Ausgabe 06/2024

Java aktuell

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

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 06

Java aktuell

iJUG Verbund
www.ijug.eu

Microservices
Verwaltung, Robustheit, Multi-Cloud

APIs
Standards für gutes API-Design

Testing
Die richtige Testmethode finden

Microservices

KI Navigator 2024

Dein Kompass in der Welt der KI

20. + 21. November in Nürnberg

KI zum Anfassen und Ausprobieren – das ist die Devise unseres gesonderten Bereichs in der AI Experience World.

Informiert euch hier über KI-Tools und -Lösungen direkt bei unseren Ausstellern. Kommt mit dem ein oder anderen Roboter ins Gespräch oder bastelt selbst eine eigene KI-Technologie.

kinavigator.eu

DOAG Heise Medien de|ge|pol

Liebe Leserinnen und Leser,

willkommen zur sechsten Ausgabe des Jahres – der letzten ihrer Art. Denn ab dem kommenden Jahr 2025 reduzieren wir die Anzahl der Ausgaben pro Jahr auf insgesamt vier. So kann das Redaktionsteam seinen Fokus besser verteilen und auch andere Medien mit spannenden Informationen für euch befüllen. Dadurch haben sich auch die Erscheinungstermine verschoben, sodass Ausgabe 1/25 nun im Februar statt bereits im Dezember erscheint. Somit lohnt es sich ab 2025 besonders, ein Jahresabonnement abzuschließen, denn der Preis bleibt bei 29,00 Euro (Vergleich Einzelheft: 7,95 Euro).

Zur Einstimmung in diese Ausgabe fasst Andreas Badelt in seinem Java-Tagebuch alle Geschehnisse und Neuigkeiten aus der Java-Community für euch zusammen. Danach folgt der vierte Teil der goldenen Regeln. Im Anschluss sind wir perfekt gewappnet, in das Titelthema Microservices einzusteigen. Im ersten Fachartikel ab Seite 14 präsentiert Nikolas Hermann Methoden und Werkzeuge zur Verwaltung von Microservices. Dann zeigt Oliver Manz, wie die Robustheit eines Microservices verbessert werden kann, um schwerwiegende Folgen zu vermeiden. Wie das Graal Development Kit für Micronaut für moderne Multi-Cloud-Umgebungen eingesetzt werden kann, um hoch-performante, portierbare und skalierbare Microservices zu erstellen, zeigt uns Marcel Uusvel-Amende in seinem Beitrag. Der Frage „Was können 4 Threads was 600 nicht können?“ geht Jonas Nagy-Kuhlen ab Seite 34 auf den Grund. Im Artikel beleuchtet er anhand eines auf WebFlux und Reactor basierenden Threading-Modells die Vorteile reaktiver Webservices.

Das Vater- und Sohn-Autoren-Duo Uwe und Noah Neukam verrät uns in ihrem Artikel ab Seite 42, worauf es bei der Entwicklung von APIs wirklich ankommt und wie durch Standards eine möglichst hohe Unabhängigkeit erreicht werden kann. Im ersten Teil der insgesamt zweiteiligen Reihe widmen sich Ralf Straßner und Peter Fichtner dem Thema Testing und zeigen Möglichkeiten auf, wie Teams die passenden Tests finden und umsetzen können. Die Ausgabe wird abgerundet vom dritten und letzten Teil von Richard Gross’ Reihe zur Vermeidung strukturzementierender Tests.

Wir wünschen euch viel Spaß beim Lesen!

Lisa Damerow
Redaktionsleitung Java aktuell

INHALT

Werkzeuge und Methoden für die Verwaltung von Microservices

Multi-Cloud mit dem Graal Development Kit für Micronaut

3 Editorial

6 Java-Tagebuch
Andreas Badelt

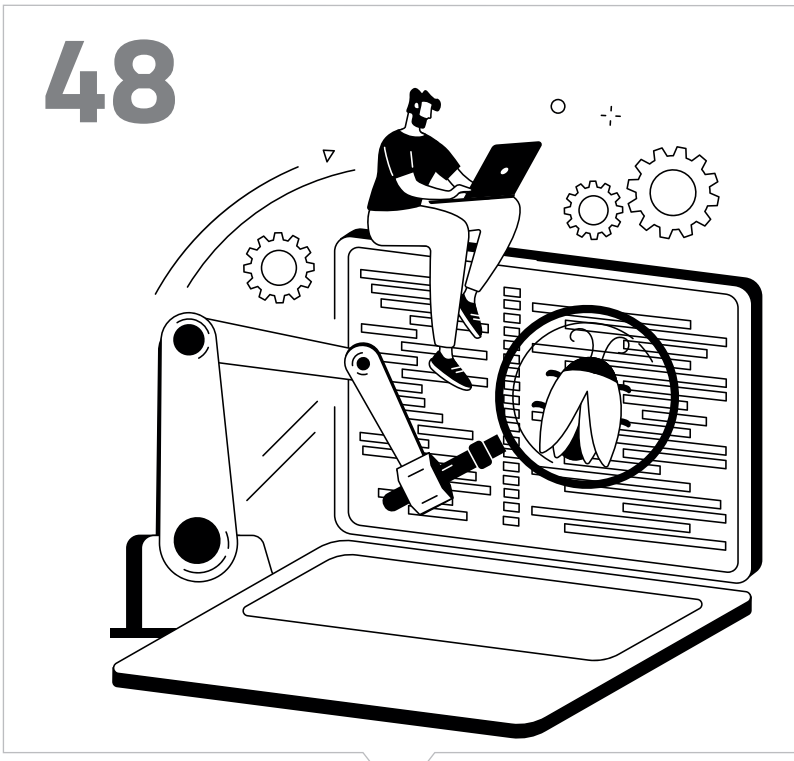
10 Die goldenen Regeln: Teil 4
Andreas Monschau

14 Und was machen wir jetzt mit all diesen Services?
Nikolas Hermann

20 Robuste Systeme
Oliver Manz

26 Multi-Cloud mit dem Micronaut Framework
Marcel Uusvel-Amende

Standards für besseres API-Design



Passende Testmodelle finden

34 Was können 4 Threads, was 600 nicht können? Reaktive Webservices mit WebFlux & Reactor
Jonas Nagy-Kuhlen

42 API-Entwicklung: Das richtige Maß macht's
Noah Neukam, Uwe Neukam

48 Risikobewusst, modular… einfach passend testen! Teil 1
Ralf Straßner, Peter Fichtner

56 Strukturzementierende Tests und wie man sie vermeidet, Teil 3: Das richtige Vorgehen
Richard Gross

67 Impressum/Inserenten

Andreas Badelt, stellv. Leiter der DOAG Cloud Native Community

JAVA TAGEBUCH

1. Juni 2024

EclipseCon geht in OCX auf

Wenn diese Tagebuch-Ausgabe erscheint, steht die neue „Open Community Experience"-Konferenz der Eclipse Foundation kurz bevor. Die traditionelle EclipseCon (bislang jeden Herbst in Ludwigsburg) wird dann Teil der größeren OCX werden, die vom 22. bis 24. Oktober in Mainz stattfinden wird. Neben der EclipseCon zum Thema „Empowering the Developer Tools Ecosystem" der Foundation wird es die Streams „Open Community for Automotive" und „Open Community for Java" geben.

7. Juni 2024

JDK 23

Das JDK 23 steht auch schon kurz vor dem geplanten Release am 17. September, ohne größere Updates kurz vor Schluss. Die JEPs 480 und 481 („Structured Concurrency", „Scoped Values") wird es in der jeweils dritten Preview geben. Und die „Flexible Constructor Bodies" drehen als zweite Preview auch eine weitere Runde auf dem Weg zum stabilen Feature.

Für das JDK 24 (Frühjahr 2025) gibt es bislang einen Kandidaten mit JEP 472 („Prepare to Restrict the Use of JNI"). Neben dem Java Native Interface soll auch die in Java 22 eingeführte Foreign Function & Memory (FFM) API zukünftig per Default eingeschränkt werden – und die benötigten Interfaces dann gezielt freigeschaltet werden können. Zunächst werden aber nur einheitliche Warnungen für beide APIs eingeführt und die Methoden in sun.misc.Unsafe als „deprecated" markiert. Die tatsächliche „integrity by default" wird mit einem späteren Release kommen. Zur Unterstützung der Entwickler:innen soll das neue Tool jnativescan dienen, um die Nutzung zukünftig eingeschränkter Methoden aufzuspüren.

Für die generelle Suche nach der Nutzung „abgekündigter" Methoden empfiehlt sich ansonsten das Java Flight Recorder „Deprecated Event", das mit JDK 22 eingeführt wurde. Erik Gahlin beschreibt, wie es genutzt werden kann, insbesondere um „3rd party libraries" auf drohendes Ungemach zu scannen *[1]*.

17. Juni 2024

Temurin Security Audit

Als eine Maßnahme seiner „Secure Supply Chain"-Initiative hat das Eclipse-Temurin-Projekt schon im vergangenen Jahr ein unabhängiges Audit seiner Build- und Verteilungsprozesse durchführen lassen und entsprechende Aktionen eingeleitet. Unter anderem wurde das statische Analyse-Tool Semgrep eingeführt, um jeden Pull Request in den Repositories automatisch zu prüfen. Nachdem die Empfehlungen der externen Berater abgearbeitet wurden, hat Eclipse den ursprünglichen Bericht und die entsprechend durchgeführten Aktionen nun veröffentlicht. Beide sind im Blog-Eintrag verlinkt, zusammen mit weiteren Informationen zum Thema Security (Audits), die auch für andere Projekte interessant sein könnten *[2]*.

28. Juni 2024

Oracle und die JDK-Lizenzen

Hendrik Ebbers (JUG Dortmund) und Falk Sippach (JUG Darmstadt) reden Tacheles *[3]*: Oracles Lizenzpolitik – insbesondere sein Vorgehen gegenüber Unternehmen, die Java einsetzen – könne zur Gefahr für das Java-Ökosystem werden. Worum geht es? In den letzten Jahren hat Oracle zunächst ein kommerzielles Lizenzmodell für das Oracle JDK eingeführt und dieses mit Java 20 noch einmal angepasst. Firmen, die (irgendwo) das Oracle JDK einsetzen, müssen nun pro Mitarbeiter eine monatliche Pauschale zahlen, was zu durchaus hohen Kosten führen kann. Soweit bekannt und auch schon Thema im Tagebuch gewesen. Seit einiger Zeit hat Oracle wohl Daten gesammelt, geht nun massiv auf Firmen zu, deren Mitarbeiter schon mal das Oracle JDK heruntergeladen haben (und sich der drohenden Lizenzkosten vielleicht gar nicht bewusst waren), und setzt sie mit einem Angebot unter Druck: Wenn sie direkt ein Lizenz-Abo abschließen, werden sie nicht zu einer Nachzahlung für etwaige Lizenzverfehlungen gezwungen. Das schreckt natürlich die Firmen auf, die sich vermehrt an professionelle Lizenzberater wenden, oder eventuell sogar über einen Abschied von Java nachdenken – aus Unwissenheit über die Alternativen.

Die Autoren fragen sich dann, „ […] was Oracle als Nächstes probieren wird, falls die aktuellen Bemühungen nicht den ausreichenden Erfolg bringen… vielleicht beschließt ein Manager bei Oracle dann, die Mittel für das OpenJDK zu kürzen." Das wäre natürlich der Gau, aber vorher wird hoffentlich die Vernunft zurückkehren; oder im schlimmsten Fall das Java-Ökosystem, in dem ja noch viele andere kleine bis große Unternehmen aktiv sind, den Verlust auffangen.

6. Juli 2024

Java und KI

Javas Status in der schönen neuen KI-Welt wird in einem Video von Nicolai Parlog auf inside.java analysiert. Kurz zusammengefasst: Der Ruf „AI in Java is bad" ist eine stark verkürzte Momentaufnahme. Es gibt drei wesentliche Kategorien: Modelle entwickeln, ausführen und in Produkte integrieren. Nur in der ersten Kategorie kann Java nicht an der Spitze mithalten und wird es wohl auch in Zukunft nicht. Zum Erfolg in der KI-Welt werden nicht zuletzt die kontinuierliche Lieferung neuer Features durch OpenJDK-Projekte wie Valhalla (Verbesserungen der Peformance und Speichereffizienz) und Panama (bessere Interoperabilität mit Native Code) beitragen – sowie weitere Plattformen oder Bibliotheken wie die TornadoVM, Tribuo oder die Deep Java Library (DJL). Die Größe und Vielfalt, und damit auch Innovationskraft des Java-Ökosystems – so könnte man es auf den Punkt bringen – werden dafür sorgen, dass Java auch hier erfolgreich ist *[4]*.

16. Juli 2024

Quarkus 3.15 (LTS)

Das Quarkus-Projekt hat das nächste Long Term Support (LTS) Release angekündigt. Das letzte, Quarkus 3.8, ist erst im Februar erschienen. Das nächste, Quarkus 3.15, soll schon Ende September kommen. Im Gegensatz zum JDK geht es hier nicht um Support-Zeiträume von vielen Jahren. Die Quarkus-Releases kommen in kurzer Folge und LTS bedeutet hier 12 Monate „Stabilität" unabhängig von erfolgten weiteren Releases, mit Support für Bug und Security Fixes sowie Performance-Verbesserungen.

26. Juli 2024

Fujitsu in der Adoptium Working Group

Die Eclipse Adoptium Working Group wächst weiter. Als neuestes Mitglied wurde jetzt Fujitsu aufgenommen. Wer sich für die laufenden Aktivitäten des Adoptium-Projekts interessiert – von Secure Supply Chain bis Marketing – schaut im 2024er Plan nach, der soeben mit dem Halbjahrs-Review aktualisiert wurde *[5]*.

21. Juli 2024

Jakarta EE 11

Jakarta EE 11 wird das Ziel Juni/Juli wohl nicht erreichen, aber laut Ivar Grimstad in Kürze den ersten Release-Kandidaten herausbringen. Freuen wir uns auf Verbesserungen rund um CDI, Jakarta Data 1.0, die Nutzung von Java 21, die nicht zuletzt auch Virtual Threads für bessere (Memory-)Performance mitbringen, und einige mehr *[6]*. Die Jakarta EE Ambassadors haben inzwischen eine Präsentation zusammengestellt, die sowohl die neuen Features mit 11 als auch den Weg dorthin von Java EE beschreibt *[7]*.

25. Juli 2024

Stackoverflow-Umfrage 2024

Die Ergebnisse des jährlichen „Stack Overflow Developer Survey" sind aufbereitet (für die Jüngeren: Stack Overflow war vor ChatGPT die bevorzugte Website für „Code Copy & Paste"). Über 65.000 „Coder" aus aller Welt haben das Survey ausgefüllt, am meisten aus den USA (19 %), aber wie immer mit überproportional starker Beteiligung einiger Länder dahinter: Deutschland (8 %), Indien (7 %), UK (6 %) und die Ukraine (5 %) – während zum Beispiel China überhaupt nicht unter den ersten 10 auftaucht. Liegt es an der Sprachbarriere? Oder wird Stack Overflow gar als potenzielle „Copy & Paste Schadcode-Schleuder" des Westens von der großen Firewall blockiert? Nein: Wie eine Suche auf Stack Overflow (wo sonst?) schnell ergibt, wird die Seite in China nicht blockiert, es gibt aber eine ähnliche chinesische Seite namens „segmentfault", die sich auf das chinesische Publikum fokussiert.

Ein paar Details aus den Ergebnissen *[8]*: Entwicklerinnen und Entwickler mussten anscheinend im letzten Jahr im Durchschnitt Einkommenseinbußen hinnehmen, sind aber in Bezug auf KI und ihre Arbeitsplätze wieder optimistisch: 70 % sagen, dass diese keine Bedrohung für ihren Job darstelle. Auch keine Bedrohung, aber die größte Frustrationsquelle im Job sind technische Schulden – mit 62 % weit vor Platz zwei und drei: (Zu) komplexe Tech Stacks für Build und Deployment.

29. Juli 2024

Schmucksteine für Java Developer

Valhalla und Panama hatten wir schon in dieser Tagebuch-Ausgabe. Zu einem anderen JDK-Projekt hat Gavin Bateman von Oracle vor kurzem einen guten „Fortschrittsbericht" gehalten: Projekt Amber („Bernstein") fokussiert sich auf die Produktivität der Entwicklerinnen und Entwickler und liefert dazu eine Reihe kleinerer Features. Unter anderem steht die datenorientierte Entwicklung auf dem Zettel, dazu wurden beispielsweise schon Records und Sealed Classes geliefert. Aber auch Pattern Matching und dazu die Verbesserungen an den Switch Statements sind Amber-Features. Zukünftige Features werden sich um Verbesserungen für das Paradigma der datenorientierten Programmierung kümmern und um Sprachkonstrukte für den leichteren Einstieg in die Java-Programmierung. Das ganze Video gibt's auf YouTube *[9]*.

Quellen

[1] *https://egahlin.github.io/2024/05/31/deprecated-event.html*
[2] *https://adoptium.net/de/blog/2024/06/external_audit/*
[3] *https://www.heise.de/blog/Kommentar-zu-Oracles-Lizenzen-Ein-Angebot-das-Sie-nicht-ablehnen-koennen-9781727.html*
[4] *https://inside.java/2024/07/04/newscast-72/*
[5] *https://www.eclipse.org/lists/adoptium-wg/pdfrJkqZTWzDf.pdf*

[6] *https://jakarta.ee/specifications/platform/11/*
[7] *https://docs.google.com/presentation/d/18V6XUphuFfCYaeNddNua3dlbrHU2f9La*
[8] *https://stackoverflow.blog/2024/07/24/developers-want-more-more-more-the-2024-results-from-stack-overflow-s-annual-developer-survey/*
[9] *https://www.youtube.com/watch?v=Kr3WOoGnelA*

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").

European NetSuite User Days

November 2024 18 - 19

at NCC Ost Nuremberg

netsuite.doag.org
#NetSuiteUserDays

Java aktuell

JAHRESABO

FÜR 29,00 €
BESTELLEN

iJUG
Verbund
www.ijug.eu

Mehr Informationen zum Magazin und Abo unter:

www.ijug.eu/de/java-aktuell

Die goldenen Regeln: Teil 4

Andreas Monschau, Haeger Consulting

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

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

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

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

Regel 4: Schreibe unlesbaren Code

Konventionen vor Konfiguration

Brechen wir direkt mal mit einem der bekanntesten Paradigmen der Softwareentwicklung: Solange sich ein Entwickler in allen Bereichen einer Implementierung an die üblichen Konventionen hält, müssen diese nicht konfiguriert werden. Wenn man aber „Konventionen" liest, schwingt da immer etwas der Geist von Gleichschaltung mit. „Konventionen" – wir sollen alles so machen, wie alle anderen. Aber sind wir nicht alle Individuen? Und überhaupt, Softwareentwickler sind kreative Köpfe, sie sind Künstler! Verzichtet daher auf dieses Paradigma, es macht den Einstieg in eure Codebasis nur unnötig einfach und das wollen wir doch vermeiden, oder?

Dinge einfach halten

KISS ist das nächste vermeintlich so wichtige Paradigma. Aber hier lässt sich ganz einfach sagen: Es war schwer, Code zu schreiben, es soll schwer sein, Code zu lesen. Sonst kann am Ende noch jeder Software entwickeln und wollen wir das wirklich? Nein! Die Anwendung von massiver Obfuskation des Codes wird empfohlen!

Namen von Variablen, Funktionen und Klassen

Obfuskation ist auch hier eine gute Sache – warum sollte denn *temp* ein schlechter Bezeichner sein *(siehe Abbildung 1)*? Es leitet sich doch alles aus dem Kontext heraus wieder her, und im besten Fall steht auch ein schicker Kommentar mit dabei, siehe hierzu Teil 3 der Artikelreihe aus der letzten Ausgabe der Java aktuell.

```
String myText = temp + temp2;
```

Abbildung 1: Klare und verständliche Benamung von Bezeichnern

Genauso verhält es sich mit den Namen von Funktionen. Man soll direkt aus dem Namen heraus erschließen können, was an dieser Stelle passiert – dann ist ein Methodenname wie `validateTextDataAndPrintToDruckstrasseOrDisplayErrorModal()` auch absolut valide. Es ist klar erkennbar, was alles an dieser Stelle passiert: Eben alles!

Liebesgrüße nach Redmond

Lass dir bloß nichts anderes erzählen: ungarische Notation ist dein Freund! Man muss einer anhand des Namens einer Variable direkt erkennen können, um welchen Datentyp es sich handelt. Es könnte ja mal der Fall sein, dass du ohne IDE an deinem Projekt weiterarbeiten musst (wobei mir nicht ganz klar ist, wann denn dieser Fall eintreten könnte). Und ignoriere Ressourcen im Internet, die gegensätzliches behaupten. Es gibt im Internet Stimmen, die ganz gezielt sagen:

> **DO NOT use Hungarian notation.”**

Wisst ihr, wer das sagt? Das sagt Microsoft *[1]*. Microsoft? Das sind die Bösen! Die wollen uns sagen, was wir machen sollen? Auf keinen Fall! (Na gut, es bezieht sich an dieser Stelle natürlich auf Guidelines für .net, aber wir kennen ja die Redmond'schen Machtansprüche…)

Dokumentation im Sourcecode

Im letzten Teil haben wir das Thema „Dokumentation" angesprochen, das ich insbesondere aber auch im Kontext meiner Hinweise zur Codequalität nochmal in Erinnerung rufen möchte. Ein „*To-do*" kann man ebenso gut in den Javadoc-Block oberhalb einer Methode packen. Auch hier gilt – irgendwer wird den Kommentar sicher irgendwann mal vervollständigen.

Die Antithese hierzu wäre, übertrieben viel zu kommentieren – ob es nun Sinn macht oder nicht. Das klassische Beispiel: „*Increments i*".

Du siehst: Es ist ganz einfach, unlesbaren bis schlechten Code zu schreiben. Und das könnt ihr alle! Frustration bei Neulingen ist da vorprogrammiert!

Beim nächsten Mal schauen wir uns etwas detaillierter an, wie man am besten kommunizieren sollte, dass ein Neuling auch absolut gar nichts davon hat. Mit etwas Glück wird es euch helfen, euer Projekt gegen die Wand zu fahren.

Referenzen:

[1] *https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions*

Andreas Monschau
Haeger Consulting
amonschau@haeger-consulting

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

DEINE VORTEILE

25 % Rabatt
auf JavaLand-Tickets

Java aktuell
Jahres-Abonnement

Java Community Process
Mitgliedschaft

JETZT MITGLIED WERDEN!

Ab 15 Euro im Jahr

www.ijug.eu

iJUG Verbund

Und was machen wir jetzt mit all diesen Services?

Nikolas Hermann, Breuninger

Viele Teams haben in den letzten Jahren ihre Architektur auf Microservices umgestellt. Statt für einen Monolithen ist ein Team plötzlich für dutzende oder hunderte Services verantwortlich. Der Aufwand für die Instandhaltung steigt entsprechend. Damit Services nicht durch Vernachlässigung zu Legacy-Services werden und der Wartungsaufwand trotzdem geringgehalten werden kann, braucht es Werkzeuge und Methoden.

In diesem Artikel sollen ein paar davon vorgestellt werden, damit auch in Zukunft noch genügend Zeit für die wertstiftenden Dinge bleibt.

In den letzten Jahren haben Microservices einen wahren Siegeszug angetreten. Sie versprechen mehr Agilität, Skalierbarkeit und Fehlertoleranz, indem Anwendungen nicht mehr als monolithische Blöcke, sondern als flexibles Netzwerk kleiner, unabhängiger Dienste betrachtet werden. Jeder Dienst kann autonom entwickelt, bereitgestellt und skaliert werden, was eine schnellere Reaktion auf sich ändernde Anforderungen ermöglicht.

Zwar sind die Vorteile verlockend, allerdings bringen Microservice-Architekturen auch neue Komplexitäten mit sich, die es zu meistern gilt. Durch das Schneiden der Applikationen in viele kleine Services erhöht sich die Anzahl der zu wartenden Code-Basen. Jeder Service muss aktuell und zu jeder Zeit kompatibel zu allen anderen Services gehalten werden. Der Betrieb verlangt oft nach Orchestrierung, zum Beispiel über Kubernetes, um mit der Vielzahl an einzelnen Deployments umgehen zu können. Libraries müssen mehrfach aktualisiert werden und Vulnerabilities müssen in allen Services erkannt werden. Zudem muss jeder Service über Observability-Komponenten wie Logging, Monitoring und (distributed) Tracing verfügen, um die Systemgesundheit der gesamten Applikation beurteilen zu können.

Herausforderungen im Detail

Je mehr Services betrieben werden, desto komplexer wird das gesamte System. Jeder Service besteht meist nicht nur aus eigenem Code und Libraries, sondern hat weitere Abhängigkeiten wie Datenbanken, Message Queues und externe Systeme. Kleine Aufgaben, wie das Ändern einer Konfiguration, müssen auf einmal in jedem Service einzeln gemacht werden – das war im Monolithen ohne die Duplikation einfacher.

Wie behält man den Überblick über den Zustand von dutzenden oder gar hunderten von Services? Herkömmliche Monitoring-Ansätze, die sich auf die Überwachung einzelner Server oder Anwendungen konzentrieren, stoßen hier schnell an ihre Grenzen. Man benötigt eine ganzheitliche Sicht auf das System, um Probleme frühzeitig erkennen und beheben zu können. Observability-Systeme sollten daher nicht nur Metriken wie CPU-Auslastung und Speicherverbrauch sammeln, sondern auch Logs analysieren und verteilte Transaktionen verfolgen, um die Auswirkungen von Änderungen auf das gesamte System verstehen zu können.

Jeder Service hat seine eigenen Abhängigkeiten, die regelmäßig aktualisiert werden müssen. Das kann schnell zu sehr hohen Aufwänden führen, wenn dies ohne Systematik und unterstützende Werkzeuge passiert. Inkonsistente Versionen von Bibliotheken oder Frameworks können zu unerwartetem Verhalten in einzelnen Teilen des Systems führen, die sich nur schwer debuggen lassen. Ebenso wird die Reaktion auf Vulnerabilities in den verwendeten Bibliotheken deutlich schwieriger. So sollte bei Bekanntwerden von kritischen Lücken wie log4shell schnellstmöglich erkannt werden können, wo die betroffene Library im Einsatz ist und aktualisiert werden muss.

Ein Fehler in einem Service kann sich über die gesamte Systemlandschaft ausbreiten und das gesamte System lahmlegen. Änderungen an einem Service können Auswirkungen auf andere Services haben. Es muss sichergestellt werden, dass nach Änderungen noch alle Systeme korrekt miteinander kommunizieren können. Die Koordination von Änderungen über mehrere Services hinweg muss sorgfältig geplant sowie orchestriert werden und Schnittstellen daher häufig abwärtskompatibel gestaltet werden.

Software-Design für effizienten Betrieb

Die Grundlage für einen reibungslosen Betrieb bilden einige Prinzipien, die in der Design- und Implementierungsphase eingehalten werden sollten. Diese helfen dabei, die Microservice-Architektur auch in Zukunft effizient betreiben zu können.

Services sollten mithilfe von strategischem Domain-Driven-Design (DDD) geschnitten werden. Eine klare Abgrenzung der Bounded Contexts hilft, Abhängigkeiten zur Laufzeit zu reduzieren. Zudem kann die Ubiquitous Language die Kommunikation innerhalb des Teams sowie zwischen Entwicklungsteam und Fachbereich erleichtern – und das unabhängig vom konkreten Service. DDD hilft, die Anwendung in überschaubare Einheiten zu zerlegen, die jeweils einen bestimmten Geschäftszweck erfüllen. Taktisches DDD ist aufgrund der überschaubaren Menge an Code pro Service hingegen weniger relevant als in großen Monolithen.

Abhängigkeiten zwischen Services sollten minimiert werden, um Änderungen zu vereinfachen und das Kaskadieren von Fehlern einzudämmen. Services sollten ohne synchrone Abhängigkeiten zu anderen Services funktionieren und Daten, wann immer möglich, asynchron replizieren. Event-getriebene Architektur in Verbindung mit einer Message Queue wie Kafka kann helfen, um die Services möglichst lose zu koppeln. Dadurch kann die Auswirkung von Fehlern in einem System gering gehalten werden, da die anderen Services zumindest noch eine Zeit lang mit einem veralteten Datenbestand weiterarbeiten können.

Technologien wie Programmiersprachen, Frameworks, verwendete Bibliotheken und Konventionen der Services sollten so einheitlich wie möglich sein. Das bedeutet aber, dass nicht immer die optimale Technologie für ein bestimmtes Problem gewählt werden kann. Wenn als Tech-Stack beispielsweise Java mit Spring Boot im Einsatz ist, sollten diese Technologien für jeden Service verwendet werden

– auch wenn nur triviale Aufgaben, wie zum Beispiel die Auslieferung einer statischen Datei, erfüllt werden müssen. Was zunächst nicht intuitiv klingt, hilft bei der späteren Wartung enorm, da Updates in allen Services gleich funktionieren. Und auch Services, die nicht mehr weiterentwickelt werden, können von jedem Teammitglied einfach verstanden und aktualisiert werden. Das erleichtert die spätere Weiterentwicklung und Wartung enorm und fördert die Wiederverwendbarkeit von bestehendem Code und Wissen. Auch die On-Call-Bereitschaft erfordert, dass neue Teammitglieder sich schnell in unbekannten Services zurechtfinden können.

Umfassende Unit-, Integrations- und End-to-End-Tests stellen die Qualität der Services sicher. Ein Test-First- oder sogar Test-Driven-Ansatz kann helfen, die Testabdeckung zu erhöhen. Da jeder einzelne Service relativ wenig Fachlichkeit beinhaltet, kann die Gesamtlaufzeit der Tests trotz umfangreicher Testabdeckung minimiert werden. Die CI/CD-Pipeline, die unbedingt implementiert werden sollte, um nicht jeden Service manuell testen und deployen zu müssen, bleibt trotzdem deutlich performanter und gibt schnelleres Feedback, als dies bei einem Monolithen der Fall wäre.

