Metriken – auch wenn diese oft nur Proxys für tatsächliche Umwelteinflüsse sind. Der Vorteil bei Green IT: Alles, was ein System effizienter, performanter oder günstiger macht, ist meist auch nachhaltiger. In vielen Fällen lohnt es sich daher, bestehende Kennzahlen im Betrieb bewusster zu beobachten:

- Anzahl der laufenden Pods pro Service (sofern Funktions- und Nutzungsspektrum konstant bleiben): Eine steigende Anzahl kann auf sinkende Effizienz hindeuten. Gleichbleibende oder sinkende Werte sind hier meist ein gutes Zeichen.
- Laufzeitkosten einzelner Services (ohne Personalkosten): Weniger Betriebskosten bedeuten oft auch weniger Energieverbrauch.
- Deploy-Zeit pro Service: Weniger Ressourcenverbrauch in der CI/CD-Pipeline kann ebenso zur Gesamtbilanz beitragen.
- Cloud-spezifische Nachhaltigkeitsreports: Viele Provider stellen mittlerweile – oft über das Abrechnungsportal – CO_2-Reports zur Verfügung. Der Zugang liegt allerdings häufig in der Finanzabteilung, nicht beim Entwicklungsteam.

Ein sinnvoller nächster Schritt ist die Verwendung eines geeigneten Werkzeugs – je nach Betrachtungsebene:

- Service-Ebene (feingranular): Hier lassen sich Tools wie ML CO2 Impact *[4]*, CodeCarbon *[5]* oder PolarSignals *[6]* einsetzen. Sie erfassen beispielsweise CPU-Auslastung, Speicherverbrauch oder Rechenzeit und erlauben Rückschlüsse auf den Energieverbrauch einzelner Komponenten, teilweise bis auf Methodenebene heruntergebrochen.
- System-Ebene (grobgranular): Für eine gesamthafte Betrachtung eignen sich das Carbon Footprint Modelling Tool *[7]*, Cloud Carbon Footprint *[8]* oder das Impact Framework *[9]*. Letzteres ist aktuell noch recht theoretisch und mit gewissem Abstand zur betrieblichen Realität – wie bisherige Anwendungserfahrung zeigt.

Wichtig ist: Kein Tool liefert allein „die Wahrheit". Aussagekraft entsteht durch Kombination, Kontext und Verlauf – etwa durch regelmäßige Vergleiche über Zeit oder Gegenüberstellung unterschiedlicher Systemteile. So werden bewusste Entscheidungen datenbasiert unterstützt – und Fortschritte sowie Rückschritte nachvollziehbar dokumentiert.

Neue Systeme: Der strategische Gestaltungsspielraum

Wer neue Systeme konzipiert, hat die Chance, Nachhaltigkeit von Anfang an mitzudenken – strukturiert, technisch fundiert und ohne Altlasten. Statt sie als Zusatzanforderung zu behandeln, sollte Nachhaltigkeit als grundlegendes Architekturprinzip etabliert werden. Dazu kann man sich beispielsweise am Software-Lifecycle orientieren.

Früh ansetzen: Planung und Requirements

Schon in der Planungsphase werden entscheidende Weichen gestellt. Nachhaltige Systeme entstehen nicht durch grüne Labels, sondern durch bewusste Entscheidungen.

Beispielhafte Fragen zur Bewertung:

- Gibt es bestehende Lösungen, die erweitert statt neu gebaut oder eingekauft werden können?
- Gibt es im System Zeiten geringer Auslastung – oder wird es rund um die Uhr genutzt?
- Welchen Mehrwert schafft das neue System tatsächlich?

Solche Überlegungen helfen, unnötige Energieverbräuche schon vor dem ersten Commit zu vermeiden – und stärken gleichzeitig die technische und wirtschaftliche Robustheit des Vorhabens.

Vorausschauend gestalten: Architektur als Hebel

Architektur ist der strategische Hebel für langfristige Wirkung. Wer hier vorausschauend plant, kann Ressourcenverbrauch massiv beeinflussen – ohne dabei auf Wartbarkeit, Sicherheit oder Skalierbarkeit verzichten zu müssen.

Bewährte Prinzipien:

- Redundanzen vermeiden: Doppelte Datenhaltung oder unnötige Schnittstellen erhöhen nicht nur Komplexität, sondern auch Energieverbrauch.
- Übertragene Daten minimieren: Schlanke APIs, gezieltes Caching und effiziente Formate reduzieren Netzlast und Speicherbedarf.
- Services nach Lastprofil schneiden: Stark belastete Komponenten sollten gezielt skalierbar sein – ressourcenschonende Funktionen hingegen möglichst leichtgewichtig.
- Modularisierung statt Monolith: Ermöglicht gezieltes Optimieren, Skalieren oder Abschalten einzelner Komponenten – ein klarer Vorteil für Green IT.

Auch die Perspektive der Nutzenden sollte einbezogen werden: Anwendungen, die auf älterer Hardware gut laufen, verlängern Lebenszyklen und reduzieren Elektroschrott. Server-Side Rendering oder adaptive Inhalte sind hier sinnvolle technische Mittel.

Effizient umsetzen: Nachhaltigkeit beim Entwickeln und Testen

In der Umsetzungs- und Testphase entscheidet sich, ob nachhaltige Architekturideen tatsächlich wirksam werden. Viele Stellschrauben sind hier technisch gut greifbar und gehen weit über die reine Implementierung hinaus.

Ein zentraler Hebel ist die effiziente Gestaltung von Build- und Testprozessen:

- CI/CD-Pipelines bewusst konfigurieren: Nicht jeder Commit muss automatisch einen Vollbuild oder vollständigen Testlauf auslösen. Selektive Builds, zielgerichtete Testauswahl und die Vermeidung unnötiger Artefakte sparen Zeit und Energie.
- Testumgebungen nur bei Bedarf provisionieren: Entwicklungs- und Testumgebungen müssen nicht dauerhaft laufen. Temporäre Instanzen mit automatisiertem Abbau helfen, Ressourcen zu schonen – besonders in großen Projekten mit vielen parallelen Umgebungen.
- Testanzahl und -laufzeit optimieren: Langsame oder überflüssige Tests verlängern Feedback-Zyklen, erhöhen Frustpotenzial und belasten Infrastruktur. Eine ausgewogene Teststrategie sorgt für Effizienz ohne Qualitätsverlust.
- Effiziente Algorithmen und Datenstrukturen: Auch im Zeitalter leistungsfähiger Hardware lohnt es sich, rechenintensive Prozesse zu hinterfragen – insbesondere bei häufig ausgeführtem Code.
- Nachhaltigkeit als Teil der Codequalität: Ressourcenschonung sollte kein Sonderziel sein, sondern integraler Bestandteil von Coding Guidelines, Reviews und Pairing-Sessions.
- Werkzeuge gezielt einsetzen: Tools wie Performance-Profiler, Impact-Metriken oder Green-Software-Linter helfen, Engpässe und Energieverschwendung sichtbar zu machen – und Optimierungen datenbasiert zu priorisieren.

Viele dieser Maßnahmen lassen sich bereits im Projektsetup verankern und fördern nicht nur Nachhaltigkeit, sondern auch Wartbarkeit, Stabilität und Teamzufriedenheit.

Langfristig denken: Nachhaltigkeit im Betrieb sichern

Im laufenden Betrieb entscheidet sich, wie effizient ein System tatsächlich ist. Während Architektur und Umsetzung die Grundlage schaffen, entfalten sich viele Umweltwirkungen erst über die Betriebszeit.

Wichtige Prinzipien für nachhaltige Betriebsmodelle sind:

- Ressourcenbedarf minimieren: Durch schlankes Design, intelligente Lastverteilung und gezielte Deaktivierung ungenutzter Komponenten lassen sich Infrastrukturkosten und Energieverbrauch senken.
- Hardwareauslastung optimieren: Bestehende Systeme sollten möglichst gut ausgelastet sein – etwa durch Konsolidierung von Diensten oder durch Nutzung ressourcensparender Betriebsmodelle wie Spot-Instanzen.
- Cloud-Ressourcen bewusst wählen: Der Standort des Rechenzentrums, die Speicherkategorie und die Verfügbarkeitsstrategie haben Einfluss auf den Energieverbrauch. Regionen mit hohem Anteil erneuerbarer Energien sind zu bevorzugen.
- Monitoring-Strategien erweitern: Neben klassischen Betriebsmetriken sollten auch Nachhaltigkeitsaspekte beobachtet werden – zum Beispiel durch Integration von CO_2-Dashboards, Strommix-Daten oder Green-Metrics.