Renovate und Dependabot für Dependency-Management

Jeder einzelne Service bringt Abhängigkeiten in Form von verwendeten Bibliotheken und Frameworks mit sich. Die Vielzahl der Abhängigkeiten kann schnell zu einem unübersichtlichen Dschungel werden. Glücklicherweise gibt es Werkzeuge wie *Renovate* [1] und *Dependabot* [2], die dabei helfen, den Überblick zu bewahren und Abhängigkeiten auf dem neuesten Stand zu halten – wenn gewünscht sogar automatisch. Während sich Renovate als weitgehend unabhängiges Tool gut in eine Vielzahl von Code-Hosting-Plattformen wie zum Beispiel GitLab integrieren lässt, ist Dependabot als mittlerweile von Microsoft betreutes Tool gut in deren Plattform GitHub integriert.

Die Grundfunktion beider Tools besteht darin, Abhängigkeiten in einem Service zu finden und Merge/Pull Requests gegen den Code zu erstellen. Diese können, je nach gewünschter Konfiguration, manuell überprüft oder automatisch gemerged werden. So kann sichergestellt werden, dass nicht-aktuelle Abhängigkeiten sofort erkannt und schnellstmöglich aktualisiert werden können, wodurch alle Services immer mit den neuesten Versionen von Bibliotheken und Frameworks laufen. Eine hohe Testabdeckung und die Verwendung von CI ist essenziell, um schnell – oder sogar automatisiert – entscheiden zu können, ob eine Aktualisierung problemlos durchgeführt werden kann.

Die Konfiguration von Renovate und Dependabot ist einfach und flexibel. Es kann festgelegt werden, welche Dateien neben den Standard-Orten wie pom.xml, build.gradle oder package.json nach Dependencies gescannt werden sollen. Zudem kann bestimmt werden, welche Abhängigkeiten automatisch aktualisiert werden und welche Abhängigkeiten zusammen in einem Merge/Pull Request aktualisiert werden. *Listing 1* zeigt beispielhaft eine Config von Renovate, die Updates der Typen Minor und Patch (nach SemVer) und alle Testabhängigkeiten automatisch merged, wenn die entsprechende Pipeline grün ist. Alle anderen Updates müssen manuell überprüft werden.

```
{
  "packageRules": [
     {
        "matchUpdateTypes": ["minor", "patch"],
        "automerge": true
     },
     {
        matchDepTypes: ["test"],
        automerge: true
     }
  ]
}
```

Listing 1: Beispiel für eine Renovate-Konfiguration

Wenn automatische Updates nicht oder nur teilweise gewünscht sind, entsteht durch diese Tools im Microservice-Umfeld oft ein neues Problem: Die Menge an Merge/Pull Requests, die durch häufige Updates entstehen, kann bei sich rasant entwickelnden Abhängigkeiten schnell überfordern. Hier können wiederum Tools wie gitlab-helper [3] helfen, gleichartige Merge Requests zu identifizieren und auf einen Schwung zu mergen. *Listing 2* zeigt die Verwendung von gitlab-helper, um alle Merge Requests, die im Namen den String „Spring Boot" enthalten, anzuzeigen und zu mergen – falls deren CI-Pipeline grün ist.

meta, ripgrep und sed für Bulk-Änderungen

Manchmal kommt es vor, dass gleichartige Änderungen in allen Services vorgenommen werden müssen. Dies kann zum Beispiel die Änderung eines API-Endpoints sein, der in mehreren Services verwendet wird. Um sicherzustellen, dass alle Vorkommnisse eines zu ersetzenden Strings gefunden und ersetzt werden, bietet sich das Trio *meta* [4], *ripgrep* [5] und *sed* [6] an.

Meta erlaubt es, übliche Befehle wie checkout, commit oder push auf mehreren getrennten Repositories gemeinsam auszuführen. Mit einem gepflegten meta-Repository kann sichergestellt werden, dass alle Repositories lokal vorhanden sind und bearbeitet werden können. In der Code-Hosting-Plattform erfordert meta lediglich ein JSON-File, in dem alle Repositories verlinkt sind, ähnlich wie das bei Git-Submodules der Fall wäre.

Ripgrep (rg) bezeichnet sich selbst als „grep on steroids". Es bietet die Möglichkeit, sehr schnell rekursiv alle Dateien von Git-Reposito-

```
# Zeigt alle Spring-Boot Merge Requests an
gitlab-helper update-merge-requests --search "Spring Boot" list
# Merged die gefunden Spring-Boot Merge Requests
gitlab-helper update-merge-requests --search "Spring Boot" merge --execute
```

Listing 2: Beispielhafter Workflow für gitlab-helper

ries zu durchsuchen und so alle Verwendungen einer Zeichenkette in allen Dateien zu finden. Dies wäre auch mit klassischem „grep" möglich, allerdings vereinfacht und beschleunigt ripgrep die Suche enorm.

Sed ist ein mächtiger Stream-Editor, der unter anderem Zeichenketten in Dateien ersetzen kann – auf Wunsch auch direkt innerhalb von Dateien und für mehrere Dateien gleichzeitig.

Die Kombination der drei Werkzeuge ermöglicht es, sehr effizient Code in unterschiedlichen Repositories zu finden und zu ersetzen. *Listing 3* zeigt, wie die Werkzeuge gemeinsam verwendet werden können, um einen API-Endpoint in allen Dateien aller Repositories zu finden, ändern, committen und dann zu pushen. Die Syntax von sed unterscheidet sich hier zwischen der GNU-Version und der Version für macOS in einem Detail: Bei macOS muss nach dem „-i" noch ein Leerstring in Anführungszeichen angehängt werden, um anzugeben, dass kein Backup erstellt werden soll.

Observability im Microservice-Umfeld

Effektive Systeme für die Observability-Triade – Monitoring, Logging und Tracing – helfen dabei, Probleme im Betrieb frühzeitig und genau erkennen zu können. Gerade im Microservice-Umfeld ist es wichtig, dass die Daten zentral und nicht nur auf den einzelnen, häufig austauschbaren Host-Nodes zur Verfügung stehen.

Die Sammlung von Metriken wie CPU-Auslastung, Speicherverbrauch, Latenz und Fehlerquoten gibt einen Überblick über die Leistung der Services. Ein Monitoring-System sollte die Metriken in Echtzeit visualisieren und bei Überschreitung von Schwellwerten alarmieren können. Solche Systeme können Daten von verschiedenen Services aggregieren und gemeinsam darstellen, sodass Probleme im Gesamtsystem auf einzelne Services heruntergebrochen werden können. Häufig kann dadurch der Root-Cause eines Problems im Gesamtsystem schnell gefunden werden. Als Beispiel für ein solches System sei hier *Prometheus [7]* in Verbindung mit *Grafana [8]* genannt und als kommerzielle SaaS-Alternative bietet sich zum Beispiel *DataDog [9]* an.

Logs enthalten detaillierte Informationen über Ereignisse, die in den Services auftreten. Durch die Analyse von Logs können Fehler erkannt und als Ursachen von Problemen identifiziert werden. Zudem lässt sich das Verhalten von Services und Änderungen in diesen im Laufe der Zeit nachvollziehen. Auf einem Unix-System werden Logs normalerweise nach stdout und stderr geschrieben – diese Logs lassen sich abgreifen und an zentrale Logging-Server weiter-

```
meta git clone git@gitlab.example.com:project/meta.git
cd meta
rg -l "example.com/api/v1" * | xargs sed -i "s|example.com/api/v1|example.com/api/v2|g"
meta git commit -am "Update example endpoint to v2"
meta git push
```

Listing 3: Zusammenspiel von meta, ripgrep und sed

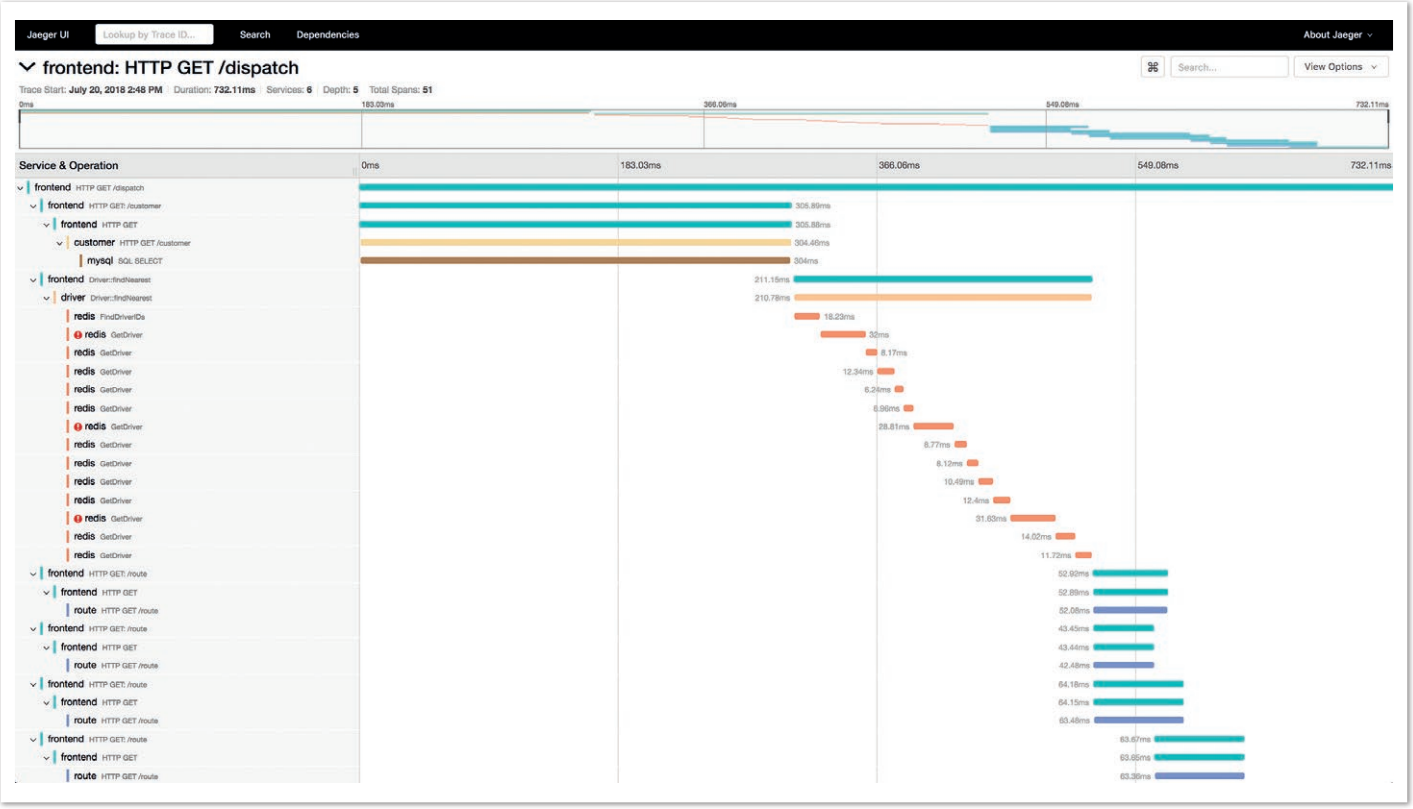


Abbildung 1: Traces aus mehreren Systemen in Jaeger (© [12])

leiten. Es hilft, die Logs in einem einheitlichen Format als „structured logs" zu erzeugen, damit diese auf dem Logging-Server indiziert und durchsuchbar gemacht werden können. Solche Logging-Server können beispielsweise mit *Elasticsearch* [10] oder dessen Fork *OpenSearch* [11] implementiert werden.

In einer Microservice-Architektur durchlaufen Anfragen oft mehrere Services, bevor sie beantwortet, beziehungsweise vollständig verarbeitet werden. Bei synchroner Kopplung kann ein Request weitere Requests zwischen den Services zur Folge haben, bei asynchronen Ansätzen verursacht ein Event entsprechend andere Events. Tracing ermöglicht es, den Weg einer Anfrage oder eines Events durch das gesamte System zu verfolgen und Engpässe oder Leistungsprobleme zu identifizieren. Dadurch kann die Bearbeitungsdauer in jedem Teilsystem analysiert und Bottlenecks gefunden werden. Tracing-Systeme bieten häufig zudem eine Übersicht, welche Systeme bei bestimmten Anfragen und Events zusammenspielen. Beispiele für Tracing-Systeme sind *Jaeger* [12], oder kommerzielle APM-Produkte wie *New Relic* [13] und *Instana* [14]. *Abbildung 1* zeigt den Weg eines Requests durch mehrere Systeme in *Jaeger*.

Im Spring-Boot-Umfeld ist Micrometer mittlerweile der Standard für den Export von Metriken und Traces geworden, logs werden häufig auf Host-Ebene von einem „log-shipper" wie *logstash* [15] oder *fluentd* [16] an die zentralen log-Server gesendet.

Fazit

Microservice-Architekturen sind ein mächtiges Werkzeug, um die Softwareentwicklung zu beschleunigen und Anwendungen flexibler und skalierbarer zu machen. Doch wie bei jeder Technologie, gibt es auch hier Herausforderungen zu meistern.

Die Wartung von Microservice-Architekturen erfordert neue Ansätze. Die Überwachung von einzelnen Servern und Anwendungen funktioniert nicht mehr, es bedarf vielmehr einer ganzheitlichen Herangehensweise. Die Komplexität von Abhängigkeiten muss beherrschbar gemacht werden und Änderungen und Updates von getrennten Services müssen handhabbar sein. Außerdem müssen effektive Observability-Systeme implementiert werden, um den Überblick über die Services zu behalten.

Mit den richtigen Strategien, Werkzeugen und Prozessen können die Vorteile von Microservice-Architekturen ausgeschöpft werden, ohne sich im Dschungel der Wartung zu verlieren. Denn Microservices sind kein Selbstzweck, sondern ein Mittel zum Zweck. Der Zweck ist es, bessere Software zu entwickeln, die den Bedürfnissen der Kunden gerecht wird.

Quellen

[1] *https://github.com/renovatebot/renovate*
[2] *https://github.com/dependabot*
[3] *https://github.com/L7R7/gitlab-helper*
[4] *https://github.com/mateodelnorte/meta*
[5] *https://github.com/BurntSushi/ripgrep*
[6] *https://www.gnu.org/software/sed/*
[7] *https://prometheus.io/docs/introduction/overview/*
[8] *https://grafana.com/*
[9] *https://www.datadoghq.com/*
[10] *https://www.elastic.co/de/elasticsearch*
[11] *https://opensearch.org/*
[12] *https://www.jaegertracing.io/*
[13] *https://newrelic.com/*
[14] *https://www.ibm.com/products/instana*
[15] *https://www.elastic.co/de/logstash*
[16] *https://www.fluentd.org/*

Nikolas Hermann

nikolas.hermann@breuninger.de

Nikolas Hermann ist seit 2019 Software-Engineer bei Breuninger. Als Teil eines Remote-Mob-Programming-Teams arbeitet er mit Begeisterung an der permanenten Weiterentwicklung und dem Betrieb des Online-Shops. Seine Leidenschaft liegt dabei in der Optimierung der Team-Performance durch Dev-Ops-Methoden, Extreme Ownership und synchrone Kollaboration.

Robuste Systeme

Oliver Manz, vaylens GmbH

Moderne Microservice-Architekturen lösen viele technische und organisatorische Probleme und haben daher weite Verbreitung gefunden. Meist entstehen so jedoch komplexe, verteilte Systeme, die mit ganz eigenen Herausforderungen einhergehen. Gerade Anwendungen in kritischen Unternehmensbereichen müssen aber eine hohe Verlässlichkeit aufweisen, da Ausfälle unter Umständen schwerwiegende Folgen haben. Nicht ausgeführte Zahlungen, versäumte Rechtsfristen, fehlende Nebenbucheinträge und vieles mehr können enormen manuellen Aufwand erzeugen, negative Presse bedeuten oder sogar aufsichtsrechtliche Konsequenzen haben. Ist ein Microservice Teil eines kritischen Datenpfades, lohnt es sich, Zeit und Energie in seine Robustheit zu investieren.

Die ANSI definiert Robustness als „*The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.*" *[1]* und nennt damit die beiden Kategorien (invalid inputs, stressful conditions), in die Probleme normalerweise einzuordnen sind:

1. Fehlende, invalide oder semantisch unbrauchbare Daten beziehungsweise syntaktisch geänderte Schnittstellen
2. Ausfälle oder Einschränkungen in der Kommunikation zwischen unserer Software und anderen Microservices oder externen Komponenten (siehe auch „8 fallacies of distributed computing").

Wollen wir einen robusten Service programmieren, sollte dieser ebenso resistent gegenüber defekten Daten und Schnittstellen sein wie gegenüber defekten Netzwerkkomponenten. Die meisten Systeme werden jedoch für den Normalbetrieb entwickelt und selten auf außergewöhnliche Lasten oder unvorhersehbare Ereignisse ausgelegt; es wird nur selten in die Entwicklung von unwahrscheinlichen Edge-Cases investiert.

Ich habe die letzten Jahre vornehmlich Integrationssysteme entwickelt – Services, die die IT-Landschaften zweier Großunternehmen verbinden. Mehrere dieser Services sind Batch-Systeme, die große Datenmengen in einer vorgegebenen Zeit verarbeiten müssen. In einem verteilten System stehen diese Microservices nicht allein, sondern sind auf Logik und Daten vieler anderer Services angewiesen. Darüber hinaus findet die Verarbeitung meistens nachts statt, also in einer Zeit, in der die IT-Operations unbesetzt ist und die Kollegen der Infrastruktur Updates und Systemneustarts planen. Bei den kommenden Beispielen mag meine Historie durchblitzen, aber auch (Near-)Realtime-Services können erheblich davon profitieren, robust entwickelt zu werden.

Allgemeine Grundsätze

Im Allgemeinen kann ein Service umso robuster gestaltet werden, je weniger komplex er ist. Dass kleine Services besser zu beherrschen sind, ist einer der wesentlichen Gründe für den Erfolg der Microservice-Architekturen. Außerdem sind Optimierungen in der Anwendungsperformance deutlich leichter zu realisieren, was bei zeitkritischer Verarbeitung zu mehr Puffer führt.

Nach meiner Erfahrung wird jedes Element, das sich zwischen der eigenen Software und einer Gegenstelle befindet, irgendwann ausfallen, sofern der Service nur hinlänglich lange in Betrieb ist. Diese Ausfälle können kurze Unterbrechungen von wenigen Sekunden sein oder Probleme, deren Behebung deutlich länger dauert, wie abgelaufene Zertifikate, gesperrte technische Benutzer oder dramatische Bandbreiteneinbußen. Ein besonders kurioser Fall war ein Intrusion Prevention System (IPS), das nach Jahren plötzlich den Service blockierte, weil die leicht gestiegene Datenmenge eine Trigger-Schwelle überschritt. Je nach Komplexität der Topologie sind diese Ausfälle mehr oder weniger wahrscheinlich, aber es wird zu Ausfällen kommen! Und: es werden Auslöser dabei sein, mit denen Sie nicht rechnen.

Idempotenz

Ein bedeutender Treiber von Problemen ist die Unkenntnis darüber, ob übertragene Daten im Folgesystem verarbeitet wurden oder nicht. Dies lässt sich umgehen, wenn Sie *Messages*, *Events* und *REST-Aufrufe* idempotent gestalten. Idempotenz bedeutet, dass eine Verarbeitung mehrfach erfolgen kann, ohne das Ergebnis zu verändern. Die Funktion `increaseBy(int step)` ist nicht idempotent, sondern die Variante `changeTo(int targetValue)` ist es. Viele Schnittstellen, die aus fachlicher Sicht nicht idempotent sind, können mit geringem Aufwand diese Eigenschaft erhalten. Nehmen Sie zum Beispiel eine REST-Schnittstelle, die die Anlage eines neuen Geschäftspartners auslöst, und die ID zurückliefert.

In der Regel entscheiden sich die Entwickler des Producers im Falle eines Mehrfachaufrufs für eine von zwei Möglichkeiten:

1. Anlage eines neuen Geschäftspartners, der in allen Eigenschaften mit dem vorher angelegten übereinstimmt.
2. Erzeugen eines Fehlers, wenn Constraints auf einem oder mehreren Geschäftsschlüsseln verletzt ist (zum Beispiel HTTP-Status 409 Conflict).

Nehmen wir an, unser Service erhält beim ersten REST-Aufruf als Consumer nun den HTTP-Status 504 (Gateway Timeout). Wir wissen nicht, ob die Request auf Seiten des Producers verarbeitet wurde und können die Schnittstelle nicht erneut auslösen. Der Dienst kann das Objekt nicht weiterverarbeiten, sofern der Zustand beim Producer nicht geklärt wird. Häufig muss in solchen Situationen ein Mensch eingreifen und eine Admin-Funktion oder gar ein SQL-Statement bemühen, um die Situation zu bereinigen. Genau dies möchten wir mit robusten Systemen vermeiden.

Eine stabilere Methode besteht darin, den Aufruf idempotent zu gestalten. Ein Weg, technische Idempotenz herzustellen, besteht darin, die übermittelten Daten mit einer ID zu versehen (zum Beispiel eine correlation- oder trace-id). Der Producer speichert diese ID bei der Anlage als Referenz auf das auslösende Event oder den

```
private void finalizeOrder(ContractEntity contract) {
  String paymentRef = paymentService.capture(contract.getId());
  String logisticsRef = logisticsService.dispatch(contract, paymentRef);
  contract.setLogisticRef(logisticsRef);

  repository.save(contract);
}
```

Listing 1: Code ohne Safepoint.

auslösenden REST-Aufruf. Wird die Verarbeitung durch den gleichen Auslöser erneut angestoßen, kann der Producer dies erkennen und mit der gleichen Rückgabe antworten wie beim ersten Aufruf. Der Aufruf wird idempotent, zumindest, solange die ID sich nicht ändert.

Safepoints

Nicht immer besteht die Möglichkeit, sich mit dem Producer auf idempotente Schnittstellen zu einigen, vor allem, wenn es sich dabei um Standard-Software handelt oder sich das Entwicklerteam außerhalb der eigenen Organisation befindet. Eine solche Situation ist in *Listing 1* exemplarisch dargestellt. Stellen Sie sich vor, der Aufruf der zweiten Schnittstelle führt zu einem Fehler und eine Exception wird geworfen. Aus Sicht der Anwendung ist der Aufruf beider Schnittstellen atomar, da kein persistenter Zwischenstand vorhanden ist. Aus externer Sicht handelt es sich aber um getrennte Vorgänge. Das Problem verschwindet, wenn nach dem ersten Aufruf ein Safepoint gesetzt wird *(siehe Listing 2)*. Unmittelbar nach dem ersten Aufruf wird der Zustand persistent in der Datenbank abgelegt. Der Prozess kann von diesem Zustand aus (zum Beispiel nach einem Neustart) weitergeführt werden.

Durch geschickte Verwendung des Transaktionsmanagements kann ein ähnlicher Effekt erzielt werden *(siehe Listing 3)* – ohne den Overhead einer zusätzlichen schreibenden Transaktion. Es muss lediglich vermieden werden, dass eine Exception zu einem Rollback führt. Natürlich ist diese Variante nicht so sicher, für viele Fälle aber durchaus ausreichend.

Tolerant Reader

In einer idealen Welt sind Schnittstellen stabil. Sie werden zielgerichtet für einen Consumer eingerichtet und ändern sich deswegen nur, wenn dieser eine Änderung benötigt (Consumer Driven, Interface Segregation Principle). In der Realität arbeiten wir aber auch mit Systemen, deren Lebenszyklus sich unserer Kontrolle entzieht. Hier werden sich die Schnittstellen weiter entwickeln, manchmal auch ohne Vorankündigung. Es ist äußerst ärgerlich, wenn unser Dienst nicht mehr funktioniert, weil in der empfangenen JSON-Datei ein zusätzliches Attribut enthalten ist, das die Deserialisierung verhindert. Viele dieser Probleme lassen sich mittels Konfiguration am Deserialiserungsframework vermeiden. Für Jackson zum Beispiel (ohne Anspruch auf Vollständigkeit):

- FAIL_ON_UNKNOWN_PROPERTIES
- READ_UNKNOWN_ENUM_VALUES_AS_NULL
- ACCEPT_SINGLE_VALUE_AS_ARRAY
- ACCEPT_CASE_INSENSITIVE_PROPERTIES

Manchmal können Probleme bereits antizipiert werden, wenn man zum Beispiel Tipp- oder Rechtschreibfehler in der Schnittstelle findet. Problematisch sind auch Felder, deren Namen von einer sonst

```
private void finalizeOrder(ContractEntity contract) {
  String paymentRef = paymentService.capture(contract.getId());
  contract.setPaymentRef(paymentRef);
  contract = repository.save(contract);

  String logisticsRef =
     logisticsService.dispatch(contract, paymentRef);
  contract.setLogisticRef(logisticsRef);
  repository.save(contract);
}
```

Listing 2: Code mit Safepoint und zwei Transaktionen.

```
@Transactional(noRollbackFor = HttpStatusCodeException.class)
public void finalizeOrder(Long id) {
  ContractEntity contract = loadContract(id); //managed
  String paymentRef = paymentService.capture(contract.getId());
  contract.setPaymentRef(paymentRef);

  String logisticsRef =
     logisticsService.dispatch(contract, paymentRef);
  contract.setLogisticRef(logisticsRef);
}
```

Listing 3: Code mit Safepoint in einer Transaktion.

vorhandenen Struktur abweichen (etwa Singular statt Plural, Nominativ statt Dativ und weitere). In solchen Fällen empfiehlt es sich, die Implementierung so zu wählen, dass auch die sprachlich korrekte Variante akzeptiert werden kann (zum Beispiel mit @JsonAlias).

Arbeit zeitlich entkoppeln

Haben Sie die Metriken Ihrer Dienste jemals überprüft? Ist die Last ihres Systems homogen? Bei den meisten ist dies nicht der Fall. Batchsysteme implizieren das, aber zum Beispiel auch die Ladevorgänge von Elektroautos sind über den Tag heterogen verteilt. Auf Grundlage historischer Daten können wir in vielen dieser Fälle recht zuverlässige Prognosen darüber erstellen, wann solche Lastspitzen auftreten werden.

Vorausgesetzt, wir haben solche Daten – ein Plädoyer dafür, Metriken möglichst lange aufzubewahren. Wenn wir mit ausreichender Wahrscheinlichkeit angeben können, wann unsere Systeme unter Last stehen, können wir vorbereitende Arbeiten einfacher einprogrammieren und gezielt ausführen. Dies ist vergleichbar mit dem Aufwärmen eines Athleten vor einem Wettkampf. Gute Beispiele aus der IT sind: das Extrahieren von Daten aus ihrer relationalen Datenbank in den lokalen Speicher oder das Aufbauen von Caches aus Datensätzen anderer Microservices.

Statt mit anderen Services um die Performance eines zentralen Dienstes (und der Netzwerkkomponenten) zu konkurrieren, erstellt Ihre Software sich rechtzeitig eine Kopie und kann in Ruhe vor sich hinarbeiten. Gerade bei „Langsamdrehern", also Datensätzen mit einer geringen Änderungsfrequenz, ist dies sinnvoll. Reduzieren Sie in Stresssituationen die Abhängigkeiten zu externen Schnittstellen, einschließlich Datenbanken. Dadurch entlasten Sie Ihr Gesamtsystem und verringern das Risiko von Problemen, falls die Gegenstelle ausfällt.

Im ersten Teil haben wir uns Methoden angesehen, wie Systeme von Grund auf robust gestaltet werden können. Doch es gibt auch konkrete Maßnahmen, die bei bestehenden Architekturen und Service eingesetzt werden können.

Retry

Viele Kommunikationsprobleme zwischen Services sind vorübergehender Natur und würden nach einigen Sekunden nicht mehr auftreten. Der Neustart einer Netzwerkkomponente oder eine Lastspitze, die ein Timeout verursacht, sind typische Vertreter solcher Probleme. Spring stellt uns für solche Fälle ein leicht einzusetzendes Werkzeug zur Verfügung: @Retryable.

```
@Retryable(
     noRetryFor = NotFoundException.class,
     maxAttempts = 4,
     backoff = @Backoff(delay = 2, multiplier = 2))
public void updateIamUser(IamUserBO updated) {
  IamUserPatchDTO payload = mapToDTO(updated);
  client.patch(payload);
}

@Recover
private void recoverIam(Exception cause){
  log.error("Communication with IAM system failed.", cause);
  //Handle error here
 ...
}
```

Listing 4: Spring Retry mit einer Recover-Methode

Sobald wir idempotente Aufrufe durchführen, können wir relativ gefahrlos zum Beispiel einen fehlgeschlagenen REST-Aufrufe wiederholen. *Listing 4* zeigt ein typisches Beispiel.

Bei der Implementierung sollten einige Details beachtet werden:

- Überschreiben Sie die Werte für maxAttempts und backoff mit für Ihren Anwendungsfall sinnvollen Werten.
- Permanente Fehler sollte nicht zu einem Retry führen. Ein HTTP-Fehler 400 (Bad Request) wird sich durch wiederholte Aufrufe nicht ändern. Ein „Socket Timeout" oder ein 504 (Gateway Timeout) sehr wohl.
- Out-of-the-box ist Spring Retry blockierend. Während das Framework auf die nächste Ausführung wartet, steht der Thread daher nicht für andere Aufgaben zur Verfügung. Unter Umständen können sich blockierte Threads akkumulieren, bis keine mehr für das reguläre Processing zur Verfügung stehen.
- Rufen Sie möglichst keine mit @Retryable annotierte Methode innerhalb einer aktiven Datenbanktransaktion auf.

Stochastisches Verhalten

Mir ist bewusst, dass Software, die sich nicht deterministisch verhält, zunächst beunruhigend wirken kann. Lassen Sie mich ein Argument aus einem konkreten Fall ableiten. Unsere Datenbank enthält Vertragsdaten, die unter anderem auch Zeitreihen enthalten. Nach dem Eingang neuer Prognosedaten müssen diese Reihen neu berechnet werden. Für beinahe alle Objektgraphen ist dies ohne Probleme möglich, lediglich ein Datensatz ist außergewöhnlich groß und passt in der Folge leider auch nicht in den Heap. Die Anwendung steigt also aus, wird vom Containermanagement beendet und startet neu.

Ist die Reihenfolge der Verarbeitung deterministisch, wird diese erneut beim ursächlichen Objekt beginnen und gleich wieder aussteigen. Meine Lösung: Schauen Sie sich den kleinen Trick in *Listing 5* an. Mit einer einfachen Zeile können wir die Verarbeitung in zufälliger Reihenfolge fortsetzen. Sobald der Service an das problematische Geschäftsobjekt gerät, wird der Fehler natürlich abermals auftreten. Allerdings besteht eine gute Chance, dass alle anderen Datensätze bearbeitet werden, auch, wenn dafür mehrere Neustarts notwendig sind. Unser Service arbeitet praktisch um das fehlerhafte Objekt herum und reduziert damit die Auswirkungen auf andere.

Supervisor

Ein Supervisor ist eine Softwarekomponente, die ausschließlich dafür zuständig ist, den Business-Code zu überwachen und bei Bedarf eine Fehlerbehandlung zu initiieren. Ein Blick zum Sicherungskasten oder den Temperaturschaltern in Geräten verdeutlicht, dass dieses Prinzip in der Technik allgegenwärtig ist. Falls Sie schon mal Mikrocontroller programmiert haben, ist Ihnen vielleicht das Prinzip des Watchdogs bekannt. Wird dieser vom laufenden Code nicht regelmäßig zurückgesetzt, nimmt der Controller an, dass das Programm hängen geblieben ist und führt einen Reset aus.

Was bedeutet das für einen Java-Service? Innerhalb des Programmcodes sollte der Quellcode, der für die Regelverarbeitung zuständig ist, vom Fehlerbehandlungscode getrennt werden.

```
public void updateActiveTimeSeries(PredictionDTO prediction) {
  List<UUID> primaryKeys = loadActiveTimeSeries();

  Collections.shuffle(primaryKeys);

  for (UUID id : primaryKeys) {
        updateTimeSeries(id, prediction);
  }
}
```

Listing 5: Shufflen einer Liste zum Erzwingen stochastischen Verhaltens. Die List-Implementierung muss bearbeitbar (mutable) sein.

Letzterer ist ausschließlich für den Fehlerfall zuständig und gehört deswegen in eigene Klassen beziehungsweise Komponenten. Spring Boot bietet hier sehr mächtige Features (*ControllerAdvice*, *AsyncUncaughtExceptionHandler* und weitere). Außerhalb der JVM arbeitet bei Microservice-Architekturen eigentlich immer ein Container-Management (zum Beispiel Kubernetes), das unsere Anwendung überwacht. Die Definition der Lifeness-Probe sollte eine Selbstverständlichkeit sein. In Spring Boot liefert */actuator/health* per Default schon ein recht umfangreiches Bild und kann Kubernetes veranlassen, eine defekte Anwendung neu zu starten. Sofern die Notwendigkeit besteht, kann man die Aussage über die Gesundheit des Services durch Implementierung von *HealthContributor* mit weiteren Tests anreichern.

Fazit

Ich erinnere mich heute noch an eine Vorlesung in Regelungstechnik, in der unser Professor uns Jungspunden einen Schaltplan für einen PID-Regler präsentierte. Der eigentliche Regler machte nur einen kleinen Teil der Elektronik aus; der größere war der Absicherung der Funktionsweise geschuldet. Die meisten von uns Java-Entwicklern haben den Luxus, dass keine Kessel explodieren, wenn unsere Programme ausfallen und wir häufig damit davonkommen, „nur den Regler" zu implementieren. Gerade aber, wenn wir kritische Anwendungen im Kernbereich unseres Unternehmens oder unserer Kunden programmieren, lohnt es sich, mehr Energie in die Absicherung zu investieren. Das Design der Schnittstellen, die Auswahl des Transports (zum Beispiel Kafka, MQ, REST) und diverse größere oder kleinere Entscheidungen beim Aufbau der Anwendung entscheiden letztendlich über die Robustheit des Services im Einzelnen und der Microservice-Landschaft im Ganzen.

Quellen

[1] IEEE 1990: Std 610.121990 - IEEE Standard Glossary of Software Engineering Terminology, New York

Oliver Manz
vaylens GmbH
oliver.manz@vaylens.com

Oliver Manz ist erfahrener Java-Backend-Entwickler. Vor seinem Wechsel zur vaylens GmbH war er lange als Berater tätig und entwickelte hauptsächlich Integrationssysteme, für die niedrige Latenzen oder hohe Durchsätze maßgeblich sind, und in der Regel unbeobachtet laufen. Als Stack setzt er bevorzugt Spring Boot zusammen mit Kafka ein. Zudem ist Oliver der Überzeugung, dass alles sowieso immer besser werde, wenn man es nur reaktiv programmiere.

Multi-Cloud mit dem Micronaut Framework

Marcel Uusvel-Amende, Oracle Deutschland B.V. & Co. KG

Das Konzept des Cloud Computing bietet schnellen Zugriff auf nahezu unbegrenzte Ressourcen, die bei Cloud-optimierter Entwicklung schonend und effizient genutzt werden können. Drei zentrale Aspekte spielen dabei eine entscheidende Rolle: Anwendungsmodule müssen für dynamische Skalierbarkeit sofort startbereit sein, aus Kostengründen minimalen Verbrauch an CPU und Speicher haben und portabel sein, um eine feste Bindung an einen Cloud-Anbieter zu vermeiden. Die Kombination aus GraalVM und einem geeigneten Microservices-Framework, wie dem Graal Development Kit für Micronaut (GDK), ermöglicht es, diese Anforderungen ohne großen Entwicklungsaufwand zu erfüllen.

GraalVM

Die GraalVM *[1]* ist eine performante, universelle virtuelle Maschine, die Anwendungen ausführen kann, die nicht nur in Java, sondern auch in JavaScript, Python und Ruby geschrieben sind. Im Gegensatz zu den klassischen Open- und Oracle JDKs, bei denen der generierte Bytecode zur Laufzeit interpretiert und in Maschinencode kompiliert wird („Just-in-Time", JIT), kann man bei der GraalVM seinen Code vorab („Ahead-of-Time", AOT) kompilieren und in ein nativ ausführbares Programm umwandeln. Auch die JIT-Kompilierung hat spezifische Vorteile: Der generierte Bytecode ist plattformunabhängig und daher portabel. Zudem kann der JIT-Compiler während der Laufzeit Optimierungen vornehmen, die auf den tatsächlichen Ausführungsmustern des Programms beruhen.

Gerade beim Cloud Computing kommen allerdings die Vorteile der AOT-Kompilierung, auch als „GraalVM Native Image" bezeichnet, zum Tragen:
Die nativ ausführbaren Programme benötigen weniger Speicher und CPU. Sie starten schneller, quasi augenblicklich, und ohne Aufwärmphase, um mit voller Leistung zu laufen. Sie sind kleiner, da sie nur die benötigten Klassen und Methoden beinhalten, und benötigen keine Laufzeitumgebung (JRE), wodurch sie in Container-Images mit minimaler Größe gepackt werden können.

Bei einer vergleichenden Performance-Messung wird man auf den ersten Blick überrascht sein, wenn die JIT-kompilierte Applikation durch mögliche Laufzeitoptimierungen einige Prozentpunkte leistungsfähiger erscheint als die AOT-Variante. Letztere braucht aber tatsächlich so viel weniger CPU und Memory, dass man auf den freien Ressourcen weitere Instanzen starten kann, um die Gesamtleistung der vorkompilierten Applikation zu vervielfachen.

Wegen der unübersehbaren Vorteile bieten die populären Frameworks Spring Boot, Micronaut, Helidon und Quarkus, wie viele andere, mittlerweile erstklassige Unterstützung für die GraalVM. In der Folge werden wir uns auf Micronaut *[2]* konzentrieren, da es insbesondere für schnelle Startzeiten und geringen Speicherverbrauch optimiert ist. Die ausgeprägten Multi-Cloud-Fähigkeiten, die es durch das Graal Development Kit für Micronaut *[3]* erhält, werden ein weiterer Schwerpunkt sein.

Micronaut

Bei Micronaut handelt es sich um ein JVM-basiertes Open-Source-Projekt unter Apache-V2-Lizenz, mit dem man auf einfache, Annotations-basierte Weise leichtgewichtige, modulare, leicht test- und wartbare Microservice-Applikationen bauen kann. Java, Kotlin, Groovy und demnächst auch Scala können als Programmiersprachen genutzt werden.

Micronaut geht neue Wege, um CPU- und Speicherverbrauch zu minimieren: Auf „Runtime Reflection" und Proxy-Generierung wird verzichtet. Stattdessen nutzt man „Dependency Injection" und Aspekt-orientierte Programmierung bereits während der Kompilierung, wodurch es eine natürliche Eignung für die GraalVM und die AOT-Kompilierung in Native Images mit sich bringt.

Die Installation von Micronaut lässt sich am einfachsten mit dem Verwaltungswerkzeug SDKMAN! *[4]* vornehmen *(siehe Listing 1)*, das einen einfachen Wechsel zwischen verschiedenen Software Development Kits und Versionen ermöglicht. Es läuft auf allen UNIX-basierten Systemen, zu denen auch MacOS und das Linux-Subsystem für Windows gehören.

```
>#  SDKMAN! Installieren
>curl -s "https://get.sdkman.io" | bash
>#  Umgebungsvariablen setzen
>source "$HOME/.sdkman/bin/sdkman-init.sh"
>#  Neueste Version von Micronaut installieren
>sdk install micronaut
>#  Micronaut 4.5 zur Verwendung auswählen und testen
>sdk use micronaut 4.5.0
>mn --help
```

Listing 1: Installation von Micronaut mit SDKMAN! per Kommandozeile

Das Micronaut Framework bietet verschiedene Möglichkeiten zur Erstellung, Entwicklung und Bereitstellung von Microservices. Neben der Kommandozeile können der Micronaut Launcher und Visual-Studio-Code-Erweiterungen genutzt werden. Bei der Projekterstellung per Kommandozeile wählt man Gradle oder Maven als Build-Werkzeug, die gewünschte Programmiersprache (zum Beispiel Java) und zusätzlich benötigte Features, wie etwa einen HTTP-Server *(siehe Listing 2)*.

Für die Erstellung eines Graal Native Image installiert man zusätzlich die GraalVM, beispielsweise per SDKMAN!, und fügt eine GraalVM-Plug-in-Konfiguration (build.gradle) zu seinem Micronaut- Projekt hinzu *(siehe Listing 3)*.

Alternativ steht der Micronaut Launcher als Webanwendung zur Verfügung, der das Erstellen neuer Micronaut-Projekte vereinfacht. Er kann direkt über die Micronaut Website *[4]* genutzt oder lokal in-

```
>#  Java Micronaut App mit Gradle Build erstellen
>mn create-app my.micronautapp --build=gradle --lang=java --features=netty-server
>cd my.micronautapp
>#  GraalVM mit SDKMAN! installieren
>sdk install java 22.1.r17-grl
>sdk use java 22.1.r17-grl
>#  Graal Native Image erstellen
>./gradlew nativeCompile
>#  Native App starten
>build/native/nativeCompile/micronautapp
```

Listing 2: Erstellen und Kompilieren eines Micronaut-Projekts per Kommandozeile

```
import io.micronaut.http.annotation.*;
import java.util.Collections;
import java.util.Map;

@Controller("/hello")
class MyController {

     @Get
     Map index() {
           return Collections.singletonMap("message", "Hello Micronauts.");
     }
}
```

Listing 4: Non-blocking HTTP-Server mit einfachen Annotationen erstellen

stalliert werden *(siehe Abbildung 1)*. Man wählt in der GUI aus Listen die gewünschten Konfigurationsparameter und lädt das vorkonfigurierte Rumpfprojekt als Zip-Datei herunter, auf Wunsch auch mit Code-Beispielen.

Wer seine IDE nicht verlassen möchte, dem stehen eine Reihe von Visusal-Studio-Code-Erweiterungen zur Verfügung, die als „Graal Development Kit for Micronaut Extension Pack" auch gebündelt erhältlich sind:

- „Launch for Micronaut framework": Eine leichtgewichtige Erweiterung für das Erstellen von Micronaut Applikationen in Visual Studio Code.
- „Tools for Micronaut framework": Hilfreiche Funktionalitäten für den Entwickler, zum Beispiel Auto-Vervollständigung, Code-, Bean- und Endpunkt-Navigatoren.
- „Graal Development Kit for Micronaut Launcher": Erstellung von Projekten, die GDK-Cloud-Module beinhalten.

Der Codierungsaufwand ist bei Micronaut gering. Eine HTTP-API lässt sich beispielsweise mit nur zwei Annotationen *(siehe Listing 4)* erstellen, die einen einfachen Web-Controller und eine REST-Methode definieren. Im angeführten Beispiel wird auf eine HTTP-GET-Anfrage reagiert und eine simple Nachricht zurückgegeben. Um viele Anfragen gleichzeitig verarbeiten zu können, ohne dass eine Anfrage eine andere blockiert, nutzt Micronaut für eine effiziente

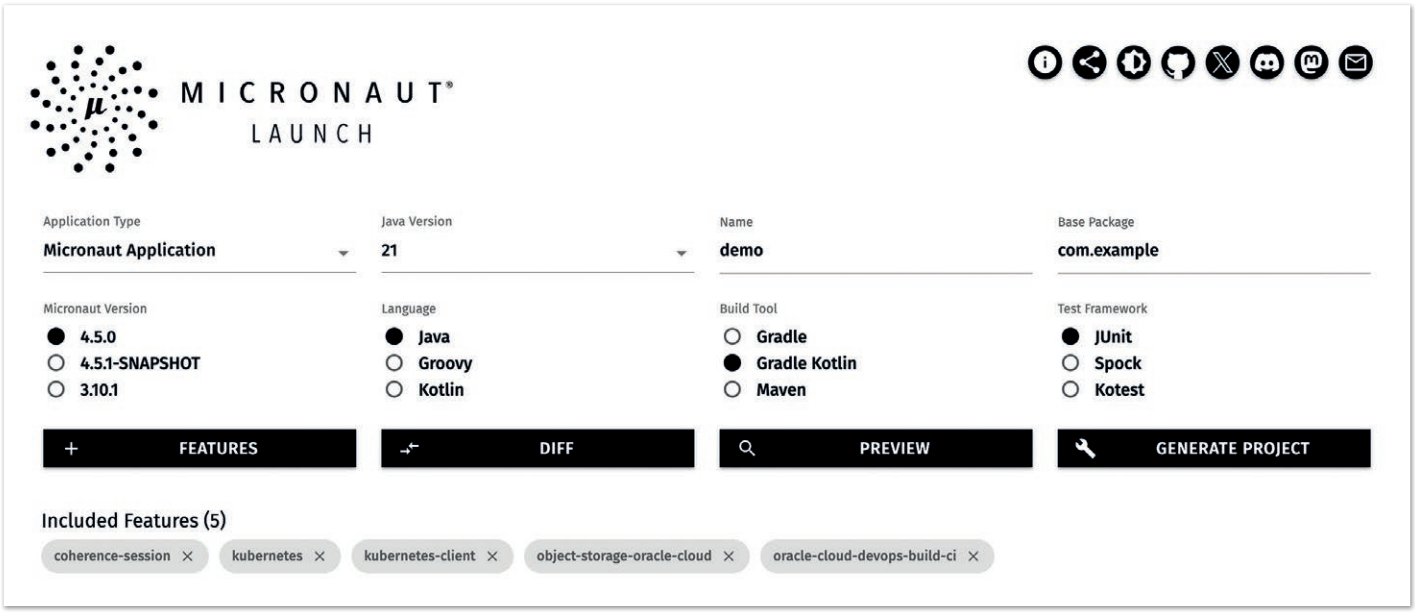


Abbildung 1: Projekterstellung mit dem Micronaut Web-Launcher (© Marcel Uusvel-Amende)

Verarbeitung von Webanfragen einen non-blocking HTTP-Server, der auf der Netty-Bibliothek basiert.

```
plugins {
     id 'org.graalvm.buildtools.native' version '0.9.10'
}

graalvmNative {
     binaries {
            main {
                  fallback = false
            }
     }
}
```

Listing 3: GraalVM-Plug-in-Konfiguration (build.gradle) in Micronaut

Eine JIT-Kompilierung resultiert bei einer solch einfachen Micronaut-REST-Applikation in ein zirka 11 MB großes Java-Archiv, das sofort auf einer aktuellen HotSpot JVM ausgeführt werden kann. Der kleine Webserver startet auf einer typischen Entwicklungsmaschine in zirka 300 ms. 5000 Requests, verteilt auf 5 parallele Threads, verarbeitet er bei einer Speicherauslastung von 149 MB und einer CPU-Zeit von 1,6 s in 440 ms, vorgewärmt in 387 ms.

Bei der AOT-Kompilierung mit GraalVM Native Image entsteht eine zirka 70 MB große, direkt nativ ausführbare Datei, die keine Laufzeitumgebung mehr benötigt. Der Webserver ist bereits nach 8 bis 12 ms (30x schneller) nach dem Start bereit, Requests anzunehmen. Für die Abarbeitung desselben Testfalls werden nun bei einer Speicherauslastung von 78 MB (minus 48 %) und einer CPU-Zeit von 0,12s (minus 92 %) nur noch 110ms (fast 4x schneller) benötigt. Man erreicht, gemäß der Erwartung, eine performantere Applikation bei deutlich geringerer Ressourcennutzung.

Es gibt aber noch weitere Argumente für die Verwendung des Micronaut Frameworks in der Cloud: Durch sein flexibles Konfigurationsmanagement kann man Cloud-native-Applikationen einfach an die verschiedensten Umgebungen adaptieren. Funktionalitäten für Service Discovery, Distributed Tracing, Load Balancing und Service Communication sind bereits beinhaltet.

Graal Development Kit für Micronaut

Eine wachsende Zahl von Unternehmen verfolgt eine Multi-Cloud-Strategie. Die Vermeidung der Abhängigkeit von nur einem einzelnen Cloud-Anbieter erhöht die Redundanz und Ausfallsicherheit. Man gewinnt die Möglichkeit, kosteneffizient die günstigsten oder funktional die leistungsstärksten Dienste aus dem Pool der Anbieter auszuwählen. Auch regulatorische Vorgaben, die verlangen, dass Daten in einer bestimmten Region oder in einer spezialisierten Cloud-Implementierung, wie einer „Sovereign Cloud", „Governance Cloud" oder einer „Cloud at Customer", gespeichert werden müssen, machen Multi-Cloud-Strategien notwendig. Eine Studie aus dem Jahr 2023 *[5]* erhebt sie zum „Goldstandard" und schätzt, dass im Jahr 2026 bereits 50 % der deutschen Unternehmen auf eine Multi-Cloud-Strategie setzen werden.

Natürlich hat eine Multi-Cloud-Strategie nicht nur Vorteile. Sie bringt auch mehrfache Onboarding-Aufwände und zusätzliche Komplexität in der Verwaltung von Cloud-Diensten mit sich. „Infrastructure as Code", die Automatisierung der Cloud-Infrastruktur-Provisionierung mit Terraform oder Ansible, die die Cloud-APIs verschiedener Anbieter abstrahieren, ist hierbei hilfreich.

Auch auf Entwickler kommen neue Herausforderungen zu: Anwendungen müssen so geschrieben werden, dass sie problemlos zwischen verschiedenen Cloud-Umgebungen verschoben werden können. Bis auf Kubernetes und das S3-Protokoll gibt es bislang kaum standardisierte Cloud-APIs, die dabei helfen könnten, portablen

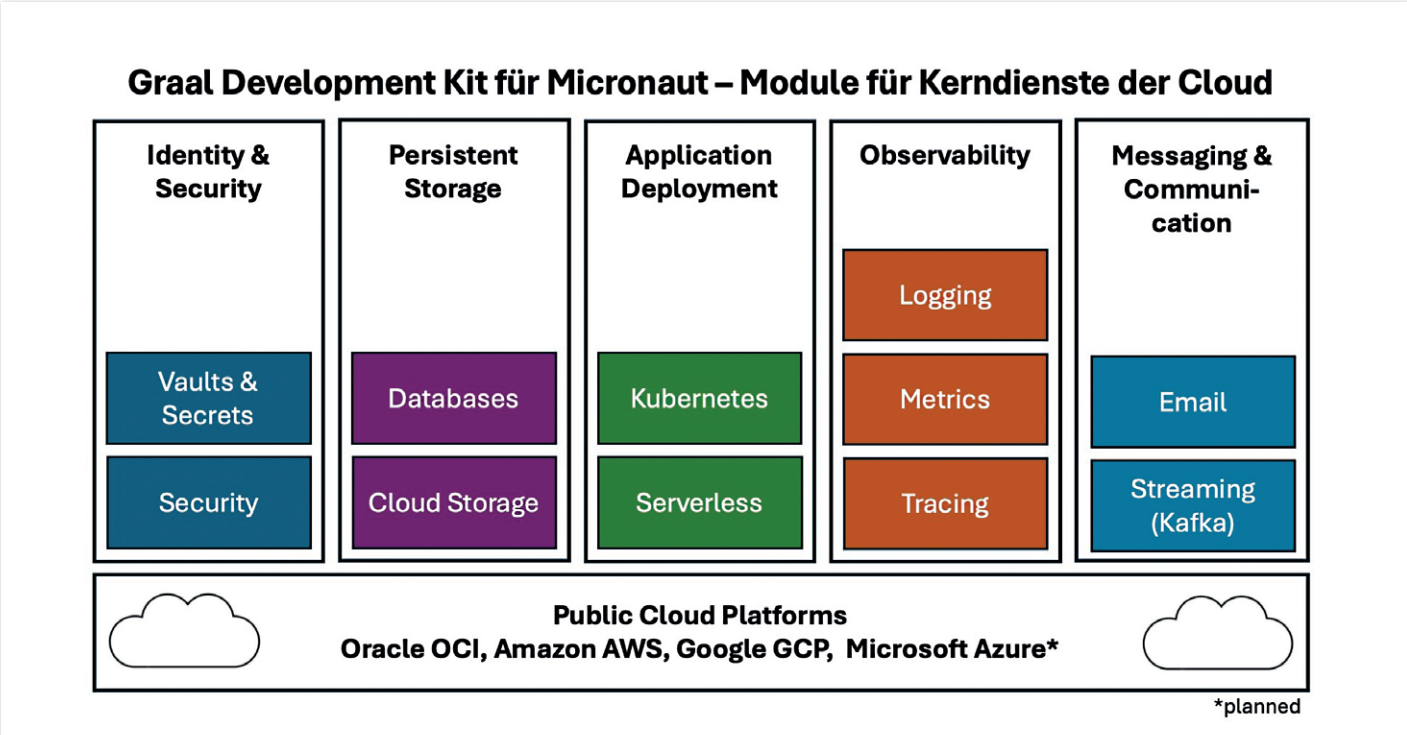


Abbildung 2: Übersicht der GDK-Module (© Marcel Uusvel-Amende)

Code zu schreiben, der Plattform-Dienste verschiedener Anbieter nutzen kann.

Das Graal Development Kit für Micronaut, ein kuratierter Satz von Open-Source-Framework-Modulen und löst die Herausforderung der Portabilität durch eine Reihe von Modulen für die Entwicklung von Microservices und die Einbindung zentraler Cloud-Dienste *(siehe Abbildung 2)*. Dazu gehören Module für die Einbindung von Datenquellen, Kubernetes und Serverless, Logging, Object Storage, Streaming, E-Mail, Security und Secrets Management, Metrics und Tracing *(siehe Abbildung 2)*. Unterstützt werden jeweils Dienste in Amazon AWS, Google GCS, Oracle OCI und demnächst auch Microsoft Azure.

Beispielhaft sehen wir uns an, wie das GDK-Modul „Data" das Konfigurationsmanagement von Micronaut nutzt, um verschiedene Datenquellen einzubinden.

GDK für das Micronaut-Modul „Data"

Mit dem Micronaut-Modul *[6]* „Data" kann man effizient auf verschiedene Arten von Datenbanken (JPA, SQL, MongoDB) zugreifen. Für eine MySQL-Datenbank wählt man bei der Erstellung der GDK-Applikation einfach den Service „Database". Die benötigten Features werden automatisch hinzugefügt. Für eine Oracle Autonomous Database geht man ähnlich vor, wählt aber zusätzlich die Cloud „OCI" und das Feature „oracle-cloud-atp" *(siehe Listing 5)*.

```
>gdk create-app sample.my-adb \
      --clouds=oci \
      --services=database \
      --features=graalvm,oracle-cloud-atp \
      --build=gradle \
      --jdk=17 \
      --lang=java
```

Listing 5: Generierung einer GDK-Datenbank-Applikation

Username, Passwort, Datenbankname, Wallet und weitere, je nach Typ der Datenquelle benötigte Zugangsparameter, können als Umgebungsvariablen gesetzt und übergeben werden.

Zentral ist die Definition von „Repository Interfaces", also von Schnittstellen, die Methoden definieren, um Daten aus einer Datenbank zu speichern und abzurufen *(siehe Listing 6)*. Micronaut Data implementiert diese dann während der Kompilierung. Die zugehörigen Datenbankabfragen werden schon im Voraus während der Kompilierung und somit bevor sie eigentlich benötigt werden, berechnet, was sie schneller und weniger ressourcenintensiv macht. Ausgeführt werden sie dann in einer leichtgewichtigen Laufzeit-Schicht.

Zusätzlich benötigt man noch eine „Domain Entity", eine Java-Klasse, die die Struktur der Datenbanktabelle (in diesem Beispiel „sample.MyTable") repräsentiert. Bei der Projekterstellung kann man sich zur Vereinfachung Beispielcode generieren lassen.

GDK für das Micronaut-Modul „Object Storage"

Zum Abschluss betrachten wir noch die Einbindung von „Object Storage", einem typischen Cloud-nativen Service, den es in ähnlicher Form bei allen wichtigen Cloud-Anbietern gibt. Das GDK-Modul „Object Storage" *[7]* kann die Nutzung von (Amazon) AWS S3, Google Cloud Storage, (Oracle) OCI Object Storage und (Microsoft) Azure Blob Storage vereinheitlichen.

Die Micronaut-Object-Storage-API stellt eine einheitliche API zur Verfügung, um Objekte zu erstellen, lesen und löschen. Zugriffe auf alle genannten Object-Storage-Dienste basieren auf dem S3-Protokoll, nutzen aber die spezifischen Sicherheitsmechanismen und APIs der verschiedenen Anbieter. Daher ist ein Zugang zu den verwendeten Clouds Voraussetzung, aber auch die lokale Installation der entsprechenden Cloud Command Line Interfaces (CLI), namentlich AWS CLI, Google Cloud CLI und/oder OCI CLI.

Danach kann man ein Bean des Typs „ObjectStorageOperations" in einen Controller oder Service der Wahl injizieren. Dieses Interface ist der „Parent", der die generische Nutzung für alle Cloud-Anbieter ermöglicht. Ist eine Applikation nicht Multi-Cloud oder nutzt sie Cloud-spezifische Implementierungsdetails, kann man auf eine konkrete Implementierung zurückgreifen, im Falle von AWS zum Beispiel „AwsS3Operations".

Natürlich kann Object Storage mehrfach und von verschiedenen Anbietern gleichzeitig konfiguriert sein. Mit „Bean Qualifiers" kann

```
package sample.repository;

import sample.MyTable;
import io.micronaut.core.annotation.NonNull;
import io.micronaut.data.annotation.Id;
import io.micronaut.data.jdbc.annotation.JdbcRepository;
import io.micronaut.data.repository.PageableRepository;
import jakarta.validation.constraints.NotBlank;
import static io.micronaut.data.model.query.builder.sql.Dialect.MYSQL;

@JdbcRepository(dialect = MYSQL)
public interface MyTableRepository extends PageableRepository< MyTable, Long> {

     MyTable save(@NonNull @NotBlank String name);
     long update(@Id long id, @NonNull @NotBlank String name);
}
```

Listing 6: Repository Interface für die Interaktion mit einer MySQL-Tabelle.

man definieren, wann und auf welchen Storage Bucket zugegriffen werden soll.

Fazit

Das Graal Development Kit für Micronaut ist eine leistungsstarke Lösung für die Herausforderungen von Multi-Cloud-Umgebungen. Durch die Kombination der effizienten GraalVM und des flexiblen Micronaut-Frameworks können Entwickler hoch-performante, portierbare und skalierbare Microservices erstellen. Die Vielzahl an verfügbaren Micronaut-Modulen und deren einfache Konfiguration machen es Entwicklern leicht, komplexe Cloud-Anwendungen zu erstellen und zu verwalten. Dies macht das Graal Development Kit für Micronaut zu einer guten Wahl für moderne Cloud-Strategien.

Quellen

[1] *https://www.graalvm.org/*
[2] *https://micronaut.io/*
[3] *https://graal.cloud/gdk/*
[4] *https://micronaut.io/launch*
[5] *https://www.computerwoche.de/a/hybride-multi-cloud-wird-zum-goldstandard*
[6] *https://graal.cloud/gdk/modules/*
[7] *https://micronaut-projects.github.io/micronaut-object-storage/latest/guide/*

Marcel Uusvel-Amende
Marcel.Amende@oracle.com

Marcel Uusvel-Amende, Dipl.-Ing. der Elektrotechnik, feiert dieses Jahr sein 25-jähriges Firmenjubiläum bei Oracle. Als Master Principal Cloud Engineer und Spezialist für Integrations- und Entwicklungsthemen bringt er seine umfassende Erfahrung ein, um Kunden des Öffentlichen Dienstes bei der Implementierung hochsicherer Lösungsarchitekturen und auf dem Weg in die Cloud zu unterstützen.