Je früher Nachhaltigkeit mitgedacht wird, desto leichter lassen sich nachhaltige Betriebsweisen realisieren. Nachbesserungen im Betrieb sind oft aufwendiger und weniger wirkungsvoll als frühzeitige Weichenstellungen.

Organisation: Nachhaltigkeit ermöglichen und leben

Technische Maßnahmen allein reichen nicht aus. Nachhaltigkeit entfaltet erst dann ihr volles Potenzial, wenn sie auch strukturell und kulturell in der Organisation verankert ist. Dazu braucht es Bewusstsein, passende Prozesse und Raum für engagierte Menschen.

Bewusstsein schaffen

Oft beginnt Veränderung bei Einzelpersonen – beispielsweise eine Entwicklerin, die Nachhaltigkeit im Review anspricht. Ein Product Owner, der kritische Fragen zur Datennutzung stellt. Eine Kollegin, die im Stand-up den Stromverbrauch der Testumgebung erwähnt. Solche Impulse wirken, verbreiten sich und werden Teil des gemeinsamen Verständnisses.

Nachhaltigkeit im Alltag sichtbar machen

Nachhaltigkeit beginnt nicht in der Strategieabteilung, sondern im Arbeitsalltag. Es sind oft kleine, praktische Maßnahmen, die Aufmerksamkeit schaffen und Wirkung entfalten:

- Nachhaltige Tools etablieren: zum Beispiel Ecosia als ökologische Suchmaschine nutzbar machen.
- Hardwarelebensdauer verlängern: regelmäßige Laptop-Reinigung oder Austausch einzelner Komponenten statt kompletter Neuanschaffung.
- Sharing-Kultur fördern: interne Marktplätze für das Tauschen, Weitergeben oder Ausleihen von Geräten oder Materialien.
- Energie sparen im Büroalltag: Erinnerungen an Lichtschaltern oder Monitoren, Heizreglern und Stoßlüftung statt Dauerbetrieb.

Auch organisatorisch lassen sich Prozesse nachhaltiger gestalten:

- Nachhaltig beschaffen: Lokale Anbieter und zertifizierte Produkte bevorzugen.
- Reisen bewusst planen: Bahn statt Flugzeug, Hotels mit Nachhaltigkeitszertifikat, Videokonferenzen statt Vor-Ort-Termine.
- Nachhaltige Teamkultur stärken: vegetarische und vegane Essensoptionen bei Events, Unterstützung umweltfreundlicher Mobilität (zum Beispiel Jobticket, Jobrad).

Je öfter über Fortschritte gesprochen wird und umso normaler die grünere Entscheidungsoption wird, desto mehr verankert sich Nachhaltigkeit im Alltag – nicht als Sonderthema, sondern als gelebte Selbstverständlichkeit.

Nachhaltigkeit in Entwicklungsteams fördern

Gerade in Architektur- und Entwicklungsteams lassen sich Wirkhebel für nachhaltige IT entfalten – vorausgesetzt, Nachhaltigkeit wird nicht nur theoretisch verstanden, sondern praktisch gelebt. Dafür braucht es mehr als Wissen über Prinzipien oder Tools: Entscheidend ist, wie bewusst diese Ansätze in den Arbeitsalltag integriert werden.

Viele ressourcenrelevante Entscheidungen entstehen im Kleinen – bei der Gestaltung von Schnittstellen, beim Logging, bei der Wahl von Bibliotheken oder bei der Architektur von Datenflüssen. Umso wichtiger ist es, Nachhaltigkeit strukturell in die Arbeitsprozesse einzubetten:

- Definition of Done erweitern: Nachhaltigkeit als Bestandteil von Akzeptanzkriterien und Review-Prozessen.

- Architekturentscheidungen dokumentieren: zum Beispiel durch Nachhaltigkeitsnotizen in ADRs (Architecture Decision Records).
- Regelmäßige Reflektion ermöglichen: Nachhaltigkeit als fester Bestandteil in Retrospektiven oder Tech Committees.
- Toolunterstützung nutzen: automatische Hinweise in Pull Requests oder CI/CD-Pipelines, die auf ineffizienten Code hinweisen.
- Grüne Aspekte in Checklisten und Guidelines integrieren: zum Beispiel bei Logging, Fehlerbehandlung, API-Design oder Speichernutzung.

Genauso wichtig wie die Prozesse ist der Austausch:

- Green-IT-Stammtische oder Fokusgruppen fördern informellen Wissenstransfer und stärken die Eigenverantwortung im Team.
- Teilnahme an Meetups oder Konferenzen bringt externe Impulse in die Organisation.
- Schulungen oder Zertifizierungen helfen, Fachwissen gezielt aufzubauen und zu vertiefen.

Wenn Nachhaltigkeit regelmäßig Teil der Gespräche ist, wird sie zunehmend als gemeinsame Aufgabe verstanden – nicht als individuelles Interesse einzelner, sondern als Qualitätsanspruch der gesamten Organisation.

Nachhaltigkeit natürlich integrieren – bewusste Entscheidungen ermöglichen

Damit Nachhaltigkeit nicht als zusätzlicher Mehraufwand empfunden wird, sondern ganz selbstverständlich mitgedacht wird, braucht es mehr als Checklisten und Tools. Nachhaltigkeit muss in Fleisch und Blut übergehen – ins „Rückenmark" aller Beteiligten.

Doch neue Denk- und Handlungsweisen entstehen nicht über Nacht. Es braucht Impulse, Routinen und Reflexion – immer wieder, bis sie zur Gewohnheit werden. Denn unser Gehirn sucht nach Automatisierung und entscheidet oft nach gelernten Mustern zu handeln. Diese sind jedoch selten auf Nachhaltigkeit ausgerichtet.

Eine einfache Frage kann helfen, innezuhalten und die eigene Perspektive zu schärfen:

„Was würde ich anders entscheiden, wenn Nachhaltigkeit wichtig wäre?"

Diese Perspektive eröffnet neue Entscheidungsräume – bei Architekturfragen genauso wie bei Feature-Umfang, Release-Taktung oder Infrastrukturwahl.

Auch scheinbar kleine Weichenstellungen entfalten in der Summe große Wirkung. Wer Nachhaltigkeit mitdenken will, sollte regelmäßig reflektieren:

- Wie beeinflussen unsere Qualitätsziele (zum Beispiel Performance, Skalierbarkeit, Wartbarkeit) den Ressourcenverbrauch?
- Welche langfristigen Auswirkungen haben Entscheidungen auf Betrieb, Wartung oder Hardwareabhängigkeiten?
- Ist die gewählte Lösung angemessen oder überdimensioniert?

Je bewusster Entscheidungen getroffen werden, desto tiefer verankert sich nachhaltiges Denken im Alltag – nicht als Sonderfall, sondern als Teil guter, verantwortungsvoller Architektur-Arbeit.

Nachhaltigkeit ist ein Systemthema

Wer nachhaltige IT will, braucht mehr als grüne Tools. Es geht um systematisches Denken, pragmatisches Handeln und kollektive Verantwortung. Technik, Organisation, Menschen, Prozesse – alles hängt zusammen. Und überall gibt es Stellschrauben.

Die gute Nachricht: Viele nachhaltige Entscheidungen sind zugleich ökonomisch sinnvoll. Effiziente Systeme sparen Energie und Geld. Klar definierte Prozesse entlasten Teams. Und sinnstiftende Arbeit motiviert.

Die Herausforderung: Es gibt keinen perfekten Plan. Aber viele erste Schritte. Und jede bewusste Entscheidung zählt – im Code, im Gespräch, in der Haltung.

Also: Lasst uns anfangen. Irgendwo. Heute.