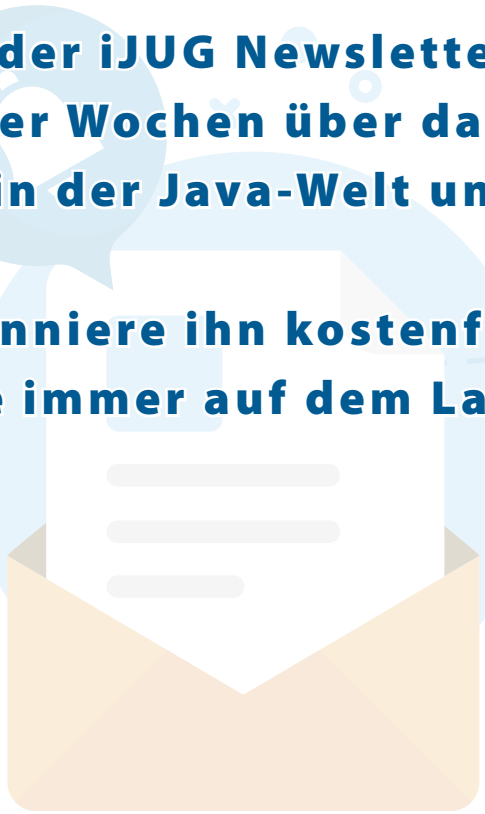
iJUG Verbund

NEWSLETTER

Anmeldung

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

Abonniere ihn kostenfrei und bleibe immer auf dem Laufenden!



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

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

Was können 4 Threads, was 600 nicht können? Reaktive Webservices mit WebFlux & Reactor

Jonas Nagy-Kuhlen, andrena objects ag

Performante Webservices sind die fundamentalen Bausteine jeder Microservice-Architektur. Das klassische synchrone Modell, also die Reservierung eines Threads pro Request, erfüllt die hohen Anforderungen an skalierbare Systeme schon längst nicht mehr. Dieser Artikel stellt den reaktiven Technologie-Stack basierend auf WebFlux und Reactor vor und zeigt, wie mit modernen, reaktiven Java-Anwendungen Performance-Bottlenecks effektiv vermieden werden können.

Damit Webservices ihren Nutzen entfalten, müssen sie mit der Außenwelt kommunizieren – entweder mit anderen Webservices oder, im Fall von Frontends, mit dem Browser der Anwenderinnen und Anwender. Das gilt in besonderem Maße für Microservice-Architekturen: Die Customer Journey ergibt sich aus dem Zusammenspiel einer Vielzahl von Webservices, die sich jeweils auf einen fachlichen oder technischen Aspekt fokussieren und über das Netzwerk miteinander kommunizieren. Während Webservices also in der Architektur die fundamentalen Bausteine bilden, hat sich HTTP als gemeinsame und plattformunabhängige „Sprache" der Services untereinander etabliert.

Das Problem mit dem synchronen Threading-Modell

Stellen wir uns ein beispielhaftes E-Commerce-System vor, dessen Architektur in *Abbildung 1* dargestellt ist. Zentraler Service ist hier der *Service Produktzusammenstellung*, der Anfragen von außen mit einer dazu passenden Zusammenstellung fiktiver Produkte beantwortet. Dafür ist hier im ersten Schritt das Abholen einer aktuellen Liste mit Produkt-IDs notwendig, bevor die Produkte (in Standard- und Spezialausführung) von weiteren Services abgefragt und abschließend zusammengestellt werden können.

Abbildung 2 zeigt die Aufrufkette durch die Servicelandschaft hindurch, die zur Abarbeitung jedes eingehenden Requests von außen durchlaufen wird. Die theoretische Parallelisierbarkeit (gegeben durch die exemplarische Fachlichkeit) wird hier deutlich: Sobald die Liste der Produkt-IDs vorliegt, können die Produkte selbst unabhängig voneinander von den zwei Produktservices abgefragt werden. Dabei ist für den *Service Produktzusammenstellung* vollkommen transparent, dass der *Service Spezialprodukte* wiederum eine Drittanbieter-API anfragt, um seine Aufgabe zu erfüllen.

Der untere Balken zeigt grün schraffiert die tatsächlichen CPU-seitigen Berechnungen unseres Services sowie grau die faktischen Wartezeiten auf Antworten anderer Services. Die Berechnungen können sowohl technischer Natur (Serialisierung der eingehenden Anfrage beziehungsweise Deserialisierung der ausgehenden Antwort) wie auch fachlicher Natur (Zusammenstellung der erhaltenen Produkte) sein. Zwar sind die Zeitanteile in diesem Beispiel willkürlich gewählt – dennoch wird sofort deutlich, dass unsere Webservices einen signifikanten Anteil der Bearbeitungszeit für einen eingehenden Request darauf warten, dass ausgehende Requests von anderen Services beantwortet werden.

Wartezeiten lassen sich in einer verteilten, nicht-monolithischen Servicearchitektur kaum vermeiden. Allerdings hängt die Performance des Service wesentlich davon ab, wie die zur Verfügung stehenden Ressourcen währenddessen genutzt werden. Webservices arbeiten nicht einen, sondern eine große Anzahl eingehender Requests gleichzeitig ab. Um diesen Grad an Parallelität zu erreichen, setzen herkömmliche Webserver wie zum Beispiel *Apache Tomcat*

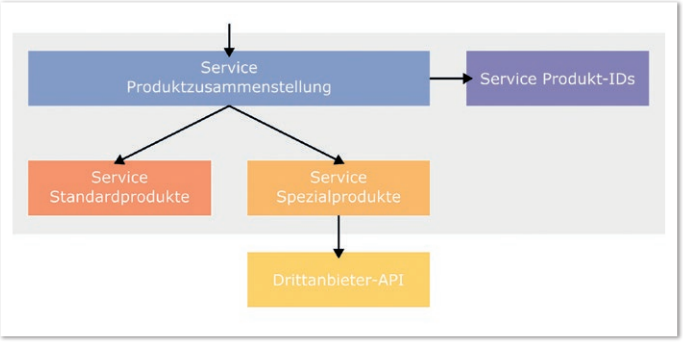


Abbildung 1: Beispielhafte Servicearchitektur als Teil eines fiktiven E-Commerce-Systems (© Jonas Nagy-Kuhlen)

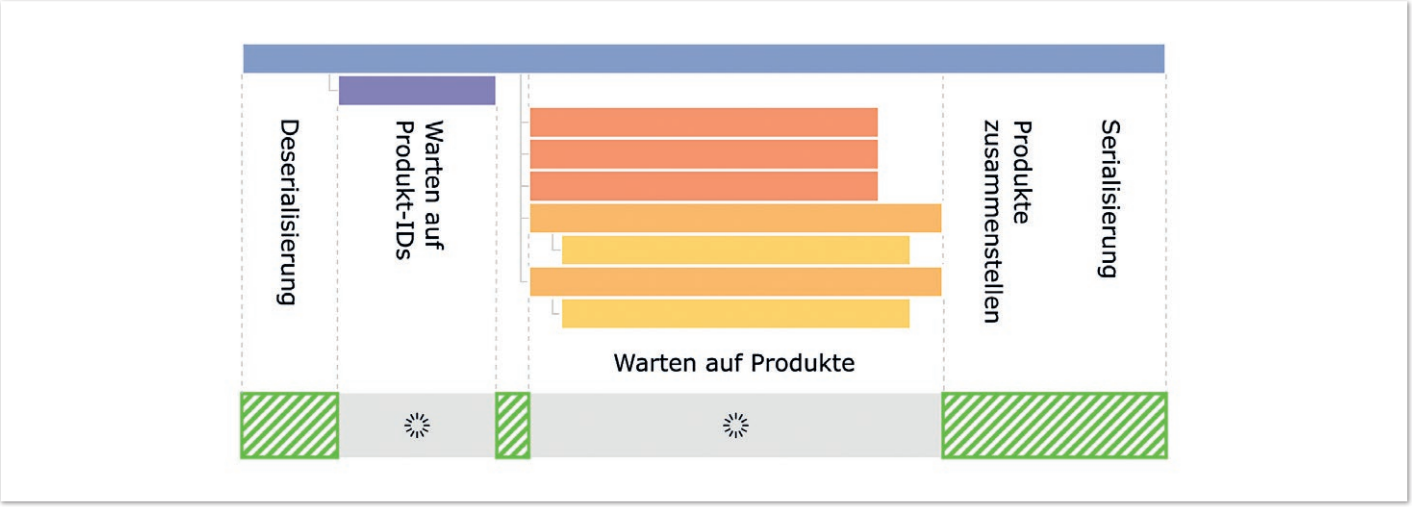


Abbildung 2: Aufrufkette der beteiligten Services sowie CPU-seitige Berechnungen (© Jonas Nagy-Kuhlen)

Abbildung 3: Zuteilung von Requests auf Threads im Thread-Per-Request-Modell (© Jonas Nagy-Kuhlen)

auf einen großen Pool von Worker Threads, die eingehenden Requests für die Dauer der Bearbeitung fest zugewiesen werden. Das sich daraus ergebende Programmierparadigma ist fest in weit verbreiteten Frameworks wie Spring Web MVC verankert *[1]*.

Nehmen wir an, unser Beispielservice basiert auf diesem synchronen *Thread-Per-Request-Modell*. Parallele eingehende Requests werden der Reihe nach freien Threads zugeteilt, wie *Abbildung 3* für einen beispielhaften Thread Pool mit vier Threads illustriert.

Für den fünften eingehenden Request steht kein freier Thread mehr zur Verfügung, sodass die Bearbeitung so lange pausiert, bis der nächste Thread (hier Thread 1) mit der Bearbeitung seines aktuellen Requests fertig ist. Wie wir sehen, hat sich die Antwortzeit des fünften Requests aus Sicht von außen fast verdoppelt – im schlimmsten Fall entscheiden potenzielle Kundinnen und Kunden anhand dieser User Experience, ob sie auf unserer Website bleiben oder entnervt wegnavigieren. Besonders problematisch dabei: Als unser Service den Request empfangen hat, befanden sich alle vier Threads im Leerlauf und haben lediglich auf Antworten von anderen Services gewartet. Es hätte also theoretisch ausreichend CPU-Leistung für die Bearbeitung zur Verfügung gestanden.

Natürlich ist der Threadpool im Normalfall deutlich größer. *Apache Tomcat* stellt standardmäßig 200 Threads bereit *[2]* und erlaubt, eine noch höhere Zahl zu konfigurieren – ein hartkodierter Wert von 600 Threads ist mir in produktiven Systemen schon begegnet. Doch damit wird das Kernproblem nicht gelöst: Die Anzahl von Requests wird immer durch den Thread Pool limitiert anstelle der real zur Verfügung stehenden Hardware-Ressourcen. Für parallele ausgehende Requests sind wiederum dedizierte Threads aus einem (normerweise separat verwalteten) Thread Pool nötig. Die richtige Dimensionierung der Thread Pools in Abhängigkeit von den Hardware-Ressourcen ist kompliziert und hängt stark von den Charakteristika der individuellen Anwendung ab. Moderne Anwendungen werden inzwischen oft mittels *Docker* und *Kubernetes* oder vergleichbaren Technologien containerisiert beziehungsweise orchestriert. In dem Fall sind die dem Service zugeteilten physischen Ressourcen extremen Schwankungen durch die Infrastruktur unterworfen, sodass eine darauf abgestimmte Konfiguration der Thread Pools nahezu unmöglich wird.

Hinzu kommt noch, dass Threads teure Betriebssystem-Ressourcen darstellen, die mit erhöhtem Speicherverbrauch und Context Switches einher gehen *[3]*. Je größer die Thread Pools, desto mehr CPU-Zeit wird für Context Switches aufgewendet, die nicht zur Ausführung von Anwendungscode beiträgt.

Event Loop to the Rescue?

Die Nutzung von CPU-Ressourcen bei netzwerkbedingten Wartezeiten ist also alles andere als optimal, wenn Threads fest einem Request zugewiesen werden. Lässt sich ein effizienteres Modell finden, indem man dem Webserver erlaubt, die Abarbeitung eines eingehenden Request auf mehrere Threads aufzuteilen? In der Tat basiert das Event-Loop-Modell genau darauf: In der *Event Queue* werden Workload-Items abgelegt und die Threads im Thread Pool arbeiten diese nacheinander ab. In unserer Beispielanwendung würde ein eingehender Request die Deserialisierungslogik und das Abschicken der Anfrage der Produkt-IDs als Workload Item ablegen. Sobald die Antwort über das Netzwerk eingetroffen ist, wird deren Bearbeitung als neues Workload Item in der Event Queue platziert. Auf diese Weise setzt sich die Verarbeitung eines eingehenden Request aus mehreren kleinen Workload-Items zusammen, die den Worker-Threads beliebig zugeteilt werden können. Dieses Verhalten ist in *Abbildung 4* dargestellt.

Der zweite Request, hier gelb eingefärbt, startet die Verarbeitung auf Thread 2 und setzt diese nach Erhalt der ersten Antwort auf Thread 1 fort. Obwohl der Thread Pool weiterhin nur aus vier Threads besteht, ist nun die Verarbeitung von acht in kurzer Folge hintereinander eintreffenden Requests ohne jegliche Verzögerung möglich. Der kaum ausgelastete *Thread 4* macht außerdem deutlich, dass noch ausreichend CPU-Ressourcen für weitere Requests zur Verfügung stehen.

Dieses Modell hat mehrere Vorteile. Der Durchsatz, also die Anzahl gleichzeitig verarbeiteter eingehender Requests, ist nun unabhängig von der Größe des Thread Pools. Nur noch der tatsächliche Workload bestimmt die CPU-Last. Umgekehrt heißt das: Wartezeiten kommen nahezu ohne Ressourcen-Kosten, da sie den Thread nicht mehr blockieren und der Thread Pool für die Verarbeitung anderer Workload-Items frei bleibt. Wie ist der Thread Pool also für das Event-Loop-Modell zu dimensionieren? Es bietet sich an, nur so vie-

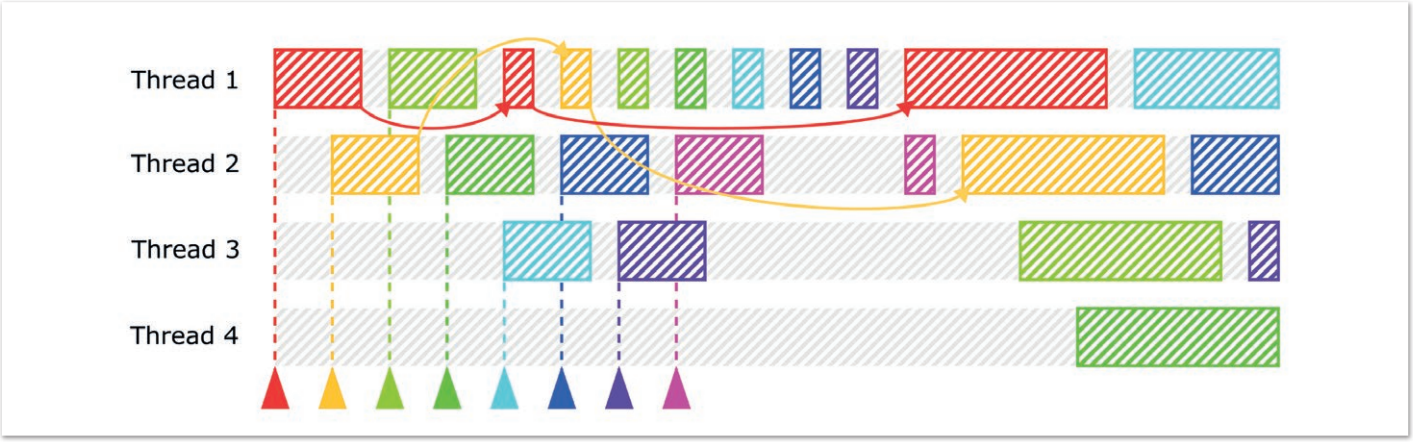


Abbildung 4: Zuteilung von Workload Items auf Threads im Event Loop-Modell (© Jonas Nagy-Kuhlen)

le Threads zu erzeugen, wie physische Prozessoren zur Verfügung stehen. Auf diese Weise wird echte Parallelität auf Hardware-Ebene erreicht. Alle weiteren Threads konkurrieren mit mindestens einem anderen Thread um CPU-Zeit und verursachen Kosten in Form von Context Switches, ohne die Abarbeitung zu beschleunigen. Daher ist auch die Anzahl der Prozessoren im Reactor Core als Standardwert für die Thread Pool-Größe konfiguriert *[4]*.

Für Threading-Modelle haben sich verschiedene Begriffe etabliert, die sich je nach Technologie-Stack und Framework unterscheiden. Oft spricht man von *synchronen* Services, wenn das Thread-per-Request-Modell gemeint ist und von reaktiven Services, wenn das Event-Loop-Modell zugrunde liegt. Letzteres wird auch regelmäßig als *asynchron* oder *nicht-blockierend (non-blocking)* bezeichnet, um die Freigabe des verarbeitenden Threads für andere Aufgaben zu unterstreichen. In unserem Beispiel haben wir ausgehende HTTP-Requests als potenziell blockierende Operationen herangezogen, das Konzept des Event Loops lässt sich aber für alle I/O-Operationen (generelle Netzwerkkommunikation, Dateisystem- und Datenbankzugriff) anwenden und die hier beschriebenen Vorteile treffen ebenso zu.

Asynchrone Codeausführung in Java

Was müssen wir als Entwicklerinnen und Entwickler also machen, wenn wir dieses Modell als Basis unseres Webservice nutzen wollen? Sicher wäre es komfortabel, wenn wir unseren Anwendungscode genauso schreiben könnten wie bisher, und ein Framework kümmert sich für uns um das Threading-Modell. Das folgende Codebeispiel soll illustrieren, warum dies ohne Anpassung der Java-Sprachdefinition unmöglich ist.

```
public MyResult getSynchronous() {
  Response response = httpClient.sendRequest(URL);
  return processResponse(response);
}
```

Listing 1: Synchroner Aufruf einer hypothetischen HTTP-Client-API

Listing 1 zeigt die synchrone Verwendung einer hypothetischen HTTP-Client-API auf Service-Seite: Ein HTTP-Request wird an eine bestimmte URL geschickt und die Response wird in einer Methode in ein fachliches Ergebnisobjekt transformiert. Das Problem hierbei: Die Methoden `sendRequest` und `processResponse` werden als zwei aufeinanderfolgende Anweisungen zwangsläufig im gleichen Thread ausgeführt, worauf auch die Implementierung des HTTP-Clients keinen Einfluss hat. Der dargestellte Code ist also für das Event-Loop-Modell nicht anwendbar. Durch eine Anpassung unserer HTTP-Client-API erhalten wir die notwendige Flexibilität, wie in *Listing 2* dargestellt.

```
public void getAsynchronous(Consumer<MyResult> callback) {
  httpClient.sendRequest(URL, response ->
      callback.accept(processResponse(response)));
}
```

Listing 2: Asynchroner Aufruf des HTTP-Clients unter Verwendung von Callbacks

Als wesentliche Anpassung gibt die Methode `sendRequest` nun keine Response mehr direkt zurück, sondern akzeptiert einen Callback als Argument, der aufgerufen werden soll, sobald die Antwort vorliegt. Damit ist die Implementierung des HTTP-Clients frei, die Verarbeitung des Callbacks auf einem anderen als dem aufrufenden Thread fortzusetzen, was die Grundlage für das Event-Loop-Modell darstellt. Leider ist die Methode aus *Listing 2* deutlich schwerer zu lesen und die fachliche Absicht des Codes rückt gegenüber technischen Details in den Hintergrund. Frameworks wie Reactor haben es sich daher zur Aufgabe gemacht, die Besonderheiten asynchroner Aufrufe bestmöglich zu kapseln, sodass der Anwendungscode fachlich fokussiert bleibt.

Reactor & WebFlux

Um asynchrone Vorgänge abzubilden, stellt das *Reactor* Framework die beiden Primitive *Mono* und *Flux* bereit. Ein Mono löst sich schlussendlich in ein oder kein Objekt auf, während ein Flux eine Sequenz von mehreren Objekten darstellt, die hintereinander eintreffen. Beide Primitive können alternativ einen Fehler in Form einer Exception propagieren. Ein Mono ist also das reaktive Äquivalent zu einem nullbaren Referenztypen, während ein Flux konzeptuell dem Java-8-Stream-Typen entspricht.

In *Abbildung 5* sind die beiden Konzepte in Form sogenannter Marble-Diagramme visualisiert *[5]*. Der farbige Kreis steht dabei für ein eintreffendes Objekt, der vertikale Strich für die erfolgreiche Terminierung der Sequenz und das rote „X" für die Terminierung durch ei-

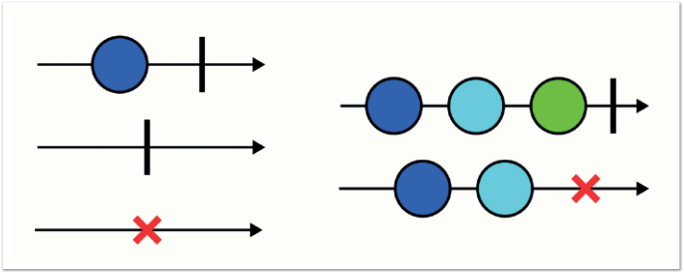


Abbildung 5: Marble-Diagramme der verschiedenen möglichen Abfolgen für die Reactor-Primitive Mono (links) und Flux (rechts) (© Jonas Nagy-Kuhlen)

nen Fehler. Die drei Ereignisse formalisieren in gewisser Weise den Callback aus dem vorhergehenden Beispiel *(vgl. Listing 2)*.

Auf den beiden Primitiven definiert das *Reactor* Framework eine große Spanne von Operatoren. Viele davon verhalten sich analog zu den Operatoren auf synchronen Streams, wie zum Beispiel die Projektion jedes Elements mittels einer Mapping-Funktion oder die Filterung von Elementen nach einem Prädikat. Die Reactor-Dokumentation beschreibt alle Operatoren detailliert (oft auch in Form von Marble-Diagrammen) und stellt eine praktische Referenz bereit, wie verschiedene Code-Konstrukte aus der synchronen in die asynchrone Welt übersetzt werden *[6]*.

Spring WebFlux ist das reaktive Äquivalent zum synchronen *Spring Web MVC* und setzt auf *Reactor* auf, um reaktive Spring-Anwendungen basierend auf dem Event-Loop-Modell zu ermöglichen *[7]*. Zur asynchronen Verarbeitung eingehender Requests erlaubt das Framework die Rückgabe von Reactor-Primitiven in Controller-Methoden, dargestellt in *Listing 3*.

```
@GetMapping(value = "/hello/world")
public Mono<String> getHelloWorld() {
  return Mono.just("Hello world!");
}
```

Listing 3: Reaktive Verarbeitung eines eingehenden Request mit WebFlux

Für ausgehende HTTP-Requests stellt *WebFlux* die Klasse `WebClient` bereit, welche die über das Netzwerk eintreffende Response in Form eines Mono- oder Flux-Objekts kapselt *(siehe Listing 4)*.

```
public Mono<MyResult> getAsynchronous() {
  return webClient.get().uri(URL)
         .retrieve()
         .bodyToMono(MyResult.class);
}
```

Listing 4: Reaktives Absenden eines ausgehenden Request mit WebFlux

Der `WebClient` produziert also Mono-/Flux-Objekte, während der Controller solche konsumiert. Die Anwendungslogik lässt sich also als funktionale Kette dazwischen betrachten, beschrieben als Abfolge reaktiver Operatoren.

Reaktiver Stack in Action

Bisher haben wir uns detailliert mit der Theorie hinter reaktiven Webservices befasst. Doch welche Auswirkungen zeigen sich in der Praxis? Um dieser Frage nachzugehen, habe ich eine Beispielanwendung sowohl synchron basierend auf *Spring Web MVC* als auch reaktiv mittels *WebFlux* implementiert, die wir für einen Lasttest-Vergleich heranziehen wollen. Der Code ist auf GitHub zu finden *[8]*. Um ein Microservice-Umfeld mit messbaren Latenzen zu simulieren, spricht der Service sequenziell zwei APIs an, die so eingestellt sind, dass sie nach exakt 300 beziehungsweise 500 Millisekunden eine Antwort liefern. Eine Antwortzeit von 800 Millisekunden bildet also für unseren Service die theoretische Vergleichslinie.

Unser Benchmark besteht aus 800 Requests, die wir per Command Line an den Service schicken. Die Ressourcenauslastung visualisieren wir mit dem Monitoring Tool *VisualVM [9]*. Die synchrone Implementierung antwortet im Durchschnitt und Median nach knapp über 800 ms bei 200 gleichzeitigen Requests, wie in *Abbildung 6* zu sehen ist. Bei 400 gleichzeitigen Requests zeigt sich ein komplett anderes Bild: Die Antwortzeiten haben sich nahezu verdoppelt! Den Grund dafür erkennen wir, wenn wir einen Blick auf den Verlauf der Threads werfen. Die maximale Größe des Thread Pools beim zugrunde liegenden *Tomcat*-Webserver ist standardmäßig auf 200 konfiguriert, weshalb die Hälfte der eingehenden Requests erst einem Thread zugeteilt werden kann, sobald die andere Hälfte erfolgreich verarbeitet wurde. Die Größe des Thread Pools stellt sich also als Bottleneck heraus.

In *Abbildung 7* sind die Ergebnisse der gleichen Testausführung gegen die reaktive Implementierung zu sehen, die als Standard-Webserver *Netty* statt *Tomcat* nutzt. Die Antwortzeiten bleiben unabhängig von der Anzahl gleichzeitiger Requests nahezu konstant, ebenso die Anzahl der Threads unter Last. Ein genauerer Blick in *VisualVM* zeigt, dass der Thread Pool zur Abarbeitung der eingehenden Requests beziehungsweise der Event Loops 16 Threads umfasst, was exakt der Anzahl der logischen Prozessoren der Testhardware entspricht.

Meine Anwendung ist synchron – und jetzt?

Es ist wahrscheinlich, dass Entwicklerinnen und Entwickler in ihrem Projekt synchrone Spring-Webservices vorfinden. Ob sich eine aktive Migration auf den reaktiven Stack lohnt, hängt immer von der Anwendung ab, allerdings sprechen folgende Voraussetzungen dafür:

- Der Durchsatz stellt ein Bottleneck dar. In diesem Fall kann das Event-Loop-Modell helfen, den Grad an Parallelität in der Anwendung zu erhöhen.
- Die CPU-Last ist (zu) hoch. Das Eliminieren von großen Thread Pools und dadurch von Context Switches und Polling sorgt dafür, dass mehr CPU-Leistung für Anwendungslogik zur Verfügung steht.
- Es werden viele parallele ausgehende Requests gesendet. In dem Fall müssen besonders viele Threads unnötig verwaltet werden, was wiederum für unnötigen Overhead sorgt, der durch das reaktive Modell vermieden werden kann.

Umgekehrt sollte ein Umstieg aufgrund des Zeitaufwands nur durchgeführt werden, wenn im Team genug Kapazität für techni-

Abbildung 6: Lasttestergebnisse der synchronen Service-Implementierung und Verlauf der Threads im Thread-Per-Request-Modell (© Jonas Nagy-Kuhlen)

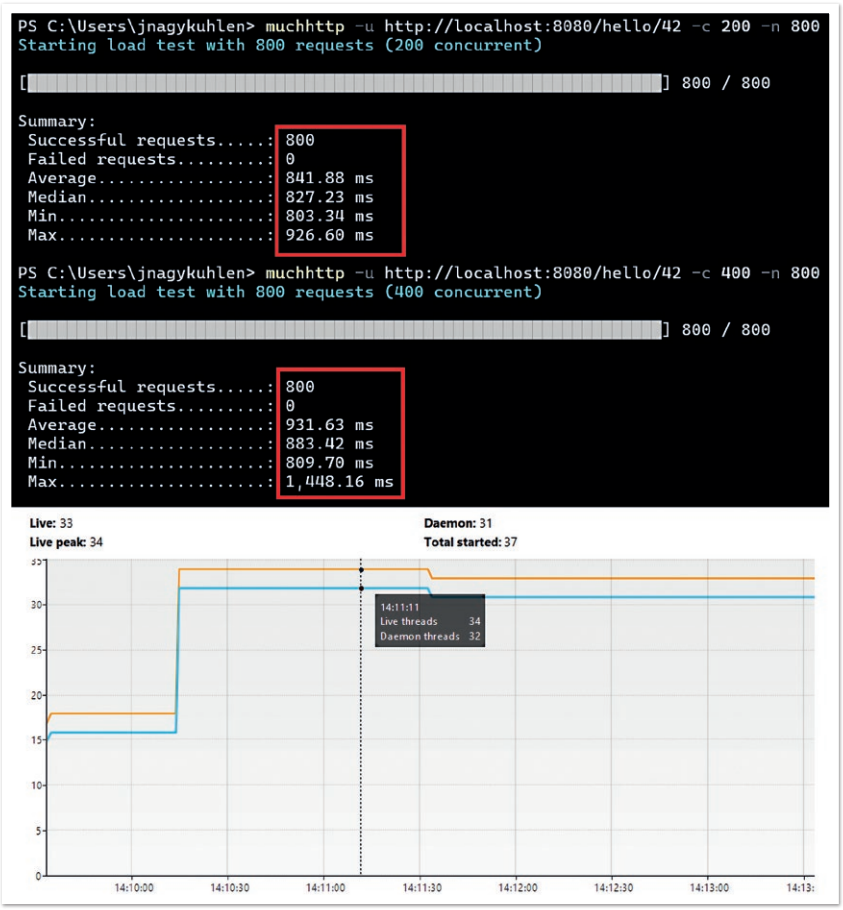


Abbildung 7: Lasttestergebnisse der reaktiven Service-Implementierung und Anzahl der Threads für das Event-Loop-Modell im Verlauf (© Jonas Nagy-Kuhlen)

sche Optimierungen zur Verfügung steht und Performance tatsächlich ein Problem darstellt. Führt die Anwendung ausschließlich CPU-intensive Berechnungen durch und entstehen somit wenig Wartezeiten durch I/O, dann ist der zu erwartende Performancegewinn gering. Wenn ein neuer Service aufgebaut wird, ist es hinsichtlich Zukunftssicherheit und Flexibilität naheliegend, direkt mit Spring WebFlux als modernem Technologie-Stack zu starten und die Software von Anfang an im Sinne des reaktiven Paradigmas zu entwickeln.