Quellen

[1] *https://finance.ec.europa.eu/capital-markets-union-and-financial-markets/company-reporting-and-auditing/company-reporting/corporate-sustainability-reporting_en*
[2] *https://patterns.greensoftware.foundation/*
[3] *https://greensoftware.foundation/*
[4] *https://mlco2.github.io/impact/*
[5] *https://codecarbon.io/*
[6] *https://www.polarsignals.com/*
[7] *https://github.com/borisruf/carbon-footprint-modeling-tool*
[8] *https://www.cloudcarbonfootprint.org/*
[9] *https://if.greensoftware.foundation/*

Sophia Resch
pentacor GmbH
sophia.resch@pentacor.de

Sophia folgt in ihrer Arbeit als Software Engineer einem roten Faden zum Thema Nachhaltigkeit. Ihr Hintergrund in Rechnungswesen, Maschinenbau und mathematischer Modellierung hilft ihr, komplexe Systeme ganzheitlich zu betrachten. In ihrer Arbeit verfolgt sie konsequent das Ziel, IT nachhaltiger und ressourcenschonender zu gestalten.

Ich weiß was, was du nicht weißt

Benjamin Garbers, BREDEX GmbH

Jeder Mensch ist einzigartig. Unterschiedliche Interessenschwerpunkte und Erfahrungslevel in Projektteams sind damit keine Seltenheit, bedeuten aber ein gewisses Maß an Risiko: Was für die eine Person wie eine simple Aufgabe aussieht, mag für jemand anderen ein unlösbares Problem darstellen. Für den Teamerfolg ist ein effektiver Wissensaustausch daher unerlässlich.

David Owens sagte in den 1990ern schon: „Only 2 percent of information gets written down – the rest is in people's heads." [1] Auch 30 Jahre später scheint sich daran nicht viel geändert zu haben, wie der Autor bereits am eigenen Leib erfahren musste und dies als Anreiz nahm, sich einmal genauer mit dem Thema zu befassen. In diesem Artikel werden zunächst ein paar Grundsätze zum Thema „Wissen" und die damit verbundenen Risiken dargestellt. Danach werden ausgewählte Methodiken beleuchtet, die einen effektivieren Wissensaustausch fördern können. Abschließend zeigen Beispiele, wie diese situationsabhängig angewendet werden können.

Arten von Wissen

Wissen lässt sich primär unterscheiden in explizites und implizites Wissen. Explizites Wissen zeichnet sich dadurch aus, dass es in Form von Sprache und Schrift kodifizierbar ist und in Struktur von Anleitungen, Regeln, Dokumentationen oder ähnlichem formalisierbar ist. Es ist somit für andere Personen leicht, auf die erfassten Informationen zuzugreifen.

Implizites Wissen hingegen liegt im Können einer Person und ist somit nur schwer oder gar nicht strukturiert erfassbar. Es kommt durch intuitiv richtige Handlungen (in actu) wie zum Beispiel das unbewusste Binden von Schnürsenkeln zum Ausdruck. Dieses Handlungswissen lässt sich, je nach Ausprägung, einfacher oder schwieriger in explizites Wissen transformieren. So kann beim Beispiel mit den Schnürsenkeln die handelnde Person selbst vielleicht keine bewussten Regeln benennen, aber eine externe Beobachterin/ein externer Beobachter wäre dazu gegebenenfalls in der Lage. Manchmal können Personen ihr Wissen nicht selbst verbalisieren. Schwieriger wird es bei nicht formalisierbarem oder erfahrungsgebundenem Wissen: Hier sind selbst externe Beobachterinnen und Beobachter nicht in der Lage, die Handlungen einer Person genau zu erklären. Als Beispiel seien hier komplexe Bewegungsabläufe im (Leistungs-) Sport genannt. Deren Grundzüge lassen sich noch durch Regeln und Prinzipien vermitteln. Eine Perfektion hingegen kann nur durch eigene Anwendung und Übung erlangt werden [2].

Fehlende Wissensverteilung als Risikofaktor

Natürlich kann nicht jeder Mensch alles wissen. Aber ein mangelnder Austausch von Informationen kann im Unternehmensumfeld, auch in Hinblick auf das eingangs erwähnte Zitat, einen nicht unerheblichen Risikofaktor darstellen. Im einfachsten Fall werden Probleme doppelt gelöst oder bereits getroffene Entscheidungen wieder und wieder mit derselben Argumentation hinterfragt, weil ein grundsätzlicher Verständnisunterschied herrscht.

Auch hängt die Qualität einer Lösung und deren Umsetzungsgeschwindigkeit stark vom vorhandenen Wissen einer Person ab. Weniger Erfahrung bedeutet im Zweifelsfall eine niedrigere Qualität (und somit oft auch Mehraufwand für Nacharbeiten) oder einfach nur vermeidbare Verzögerungen. Langfristig kann dies die Kundenzufriedenheit und auch die der betroffenen Personen beziehungsweise die des Teams negativ beeinflussen.

Eines der größten Risiken ergibt sich durch sogenannte Wissensinseln, also Personen, die bestimmtes Fachwissen als einzige besitzen und dieses auch nirgends für andere zugreifbar abgelegt haben. Dann sind Sätze wie: „Ja, Bob, bei diesem Problem musst du Alice fragen." keine Seltenheit. Besonders spannend wird es dann, wenn Alice gerade im Urlaub ist oder sogar vor drei Monaten die Firma verlassen hat. Wie *Abbildung 1* andeutet, kann es aber auch keine Lösung sein, einfach alles Wissen im Unternehmen immer explizit machen zu wollen. Neben dem kaum zu leistenden Pflegeaufwand stellt sich hier nämlich die Frage: Wie finde ich zeitnah das richtige Wissen in diesem endlos großen Haufen wieder? Die richtige Balance zwischen explizit und implizit vorhandenem Wissen und dem Verteilungsgrad innerhalb der Organisation zu finden ist nicht einfach und stark situationsabhängig.

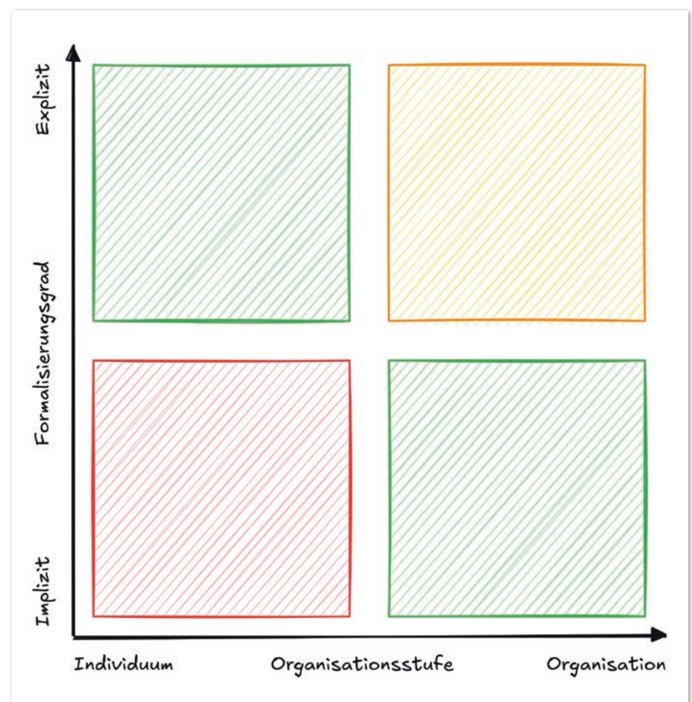


Abbildung 1: Risiko verschiedener Arten von Wissensverteilung (© Benjamin Garbers)

Wie funktioniert Wissensvermittlung in der Theorie?

Der generelle Mechanismus der Wissensvermittlung und -gewinnung im Unternehmensumfeld wurde bereits in den 1990er Jahren von Ikujirō Nonaka und Hirotaka Takeuchi durch das SECI-Modell beschrieben [3]. Es gliedert sich in die vier Schritte:

- *Sozialisierung:* Austausch von implizitem Wissen durch Beobachtung, Nachahmung oder praktische Anleitung.
- *Externalisierung:* Formalisierung der gewonnenen Erkenntnisse in explizites Wissen, sodass diese für andere zugreifbar werden.
- *Kombination:* Verknüpfung mit bestehendem explizitem Wissen zu neuen Erkenntnissen durch Diskussion, Brainstorming, Prototyping oder ähnliche Formate.
- *Internalisierung:* Überführung des neu gewonnenen Wissens in eine implizite Form, womit es intuitiv anwendbar wird.

Diese werden dabei iterativ immer wieder durchlaufen *(siehe Abbildung 2)* und überführen dabei das Wissen einzelner Personen hin zu neuem Organisationswissen.