Fazit

In diesem Artikel wurden die Vorteile reaktiver Webservices beleuchtet und dabei detailliert auf das zugrundeliegende Threading-Modell eingegangen, für das *Spring* mit *WebFlux* gute Framework-Unterstützung bereitstellt. *Reactor* wirkt ausgereift und bietet zwei fundamentale reaktive Datentypen mit einer Vielzahl nützlicher Operatoren an, die hervorragend dokumentiert sind. Leider gibt es in Java nach wie vor keine Sprachunterstützung für asynchrone Logik, wie sie C# und JavaScript mit den Keywords `async/await` sowie Kotlin mit Coroutines schon lange mitbringen. Die stattdessen eingeführte funktionale Schreibweise von Logik ist oft umständlicher und weniger kompakt im Vergleich, was für eine steilere Lernkurve sorgt. Die Verwendung von *WebFlux* und *Reactor* in Java bleibt somit ein Trade-off zwischen Performance und Skalierbarkeit auf der einen und Lesbarkeit sowie einfacherem Debugging auf der anderen Seite. Dennoch ist es wichtig, die zur Verfügung stehenden Werkzeuge zu kennen und bei Bedarf anwenden zu können.

Quellen

[1] Broadcom 2024: Web on Servlet Stack *https://docs.spring.io/spring-framework/reference/web.html*

[2] The Apache Software Foundation 2024: Apache Tomcat 11 Configuration Reference *https://tomcat.apache.org/tomcat-11.0-doc/config/executor.html*

[3] Ben Evans 2023: Exploring the design of Java's new virtual threads *https://blogs.oracle.com/javamagazine/post/java-virtual-threads*

[4] VMware Inc. 2024: Reactor Core Schedulers *https://github.com/reactor/reactor-core/blob/main/reactor-core/src/main/java/reactor/core/scheduler/Schedulers.java#L81*

[5] Broadcom 2024: How to read marble diagrams? *https://projectreactor.io/docs/core/release/reference/#howtoReadMarbles*

[6] Broadcom 2024: Which operator do I need? *https://projectreactor.io/docs/core/release/reference/#which-operator*

[7] Broadcom 2024: Spring WebFlux *https://docs.spring.io/spring-framework/reference/web/*

webflux/new-framework.html

[8] Jonas Nagy-Kuhlen 2024: WebFlux Demo *https://github.com/jnagykuhlen/WebfluxDemo*

[9] Oracle Corporation 2024: VisualVM *https://visualvm.github.io*

MITMACHEN UND AUTOR/IN WERDEN!

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

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

Wir freuen uns, von euch zu hören!

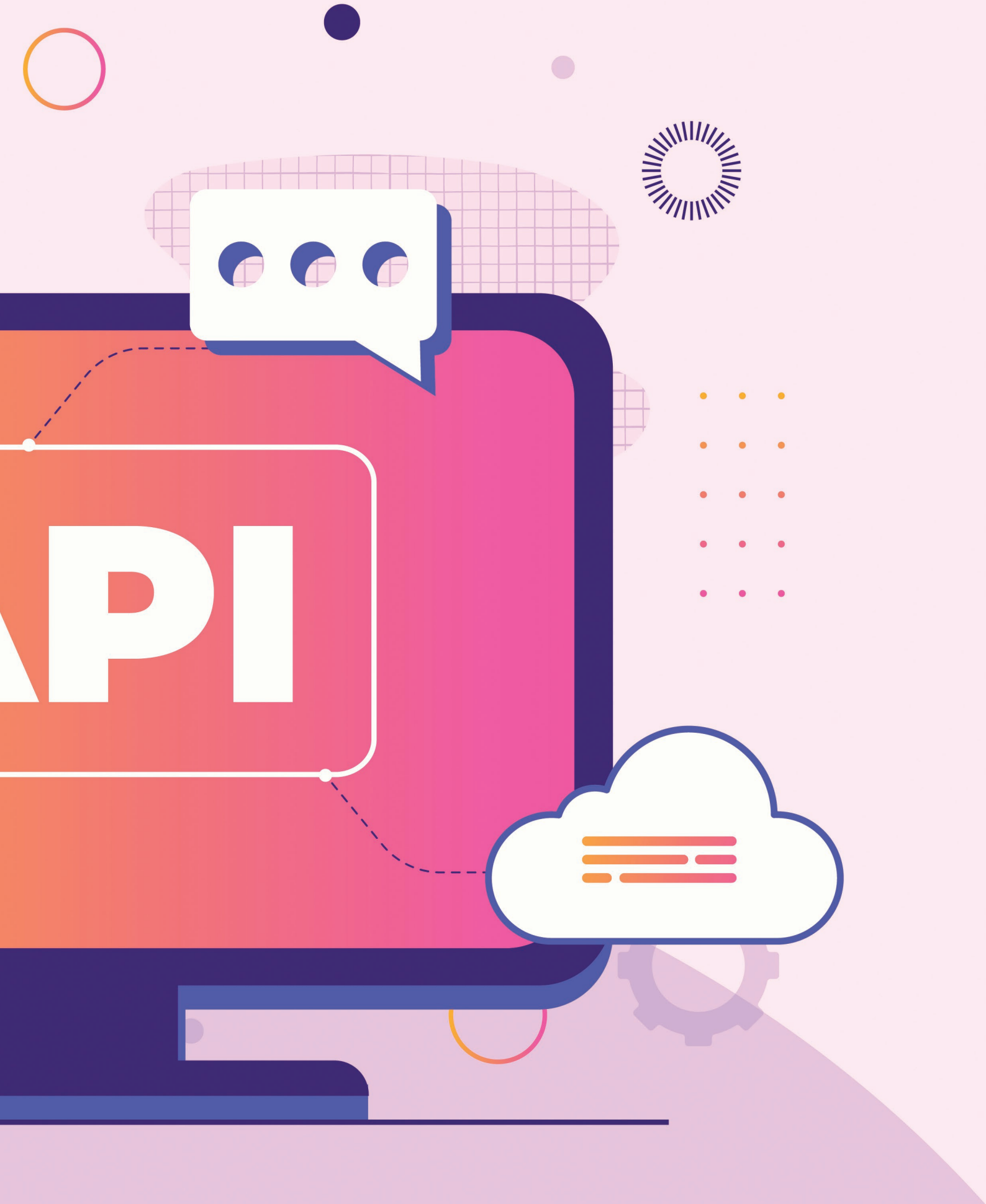
iJUG Verbund

Jonas Nagy-Kuhlen
andrena objects ag
jonas.nagy-kuhlen@andrena.de

Jonas Nagy-Kuhlen ist als Architekt und agiler Softwareentwickler maßgeblich verantwortlich für die Konzeption und Umsetzung von Lösungen in komplexen Systemlandschaften basierend auf dem Java-Spring-Stack. Seit seinem Einstieg in die .NET-Entwicklung vor mehr als zehn Jahren konnte er insbesondere Erfahrung mit performancekritischen, nebenläufigen Systemen sammeln und diese erfolgreich in verschiedenen Projekten führender Unternehmen einbringen. Dabei legt er großen Wert auf agile Methoden und sauberen, getesteten Code.

API-Entwicklung: Das richtige Maß macht's

Noah Neukam, ITech Progress GmbH
Uwe Neukam, DATEV eG


API

Eine Welt ohne Schnittstellen gibt es nicht. Dennoch ist die Formulierung von APIs weiterhin eine große Herausforderung. Beim API-Design kommt der Unabhängigkeit ein hoher Stellenwert zu. Mithilfe von Standards lässt sich diese erreichen und direkt in Entwicklungsprozesse einbinden.

Bereits in der Früh werden wir täglich mit Schnittstellen konfrontiert, und zwar in unzählbarer Menge. Von der Zeit, die zwischen uns und unserem Wecker ausgemacht ist und dem damit geteilten allgemeinen Verständnis über Zeit, bis hin zu allen Transistoren, technischen Spezifikationen und Spannungsangaben. Ohne Schnittstellen geht nahezu nichts. Bei sehr vielen Tätigkeiten gilt die Aussage: „Alleine geht es nicht", und um Zusammenarbeit zu ermöglichen, bedarf es eines Grenzsteines. Dieser verdeutlicht, was wir machen, was die anderen machen und wie die verrichtete Arbeit dann zusammengebracht wird. Eine Schnittstelle klärt damit Rechte und Pflichten.

Eine besondere Unterart der Schnittstelle stellt die API *[1]* dar. Sie grenzt sich durch den Bezug zur Softwareentwicklung ab. Redhat *[2]* formuliert eine API als „eine Sammlung von mehreren Definitionen und Protokollen zur Entwicklung und Integration von Anwendungssoftware. Eine API ermöglicht es, unabhängigen Anwendungen miteinander zu kommunizieren".

Die beliebteste Form unabhängiger Kommunikation ist dabei die HTTP-API. Im Bereich der HTTP-APIs wird grundsätzlich noch weiter *[3]* unterschieden: zwischen Platform-APIs und Business-APIs. Welcher Kategorie eine API zugeordnet ist, hängt vom Verwendungszweck ab und bestimmt maßgeblich die Gestaltung der API. Platform-APIs sind dabei die „einfacheren" APIs. Denn bei ihnen geht es ausschließlich um das Einspielen oder Abrufen von Daten; sie sind datenorientiert. Das ist es auch, was sie von der Business-API unterscheidet. Business-APIs folgen Geschäftsregeln, also definierten Fachlichkeiten; sie sind eher Use-Case-orientiert.

Und was sagt uns das jetzt? Platform-APIs sind simpler und mit Business-APIs verdiene ich Geld? Wie immer im Leben, sind die Antworten nicht ganz so einfach. Sicher, Platform-APIs sehen auf den ersten Blick simpler aus, sie sind gestaltungstechnisch stringent und die Daten bestimmen den Umfang. Aber auch diese sind technisch gebunden. Business-APIs sind in ihrer Erstellung nicht unbedingt aufwendiger, nur weil sie in fachliche Prozesse integriert werden sollen, jedoch können die Prozesse variieren und somit die Komplexität durchaus erhöhen. Interessanterweise können beide Typen für ein Unternehmen einen Geschäftswert darstellen und somit zu dessen Geschäftserfolg beitragen. „API as a Product" ist hier das Stichwort.

Business-APIs brauchen aktive Gestaltung

Mit dem „Upgrade" einer API zu einem Produkt *[4]* entsteht nicht nur eine mögliche neue Einnahmequelle, es bedarf auch eines „Entwurfs- und Herstellungsprozesses". In der klassischen Betriebswirtschaft ist dieser Prozess dafür verantwortlich, sicherzustellen, dass ein Produkt entsteht, das Kunden begeistern kann und eine dem Preis angemessene Qualität aufweist. Der Herstellungsprozess, inklusive eines Verbesserungs- und Verfeinerungsprozesses, stellt die eigentliche Ware her oder bietet sie an und prüft die Qualität. In der IT gestalten sich diese Prozesse zwar anders (Code muss schließlich „nur einmal" geschrieben werden), aber die Grundideen sind gleich. Klar ist, die Zielgruppe sind Entwickler, doch was macht ein für Entwickler attraktives Produkt aus?

Dem Grundsatz „Form follows Function" folgend lassen sich einige Kategorien an Anforderungen ableiten, die bei „API as a Product" immerwährend auftreten:

- Fachliche Inhalte und Umfang
- Lifecycle-Management und Updates
- Qualität und Zuverlässigkeit
- Standards

Alle vier Punkte werfen dabei eigene grundlegende Gestaltungsfragen *[5] [6]* auf und alle bedürfen weiterer Erläuterung.

Fachliche Inhalte und Umfang

Gemäß dem Ausdruck „Die richtigen Dinge tun, statt Dinge richtig tun", von Peter Drucker stellen wir die Inhalte an Platz eins. Fachlich korrekte Inhalte in einer API anzubieten, die Abnehmer auf dem Markt findet, ist dabei allerdings nur der erste Schritt. Um erfolgreich zu sein, müssen auch die Darstellung und der Umfang stimmen. Die Darstellung ist dabei nicht als eine technische Diskussion über Vor- und Nachteile von JSON gegenüber XML zu verstehen, sondern ob die Stakeholdergruppe, die von der API adressiert werden soll, erfolgreich abgeholt wird. Das beinhaltet, ihre Sprache zu benutzen und die API nach ihren Anwendungsfällen und ihrer Logik zu strukturieren. Denn diese kann, trotz der gleichen Inhalte, von Stakeholder zu Stakeholder variieren *(siehe Abbildung 1)*.

Zuletzt muss der Umfang stimmen. Ist die API zu schmal (*under-fetching*) geschnitten und es bedarf mehrerer APIs, damit der Stakeholder sein Interesse erfüllen kann, werden nur unnötiger Aufwand und im Zweifelsfall zusätzliche Kosten erzeugt. Auch wenn eine API zu umfangreich (*over-fetching*) ist, sind Aufwand und Kosten führende Themen. Eine übermäßig umfangreiche API ist verwirrend, liefert zu viele Daten, die in dem Benutzerkontext nicht benötigt werden und ist somit schwer zu bedienen.

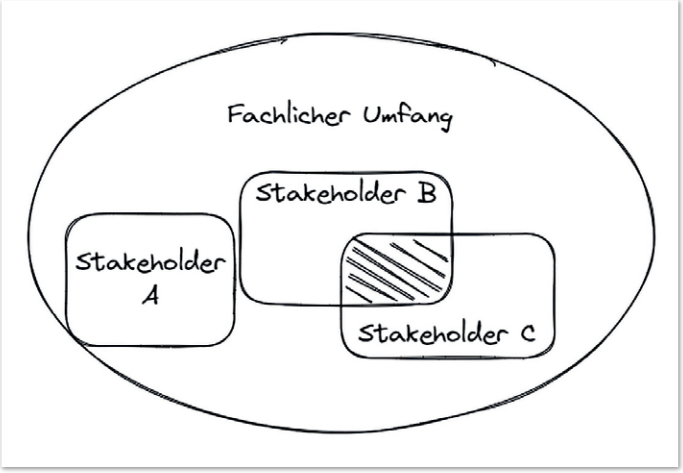


Abbildung 1: Stakeholder-Interessen überschneiden sich. Die gleiche Sprache sprechen sie deswegen nicht immer.

Um diesen Unannehmlichkeiten zu entgehen, gibt es einige, wohl etablierte, Herangehensweisen, die wir als hilfreich empfinden. An vorderster Front steht dabei das Event Storming. Als Methode, die viel im Domain-Driven Design genutzt wird, gibt uns das Event Storming Aufschluss über Leseinteressen der Nutzer, über Zustände, die von Interesse sind, und wann ein Stakeholder in einen Fachprozess involviert ist. Hat man die richtigen Leute, gibt es Aufschluss über die „Sprache" dieser Stakeholdergruppe. Mit diesen Informationen ist man in der Lage, alle Grundlagen für eine fachlich wohlgeformte API zu erschaffen. Neben dem Event Storming stehen viele andere Methoden zur Verfügung, die je nach Einsatzzweck tiefer blicken lassen: Zum Beispiel die „Produktkarton-Methode", wenn spezifisch die „unique selling proposition" der API herausgearbeitet werden soll. Die Gemeinsamkeit der Methoden und Quelle ihres Mehrwertes ist ihre explorative Natur.

Wer scheitern möchte, kann es mit einigen Antipatterns probieren. Ein Klassiker ist dabei, die eigene Perspektive nicht zu hinterfragen. Gemäß Epiktet kann „jener, der denkt alles zu wissen, nichts lernen", und sich auf die Sprache oder Bedürfnisse einer Stakeholdergruppe nicht einzulassen, kann fatal sein. Auch einen Fachprozess einheitlich und für alle Stakeholder in eine API zu bringen, ist ein Rezept für Desaster. Zu viele Köche verderben den Brei. Zu viele Stakeholder auch!

Lifecycle-Management und Updates

„Für ihr Betriebssystem stehen Updates bereit", ist ein Satz, der heutzutage weniger Schrecken verursacht als früher. Was damals mit einem anderen Wording versehen war, hatte auch eine andere Implementierung: Der Computer startete sich gegebenenfalls ohne Vorwarnung neu, um die Updates zu installieren. Das machte nicht nur den Rechner für die Zeit der Installation unbrauchbar, sondern verwarf gerne die Arbeit, die bis dahin erledigt worden war. Für APIs gilt das Gleiche. Wird ein Breaking Change an einer API durchgeführt, so ist die Nutzung der API blockiert, bis ein Update des Clients vorgenommen wurde. Das ist für keinen der Beteiligten schön und sollte unbedingt verhindert werden.

Einige Herangehensweisen möchten wir hier zeigen, die den Schmerz im API-Lifecycle lindern können. Nummer eins ist hierbei das „Non-Breaking Changes only"-Prinzip. Hierbei verpflichtet man sich, als Anbieter in einer API nur Änderungen vorzunehmen, die mit bisherigen Clients kompatibel sind. Anstatt einen Breaking Change zu machen, kann eine zweite API angeboten werden. Je nach Situation wird auf Dauer eine der beiden APIs wieder abgeschaltet. Diese Option lässt viel Freiraum:

- ob Daten geteilt werden,
- ob es eine Migrationsmöglichkeit zwischen den APIs gibt,
- ob ein Parallelbetrieb möglich ist.

Das Prinzip des Mehrfachbetriebs ist mächtig, hat aber auch eine Voraussetzung, die wir klar machen wollen. Der Business-Code darf nicht direkt an die Ausgestaltung der API gebunden sein. Und diese Trennung möchten wir als zweite hilfreiche Herangehensweise aufzeigen. Eine Trennung von API und Business-Code sowie auch technischen Implementierungsdetails erlaubt nicht nur Flexibilität in der Gestaltung, sondern verhindert auch, dass API-Nutzer von der technischen Realität des Entwicklungsteams beeinflusst werden.

Wer mit seinem Lifecycle Probleme bekommen möchte, für den haben wir natürlich auch Rat. Bindet eure API-Klassen direkt in den Business-Code! Lasst die Details von Datenbank und Plattform bis an die API durchscheinen! Das verstärkt den Vendor-Lock-in. Vereinfacht und nutzt anstelle von „Keine Breaking Changes" den Grundsatz „Nur hinzufügen zur API" und fügt regelmäßig neue Enum-Werte in Auflistungen hinzu. Clients bekommen dann zwar Werte, die sie nicht gelernt haben, zu interpretieren, aber so können sie ihre Resilienz testen! Als Zugabe macht mindestens zweimal im Jahr einen Breaking Change oder kündigt eine API ohne großen Vorlauf ab.

Qualität

Die Qualität der API ist nicht direkt auf der Verpackung zu erkennen. Dennoch ist die Qualität für einen API-Nutzer stets merklich, zum Beispiel in der Latenz der Anfragen. Mit der Qualität verhält es sich also wie bei anderen Softwareprodukten auch. Wie bei vielen Open-Source-Bibliotheken besteht die Nutzergruppe allerdings aus Entwicklern. Wer schon einmal ein Framework oder eine Bibliothek ausgewählt hat, kennt die hier üblichen Fallstricke. Das angebotene Interface soll meist leicht zu nutzen sein, verständlich und doch einen gewissen Grad an Flexibilität bieten. Diese Spezifikation gilt zwar nicht für jede API, zeigt uns aber die Dimensionen auf, die bei der Qualität angelegt werden und ebenso, dass sie durch Qualitätsmodelle beschrieben werden können.

Die wichtigste These, die wir für das Erreichen von Qualität in APIs aufstellen wollen, ist das Erfassen der Qualitätsziele einer Stakeholdergruppen-spezifischen API. Ansonsten weichen APIs nicht besonders von üblicher Entwicklung ab. Qualitätsszenarien sollten erstellt werden, um zu prüfen, inwieweit die Qualitätsziele erreicht sind. Entscheidungen in der Architektur sollten durch die Qualitätsziele getrieben sein und ihre Erreichung fördern.

Während wir bisher nahegelegt haben, die Implementierung der API und der Fachlichkeit getrennt zu betrachten und API-Objekte von der Fachlichkeit zu trennen, möchten wir an dieser Stelle auch auf das Gegenteil aufmerksam machen. Wenn eine API bestimmte Qualitäten aufweisen soll, so muss diese Qualität auch in den beteiligten Komponenten zu finden sein. Es wäre auch sehr ungewöhnlich, wenn eine API eine Antwortzeit von unter 200 ms einhalten könnte, während eine (nicht-cachebare) Abfrage an eine der datenliefernden Komponenten 500 ms bräuchte. Wir empfehlen daher, die Qualitätsanforderungen an die API auch auf betroffene Komponenten zu mappen. Das hilft dem Verständnis und erlaubt, bei geänderten Qualitätsanforderungen gezielt und informiert zu agieren, insbesondere bei in Konflikt stehenden Anforderungen oder überladenen Komponenten. Ob nun Separation, Duplikation, Trade-offs oder andere Maßnahmen korrekt sind, ist egal – durch das Aufstellen und mappen der Qualitätsziele können wir den Übergang in die üblichen Methoden und Vorgehen der Softwarearchitektur möglich machen.

Für diejenigen, die Scheitern wollen, haben wir natürlich auch etwas dabei: Formuliert alle Qualitätsziele global! Es soll immerhin eine einheitliche Erfahrung entstehen. Ob einzelne Stakeholder besondere Interessen haben, ist hier nicht so wichtig. Differenziert nicht zwischen den Qualitätszielen für Entwickler und denen für den fachlichen Stakeholder, den sie an der API vertreten, das ist auch zu kompliziert. Am besten werden alle APIs, falls überhaupt mehrere, von einer einzigen Komponente bedient. Dann ist die Fachlichkeit wiederverwendbar an einem Ort und das bringt ja schließlich den größten Business Value!

Standards

Einen Standard in der Softwareentwicklung verbinden wir oftmals mit dem Thema Qualität. Erlernbarkeit, Struktur, Komplexitätsvermeidung, Wiederverwendbarkeit sind hier Worte, die uns in den Kopf kommen können. Und auch aus Produktsicht ist ein Standard eine begehrenswerte Sache, denn er verspricht Wiedererkennungswert und wird mit Vertrauen und Zuverlässigkeit assoziiert. Wer etwas anders machen möchte, muss sich erst einmal am Standard messen! Diesen Satz möchten wir auch direkt als erste Empfehlung mitgeben. Wer einen Standard nutzt, sollte um den Zeitpunkt wissen, wann es davon abzuweichen gilt. Wir machen das am Mehrwert fest. Gibt uns eine Abweichung einen signifikanten Vorteil, ob in Verständlichkeit, Erlernbarkeit, Einfachheit oder einem anderen hoch gewichteten Qualitätsziel, so ist diese Abweichung in Betracht zu ziehen. Zeichnet sich ein solcher Vorteil nicht ab, legen wir die Nutzung des Standards nahe.

Um unsere Herangehensweise nutzbar zu machen, müssen wir auch erklären, was wir in einem Standard erwarten und wo wir ihn ansetzen. Auf oberster Ebene gibt es dabei Web-Standards, die unternehmensweit als Standard eingesetzt werden können. (Beispiel: Um den Scope dieses Artikels einzuschränken, haben wir hier lediglich von HTTP+JSON gesprochen.) Es muss dabei genug Auswahl geben, um übliche Integrationsmuster abzudecken, ebenso sollte eine klare Präferenz an Technologien geäußert werden.

Weitere Standards können dann auf der Ebene von Domänen oder Produktkategorien sowie in einem einzelnen Produkt festgelegt werden. Sofern mehrere Standards zum Einsatz kommen, möchten wir nahelegen, Übersicht zu schaffen. Einerseits durch eine klare Kommunikation, andererseits dadurch, sich in einem Standard nur zu Sachverhalten zu äußern, die es wirklich bedürfen! Je abstrakter ein Standard ist, desto kürzer sollte er sich äußern und nach Möglichkeit auch eine weniger verpflichtende Natur besitzen. Das Ziel sollte sein, für 80 % der Betroffenen eine Lösung anzubieten, die sowieso in Erwägung gezogen worden wäre. Für die übrigen 20 % sollte es einen guten Grund geben, bewusst abzuweichen. So schaffen wir ein Umfeld, das durch seinen Wiedererkennungswert glänzen kann, aber das sich an entscheidenden Stellen auch von der Konkurrenz absetzen kann.

Damit wären wir auch bei den Tipps zum Scheitern angekommen. Zum Abschluss haben wir als Bonus gleich zwei Routen zum Scheitern konstruiert. Denn insbesondere Standards bedürfen einer gesunden Mitte und wir zeigen mit Freuden die Abgründe beider Seiten auf.

Als Erstes haben wir den Verfall in den Dogmatismus. Wir machen also Standards verpflichtend. Idealerweise hat dabei jede Ebene (Unternehmensarchitekt, Domänenarchitekt, Abteilungsarchitekt, Produktarchitekt) einen eigenen Standard entworfen. Mindestumfang sind dabei 30 Seiten A4 – schließlich sind einige gute Ideen und Patterns zu vermitteln. Jede Abweichung wird als Bug geahndet und stets kurz vor der Freigabe eingestellt. Dass dies die APIs des Unternehmens in eine nicht identifizierbare graue Masse verwandelt, ist dabei nicht wichtig. Immerhin hat der Grauton wiedererkennungswert! Irgendwie scheinen die (nicht regulierten) APIs auf dem Markt aber besser zu performen, also könnten wir…

Als Zweites: einfach in die totale Anarchie verfallen. Wer Vorgaben macht, hat kein Vertrauen und die Mitarbeiter in den Produkten kennen sich schließlich besser aus. Es mag zwar passieren, dass wir für jede API eine andere Lösung erarbeiten, aber diese Lösungen sind optimiert. Außerdem braucht man pro API gerade einmal 40 Seiten A4, um sie zu verstehen. Es kann auch keine Standard-Bibliothek genutzt werden, weshalb jeder Nutzer extreme Implementierungsaufwände hat, aber gute Qualität kostet bekanntermaßen. Das zeigt sich dann in den Preisen, denn auch der Anbieter zahlt für jede Abweichung vom Standard – in der Konzeption und in der Implementierung.

Quellen

[1] Was ist API? | IBM – *https://www.ibm.com/de-de/topics/api*

[2] Was ist eine API? - Was ist eine API? Das 1x1 für Programmierschnittstellen – *http://www.redhat.com*

[3] Platform API vs Service API | Hackney API Playbook – *https://playbook.hackney.gov.uk/API-Playbook/platform_api_vs_service_api*

[4] On the path to evolutionary architecture with APIs – *https://speakerdeck.com/danielkocot/on-the-path-to-evolutionary-architecture-with-apis*

[5] 10 Lessons from 10 Years of Amazon Web Services | All Things Distributed – *https://www.allthingsdistributed.com/2016/03/10-lessons-from-10-years-of-aws.html*

[6] API-Design – Wie man es besser (nicht) macht | Uwe Friedrichsen – YouTube – *https://www.youtube.com/watch?v=2xgplCQS1bY*

Uwe Neukam

uwe.neukam@datev.de

Uwe Neukam ist Software Architect bei DATEV eG und hat in mehr als 20 Berufsjahren in mehreren Unternehmen den Wandel von On-Premise-Entwicklung hin zur Web-Entwicklung begleitet. In individuellen Analysen und auf das Unternehmen zugeschnittenen Umsetzungen sieht er den Schlüssel dafür.

Noah Neukam

n.neukam@itech-progress.com

Noah Neukam ist als Grenzgänger zwischen Software Engineering und Software Architecture in Web-Architekturen heimisch. Seine Steckenpferde sind qualitätsgetriebene Softwarearchitekturarbeit, Kommunikation und Umfeld sowie die Auswirkungen von Strukturen und Strukturmustern auf Software.

DIE DOAG ANWENDERKONFERENZ

ANWENDERKONFERENZ.DOAG.ORG

SAVE THE DATE

19. BIS 22. NOVEMBER

IN NÜRNBERG

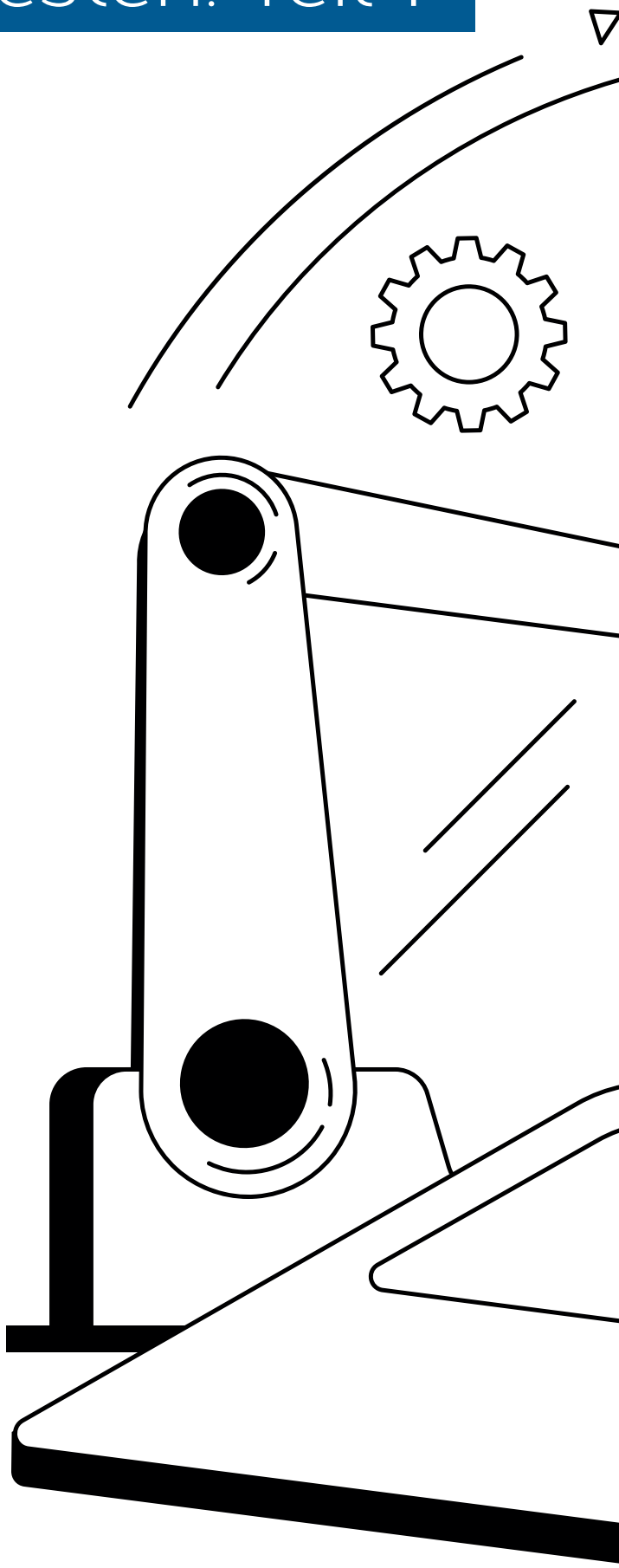
2024

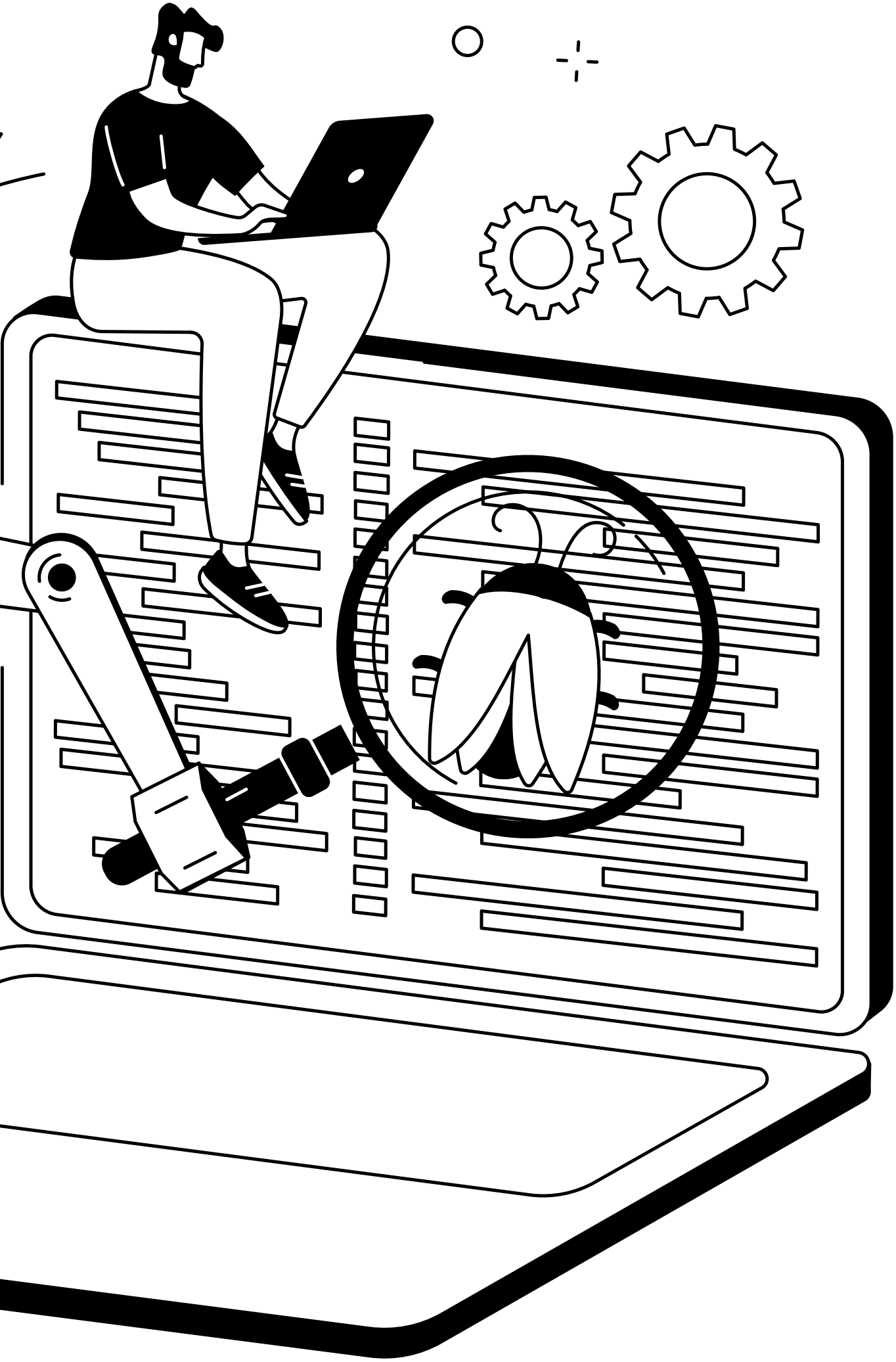
DOAG

Konferenz + Ausstellung

Risikobewusst, modular... einfach passend testen! Teil 1

Ralf Straßner, Peter Fichtner, Atruvia AG





In vielen Teams spielt das Testen eine immer wichtigere Rolle im Arbeitsalltag. Da es zahlreiche Möglichkeiten gibt, was und wie man automatisiert oder teilweise auch manuell testen könnte, sind die Teams oft unsicher, welche Arten von Tests sie brauchen, damit sie ihr Produkt gut absichern. In dieser zweiteiligen Artikelreihe zeigen wir auf, wie Teams vorgehen können, welche Fragen eine Rolle spielen und welche Modelle, Prinzipien und Hilfsmittel bei der Lösungsfindung hilfreich sind.

Wenn Teams die passenden Tests für ihre Anwendung finden wollen, stehen sie zahlreichen Herausforderungen gegenüber. Es sollen weder unbewusst Lücken entstehen noch bestimmte Aspekte mehrfach getestet werden, was unnötige Aufwände zur Folge hätte. Darüber hinaus sollen Tests oft noch mehr leisten: Sie sollen das Team leiten, das fachliche Verständnis schärfen oder das Produkt beziehungsweise die umgesetzte Lösung hinterfragen.

Hieraus ergeben sich in der Regel folgende Fragen:

- Wie schreiben wir passende Tests für unser Produkt?
- Was bedeutet „passend" überhaupt? Wann ist ein Test passend?
- Brauchen wir nicht sowieso Tests für alles?
- Wie können wir unsere Kapazitäten sinnvoll einsetzen, um einen möglichst großen Nutzen durch unsere Tests zu generieren?
- Wie behalten wir den Überblick über alle Testarten, also welche Aspekte unseres Produkts wie genau getestet werden?

Wir zeigen nachfolgend Möglichkeiten auf, wie sich ein Team einen Testansatz erarbeiten kann, der diese Fragen beantwortet. Insbesondere für Entwickler:innen scheinen hierbei die Testpyramide und ähnliche Testmodelle eine Lösung zu liefern. Daher klären wir zunächst, was diese tatsächlich beitragen können.

Wie hilfreich sind Testmodelle?

Es gibt eine wachsende Zahl von Modellen, die sich damit beschäftigen, auf welche Testarten man den Fokus legen sollte und welche Verteilung die Tests insgesamt einnehmen sollten – wir nennen sie nachfolgend schlicht Testmodelle. Solche Testmodelle werden anhand der Form benannt, die durch die propagierte Verteilung der Tests entsteht. Das älteste und bekannteste Testmodell ist die ursprünglich von Mike Cohn entworfene Testpyramide *[1]*, bekannter mit den wahrscheinlich erstmals von Mike Wacker geänderten Begrifflichkeiten *[2]*. Daneben gibt es beispielsweise den Testpokal (Testing Trophy) von Kent C. Dodds *[3]* und die Testhonigwabe (Testing Honeycomb) von André Schaffer und Rickard Dybeck *[4]*. Diese drei Testmodelle sind in *Abbildung 1* dargestellt.

In unseren Team-Coachings erleben wir oft, dass die Entwickler:innen sich bei der Suche nach einem Testansatz ausschließlich damit beschäftigen, welches Testmodell das Beste ist – für ihren Tech Stack oder gar allgemein. Durch die Entscheidung für ein Testmodell legt das Team fest, dass die im Testmodell empfohlene Verteilung der Tests anzustreben ist und manche Testarten entweder bevorzugt oder vermieden werden sollen.

Betrachten wir exemplarisch für die Testpyramide, was das bedeutet: Um die Pyramidenform zu erreichen, konzentrieren sich die Entwickler:innen darauf, möglichst viele Unit-Tests zu schreiben. Damit diese schnell und zuverlässig sind, werden alle Abhängigkeiten, die dies verhindern würden, in den Unit-Tests ausgeklammert. Stattdessen werden diese Abhängigkeiten in separaten Integrationstests geprüft, für die jedoch weniger Aufwand betrieben wird, weshalb deren Anzahl deutlich geringer ausfällt. End-to-End-Tests werden nur im Ausnahmefall geschrieben, wenn etwas nicht anders geprüft werden kann. Bei allen Tests handelt es sich um automatisierte Tests, manuelles Testen wird vermieden.

Das ist grundsätzlich nicht falsch. Dennoch stellt sich bei den Teammitgliedern oft eine gewisse Verwirrung ein, wenn die anvisierte Verteilung der Tests nicht zustande kommt. Im schlimmeren Fall wird die Verteilung durch ein dogmatisches Einhalten des Testmodells zwar erreicht, aber immer wieder können gravierende Fehler nicht durch die Tests verhindert werden.

Eine typische Reaktion ist dann, die Auswahl des Testmodells zu hinterfragen und nach einem besseren Testmodell zu suchen. Doch hier ist Vorsicht geboten, wie auch Martin Fowler aufzeigt *[5]*: Denn jedes Testmodell bringe ein eigenes, unterschiedliches Verständnis von Begrifflichkeiten wie Unit-Test und Integrationstest mit sich, die es zu verstehen gelte. Zum Beispiel wurde in der originalen Testpyramide von Mike Cohn nicht von End-to-End-Tests, sondern von

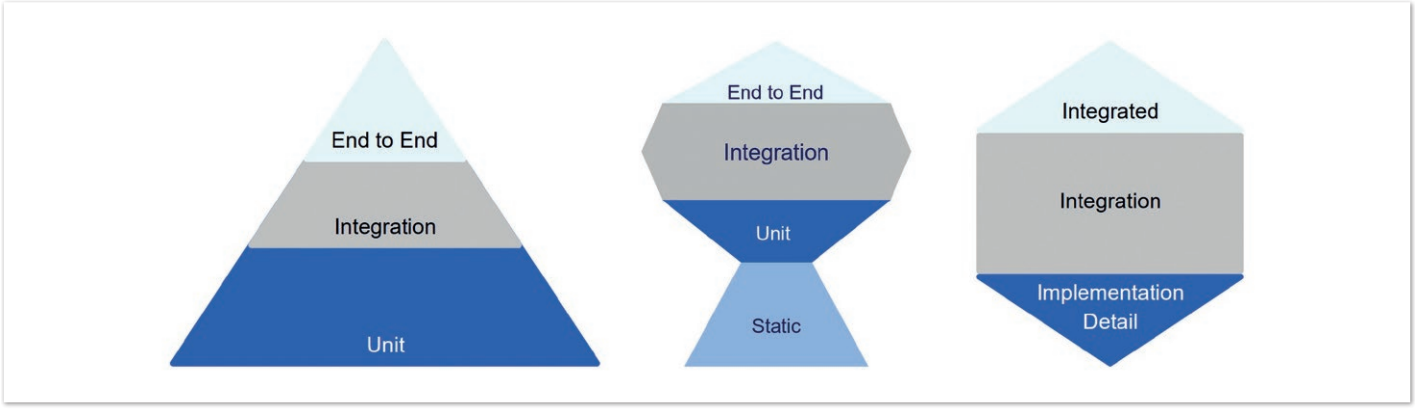


Abbildung 1: Testpyramide, Testpokal und Testhonigwabe (© Ralf Straßner, Peter Fichtner, in Anlehnung an [2], [3] und [4])

User-Interface-Tests gesprochen und statt dem Begriff Integrationstest wurde der Begriff Service-Test verwendet [1]. Die Testmodelle miteinander zu vergleichen, erweist sich daher als schwierig und lenkt davon ab, die wesentlichen Ziele und gewünschten Eigenschaften der Tests zu schärfen, wie auch Justin Searls feststellt [5]: *„People love debating what percentage of which type of tests to write, but it's a distraction. Nearly zero teams write expressive tests that establish clear boundaries, run quickly & reliably, and only fail for useful reasons. Focus on that instead."*

Aus unserer Sicht kann dennoch ein Nutzen aus den Testmodellen gezogen werden. Wenn Teams die Prinzipien der Testmodelle anwenden, wird ihre Handlungsfähigkeit verbessert und sie können schneller auf Fehler reagieren. Die Prinzipien der erwähnten Testmodelle ähneln sich und lassen sich folgendermaßen zusammenfassen:

- **Teste möglichst eine Verantwortlichkeit einzeln und isoliert.**
  - Isolation bedeutet dabei, dass man die zu testende Unit (auch *System Under Test* genannt) so wählt, dass die für das aktuelle Testziel irrelevanten Aspekte nicht enthalten sind.
  - Dieses Prinzip ist aus der Testpyramide ableitbar und sorgt für eine hohe Präzision der Tests. Durch das Ausklammern von sogenannten unangenehmen Kollaborateuren (*Awkward Collaborators [6]*) wird die volle Kontrolle über die Testdaten und somit über das Testszenario erreicht. Darüber hinaus liefern die Tests ein frühes sowie schnelles Feedback.
- **Teste möglichst wenige Implementierungsdetails.**
  - Der Testpokal und die Testhonigwabe forcieren den Gedanken, Tests möglichst fachlich, also mit Bezug zur Domäne, zu gestalten. Dies ist kein Widerspruch zur Testpyramide, sondern schärft die Eigenschaften guter Unit-Tests: Das *System Under Test* soll nicht zu feingranular anhand des technischen Schnitts gewählt werden, indem zum Beispiel Methoden oder Klassen nur einzeln getestet werden. Stattdessen testet man eine einzelne **fachliche Verantwortlichkeit**, wobei durchaus auch mehrere Klassen und Methoden gemeinsam, aber dennoch isoliert getestet werden (sogenannte *Sociable Tests [7]*). Das *System Under Test* besitzt eine domänennahe Schnittstelle, wodurch sich die Tests auf weniger Implementierungsdetails stützen.
  - Dieses Prinzip erhöht die Robustheit der Tests gegenüber Refactorings. Zudem werden die Tests durch die Nähe zur Fachlichkeit besser nachvollziehbar.
- **Teste die Integration und das Zusammenspiel von Komponenten separat und explizit.**
  - Integration wird nicht nebenbei **mit**getestet, sondern stellt eine eigene Verantwortlichkeit dar und es gibt Tests, die sich ausschließlich darauf fokussieren. Das bedeutet, dass die beteiligten Komponenten bei diesen Tests als in sich korrekt betrachtet werden, da sie bereits isoliert getestet worden sind, und daher nur deren korrektes Zusammenspiel an einem Interaktionspunkt [4] geprüft wird.
  - Auch dieses Prinzip erhöht die Präzision der Tests.
- **Teste nicht manuell und möglichst wenig über End-to-End-Tests.**
  - Manuelle Tests sollen generell vermieden werden. Da die meisten Aspekte durch die zuvor beschriebenen Testarten abgesichert werden, benötigt man nur für die verbleibenden, eher technischen Aspekte noch End-to-End-Tests. Lediglich bei diesen Tests werden alle Komponenten des Systems integriert, man spricht daher auch von „Integrated Tests" (siehe Testhonigwabe).
  - Die End-to-End-Tests liefern zwar ein späteres Feedback als die anderen automatisierten Tests und bringen in der Regel einen höheren Erstellungs- und Pflegeaufwand mit sich, sind jedoch immer noch wirtschaftlicher als ein mehrmaliges manuelles Testen.

Für eine schnelle Handlungsfähigkeit sollte zudem das Testen in seiner Gesamtheit möglichst wenig Aufwand verursachen. Bezogen auf eine einzelne Testart bedeutet dies, das Verhältnis von Nutzen zu Aufwand zu optimieren. Eine Testart wird also bevorzugt, wenn sie entweder einen hohen Nutzen mit sich bringt oder einen geringen Aufwand für Erstellung und Pflege – im Idealfall aber über beides verfügt. Und hier steckt in den Testmodellen der entscheidende Unterschied, nämlich wenn es darum geht, mit welcher Testart dies erreicht werden kann. Sind beispielsweise Unit-Tests oder Integrationstests zu bevorzugen? Das Problem ist, dass die Testmodelle den Kontext, in dem sie angewendet werden, kaum berücksichtigen können. Es gibt lediglich grobe Vorschläge, zum Beispiel dass sich die Testhonigwabe für ein verteiltes System eignet [4] oder der Testpokal für JavaScript-Anwendungen [3]. Daher steckt in jedem Testmodell eine pauschale Annahme, wie aufwändig und wie nützlich eine Testart ist.

Wir sind jedoch der Meinung, dass dies nicht pauschal beantwortbar ist. So wird der Aufwand für die Erstellung und Pflege von Tests von vielem beeinflusst. Neben der Art und Architektur der Anwendung (zum Beispiel verteiltes System) sind dies zum Beispiel auch die angewandten Designprinzipien. Als noch wichtiger erachten wir es jedoch, den individuellen Nutzen einer Testart zu kennen. Dieser lässt sich aber nur einschätzen, wenn Klarheit darüber besteht, was zum Erfolg des Produktes beiträgt und welche Rolle die Testart hierbei spielt. Erst dann können darauf basierend die passenden Testarten ausgewählt werden. Als weiteres Problem der Testmodelle fällt die sehr grobe Klassifizierung der Testarten ins Gewicht. Unser Ziel wird nachfolgend sein, die Testarten konkreter und vor allem individueller zu bestimmen, um eine passende Fokussierung erreichen zu können.

Welchen Nutzen ziehen wir also aus den Testmodellen? Sie liefern uns wichtige Prinzipien, auf deren Einhaltung wir achten sollten und die uns eine gute Grundlage liefern, um passende Tests zu finden. Unsere Empfehlung ist es jedoch, der konkreten Form keinen hohen Stellenwert zuzuschreiben. Besser ist es, offen dafür zu bleiben, welche Verteilung der Tests sich als Konsequenz aus unserem Testansatz ergibt. Die Form steht also ganz am Ende und nicht am Anfang unserer Überlegungen. Wir empfehlen, beim Testansatz den individuellen Nutzen der Testarten zu berücksichtigen und schrittweise vorzugehen. In den nachfolgenden Kapiteln betrachten wir, wie diese Schritte gestaltet werden können.

Der eigene Testansatz: Alles beginnt mit dem Kontext und den Risiken.

Testen ist Risikomanagement! Um den eigenen Testansatz zu finden, der zu den passenden Tests führt, muss somit all das transparent gemacht werden, was letztendlich Risiken mit sich bringt. Bei genauer Betrachtung spielt dabei nicht nur das Produkt selbst mit seiner Fachlichkeit und den technischen Aspekten der Umsetzung eine Rolle, sondern beispielsweise auch die Marktsituation für das

Produkt: Sind wir uns sicher, dass es das richtige Produkt ist, mit dem wir die bestehenden Kunden glücklich machen und neue Kunden gewinnen können? Ebenfalls wichtig ist es, sich das Team, das für die Umsetzung verantwortlich ist, genau zu betrachten: Welche Skills und Erfahrungen weist es auf, welche nicht? Ist das Team erst seit kurzem zusammengestellt und muss es erst noch in eine effektive Zusammenarbeit finden?

Dies ist eine Auswahl von Fragen, die aufzeigen können, woraus sich Risiken ergeben. Dabei ist jeder Kontext individuell und oft lauern Risiken an Stellen, die man beim reinen Blick auf das Produkt und seiner Fachlichkeit nicht wahrnimmt, die aber mitunter schwer wiegen. Wenn wir unser Testen nicht darauf adaptieren, werden wir zwangsläufig Aufwände für Tests eingehen, die uns wenig Nutzen liefern. Diese Kapazitäten fehlen dann aber in der Regel an anderer Stelle, weil eine hundertprozentige Absicherung in der Praxis nicht wirtschaftlich ist. Legen wir jedoch den Fokus auf die Tests, die uns am besten gegen unsere größten Risiken absichern, werden wir nicht nur die Anzahl, sondern auch die Qualität dieser Tests erhöhen. Letztendlich wird dadurch die Wirksamkeit unseres Testens erhöht und es trägt stärker zum Projekterfolg bei.

Der Kontext einer Beispielanwendung

Um dies zu veranschaulichen, beschreiben wir nachfolgend den Kontext einer Beispielanwendung und wenden die weiteren Schritte darauf an.

Fachlichkeit:
Es soll die aktuelle Fußball-Bundesligatabelle aus den einzelnen Spielergebnissen der Saison berechnet und dargestellt werden. Darin sollen die Daten, die in Bezug auf die Platzierung des jeweiligen Teams relevant sind, dargestellt werden. Dabei gibt es diverse fachliche Regeln, wie eine Platzierung zustande kommt, insbesondere sind die Kriterien wichtig, anhand welcher punktgleiche Teams in welcher Reihenfolge verglichen werden.

Technik:
- Frontend: JavaScript, Angular
- Backend: Java, Spring Boot
- Keine eigene Persistenz
- Der Bezug der Spieltagsdaten erfolgt von einer frei verfügbaren Web-API (Third-Party).

Markt:
Das Produkt soll kostenfrei im Web zur Verfügung gestellt werden. Es gibt hier bereits ein umfangreiches kostenfreies Angebot auf verschiedenen Sport- und Nachrichtenseiten. Das Produkt unterliegt keiner Regulatorik.

Team:
Das Team wird neu zusammengesetzt und die Teammitglieder haben noch nie zusammengearbeitet. Es besteht aus zwei Entwicklerinnen und einem Fachexperten. Die Entwicklerinnen verfügen beide über viel Erfahrung mit Java und Spring Boot und haben auch schon mit JavaScript entwickelt. Jedoch besitzen beide keine Erfahrung im Einsatz von Angular. Der Fachexperte kennt sich gut mit den fachlichen Regeln aus, verfügt jedoch über keine technischen Erfahrungen aus der Entwicklung.

Die Risiken der Beispielanwendung

Welche potenziellen Risiken ergeben sich nun aus diesem Kontext, die sich negativ auf den Projekterfolg auswirken können? Grundsätzlich ist eine ganze Reihe davon denkbar, wie zum Beispiel:

- **Funktionales Risiko durch diverse fachliche Regeln:** Wenn es nicht möglich ist, alle fachlichen Konstellationen auf eine einfache Weise zu prüfen, könnten sich fachliche Fehler einschleichen oder große Testaufwände entstehen, beispielsweise weil die Daten auch im Test von der Web-API (Third Party) bezogen werden.
- **Know-How- bzw. Lernrisiko in Bezug auf Angular:** Wenn durch fehlende Kenntnisse verursachte Fehler in den Angular-Komponenten erst spät bemerkt werden, hat dies ein langsames Lernen und Vorankommen der Entwicklerinnen zur Folge.
- **Integrationsrisiko beim Bezug der Spieltagsdaten von Third-Party:** Änderungen an der Third-Party-Schnittstelle oder an den gelieferten Daten könnten hohe Aufwände für die Anpassungen an der Anwendung und den dazugehörigen Tests verursachen. Wenn diese Änderungen nicht schnell genug bemerkt werden, wird möglicherweise an einer nicht mehr funktionierenden Anwendung weiterentwickelt.
- **Risiko durch bereits vorhandenes, umfangreiches Angebot am Markt:** Es könnte ein Produkt umgesetzt werden, das am Markt gar nicht benötigt wird. Liefert das geplante Produkt dem Endanwender einen Mehrwert?
- **Risiko von Fehlkommunikation im neu zusammengesetzten Team:** Die Zusammenarbeit der Teammitglieder muss sich erst einspielen. Es sind Missverständnisse beim fachlichen Verständnis und dem Transfer auf die technische Umsetzung zu erwarten. Dies wird durch die fehlende technische Erfahrung des Fachexperten verschärft.

Die Risiken stehen auch in Wechselwirkung zueinander. Wie bereits erwähnt, wirkt sich die Abhängigkeit zur Third-Party-Schnittstelle auch auf das Testen der fachlichen Regeln aus. Es wäre ineffektiv und ineffizient, wenn wir nur mit den echten Daten von der Third-Party-Schnittstelle testen würden, da diese sich in einer laufenden Saison immer wieder ändern und somit würden sich auch unsere Testszenarien ändern. Ebenso wirkt sich das Risiko von Fehlkommunikation nicht nur auf die Umsetzung der fachlichen Regeln aus, sondern könnte auch dazu führen, dass die Entwicklerinnen und der Fachexperte jeweils eigene Tests für die fachlichen Regeln durchführen. Gründe hierfür sind ein fehlender Überblick, was genau bereits getestet wurde, aber auch ein mangelndes Vertrauen in die automatisierten Tests, insbesondere wenn diese nah an der Technik orientiert sind. Der Fachexperte kann sie dann inhaltlich nicht nachvollziehen und ist sich unsicher, ob seine fachlichen Vorgaben richtig verstanden und umgesetzt wurden. Als Folge entstehen vermeidbare Mehrfachaufwände für das Testen. Unserer Beobachtung nach passiert dies auch in bereits länger zusammenarbeitenden Teams.

Bei der Identifizierung von Risiken sind die Erfahrungen aus vergleichbaren Produkten oder Projekten wertvoll. Um nicht jedes Mal das Rad neu zu erfinden oder sich rein auf die Erfahrungen der beteiligten Teammitglieder zu verlassen, kann es hilfreich sein, einen Risikokatalog aufzubauen, der basierend auf einem knapp charakterisierten Kontext das potenzielle Risiko aufzeigt. Dieser könnte wie in *Tabelle 1* gestaltet werden.

Kontext	Risiko
Komplexe Domänenlogik, komplizierter Algorithmus	Funktionales Risiko
Viele Fremdsysteme	Integrationsrisiko
Die Hauptaufgabe ist es, andere (unternehmensinterne) Komponenten/ Services einzubinden und zu kombinieren.	Orchestrierungsrisiko
Bereitstellen einer Open-Source-Bibliothek	API-Design-Risiko
Extrem hohe Benutzeranzahl (Last), performancekritische Prozesse, …	Schwer erreichbare Qualitätsmerkmale
Unklar, ob es das richtige Produkt ist.	Marktrisiko
Schwierige rollenübergreifende Zusammenarbeit, kein gemeinsames fachliches Verständnis	Risiko von Fehlkommunikation
Bedarf für technische Umbauten, hohe technische Schulden	Risiko beim Refactoring
Fehlendes (praktisches) Know-How bezüglich eingesetzter Technologien o. ä.	Know-How- bzw. Lernrisiko
…	…

Tabelle 1: Kontext und Risiken (© Ralf Straßner, Peter Fichtner, in Anlehnung an [8])

Das grobe Erfassen der potenziellen Risiken ist nur ein erster Schritt und die Risiken müssen genauer analysiert, bewertet und priorisiert werden. Neben den Tests können auch andere Maßnahmen umgesetzt werden, welche die Auswirkung eines Risikos oder dessen Eintrittswahrscheinlichkeit senken. Diese Schritte klammern wir in unserer Artikelreihe jedoch bewusst aus, da sie für das ganzheitliche Verständnis der von uns beschriebenen Herangehensweise nicht notwendig sind. Als nächstes kommen wir daher zu der Frage, wie wir mit den Risiken nach deren Priorisierung weiter umgehen. Welche Tests benötigen wir dafür? Hier bietet sich als weiterer Zwischenschritt an, basierend auf dem jeweiligen Risiko ein geeignetes Testziel zu formulieren.

Die Testziele und die passenden Testarten

Ein Testziel hilft dabei, die Wahrscheinlichkeit für den Eintritt eines Risikos zu senken oder aber schnell zu erkennen, wenn dieses Risiko eintritt. Es wird als eine Aussage formuliert, die noch nicht konkret die Testart definiert, sondern vielmehr die Absicht und die benötigten Eigenschaften des Testens beschreibt. *Tabelle 2* gibt für die Risiken aus dem im vorherigen Kapitel eingeführten Risikokatalog mögliche Testziele an.