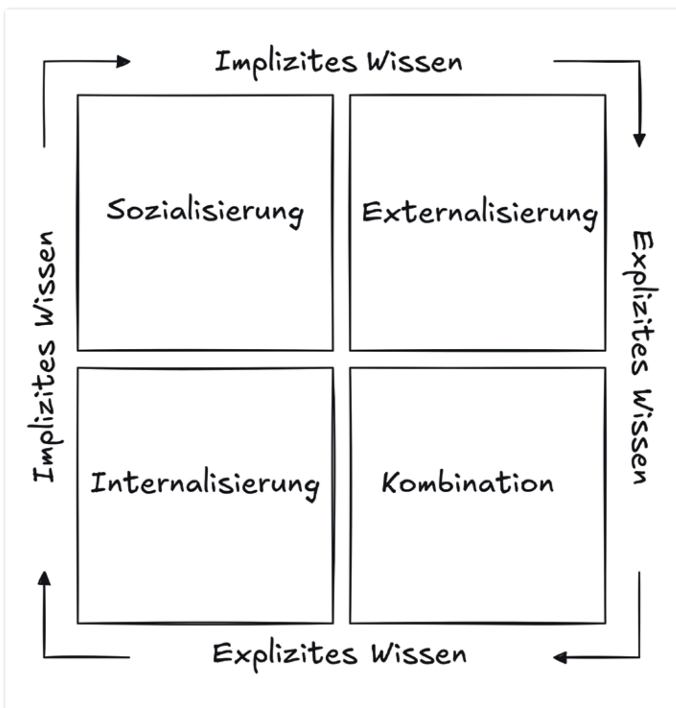


Abbildung 2: SECI-Modell (© Benjamin Garbers)

In der Praxis: Wie fängt man an?

Als Einstiegspunkt in einen Verbesserungsprozess kann man eine Skillmatrix aufstellen. Mit ihr lässt sich der aktuelle Erfahrungsstand eines Teams erfassen, was hilft, vorhandene Stärken und Schwächen aufzuzeigen. Sie ermöglicht es damit, gezielt weitere Maßnahmen auszuwählen, um bestimmte Themenkomplexe oder Teammitglieder zu fördern.

Tabelle 1 zeigt eine exemplarische Matrix mit Skills und der jeweiligen Selbsteinschätzung der Teammitglieder. Die Granularität der Erfahrungsstufen ist beliebig wählbar, sollte aber nicht zu detailliert sein. Die beispielhafte Skala reicht hier von 0 („Nie davon gehört") bis 4 („Kann ich anderen beibringen"). Auch sollte man sich auf die in der aktuellen Situation wichtigen Skills beschränken. Eine Matrix mit 50 Einträgen ist überladen und verliert ihren Mehrwert als einfaches Übersichtsmittel. Aus der Selbsteinschätzung der drei Personen lässt sich ablesen, dass jedes Teammitglied in bestimmten Themengebieten Lücken hat. Kris hat grundlegendes Wissen über fast alle Bereiche, Peter und Carla haben zusätzlich noch besondere Expertise in zwei Skills. Das Team als Ganzes deckt fast alle relevanten Skills gut ab. Der Punkt „Testmanagement" jedoch sticht durch die drastischen Wissensunterschiede als mögliches Projektrisiko hervor.

	Peter	Carla	Kris
Frontendentwicklung	4	1	2
UX-Design	4	1	2
Domänenwissen	1	1	2
Risikomanagement	1	4	2
Testmanagement	0	4	0

Tabelle 1: Eine Skillmatrix

Schreib's auf!

Eine effektive Methode zur nachhaltigen Erfassung von Wissen ist das simple Dokumentieren. Durch bloßes Niederschreiben von Informationen werden diese nicht nur einfacher zugreifbar, sondern man lernt als Autorin oder Autor selbst auch noch dazu, wenn man die eigenen Gedanken einfach und verständlich ausformulieren muss. Dabei sollte man allerdings nicht der Devise „Viel hilft viel!" folgen, sondern auf den jeweiligen Kontext achten. Dieser beeinflusst stark die Wahl von Sprache, Zielmedium, Struktur, vorausgesetztem Wissen und weiteren Eigenschaften.

Eine Orientierungshilfe bietet der Diátaxis-Ansatz [4]. Er enthält neben der Beschreibung verschiedener Dokumentationsformate, wie zum Beispiel Tutorien oder Kurzanleitungen, vor allem Entscheidungshilfen: Welche Informationen sind im aktuellen Kontext relevant? Wie sollte man diese am besten ordnen? Und welcher Schreibstil könnte passen?

Ein gutes Beispiel für eine zielorientierte Dokumentationsform im Softwarebereich ist das bekannte arc42-Format [5]. Neben einer übersichtlichen Struktur liegt seine Stärke in der Beschränkung auf die relevantesten Informationen zur Architektur eines Softwaresystems. Sollte man noch weitere Themen aufschreiben wollen, muss dies in einem zusätzlichen, wiederum zielorientiertem, Dokument erfolgen (Nutzerhandbuch, Betriebshandbuch und viele andere).

Dokumentation ist oft kein Einzelereignis, sondern ein Prozess. Eine gute Dokumentation muss daher kontinuierlich aktualisiert und gepflegt werden, damit sie ihren Mehrwert erhält. Es lohnt sich daher nicht nur das passende Format zu finden, sondern auch einen Blick auf die Werkzeuge zu werfen. Ansätze wie Docs-as-code helfen dabei, manuelle Aufwände gering und den Grad an Automatisierung hoch zu halten [6].

Wissensverteilung durch Kollaboration

Um eine Sozialisierung von Wissen zu fördern, lohnt es sich, regelmäßig seine Arbeitsweise zu hinterfragen und der jeweiligen Situation anzupassen. Die klassische Einzelarbeit bietet gute Möglichkeiten zur

Fokussierung, was beim Selbststudium oder einer Prüfungsvorbereitung hilfreich sein kann. Der Wissensaustausch zwischen Mitarbeiterinnen und Mitarbeitern ist hier jedoch oft nur begrenzt möglich.

Schlanke Kollaborationsformen wie *Pair Programming* sind hier schon wesentlich effektiver: Durch Beobachtung und konstanten Austausch zwischen den Beteiligten können neue Ideen gewonnen und (Verhaltens-)Muster abgeschaut werden, die die eigene Arbeitsweise verbessern. Man sollte jedoch auf die Zusammensetzung der jeweiligen Paare achten: Arbeiten zwei weniger erfahrene Teilnehmerinnen und Teilnehmer zusammen, fehlen oft die richtigen Anreize für gezielte Verbesserungen. Gleichzeitig laufen zwei erfahrene Personen zusammen Gefahr, in ihren hergebrachten Mustern zu verbleiben und wenig neues auszuprobieren. Eine Mischung aus verschiedenen Erfahrungsstufen schafft für beide Beteiligten Anreize aus etablierten Praktiken zu lernen und diese auch zu hinterfragen (siehe Kasten „Stufen der Kompetenzentwicklung").

Mit *Software Teaming* beschreiben Woody Zuill und Kevin Meadows einen Ansatz, der die Kollaborationsidee noch viel weitreichender auslegt: Hier arbeitet das *gesamte* Team zur selben Zeit, am gleichen Ort und am selben Rechner gemeinsam an einem Problem *[7]*. Eine derartige Zusammenarbeit bietet noch wesentlich mehr Möglichkeiten zur Sozialisierung von Wissen durch Beobachtung und Interaktion durch vielfältigere Ideen und Sichtweisen.

Die Autoren beobachten bei ihrer Methode zudem bessere Ergebnisse bei geringeren Durchlaufzeiten sowie weniger Stress für die einzelnen Beteiligten. Gleichzeitig stellt sie aber auch sehr hohe Ansprüche an eine ergonomische Arbeitsumgebung, die für eine große Anzahl an Teilnehmerinnen und Teilnehmern angemessen sein muss (Sitzgelegenheiten, Lesbarkeit des Monitors/Projektors, mögliche Ablenkungen und vieles mehr). Zudem erfordert das Software Teaming für eine effektive Durchführung Disziplin und Kommunikationsfähigkeit. Zum allgemeinen Versprechen der höheren Produktivität gibt es jedoch widersprüchliche Studien, die zeigen, dass diese Methodik nicht immer passend sein kann *[8]*.

Stufen der Kompetenzentwicklung nach Gordon Training International

Die vier Stufen der Kompetenzentwicklung beschreiben die Entwicklung der Fähigkeiten von Personen *[9]*:

- *Unbewusste Inkompetenz:* Personen handeln oft falsch, sind aber selbst nicht in der Lage, die eigenen Defizite zu erkennen und zu überwinden. Für eine Weiterentwicklung benötigen sie externe Hilfe.
- *Bewusste Inkompetenz:* Personen können teilweise intuitiv richtig handeln, ohne zu wissen, warum. Sie sind sich ihrer Defizite aber bewusst und können somit aus Fehlern lernen.
- *Bewusste Kompetenz:* Personen handeln korrekt, indem sie bewusst formale Regeln auf ein Problem anwenden. Dies erfordert allerdings ein hohes Maß an Konzentration.
- *Unbewusste Kompetenz:* Personen handeln zwar intuitiv richtig, können dies aber oft nicht formal begründen und damit effektiv weitervermitteln.

Lernkultur über Projektgrenzen hinaus

Auch auf der Organisationsebene lassen sich Maßnahmen etablieren, die einer Wissensverteilung und -gewinnung zuträglich sind. In den letzten Jahren sind verschiedene Formate entstanden, die sich gezielt mit einzelnen Themenschwerpunkten befassen – zum Beispiel die „Samman Coaching Method" für *technisches Coaching [10]*. Zudem existieren Angebote wie *Lerncoachings*, die nicht auf punktuellen Wissenserwerb abzielen, sondern die Lernkompetenz (Konzentration und Motivation) einer Person an sich unterstützen und fördern sollen. Durch verschiedene Beratungstechniken werden die eigenen Stärken und Schwächen reflektiert und somit die Möglichkeit geschaffen, den eignen Lernprozess besser zu gestalten. Da dieses Vorgehen hochgradig individuell für jeden Menschen ist kann es besonders für Organisationen schwierig sein, solch ein Angebot in der Breite bereitzustellen.

Einfacher gestaltet es sich, Wege zu schaffen, die die Sichtbarkeit und einen *proaktiven Austausch* von Wissen fördern. Kleine Maßnahmen wie ein *„Today I learned"*-Kanal im Unternehmenschat können bereits große Wirkung zeigen: Neben den veröffentlichten Inhalten selbst erzeugen sie Sichtbarkeit für Wissensträger und motivieren andere dazu, ebenfalls Wissen zu teilen. Mehr Investitionsbereitschaft erfordern Angebote wie klassische *Communities of Practice*, in denen sich themenspezifisch ausgetauscht werden kann oder eine *Conference WatchParty*, bei der gemeinsam Konferenzaufzeichnungen geschaut und anschließend diskutiert werden. *Projektbesuche* zählen ebenfalls zu dieser Kategorie. Hier hospitieren einzelne Teammitglieder gezielt in anderen Projekten im Unternehmen, um Technikwissen auszutauschen und andere Arbeitsweisen zu beobachten.

Neben der internen Arbeit kann man zudem auch einen Austausch über Organisationsgrenzen hinweg anbieten und fördern. Hierzu zählen zum Beispiel die Unterstützung von lokalen User Groups, die Teilnahme an Veranstaltungen wie dem Global Game Jam *[11]* oder der klassische Konferenzbesuch einzelner Mitarbeiter.

Erfahrungen aus dem Projektalltag

Das erste Projekt wurde von einem eingespielten und erfahrenen Team betreut. Jeder wusste um die Stärken und Schwächen der anderen und Arbeitspakete wurden sehr oft mithilfe von Pair Programming bearbeitet, um voneinander zu lernen. Da eine sehr aktive Kommunikationskultur vorherrschte, mussten nur die wichtigsten Konzepte und Entscheidungen verschriftlicht werden. Hierbei lag der Fokus mehr auf dem fachlichen „Was?" und „Warum?" und weniger auf dem „Wie?". Konkrete technische Detailfragen konnten sich die Mitglieder entweder selbst erarbeiten oder durch Zusammenarbeit klären.

Die zwei größten Herausforderungen im nächsten Beispiel bestanden in dem Fakt, dass das Team hybrid arbeitete (eine Hälfte vor Ort, eine Hälfte remote) und es extreme Wissensinseln gab. Der erste Punkt sorgte für ein Ungleichgewicht der Informationsverteilung: Wer vor Ort arbeitete hatte ganz andere Chancen beiläufig wichtige Informationen (und auch Entscheidungen!) aus den Bürogesprächen aufzuschnappen als die Remotegruppe. Vom zweiten Punkt waren alle betroffen, was sich auch als großes Projektrisiko herausstellte, da Arbeitsabläufe teilweise von diesen alleinigen Wissensträgern limitiert wurden (Bottlenecks). Der Fokus an Maßnahmen lag daher auf der Sozialisierung und Extraktion von Wissen. Durch mehr Zu-

sammenarbeit in Pairing- und Teamingformaten konnten wichtige Kernaspekte ausgetauscht und später zum ersten Mal verschriftlicht (oder sogar automatisiert) werden. Abläufe wurde so auf einmal für alle reproduzierbar und bisherige Entscheidungen konnten besser verstanden und hinterfragt werden.

Das letzte Beispiel stammt aus einem Projekt mit sehr unterschiedlicher Teamzusammensetzung. Fast alle Mitglieder hatten unterschiedliche Erfahrungslevel, kamen aus verschiedensten Kulturkreisen und hatten verschiedene Muttersprachen. Zu guter Letzt arbeiteten fast alle auch noch zeitlich und räumlich verteilt. In dieser Situation bestand eine wichtige Aufgabe darin, eine kollektive Basis für eine Zusammenarbeit zu schaffen. Die verfügbare gemeinsame Zeit wurde hauptsächlich genutzt einen effektiven Arbeitsmodus zu finden, der die Verteilung und Asynchronität des Teams berücksichtigt. Bei der Wissensvermittlung lag der Fokus zu Beginn stark auf dem „Wie?", das heißt, konkretem technischen Wissen, und wechselte erst nach einiger Zeit stärker auf ein fachliches „Was?" und „Warum?". Zum Einsatz kamen dafür Methodiken aus dem technischen Coaching und gezieltes Pair Programming. Mit Software Teaming wurden in diesem Projekt aufgrund der herausfordernden Kommunikationssituation weniger gute Erfahrungen gemacht.

Im Vergleich zu den anderen Beispielen existierte auch viel mehr explizites Wissen in Form von READMEs, Systemdokumentationen, Handbüchern und ähnlichem, um Dinge nachschlagen zu können, wenn Teammitglieder einmal nicht sofort verfügbar waren.

Zusammenfassung

Allen genannten Ansätzen ist gemein, dass sie keine allgemeingültigen Empfehlungen sind, sondern eher als Bausteine eines Werkzeugkastens zu verstehen sind, die zum jeweiligen Unternehmen, Team und Problem passend ausgewählt werden müssen. Dem Ökonom Peter Drucker wird oft das Zitat zugeschrieben: „Culture eats strategy for breakfast". Bedeutet: Wenn in einem Unternehmen kein Bewusstsein und keine Wertschätzung für einen aktiven Wissensaustausch herrschen, wird man es mit manchen Maßnahmen sehr schwer haben. Ist diese Hürde jedoch überwunden, hat man die Möglichkeit, Projekte und die Mitarbeiter- sowie Kundenzufriedenheit nachhaltig zu verbessern.

Quellen

[1] Michael Hickins (2000): Xerox Shares Its Knowledge, Routledge, London.
[2] Georg Hans Neuweg (2005): Implizites Wissen als Forschungsgegenstand, Bertelsmann, Bielefeld.
[3] Ikujirō Nonaka, Hirotaka Takeuchi (1995): The knowledge creating company: how Japanese companies create the dynamics of innovation, Oxford University Press, New York.
[4] *https://diataxis.fr/*
[5] *https://arc42.org/*
[6] *https://docsascode.org/*
[7] Woody Zuill und Kevin Meadows (2022): Software Teaming: A Mob Programming, Whole-Team Approach, Eigenverlag.
[8] Daniel Ståhl und Torvald Martensson (2021): Mob programming: From avant-garde experimentation to established practice, Elsevier, Amsterdam.
[9] *https://de.wikipedia.org/wiki/Kompetenzstufenentwicklung*
[10] *https://www.sammancoaching.org/*
[11] *https://www.globalgamejam.org/*

Benjamin Garbers
BREDEX GmbH
benjamin.garbers@bredex.de

Benjamin befasst sich in seiner Rolle als Principal Software Architect neben dem klassischen Projektgeschäft vor allem mit der internen Aus- und Weiterbildung im Bereich Softwarearchitektur.