Nehmen wir an, wir hätten die für unsere Beispielanwendung identifizierten Risiken wie folgt priorisiert:

1. **Funktionales Risiko** durch diverse fachliche Regeln
2. **Risiko von Fehlkommunikation** im neu zusammengesetzten Team
3. **Integrationsrisiko** beim Bezug der Spieltagsdaten von Third-Party
4. **Marktrisiko** durch bereits vorhandenes, umfangreiches Angebot am Markt
5. **Know-how- beziehungsweise Lernrisiko** in Bezug auf Angular

Wir veranschaulichen die weitere Vorgehensweise anhand des funktionalen Risikos sowie des Risikos von Fehlkommunikation, weil diese beiden am höchsten priorisiert sind. Aufgrund der Wechselwirkungen zwischen diesen beiden Risiken lassen sie sich gut durch ein gemeinsames Testziel adressieren. Im Hinblick auf das funktionale Risiko sollen unsere Tests der fachlichen Regeln die Umsetzung vereinfachen und diese sicherer zum Erfolg führen. Dazu brauchen die Entwicklerinnen eine frühzeitige und ständige Rückmeldung, ob ihre Implementierung funktioniert und sie auf dem richtigen Weg sind. Das bedeutet, dass automatisierte Tests benötigt werden, die frühzeitig sowie lokal ausführbar sind und eine kurze Laufzeit aufweisen. Darüber hinaus sollen die Tests in Bezug auf das Risiko von Fehlkommunikation auch den fachlichen Austausch im gesamten Team verbessern. Sie sollen daher rollenübergreifend zusammen erstellt werden.

Risiko	Mögliches Testziel
Funktionales Risiko	Schnelles und präzises Feedback bei der Entwicklung, ob die Logik korrekt ist.
Integrationsrisiko	Annahmen zu den Fremdsystemen verifizieren; Unabhängigkeit bei der Entwicklung, insbesondere in Bezug auf das Testen
Orchestrierungsrisiko	Die orchestrierende Komponente isoliert testen; Vereinbarungen zwischen den Serviceanbietern sicherstellen
API-Design-Risiko	API aus Sicht der Verwender testen.
Schwer erreichbare Qualitätsmerkmale	Frühzeitig mangelhafte Qualität und technische Sackgassen erkennen.
Marktrisiko	Frühes End-User-Feedback einholen.
Risiko von Fehlkommunikation	Tests werden rollenübergreifend zusammen erstellt.
Risiko beim Refactoring	Möglichst robuste, fachlich orientierte Tests
Know-How- bzw. Lernrisiko	Tests ermöglichen eine sichere, kleinschrittige Umsetzung, bei der Lösungsansätze ausprobiert werden können.
…	…

Tabelle 2: Risiken und mögliche Testziele (© Ralf Straßner, Peter Fichtner)

Daraus leiten wir folgende Formulierung des Testziels ab:

> **Testziel „Fachliche Regeln“:** Die Tests für die fachlichen Regeln werden rollenübergreifend zusammen erstellt und erhöhen die Sicherheit bei der Umsetzung der fachlichen Regeln durch ein frühes und schnelles Feedback während der Entwicklung.

Was ist nun die passende Testart für dieses Testziel? Eine Möglichkeit, um dies herauszufinden, ist der Einsatz des Modells der Agilen Testquadranten, die erstmals von Brian Marick definiert *[9]* und später von Lisa Crispin und Janet Gregory in der Agile-Testing-Buchreihe weiter ausgestaltet wurden *[10]*. *Abbildung 2* zeigt die Agilen Testquadranten. Wir haben diese zur Einfachheit im Vergleich zum Original auf weniger Testarten reduziert und in Bezug darauf, wer testet (grauer Kasten) und wie getestet wird (türkiser Kasten), ergänzt. Die Nummerierung der Testquadranten von Q1 bis Q4 hat keine Bedeutung hinsichtlich Priorität oder Reihenfolge, sondern dient lediglich der leichteren Referenzierung des jeweiligen Testquadranten.

Die Agilen Testquadranten helfen uns zunächst dabei, das Testziel weiter zu schärfen:

- Wollen wir das Umsetzungsteam unterstützen oder das Produkt hinterfragen?
- Wollen wir uns beim Testen stärker an der Fachlichkeit oder an der Technik orientieren?

Durch Beantwortung dieser beiden Fragen kann im Modell ein Testquadrant bestimmt werden, der uns passende Testarten aufzeigt. Wollen wir beispielsweise das Produkt in Bezug auf die Fachlichkeit hinterfragen (=> Testquadrant Q3), bieten sich manuelle Tests durch die Endanwender:innen, also die Kund:innen, an. Je nach Testziel können dies User-Acceptance-Tests, explorative Tests oder Usability-Tests

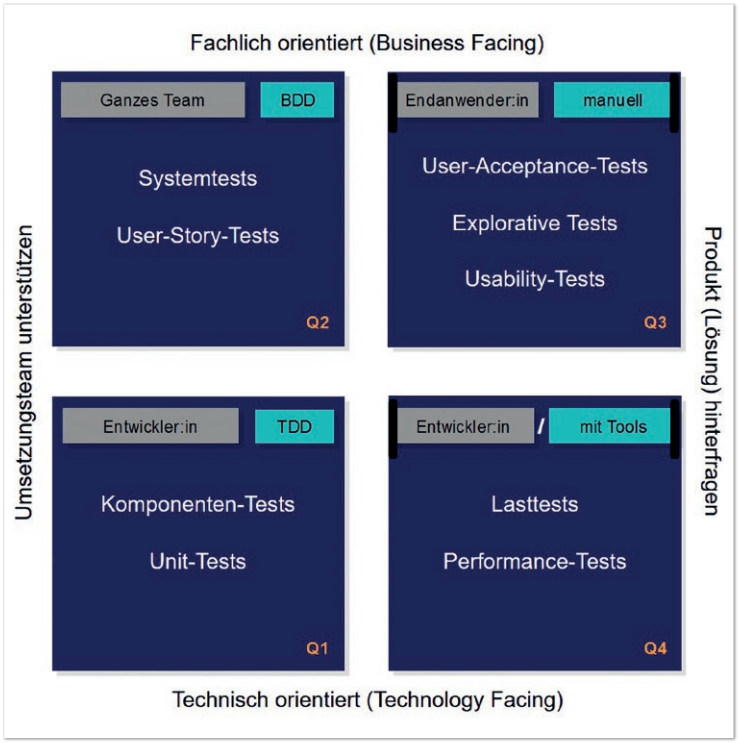


Abbildung 2: Agile Testquadranten (© Ralf Straßner, Peter Fichtner, in Anlehnung an [10])

sein. Sind wir hingegen unsicher, ob die technische Umsetzung des Produkts den benötigten Qualitätskriterien genügt, sollte der Fokus auf entsprechenden technisch orientierten Tests liegen (=> Testquadrant Q4), wie beispielsweise Performance- oder Lasttests. Hier wird also hinterfragt, ob mit den gewählten technischen Lösungsansätzen die Qualitätskriterien erreicht werden können. Diese Tests werden in der Regel über spezialisierte Tools realisiert, es könnten aber auch automatisierte Tests durch die Entwickler:innen geschrieben werden.

Geht es uns hingegen darum, das Umsetzungsteam zu unterstützen, damit es durch die Tests eine höhere Sicherheit erhält und sich gegebenenfalls auch von ihnen leiten lassen kann, kommen die beiden Testquadranten Q1 und Q2 in Frage. Um den konkreten Testquadranten zu bestimmen ist auch hier wieder die Frage, ob wir fachlich oder technisch orientiert testen wollen.

Technisch orientierte Tests, die das Umsetzungsteam unterstützen (=> Testquadrant Q1), werden von den Entwickler:innen geschrieben. Im Modell werden hier Unit- und Komponenten-Tests aufgeführt. Im Wesentlichen geht es darum, das System under Test möglichst zielgerichtet zuzuschneiden, um eine einzelne Verantwortlichkeit isoliert zu testen. Dies führt insbesondere zu kurzen Laufzeiten der Tests und dementsprechend zu schnellem Feedback. Dadurch können die Tests häufig ausgeführt werden und es bietet sich eine testgetriebene Entwicklung (Test Driven Development – TDD) an.

Fachlich orientierte Tests, die das gesamte Umsetzungsteam unterstützen (=> Testquadrant Q2), helfen diesem, ein gemeinsames fachliches Verständnis aufzubauen. Brian Marick nennt diese auch „business facing examples“ und formuliert dafür drei Ziele *[11]*: Sie sollen…

- … die Programmierer:innen dazu bringen, den richtigen Code zu schreiben.
- … die Gespräche zwischen Technologieexpert:innen und Fachexpert:innen verbessern.
- … den Fachexpert:innen dabei helfen, die mit dem Produkt verbundenen Möglichkeiten schneller zu erkennen.

Diese Ziele decken sich mit den Zielen von Behaviour Driven Development (BDD) und Domain-driven Design (DDD), deren Anwendung sich daher anbietet. Traditionell werden bei BDD, wie auch Brian Marick für diesen Testquadranten vorschlägt, Tests gegen diejenige Schnittstelle durchgeführt, welche die Endanwender:innen bedienen („whole-product interface“) *[11]*. Unserer Erfahrung nach ist jedoch ein präziserer Testschnitt möglich, wenn die Fachexpert:innen in die Lage versetzt werden, die Testfallbeschreibung auch für Tests gegen eine auf den ersten Blick eher technische Schnittstelle eigenständig zu erstellen.

Eine Voraussetzung dafür ist, dass die Testfallbeschreibung im Sinne des Testquadranten fachlich orientiert ist und inhaltlich der echten Benutzerschnittstelle ähnelt. Dies wiederum kann durch die konsequente Anwendung von Domain-driven Design erleichtert werden. Dann setzt die Schnittstelle durch die „allgegenwärtige Sprache“ (Ubiquitous Language), welche alle Teammitglieder sprechen, auf den Begrifflichkeiten der fachlichen Domäne auf und ist somit für alle Teammitglieder verständlich *[12]*. Es wird dadurch besser nachvollziehbar, wie die Daten aus der Testfallbeschreibung korrekt auf

die Schnittstelle gemappt werden, sodass das Vertrauen in diese Tests bei allen Teammitgliedern steigt.

Ein weiterer Vorteil dieses Vorgehens ist es, dass es leichter fällt, alle Testszenarien als automatisierte Tests umzusetzen, die zudem leichtgewichtig und schnell sind. Dabei helfen insbesondere die bei den Testprinzipien aufgezeigten Möglichkeiten des isolierten Testens mit voller Kontrolle über die Testdaten. Werden die BDD-Tests auf diese Weise umgesetzt, verschwimmt somit die Grenze zwischen den Quadranten der fachlich und der technisch orientierten Tests zur Unterstützung des Umsetzungsteams.

Wie können wir dies für das Testziel „Fachliche Regeln" nutzen? Die dazugehörigen Tests sollen hier *fachlich orientiert* sein und möglichst nicht von technischen Aspekten abhängen. Gleichzeitig sollen sie das fachliche Verständnis im gesamten Umsetzungsteam schärfen und so das *Umsetzungsteam unterstützen*. Es bieten sich also grundsätzlich Tests entsprechend der Testarten aus dem Testquadranten Q2 an, die durch eine gemeinsame Formulierung nach BDD das Team zur richtigen Lösung leiten sollen. Gleichzeitig sollen, wie oben beschrieben, keine schwergewichtigen Systemtests durchgeführt werden. Eine passende Lösung sind somit User-Story-Tests, die allerdings präzise zugeschnitten werden müssen. Wie dieser Testschnitt konkret aussieht und welcher Anwendungsschnitt dafür hilfreich ist, betrachten wir in der nächsten Ausgabe.

Zwischenfazit und Ausblick

Wir sind mit den Prinzipien der gängigsten Testmodelle gestartet und haben darauf aufbauend einen Weg aufgezeigt, wie wir gezielt und risikobasiert die zu unserem Kontext passenden Testziele finden und erste Ideen für die dazu passenden Testarten erhalten können. Dennoch ist dies nur ein Zwischenstand. Wie geht es weiter?

Nach unserer Erfahrung müssen die Testarten noch konkreter definiert werden. Die nächste Frage ist daher, wie das jeweilige System under Test ausgestaltet wird. Dazu werden wir den Komponentenschnitt unserer Beispielanwendung betrachten und aufzeigen, wie wir Tests so modular gestalten können, dass sie den Komponentenschnitt und das jeweilige Testziel bestmöglich in Einklang bringen.

Abschließend werden wir uns damit beschäftigen, wie der Testansatz so dokumentiert werden kann, dass er leicht zu überblicken und anzupassen ist.

Quellen

[1] Mike Cohn (2009): The Forgotten Layer of the Test Automation Pyramid (*https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid*)

[2] Mike Wacker (2015): Just Say No to More End-to-End Tests (*https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html*)

[3] Kent C. Dodds (2018): The Testing Trophy (*https://x.com/kentcdodds/status/960723172591992832*)

[4] André Schaffer, Rickard Dybeck (2018): Testing of Microservices (*https://engineering.atspotify.com/2018/01/testing-of-microservices/*)

[5] Martin Fowler (2021): On the Diverse And Fantastical Shapes of Testing (*https://martinfowler.com/articles/2021-test-shapes.html*)

[6] Rick Janda, Immo Hüneke (2014): Stop Mocking and Get Real (*https://de.slideshare.net/slideshow/stop-mockingandgetreal/36551014 – Folie 20*)

[7] Martin Fowler (2014): Unit Test (*https://martinfowler.com/bliki/UnitTest.html*)

[8] Victor Chircu (2019): Defining a Test Strategy for Continuous Delivery (*https://www.simpleorientedarchitecture.com/test-strategy-for-continuous-delivery/* – Abschnitt „Why: Identifying Risk")

[9] Brian Marick (2003): My Agile testing project (*http://www.exampler.com/old-blog/2003/08/21.1.html#agile-testing-project-1*)

[10] Lisa Crispin (2011): Using the Agile Testing Quadrants (*https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/*)

[11] Brian Marick (2003): Agile testing directions: business-facing team support (*http://www.exampler.com/old-blog/2003/09/05.1.html#agile-testing-project-4*)

[12] Martin Fowler (2006): Ubiquitous Language (*https://martinfowler.com/bliki/UbiquitousLanguage.html*)

Ralf Straßner

ralf.strassner@atruvia.de

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

Peter Fichtner

peter.fichtner@atruvia.de

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

Strukturzementierende Tests und wie man sie vermeidet, Teil 3: Das richtige Vorgehen

Richard Gross, MaibornWolff GmbH

Tests sollten auf Verhaltensänderungen reagieren aber dennoch unempfindlich gegenüber Strukturveränderungen sein. Tests, die die zweite Bedingung nicht erfüllen, nennen wir strukturzementierend. Sie können auf zwei Arten entstehen. In Teil 1 haben wir uns mit der ersten Art beschäftigt: der Strukturzementierung durch redundantes Test-Setup. In diesem Teil erarbeiten wir, wie wir die zweite Art der Strukturzementierung verhindern können: die durch Tests auf falscher Ebene.

Tests sind auf zwei Arten strukturzementierend:

1. Durch redundantes Test-Setup: Tests nutzen an mehreren Stellen immer wieder dieselbe Struktur. Je öfter dieselbe Struktur (Klasse, Methode) verlangt und je exzessiver sie gemockt wird, desto zementierter ist das Design.
2. Durch Tests auf falscher Ebene: wenn instabile Elemente anstelle von Modulen getestet werden.

Die erste Art der Zementierung sind wir durch die in Teil 1 implementierte TestDsl (Domain-Specific Language *[1]*) umgangen. Diese ist eine Abstraktionsschicht zwischen Test und Test-Setup, mit der man den eigentlichen Test von den Vorbedingungen, die man zum Testen herstellen muss, entkoppelt.

Die TestDsl beantwortet uns aber nicht die Frage, ob wir etwas testen sollen. Viele Elemente in unserem Code sind Implementierungsdetails von Modulen und damit sehr instabil. Tests dieser Details sind dann trotz Dsl strukturzementierend.

Im Folgenden schauen wir uns genauer an, wie Tests diesen Effekt haben können, was wir stattdessen testen sollten und schließlich, was das richtige Vorgehen ist, damit wir frei bleiben, Struktur zu verändern.

Die fünf Säulen guter Tests

Zuerst sollten wir anschauen, was einen guten Test eigentlich ausmacht. Das Buch Unit Testing Principles, Practices, and Patterns *[2]* benennt vier Säulen von guten Unit Tests:

1. Schützt vor Regressionen (Protection against regressions)
2. Frei von Strukturzementierung (Resistance to refactoring)
3. Wartbarkeit (Maintainability)
4. Schnelles Feedback (Fast Feedback)

Damit wir den Anwender bestmöglich vor Regressionen schützen, sollten unsere Tests zum einen so nah wie möglich an ihm dran sein. Bestenfalls sollten die Tests genau dieselbe Perspektive einnehmen und genau das machen, was der Anwender auch macht. Zum anderen sollte der Test so viel Code wie nur möglich ausführen. Je mehr Code, desto höher die Chance, dass wir Regressionen finden.

Die zweite Säule besagt, dass Tests die Struktur nicht zementieren sollen. Wir fokussieren uns hier auf die Zementierung durch Tests, die auf falscher Ebene ausgeführt werden. In *Abbildung 1* sehen wir, dass wir eine 1:1-Beziehung zwischen Tests und Code-Elementen haben. Das führt zu maximaler Zementierung, denn jedes Element x wird durch ein `xTest` festgehalten. Es ist uns gar nicht erst möglich, Verhalten von B nach C zu verschieben, denn dadurch würden sich die Tests rot färben.

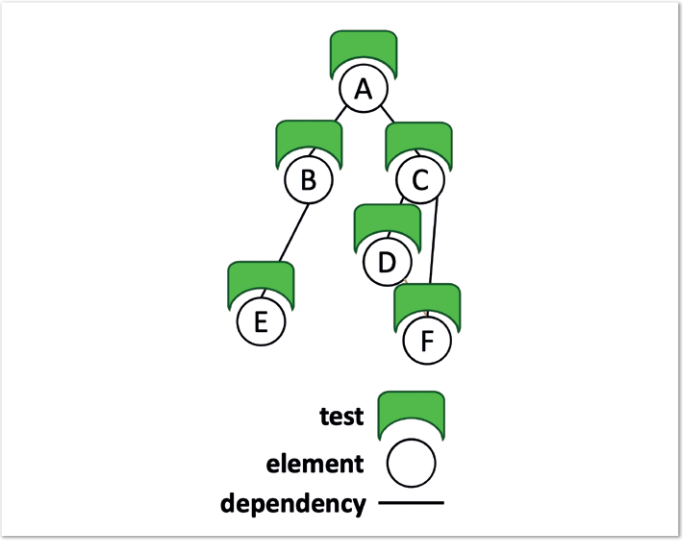


Abbildung 1: Test zementieren auf jeder Ebene. (© Richard Gross)

Tests sollten entsprechend der dritten Säule auch wartbar sein. Gemeint ist damit, ob sie leicht lesbar und ausführbar sind sowie wie lang und komplex der Test-Code ist. Ein 50-Zeilen-Test, für den wir fünf Docker-Container starten müssen, ist deutlich schwieriger zu warten als ein Drei-Zeiler, den wir mit einem Click starten können.

Die vierte Säule guter Unit-Tests bezieht sich darauf, wie schnell die Tests Feedback geben.

Wir erlauben uns an dieser Stelle, die Aufzählung noch um eine **fünfte Säule** zu erweitern: „Präzises Feedback". Je präziser das Feedback, desto weniger Code müssen wir sichten, um Fehler zu finden.

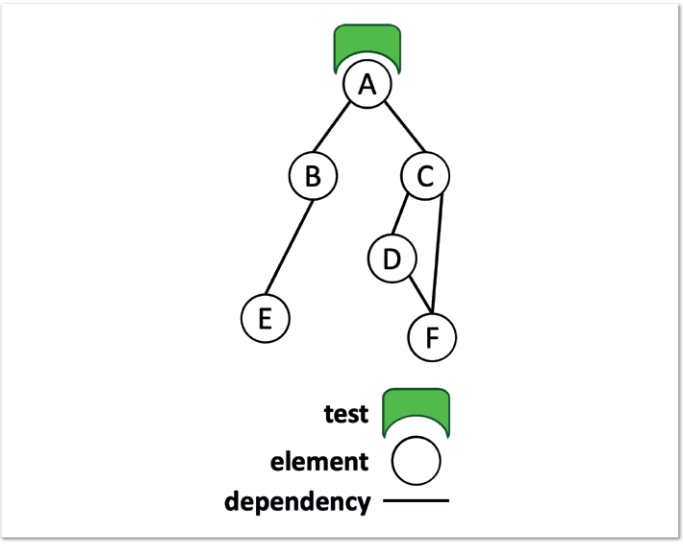


Abbildung 2: Ein High-Level-Test (© Richard Gross)

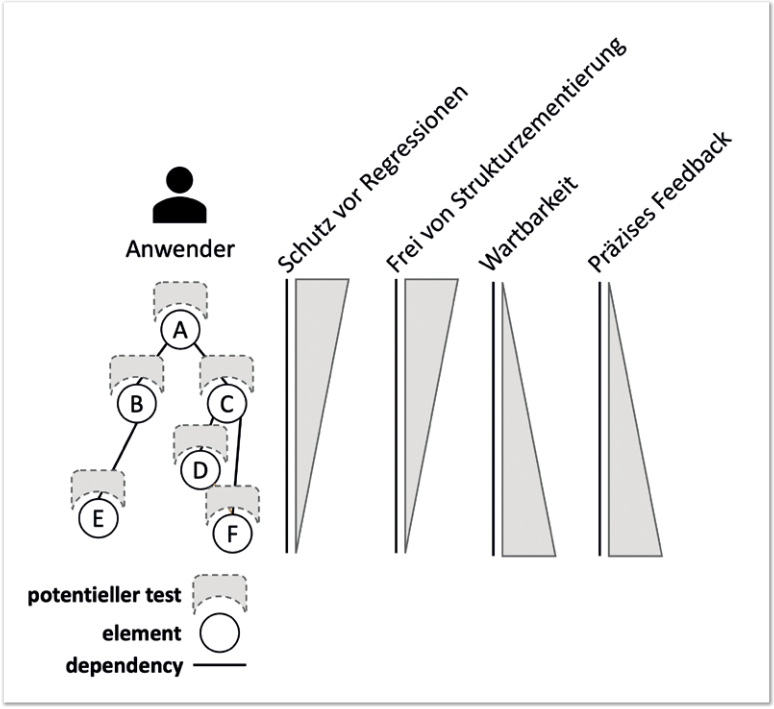


Abbildung 3: Wir müssen zwischen den Testsäulen abwägen. (© Richard Gross)

In *Abbildung 2* testen wir zwar A, aber wenn der Test fehlschlägt, kann sich der Fehler auch in B, C, D, E und F verstecken. Um das klar herauszustellen, führen wir noch die fünfte Säule ein.

Die Säulen können nicht alle vollständig erfüllt werden. Pro Test müssen wir uns entscheiden, was wir haben wollen. Ein Test von sehr weit außen führt sehr viel Code aus, ist damit gut gegen Regressionen geschützt und kaum zementierend. Wenn dieser Test allerdings rot wird, dann ist das Feedback sehr unpräzise. Des Weiteren benötigt man sehr viel Test-Setup-Code, was die Wartbarkeit reduziert *(siehe Abbildung 3)*. Hinzu kommt, dass man so weit außen viele Randfälle entweder gar nicht oder nur mit sehr komplexem Setup überhaupt nachstellen kann.

Mit Hilfe zweier Kniffe sieht das Bild allerdings schon etwas angenehmer aus. Durch die TestDsl können wir die Wartbarkeit von Tests mit viel Setup signifikant verbessern. Mit dem richtigen Vorgehen und der richtigen Architektur können wir die Präzision des Feedbacks erhöhen *(siehe Abbildung 4)*.

In unserer Übersicht haben wir dabei eine Säule bewusst ausgelassen, nämlich die des schnellen Feedbacks. Diese Säule hängt nicht davon ab, auf welcher Ebene wir testen, sondern wie viel Außenwelt involviert ist. Wenn ein Element direkt mit der Außenwelt kommuniziert, dann ist jeder Test, der dieses Element involviert, signifikant langsamer. Wir müssen schließlich die Außenwelt bereitstellen und die Kommunikationswege (Netzwerk, Browser-Clicks) kosten auch noch Zeit. Je mehr Außenwelt wir nutzen, desto langsamer wird ein Test. Daher ist T2 signifikant schneller als T3 und T3 wiederum deutlich schneller als T1 *(siehe Abbildung 5)*.

Die Bilder sprechen eine klare Sprache. Wenn wir den Fokus auf Schutz vor Regressionen und auf das Vermeiden von Strukturzementierung legen, dann müssen wir von weit oben testen. Doch nicht zu weit oben, denn Randelemente kosten uns zu viel Ausführungszeit. In Kauf nehmen müssen wir dann den erhöhten Wartungsaufwand und geringere Präzision, wenn ein Test fehlschlägt. Den Spagat können wir nur mit dem richtigen Vorgehen lösen. Bevor wir über dieses reden, müssen wir aber noch schnell klären, was Module sind, denn nur diese lohnt es sich überhaupt zu testen.

Modularität

Unsere Software ist ein großer Baum von Elementen (Funktionen oder Klassen), die sich direkt oder indirekt aufrufen. Es gibt eine Wurzel („A" in *Abbildung 1* bis *Abbildung 5*), die transitiv alle Elemente kennt, und ganz viele Blattknoten, die niemanden kennen (zum Beispiel „E" und „F" in *Abbildung 5*). Manche Abschnitte des Baumes sind dabei Module (und in diesen sind wieder Submodule), viele andere sind es nicht, denn Module bringen zwar viele Vorteile, man muss allerdings auch (viel) Aufwand in ihr Design stecken. Letzteres führt dazu, dass so manche Software zwar ein Modulsystem nutzt, es de facto aber keine Module beinhaltet. Die Vorteile von Modulen liegen dabei klar auf der Hand (*[3]*):

1. Wenig Kommunikation nötig – jede Gruppe kann an ihrem Modul arbeiten, ohne sich mit anderen austauschen zu müssen.
2. Produktflexibilität – drastische Änderungen an einem Modul sind möglich, ohne dass andere danach geändert werden müssen.
3. Verständlichkeit – ein System kann modulweise untersucht werden. Um ein Modul zu verstehen, müssen wir nicht erst alle umliegenden verstehen. Das gesamte System kann daher besser entworfen werden, da es besser verstanden wird.

Kurz, wir haben Module, wenn wir Systeme so gestalten, dass Developer zu jedem beliebigen Zeitpunkt nur einen kleinen Teil der Komplexität im Kopf behalten müssen.

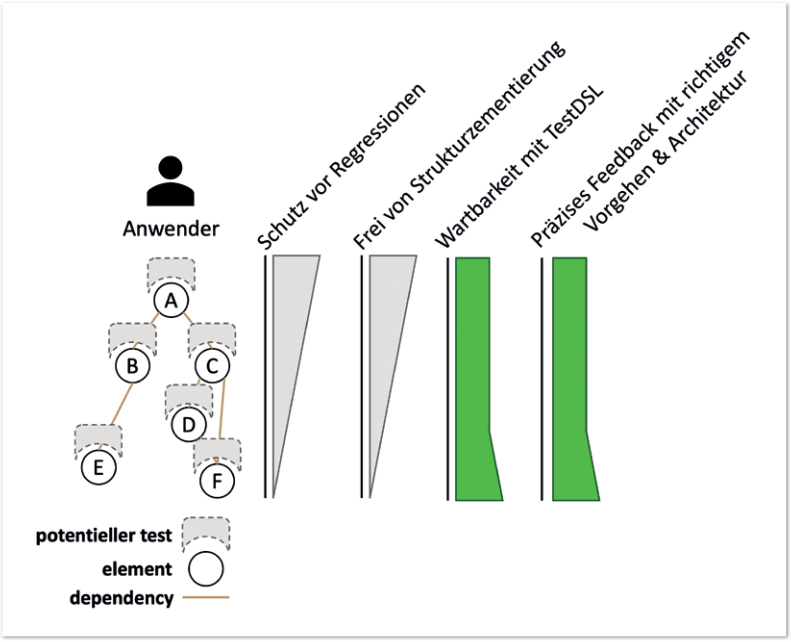


Abbildung 4: Mit TestDsl und Vorgehen wird das Abwägen leichter. (© Richard Gross)

Ob wir diese Vorteile haben, erfahren wir durch Unit-Tests, denn sie sind Tests der Modularität [4]. Zum einen im Negativen: Wenn unser Test-Setup sehr anstrengend ist, wenn es schwierig ist, zu beschreiben, was der Test macht, wenn Tests sich andauernd rot färben, obwohl wir nicht an diesem Modul gearbeitet haben; kurz, wenn der Test eine Strafe ist, dann haben wir vermutlich kein Modul. Wenn die Tests dagegen gut beschreiben, was das Element macht und sie uns helfen, das Verhalten als Ganzes zu verstehen, dann testen wir vermutlich ein Modul.

Doch Tests leiten uns nicht dabei, Module zu designen. Diese Aufgabe übernehmen drei Prinzipien: Module sollten so wenig wie möglich über ihre interne Implementierung preisgeben (*information hiding principle* [3]), ihre Schnittstellen sollten keine Überraschungen bieten (*principle of least astonishment* [5]) und garantieren, dass sie immer im erlaubten Zustand sind (*encapsulation* [6]).

Das erste Prinzip fordert, dass alles, was ein Modul nach außen preisgibt, minimal ist. Das Modul sollte möglichst wenig Methoden mit möglichst wenig Parametern bereitstellen und in diesen sollten möglichst wenig Felder sein. Ein `rentBook(Book, User)` ist beispielsweise schlechter als ein `rentBook(BookId, UserId)`. Auch sollte das Modul möglichst wenig Abhängigkeiten nach außen preisgeben. Eine Abhängigkeit auf ein generisches interface ist dabei besser als eine Abhängigkeit auf eine `class`. Ein `interface` beschreibt nur, was das Modul braucht, und nicht, wie es zu erfüllen ist. Wir sollten aber auch nicht alle Klassen durch generische ersetzen. Ein generisches `interface` fördert zwar Modularität, da wir nicht wissen müssen, wer das `interface` implementiert, wir verlieren dadurch aber den Blick, wie Informationen durch das System fließen.