Ich hab' nichts zu verbergen!

Philipp Houzar, iSecNG

Ein Satz, den viele von uns schon einmal gehört, einige sogar bereits selbst benutzt haben. Er wirkt so harmlos, so gesetzestreu – und ist doch so trügerisch. Was, wenn die dahinterliegende Haltung die Grundlage für ein massives gesellschaftliches Problem darstellt? Nämlich eine Kultur, in der Überwachung zur Norm wird und Privatsphäre zum Luxus. Zu einem Luxus, der nur noch wenigen Individuen vorbehalten bleibt, die bereit sind, den enormen Aufwand dafür aufzubringen. In diesem Artikel hinterfragen wir den Satz „Ich hab' nichts zu verbergen." Wir beschäftigen uns mit der Frage, warum er so leichtfertig als Argument dafür verwendet wird, dass allgegenwärtige Überwachung legitim ist, und warum er nicht nur kurzsichtig, sondern sogar gefährlich für uns alle ist.

Was steckt hinter dem Satz?

Nichts zu verbergen zu haben, klingt wie ein erstrebenswerter Zustand im Einklang mit dem Gesetz. In Wahrheit verbirgt sich dahinter ein weit verbreiteter Trugschluss, der Folgendes impliziert: Wenn Sie etwas zu verbergen haben, haben Sie etwas Falsches getan, was Sie jetzt verheimlichen müssen.

Obwohl kriminelle Machenschaften im Verborgenen liegen, heißt das nicht, dass alles, was verborgen bleiben soll, zeitgleich kriminell ist. Diese Umkehrung nennt man Inversionsfehler *[1]*.

In Wahrheit haben wir alle etwas zu verbergen! Oder würden Sie Ihre Bankkarte mit der zugehörigen PIN einer wildfremden Person auf der Straße in die Hand drücken? Lassen Sie etwa die Tür Ihres Badezimmers, eines Hotelzimmers oder Ihres Zuhauses immer geöffnet, damit jeder alles sehen kann? Wie sieht es mit Ihren intimsten Gedanken, Ihren Schwächen, Gefühlen, Vorlieben oder auch politischen Überzeugungen aus? Erzählen Sie das alles jeder Person, der Sie begegnen?

Höchstwahrscheinlich können Sie mindestens eine dieser Fragen mit einem klaren Nein beantworten. Und wissen Sie, wieso? Weil keine dieser Informationen jede andere Person auf der Welt etwas angeht! Aber genau das ist die Kernaussage der Behauptung, nichts zu verbergen zu haben.

Warum Privatsphäre wichtig ist – für alle

Privatsphäre ist nicht nur ein individueller Luxus, sondern vielmehr ein Grundpfeiler unserer Freiheit. Wer nichts zu verbergen hat, braucht auch keine Wahlkabinen mehr und wer das eigene Wahlverhalten nicht geheim halten kann, ist erpressbar und somit manipulierbar. Außerdem ist ohne Vertraulichkeit keine freie Meinungsbildung möglich. Wenn wir diese Punkte einmal in Zusammenhang zueinander betrachten, können wir daraus schlussfolgern, dass eine Demokratie somit nicht mehr möglich wäre.

Auch scheinbar harmlose Daten wie Geburtsdatum, Wohnort oder unser Einkaufsverhalten werden durch Algorithmen zur Erstellung detaillierter Profile verwendet. Diese können nicht nur zur Analyse unseres bisherigen Verhaltens, sondern auch zur Vorhersage oder sogar Steuerung unserer zukünftigen Entscheidungen genutzt werden. Was dabei entsteht, ist eine asymmetrische Machtverteilung: Wer über die Daten verfügt, kann über andere urteilen, sie kontrollieren oder manipulieren. Moderne Technologien (zum Beispiel personalisierte Werbung, Anzeige/Filterung von Beiträgen in sozialen Medien) erleichtern eine solche Kontrolle oder Manipulation ungemein.

Gefahr durch Machtasymmetrien

Daten sind Macht. Wer Zugriff auf unsere Kommunikation, Standorte, Interessen und Gewohnheiten hat, kann uns gezielt beeinflussen. Ein bekanntes Beispiel dafür ist das Datenanalyse-Unternehmen Cambridge Analytica (2014–2018) *[2]*. Es sammelte und analysierte Wählerdaten, um maßgeschneiderte Botschaften zu erstellen, die gezielt das Wahlverhalten beeinflussen sollten – und das mit beunruhigendem Erfolg.

Überwachung ist nie neutral. Sie trifft nicht alle gleich, sondern wirkt besonders stark auf diejenigen, die ohnehin weniger gesellschaftliche oder politische Macht besitzen: politisch Engagierte, Whistleblower, Minderheiten. Hinzu kommt der sogenannte Chilling-Effekt *[3]*: Menschen, die sich beobachtet fühlen, beginnen, ihr Verhalten anzupassen. Sie äußern sich vorsichtiger, vermeiden unbequeme Positionen und ziehen sich ins Unauffällige zurück – nicht aus Überzeugung, sondern aus Angst.

So verändert Überwachung nicht nur, was wir tun, sondern auch wer wir zu sein wagen.

Gesellschaftlicher Wandel und Rückschritte

Unsere Gesellschaft verändert sich – langsam, aber stetig. Das lässt sich gut anhand eines Phänomens aus der Fischereibiologie erklären: Das „Shifting Baseline Syndrom" *[4]*, zu Deutsch „Verschiebung der Ausgangsbasis", beschreibt die Tatsache, dass jede Generation von dem Zustand der Umwelt ausgeht, den sie in ihrer Jugend erlebt hat – und diesen als „normal" betrachtet. So kann zum Beispiel ein überfischter Ozean als gesund wahrgenommen werden, da der eigentlich gesunde Zustand zu einer Zeit vor der eigenen Wahrnehmung existiert hat und somit nach und nach aus dem kollektiven Bewusstsein gerät.

Übertragen auf die Themen Überwachung und Datenschutz bedeutet das Folgendes: Jede Generation gewöhnt sich an ein höheres Maß an Kontrolle und Datenpreisgabe – ohne zu realisieren, wie viel Freiheit dabei verloren geht. Wenn Kinder heutzutage mit GPS-Trackern im Schulranzen aufwachsen, mit digitalen Assistenten sprechen, beinahe an jeder Haustür eine Türklingel mit Kamera verbaut und Gesichtserkennung auf Bahnhöfen allgegenwärtig ist, dann verschiebt sich der Maßstab dessen, was zumutbar oder sogar wünschenswert ist.

Hinzu kommt, dass Regierungen kommen und gehen – was heute legal ist, kann schon morgen als Straftat erachtet werden. Die deutsche Geschichte zeigt, dass gesammelte Informationen über die Bevölkerung in den Händen von radikalen Regimen ein erschreckendes Missbrauchspotential entfalten. Ob Nazi-Regime oder Stasi-Apparat – es waren genau jene „zur Sicherheit" erhobenen Daten, die später zur Unterdrückung von Kritik und Widerstand eingesetzt wurden. Der Schutz unserer Privatsphäre wirkt sich nicht nur auf unser Jetzt aus, sondern schützt uns auch vor zukünftigem Missbrauch dieser Daten.

Warum das Argument privilegiert ist

Der Ausdruck „Ich hab' nichts zu verbergen" ist ein Ausdruck von Privilegiertheit. Nur, wer sich sicher fühlt und nicht mit Diskriminierung, Verfolgung oder Unterdrückung rechnen muss, kann so sprechen. Doch nicht alle haben diese Freiheit. Für Menschen, die aufgrund ihrer Hautfarbe, Herkunft, Religion, Sexualität oder politischer Haltung im Fokus stehen, ist Privatsphäre ein lebensnotwendiger Schutz. Wer ein eigenes Zimmer hat, weiß oft nicht, wie viel er oder sie darin verbirgt. Wer keins hat, sehnt sich danach. Erst der Verlust von Sicherheit macht bewusst, wie wertvoll Schutzräume sind.

Doch nicht nur Menschen, die politisch oder anderweitig verfolgt werden, sind von den Auswirkungen dieser privilegierten Haltung betroffen. Oft ist auch die fehlende Wahlfreiheit im alltäglichen Leben eine Einschränkung der Persönlichkeitsrechte. Wenn etwa die Schule oder der Verein bestimmte Messenger, wie zum Beispiel WhatsApp, verlangt, muss sich der oder die Einzelne anpassen, sonst droht Ausschluss oder Benachteiligung. Dazu zählt, dass Informationen zu eventuellen Änderungen von Abmachungen nicht (rechtzeitig) ankommen oder der Kontakt zu bestimmten Personen/Gruppen kaum bis gar nicht möglich ist.

Anders ausgedrückt: Die Akzeptanz der Mehrheit gegenüber Individuen, die sich nicht überwachen lassen und entsprechend andere Apps/Programme nutzen möchten, ist sehr gering. Oft wird angenommen, es sei „einfacher", wenn sich wenige der Mehrheit anpassen. Diese Annahme zwingt jedoch diese wenigen dadurch praktisch zu einer Handlung, die sie im Normalfall aus guten Gründen nicht getätigt hätten.

Das Argument der Überwachung

„Ich hab' nichts zu verbergen" trägt dazu bei, Überwachung zu normalisieren – und das ist gefährlich. Wer diesen Satz sagt, legitimiert stillschweigend die permanente Kontrolle des Alltags. Je mehr Menschen diese Haltung vertreten, desto eher verwandelt sich ein Ausnahmezustand in eine Selbstverständlichkeit. Was in einem Fall als Problem angesehen wird, wird im anderen Fall als persönliches Versagen gewertet. Wer etwas zu verbergen hat, scheint etwas falsch gemacht zu haben.

Doch Überwachung darf niemals zur Norm werden. Sie verlagert die Verantwortung für die Wahrung der Privatsphäre vom System auf das Individuum. Der Fokus verschiebt sich vom Schutz durch Gesetze und Technik hin zur individuellen Anpassung. Wer aus der Reihe tanzt, fällt auf – und wird damit verdächtig. Genau darin besteht die größte Gefahr: Wenn wir Überwachung als normal akzeptieren, machen wir sie zum Maßstab für gutes Verhalten. Dann sind es nicht mehr die Überwachenden, die sich rechtfertigen müssen, sondern die Beobachteten.

Die Technologie für diese Art von Überwachung existiert längst: Moderne Fahrzeuge verfügen zum Beispiel über eine Funktion, die ihre Umgebung im geparkten Zustand mit Kameras überwacht und aufzeichnet. Offiziell soll das unter anderem dem Diebstahlschutz dienen. Doch von außen ist nicht zu erkennen, ob die Kameras gerade aktiv sind.

Stellen Sie sich vor, Sie befinden sich in einer Situation, in der Sie dringend ein Gebüsch aufsuchen müssen – eine Notlage, wie sie jedem Menschen passieren kann. In unmittelbarer Nähe steht ein solches Fahrzeug. Möchten Sie, dass jemand mitbekommt, dass Sie sich im Gebüsch erleichtert haben? Oder dass es sogar aufgezeichnet und an einem für Sie unbekannten Ort gespeichert wird?

Was wir tun können

Privatsphäre ist nicht nur ein Grundrecht, sondern auch eine Verantwortung. Sie beginnt im Kleinen – bei der bewussten Entscheidung, welche Inhalte wir in sozialen Medien teilen – und reicht bis zur sorgfältigen Auswahl privatsphäreorientierter Software und Dienste. Allerdings beschränkt sich die Wahrung unserer Privatsphäre nicht nur auf das digitale Umfeld. Es gehört auch dazu, unerwartete oder übergriffige Datenabfragen im Alltag kritisch zu hinterfragen, anstatt sie reflexhaft hinzunehmen. Oft genug wird man an der Kasse nach der Postleitzahl gefragt oder das Kennzeichen auf der Autobahn und Parkplätzen automatisch per Kamera erfasst.

Wer die Kontrolle über seine Daten behalten will, muss lernen, achtsam mit ihnen umzugehen und Situationen einschätzen zu können, die eine nicht notwendige Datenerfassung bedeuten können. Dabei helfen uns die folgenden Gewohnheiten und Entscheidungen:

Hinterfragen Sie (zumindest für sich selbst) IMMER das Geschäftsmodell eines Dienstleisters oder Anbieters. Kein Unternehmen wird Ihnen aus reiner Nächstenliebe einen kostenlosen Dienst anbieten. Oft sind genau die Daten, die Sie durch die Nutzung eines solchen Angebots preisgeben, wertvoller als das Geld, welches ein vergleichbares Produkt kosten würde.

Zum einen können diese Daten mehrfach – und oft unbemerkt – an verschiedenste, mitunter dubiose Dritte verkauft werden. Was mit Ihren Informationen geschieht, bleibt für Sie meist vollkommen intransparent.

Zum anderen werden Ihre Daten zur gezielten Verhaltensanalyse genutzt. Ein Beispiel, das bestimmt jeder kennt: Kundenkarten beim Einkaufen – angeblich, um „Ihre Treue zu belohnen". In Wahrheit dienen diese Programme der umfassenden Auswertung Ihres Einkaufsverhaltens. Sie geben einem Supermarkt zum Beispiel genaueste Einblicke in die Häufigkeit Ihrer Einkäufe. Weiterhin kann er sehr gut kombinieren, welche Produkte zusammen gekauft werden. „Wenn Sie Äpfel kaufen, kaufen Sie auch Joghurt – aber nur, wenn dieser im Angebot ist", wäre ein leichtes Beispiel für eine solche Auswertung. Da bei Aktionen mit Kundenkarten viele Menschen teilnehmen, lassen sich auf dieser Basis wertvolle Erkenntnisse gewinnen. Diese ermöglichen zum Beispiel eine optimierte Platzierung von Waren oder gezielte Preisstrategien – und bringen

dem Anbieter mitunter deutlich mehr ein, als Sie durch Ihren Rabatt sparen.

Eine weitere gute Gewohnheit in Bezug auf Datenschutz ist das bewusste Teilen von Informationen (Stichwort „Datensparsamkeit"). Seien Sie ruhig geizig mit Ihren Daten – einmal herausgegeben, können Sie diese meist nicht mehr oder nur mit viel Aufwand zurücknehmen.

Die DSGVO ermöglicht es Ihnen, von jedem verantwortlichen Unternehmen, öffentlichen Stellen, Geschäftspartnern und Dienstleistern eine Auskunft über die zu Ihrer Person gespeicherten Daten zu verlangen. Versuchen Sie es mal – Sie werden überrascht sein, wie viel ein einzelnes Unternehmen über Sie wissen kann.

Als Nächstes könnten Sie sich überlegen, ob Sie nicht anstatt WhatsApp einen datenschutzfreundlicheren und sicheren Messenger verwenden wollen. Apps wie Threema oder Signal respektieren Ihre Privatsphäre und bieten einen sehr vergleichbaren Funktionsumfang.

Auch viele weitere Programme und Apps bieten Ihnen Einstellungsmöglichkeiten, um die Erfassung Ihrer Daten einzuschränken. Besonders im Browser lohnt sich ein Blick in die Datenschutz- und Sicherheitseinstellungen – unabhängig davon, welches Programm Sie nutzen. Detaillierte Anleitungen zu Ihrem Browser finden Sie leicht online.

Privatsphäre braucht Gewohnheiten – und Mut zur Entscheidung. Wenn Sie diese oder ähnliche Punkte verinnerlichen, sind Sie bereits auf einem guten Weg. Jede bewusste Wahl zählt.

Fazit: Haben wir wirklich nichts zu verbergen?

Die Aussage „Ich hab' nichts zu verbergen" mag bequem erscheinen, aber sie ist naiv, historisch kurzsichtig und demokratiegefährdend. Privatsphäre ist kein Versteck für Schuld, sondern Raum für Freiheit. Wer über sich selbst bestimmen will, braucht die Möglichkeit, Informationen zu kontrollieren.

Wir sollten unsere Geheimnisse nicht nur verteidigen, sondern feiern. Denn sie machen uns zu freien Menschen.
Privatsphäre bedeutet nicht, dass man etwas Falsches macht – sie bedeutet, dass man selbst entscheidet, wer was über einen wissen darf. Es geht nicht um Heimlichkeit, sondern um Selbstbestimmung. Eine Gesellschaft, in der alle Menschen alles über alle wissen (oder wissen könnten), ist nicht frei, sondern misstrauisch, angepasst und kontrolliert.

Gerade deshalb sollten wir das Argument „Ich hab' nichts zu verbergen" nicht mehr unkommentiert stehen lassen. Es mag aus Gleichgültigkeit oder Gutgläubigkeit entstehen – aber es schwächt den gesellschaftlichen Konsens darüber, dass Menschen ein Recht auf Rückzug, Schutz und Geheimnisse haben. Und es gefährdet letztlich genau das, was wir schützen wollen: unsere Freiheit.

Es ist Zeit, dass wir unsere Privatsphäre nicht mehr als Luxus oder Ausnahme begreifen – sondern als Fundament eines würdevollen, offenen und freien Lebens.

Hinweis: Dieser Artikel wäre nicht entstanden, wäre nicht der Verein Digitalcourage e. V. mit seinem kompakten, klugen Flyer/Folder „Moderner Mythos: Nichts zu verbergen" *[1]* als Inspirationsquelle vorangegangen. Viele Gedanken, Perspektiven und Argumentationslinien basieren auf diesem Material. Danke dafür!

Quellen

[1] Digitalcourage e.V.: Folder „Moderner Mythos: Nichts zu verbergen" (*https://shop.digitalcourage.de/files/Digitalcourage_-_Flyer_Nichts_zu_Verbergen_8-Seiten_-_Layout_v2_1_2023-07-19_flyeralarm.pdf*)
[2] Wikipedia: Cambridge Analytica (*https://de.wikipedia.org/wiki/Cambridge_Analytica*)
[3] Wikipedia: Chilling Effect (*https://de.wikipedia.org/wiki/Chilling_effect*)
[4] Wikipedia: Shifting Baseline (*https://en.wikipedia.org/wiki/Shifting_baseline*)

Philipp Houzar
iSecNG
ph-privacy@houzar.de

Philipp Houzar ist Security Engineer bei iSecNG und hat ein starkes Interesse an gesellschaftlichen Fragen rund um Datenschutz und Privatsphäre. Er engagiert sich für digitale Selbstbestimmung und setzt sich kritisch mit Überwachung und technologischer Machtverteilung auseinander.

FÜR 29,00 €
BESTELLEN

Java aktuell

JAHRESABO

Mehr Informationen zum Magazin und Abo unter:
www.ijug.eu/de/java-aktuell

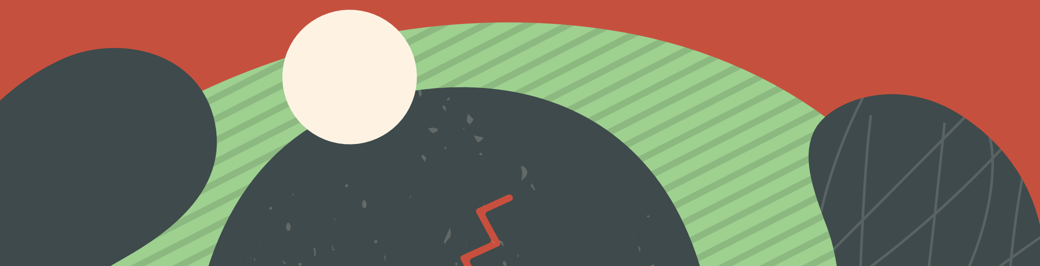
2025

DOAG

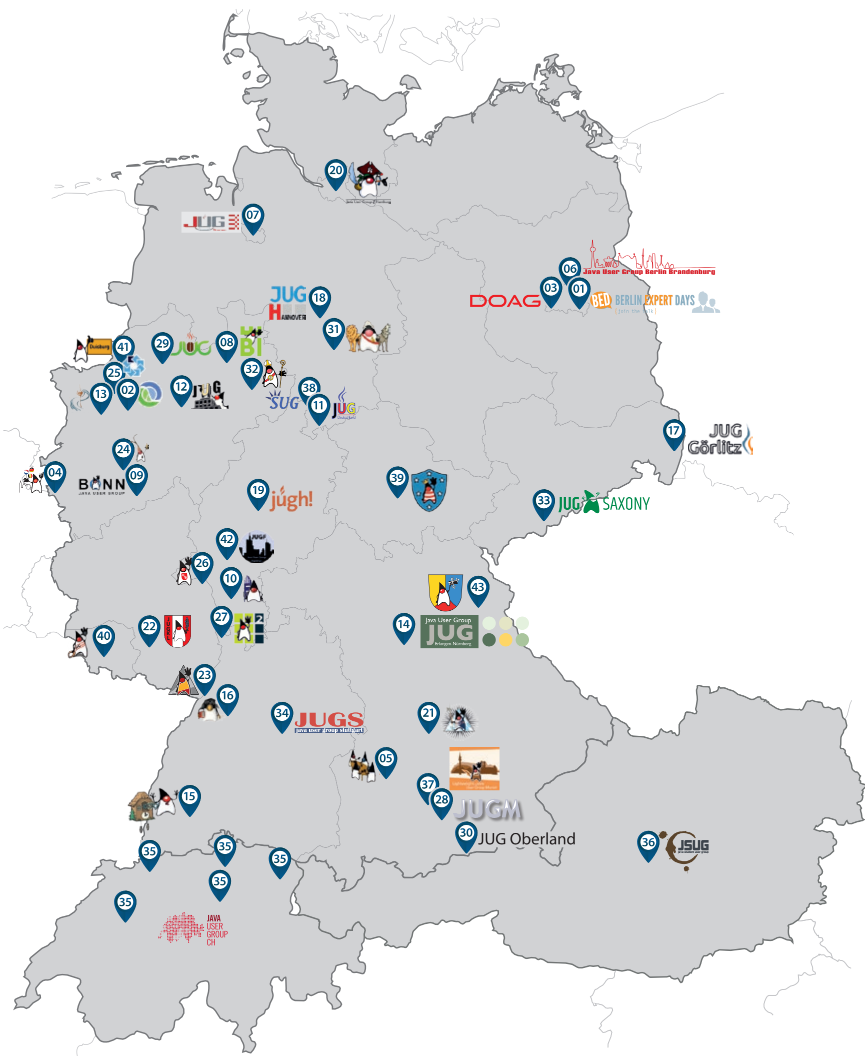
Konferenz + Ausstellung

Die ORACLE Anwenderkonferenz

Nürnberg | 18. – 21. Nov.

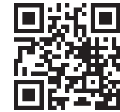


anwenderkonferenz.doag.org



Mitglieder des iJUG

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



www.ijug.eu

Impressum

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

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

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

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

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

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

Bildnachweis:
Titel: Bild © Oleksandr
https://stock.adobe.com
S. 10 + 11: Sloth McSloth
https://stock.adobe.com
S. 18 + 19: Bild © galyna_p
https://stock.adobe.com
S. 26 + 27: Bild © miss irine
https://stock.adobe.com
S. 32 + 33: Bild © Pavel
https://stock.adobe.com
S. 42 + 43: Bild © IsarStudio
https://stock.adobe.com
S. 50 + 51: Bild © Chaosamran_Studio
https://stock.adobe.com
S. 62 + 63: Bild © anantachat
https://stock.adobe.com
S. 70 + 71: Bild © Sandiip
https://stock.adobe.com
S. 76 + 77: Bild © aicandy
https://stock.adobe.com
S. 82 + 83: Bild © QAISAR
https://stock.adobe.com

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

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

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

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

Inserentenverzeichnis

DevLand GmbH i.G.	U2
DOAG e. V.	S. 8-9, S. 88-89, U 3
iJUG e.V.	S. 13, S. 61, S. 87
JavaLand GmbH	U 4

SUPER-SAVER

Bis 31.10.2025

Heide Park Soltau

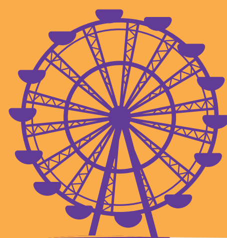
APEX connect

18. - 20. Mai 2026

DOAG 2026 Datenbank

mit Cloud Infrastructure

18. - 19. Mai 2026

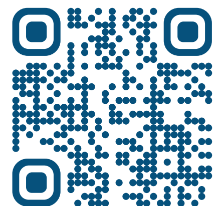


JavaLand

GROßES COMMUNITY-PROGRAMM!

im EUROPA PARK® in Rust

10. - 12. MÄRZ 2026



| #JavaLand | www.javaland.eu

Präsentiert von:

heise medien DOAG

Veranstalter: JavaLand