Handelt man entsprechend des *information hiding principle* [3] erhält man bereits Module. Die anderen zwei Prinzipien stärken dies nur. Das *principle of least astonishment* [5] fordert gut gewählte Namen oder, falls das nicht möglich ist, ein Redesign des Moduls, damit es danach überraschungsfreie Namen besitzt. *Encapsulation* [6] fordert schließlich, dass die Schnittstelle des Moduls nichts zulässt, was einen falschen Zustand erzeugen würde.

Diese Prinzipien führen dazu, dass ein Modul immer einen exponierten Teil hat (die Schnittstelle oder API des Moduls), und einen internen Teil (die Implementierungsdetails). Wenn wir in einer Programmiersprache ohne Modulsystem leben, dann tun wir gut daran, diese Unterscheidung über Ordner abzubilden:

Andere Module dürfen von außen auf *exposed* zugreifen, aber niemals auf *hidden*. Diese Struktur können wir als Developer per gemeinsamer Konvention einhalten, oder mit Tools wie *ArchUnit* [7] zur Test-Time verifizieren. In der Programmiersprache Java haben wir zusätzlich noch drei Ebenen, in denen wir technisch Module erzwingen können:

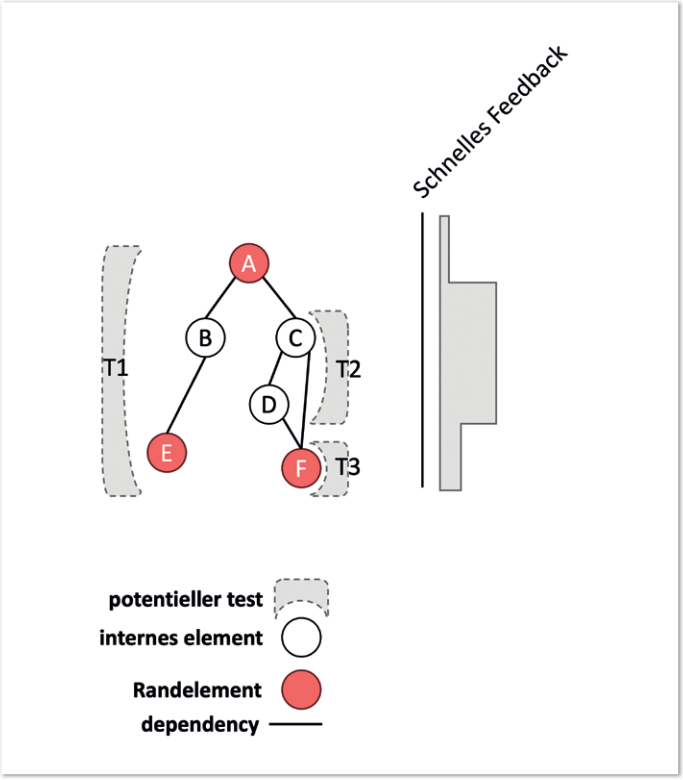


Abbildung 5: Mit TestDsl und Vorgehen wird das Abwägen leichter. (© Richard Gross)

- `public`/`private` in Klassen → Unser Modul kann dann aber nur so groß sein wie eine Datei.
- Packages, in denen *hidden* Klassen `package` *private* sind und nur *exposed* Klassen als `public` deklariert werden → Erlaubt große Module, aber keine Gruppierung innerhalb der Module.
- Für ganze `jars` per JPMS [8] → Erlaubt große Module und Gruppierung. Das Deklarieren einer `jar` erzeugt allerdings Overhead, der sich nur für *root-level* Module lohnt.

Es gibt also viele Möglichkeiten, ein Modul zu deklarieren. All diese Möglichkeiten helfen uns aber nur, wenn wir auch so viel wie möglich geheim halten.

> *Maintainability [and modularity] is inversely proportional to the number of [exposed] [...] classes, dependencies, microservices*
>
> *– Simon Brown [9]*

Es ist dabei durchaus erlaubt, dass ein Modul intern wieder aus Modulen besteht. Diese Submodule entstehen typischerweise bei der täglichen Arbeit, wenn wir Funktionalität auslagern. Ein leicht zu schreibender Unit-Test hilft uns dabei, herauszufinden, ob wir hier ein Submodul ausgelagert haben.

Es kann beim Arbeiten allerdings auch vorkommen, dass wir kleine Teile aus Modulen herauslösen, die dann nicht für sich stehen, sondern nur im Kontext vom ursprünglichen Modul Sinn ergeben. Das sind Implementierungsdetails, instabile Elemente. Wir sollten uns

jederzeit entscheiden können, sie wieder in das Modul zu re-integrieren, umzuschreiben oder zu löschen. Für diese Elemente schreiben wir daher meist keine Tests. Sie werden mit dem eigentlichen Modul mitgetestet. Wenn wir uns allerdings doch entschließen, Tests zu schreiben, dann müssen wir sie so auslegen, dass sie ein **Wegwerfprodukt** sind. Das bedeutet, dass wir Fachlichkeit niemals exklusiv an einem instabilen Element testen, sondern immer auf höherer Ebene. Damit können wir auf dieser instabilen Ebene nur Implementierungsdetails testen. Tests, die diese Details zementieren, sollten wir wegschmeißen können, wenn unser instabiles Element sich zu sehr verändert.

Das „richtige" Vorgehen

Das richtige Vorgehen – das Vorgehen, das alles retten wird, der Weisheit letzter Schluss. Hochtrabender hätte man es nicht ankündigen können. Deshalb zwei Dinge vorweg: Erstens sind andere Teams zu ähnlichen Ideen gekommen *[10]*, *[11]*. Neu ist vielleicht die Kombination mit der TestDsl, aber auch hier gibt es vermutlich mir unbekannte Vordenker. Zweitens: „das **richtige** Vorgehen" ist immer Team-spezifisch. Wenn ihr, liebe Leser, bereits ein Vorgehen habt, das für euch funktioniert und Strukturzementierung vermeidet, dann habt ihr bereits das richtige Vorgehen gefunden.

Das richtige Vorgehen besteht aus zwei Aspekten: eine auf Testbarkeit ausgelegt Struktur (*Design for Testability* *[12]*) und Feedbackschleifen auf unterschiedlichen Ebenen.

Design for Testability

Design for Testability (DFT *[12]*) ist ursprünglich eine Sammlung an Techniken, um Hardware während der Fertigung testbar zu machen. Das Ziel ist, festzustellen, ob sich Fehler während der Fertigung eingeschlichen haben. Um das umzusetzen, verankern wir Testbarkeit in unserem Design.

Bei Software ist die Situation ähnlich. Wir müssen Testbarkeit von Anfang an einplanen. Wir sehen in unserer Struktur explizite Schichten und Strukturen vor, an denen wir testen können. Testbarkeit im Nachgang einzubauen ist möglich, aber um Größenordnungen schwieriger, weshalb es oft nicht getan wird.

Design for Testability auf Software zu übertragen, setzt weit vor der Fertigung an und versucht, das Design so auszulegen, dass wir auf unterschiedlichen Ebenen Tests fahren können. Wir betrachten hier drei Techniken, die wir anwenden können: Layering, root-level Module und Orchestrierung über eine Facade, um die Testfall-Explosion zu vermeiden.

Layering

Für eine gute Testbarkeit teilen wir unseren Code in Presentation, Domain oder Data *[13]* auf. Wenn diese Layer aufeinander zugreifen wollen, dann tun sie das über Ports *[14]*. Ports sind Interfaces zur Entkopplung dieser Schichten. Implementiert werden sie entweder von Domain oder von Data (siehe auch Teil 1 des Artikels). Über primäre Ports (P1, P2, da sie oben liegen, nenne ich sie auch *Canopy Ports*) wecken wir das Modul aus dem ruhenden Zustand und lösen Verhalten aus. Über sekundäre Ports (da sie unten liegen, nenne ich sie auch *Floor Ports*) löst das Modul Verhalten in der Außenwelt aus. Damit haben wir unsere Domäne klar von Randelementen abgetrennt *(siehe Abbildung 6)*. Wir gewinnen zum einen Modularität. Die

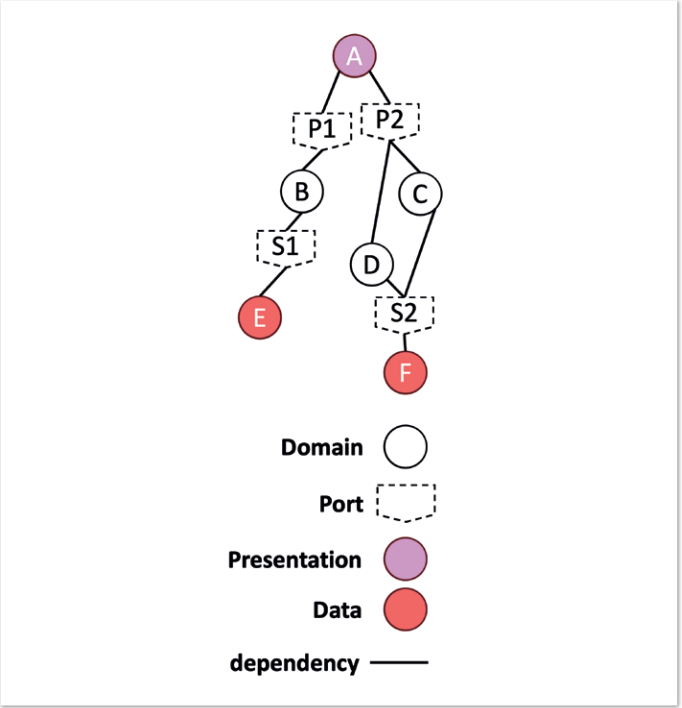


Abbildung 6: Port, Presentation, Domain, Data Layering (© Richard Gross)

Domäne weiß über die Außenwelt nur, was die Ports deklarieren. Zum anderen sind unsere Tests der Domäne schneller, weil die Ports uns erlauben, die Außenwelt zu stubben oder zu faken.

Schneiden von Systemen

Als nächstes schneiden wir unser System in fachliche root-level Module *(siehe Abbildung 7)*. Der einzige Weg in das Modul hinein geht über den bereits eingeführten primären Port. Module können mehrere primäre und sekundäre Ports haben. Die Module selbst können sich gegenseitig aufrufen, allerdings nur über den primären Port. Die

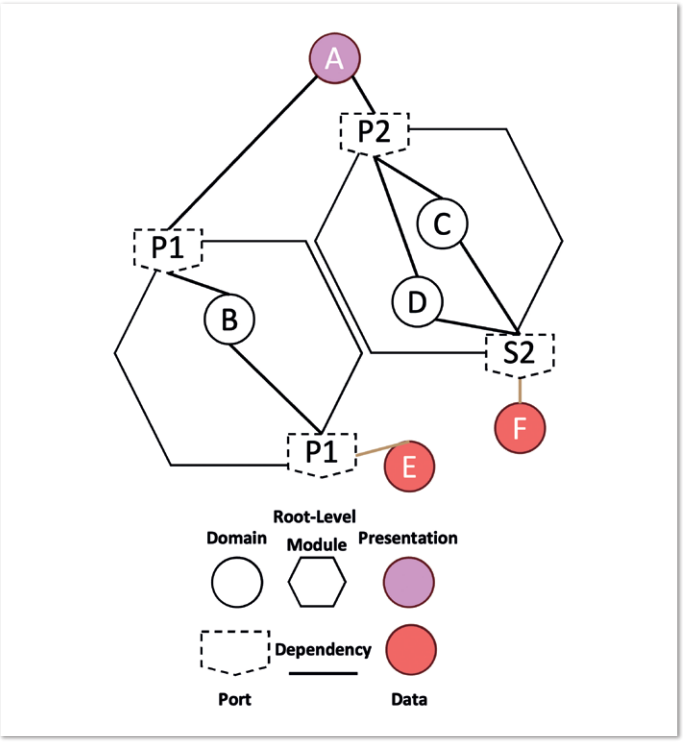


Abbildung 7: Modulschnitt (© Richard Gross)

primären Ports sind *exposed* für andere Module, die Interna der Module („B", „C", „D") sind für andere root-level Module *hidden*.

Beim Schneiden der Module orientieren wir uns an der Domäne, die wir modellieren, denn nur wenn Fachlichkeit zuerst kommt, können wir ihre Weiterentwicklung beschleunigen. Maßgabe hier ist „wire together what fires together" (angelehnt an die Hebbsche Lernregel *[15]*). Wir orientieren uns nicht an den Nomen, sondern an den Verben der Domäne, denn diese repräsentieren die möglichen Aktionen und wir können diese Aktionen direkt als Methoden (`rentingService.rentBook`) in unsere Ports schreiben.

Die primären Ports sind exzellente Ziele für rasend schnelle Unit-/Small-Scope-Tests. Sie sichern gut gegen Regressionen ab und sind refactoring-stabil. Außerdem sind sie schnell, weil sekundäre Ports gefaket oder gestubbet werden können. Vor allem aber sichern sie die fachlich gewünschte Aktion ab. Mit einer TestDsl kann man sie auch wartbar schreiben.

Weiter außen testen sollte man nur in Außnahmefällen, denn dann ist man auf dem *Presentation Layer* unterwegs und das birgt einige Tücken. Zum einen kann dieselbe Fachlichkeit über die Jahre auf unterschiedliche Art nach außen repräsentiert werden. Man fängt vielleicht an mit HTTP+JSON, erweitert später um einen CSV-Import und schließlich kommt noch gRPC hinzu. Testen wir auf Präsentationsebene, müssen wir jetzt jede Fachlichkeit drei Mal testen. Es ist deutlich effektiver, die Fachlichkeit nur einmal mit dem primären Port zu testen.

Testen wir gegen den *Presentation Layer*, haben wir aber noch ein Problem: Wir haben deutlich weniger Informationen und Möglichkeiten des Setups. Typischerweise präsentieren wir nach außen nur minimale APIs und minimale Daten. Zum einen aus Sicherheitsgründen und zum anderen, damit wir in der Weiterentwicklung nicht gehemmt sind (*information hiding principle [3]*). Je mehr wir präsentieren, desto mehr müssen wir dauerhaft supporten. Tests auf Präsentationsebene können daher viel weniger sehen sowie asserten und haben eventuell auch gar nicht die APIs, um das gewünschte Test-Setup herzustellen. Im Test-Setup dann teilweise Methoden der Domäne zu nutzen, führt instantan zu schwer lesbaren und unwartbaren Tests. Wenn wir den *Presentation Layer* testen, dann machen wir das also nicht mit dem Ziel, Fachlichkeit zu testen. Unser Ziel ist, zu verifizieren, dass *Presentation Layer* und *Domain Layer* richtig zusammenarbeiten. Das können wir dann situativ für wichtige Happy und Sad Cases durchführen.

Auf den ersten Blick sollten wir Fachlichkeit also immer gegen die primären Ports testen und benötigen gar keine andere Testebene. In der Praxis sollten wir aber noch etwas tiefer im Modul testen, damit wir eine Testfall-Explosion vermeiden. Wir müssen allerdings mit Bedacht tiefergehen, damit unsere Tests gleichzeitig möglichst refactoring-stabil bleiben.

Vermeiden der Testfall-Explosion

Root-level Module muss man auf eine bestimmte Art und Weise schneiden, damit man die von J. B. Rainsberger geschilderte **Testfall-Explosion** *[16]* vermeidet:

Stellen wir uns vor, unser Modul würde aus drei Elementen bestehen. In den Elementen sind fachliche Verzweigungen mit Hilfe von `if`- und `switch`-Statements modelliert. Das erste Element hat 3 Pfade, das zweite 4 und das dritte ganze 5 Pfade. Wenn die Elemente sich der Reihe nach Aufrufen, gibt es 60 Pfade (3 * 4 * 5) durch das Modul (siehe *Abbildung 8*, stark angelehnt an eine Grafik von Urs Enzler *[11]*).

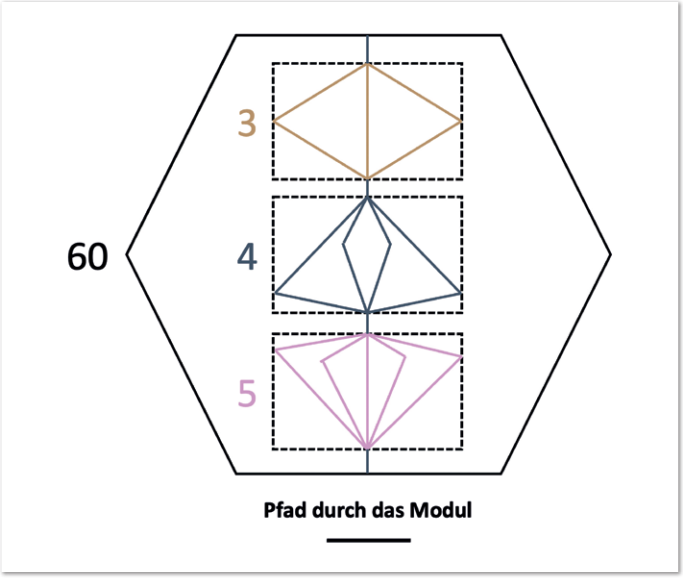


Abbildung 8: Die Testfall-Explosion (© Richard Gross) – im Original Urs Enzler

Wenn wir von außen testen, müssen wir 60 Testfälle schreiben. In einem echten System hätten wir noch viel mehr Verzweigungen und würden aus dem Testschreiben nicht mehr herauskommen.

Die von Urs Enzler beschriebene Lösung *[11]* ist, eine Facade einzuführen, die die Orchestrierung der Subelemente übernimmt. Da die Facade nur der Reihe nach aufruft, gibt es nur 13 Fälle zu testen (1 + 3 + 4 + 5).

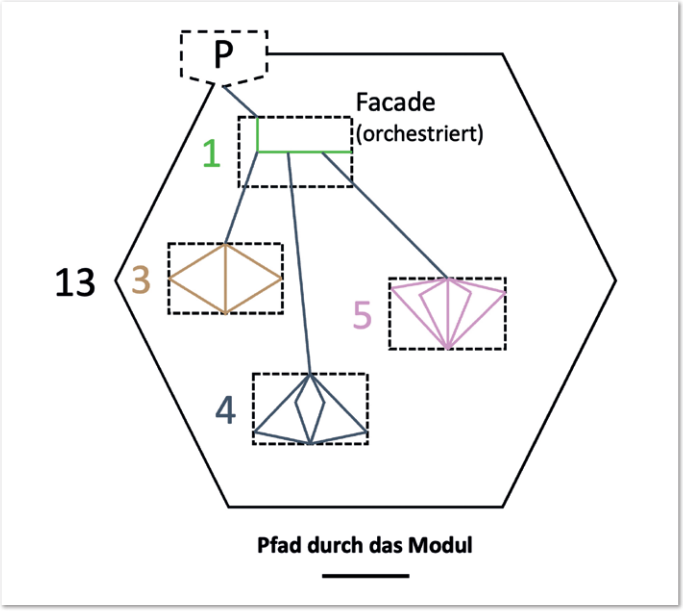


Abbildung 9: Mit Orchestrierung die Testfall-Explosion vermeiden. (© Richard Gross) – im Original Urs Enzler

Damit wir die Testfall-Explosion vermeiden, sollte unser Port `interface` also von einer `class`-Facade implementiert werden, die alle Elemente der Reihe nach orchestriert. Gegen den Port fahren wir dann **Modul**-Tests.

Was die Facade orchestriert, sollten wiederum Module sein, denn bei instabilen Elementen lohnt sich der Test nicht. Diese Module fallen in eine von zwei Kategorien: sie sind entweder *Calculations* oder *Actions* (die Begriffe stammen aus Grokking Simplicity *[17]*). Diese Kategorisierung gibt uns nicht nur eine Struktur vor *(siehe Abbildung 10)*, sondern beeinflusst auch unser Testvorgehen.

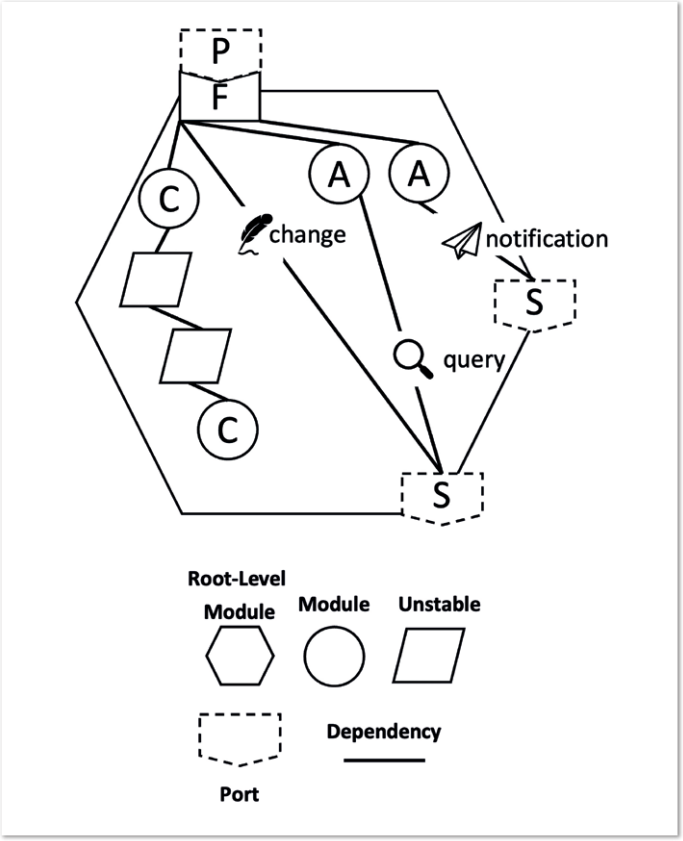


Abbildung 10: Ein Modul besteht aus Calculations und Actions. (© Richard Gross)

Calculations haben keinen Effekt auf die Außenwelt und greifen daher nie auf Ports zu. Sie sind deterministisch. Für einen gegebenen Input werden sie immer den gleichen Output berechnen. Calculations sind sehr angenehm zu testen, da das Test-Setup sehr kompakt ist.

Actions dagegen haben einen Effekt auf die Außenwelt. Sie können Datenbank-Operationen ausführen oder Nachrichten verschicken. Actions selbst sind noch einmal aufgeteilt in Queries, Changes und Notifications.

Query-Actions, wie zum Beispiel ein lesender Zugriff auf die Datenbank, verändern die Außenwelt nicht. Man kann sie relativ gefahrlos einsetzen. *Queries* zu unittesten, ist zwar möglich, birgt aber eine Gefahr: Es gibt Queries, die einfach nur das Resultat vom sekundären Port durchreichen. Ein Test ist dann Zeitverschwendung, da der Port bereits getestet ist. Es sollten eher komplexe Queries getestet werden, in denen die Logik in der Domain implementiert ist.

Change-Actions, beispielsweise schreibende Datenbank-Operationen, sind gefährlich. Sie verändern den Zustand und machen damit unser Modul nicht-deterministisch. Damit man sie schnell findet, sollten sie nur im primären Port zum Einsatz kommen und sich nicht überall verteilen. Der Test dafür ist also immer ein Modul-Test.

Bei **Notification-Actions** sieht es ähnlich aus wie bei Changes. Wenn ein Modul eine Notification rausschickt, beispielsweise über eine Message in *RabbitMQ [24]*, dann wissen wir nicht, wer diese Notification konsumiert und was er damit macht. Eine Notification ist daher auch ein Change, denn sie führt zu Nicht-Determinismus in konsumierenden Modulen. Vom Code her ist eine Notification allerdings einfach. Sie übersetzt eine *hidden* Message in eine *exposed* Message und schickt letztere weiter in die Außenwelt. Der Test hierfür ist trivial. Wichtiger ist es, einen Modul-Test zu schreiben, der überprüft, dass die Notification unter den richtigen Bedingungen rausgeht.

Wenn wir die Modul-Kategorien und die Facade auf unsere root-level Module anwenden, dann ergeben sich in der Praxis drei Ebenen, auf denen wir fast immer gute Module finden und das mit Unit-/Small-Scope-Tests verifizieren können:

- Primärer Port des Moduls → Modul-Tests
- Orchestrierte Calculations und Actions → Calculation- oder Action-Tests
- Calculations, die Blattknoten sind. → Calculation-Tests

In der trüben Mitte (*murky middle*) eines root-level Moduls finden wir dagegen oft Implementierungsdetails, weshalb wir hier nicht testen *(siehe Abbildung 11)*.

Damit haben wir eine Struktur, die auf das Testen ausgelegt ist. Im Folgenden sehen wir noch, wie wir unsere Tests aufteilen, damit wir mit möglichst wenig Tests mit möglichst viel Feedback bekommen.

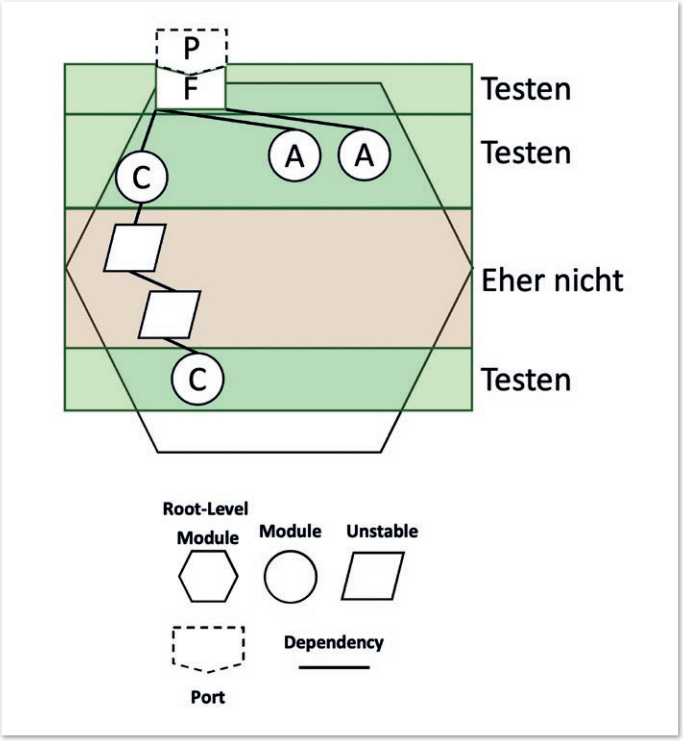


Abbildung 11: Lieber nicht in der „trüben Mitte" testen. (© Richard Gross)

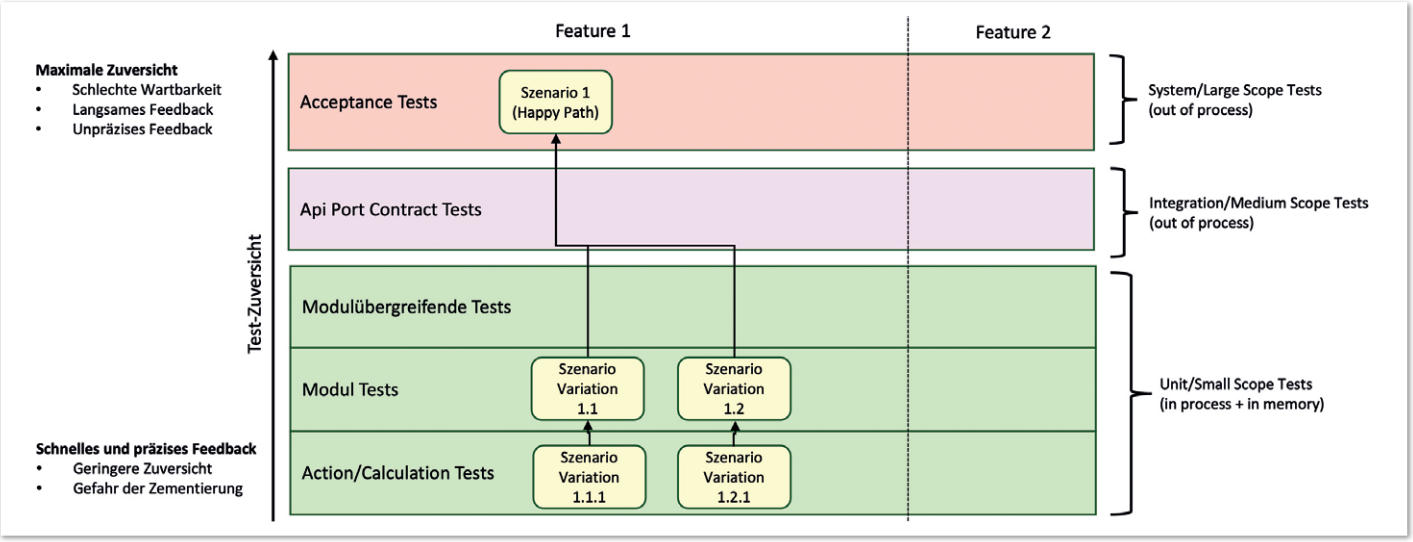


Abbildung 12: Test-Rezepte visualisiert (© Richard Gross)

Feedbackschleifen

> I get paid for code that works, not for tests, so my philosophy is to test as little as possible to reach a given level of confidence
>
> – Kent Beck
>
> (in einer Diskussion über die Menge an Unit-Tests) *[18]*

Wir Developer werden für Features bezahlt, nicht für Tests. Die Tests sind für uns nur Feedbackschleifen, die uns sagen, ob wir das richtige gebaut haben. Selbst Kent Beck, der Vater von Test-Driven Development, weist uns freundlich darauf hin, nur so viel Feedback zu holen, wie man wirklich braucht.

Das vorgestellte Design for Testability liefert die Punkte, an denen wir unsere Tests ansetzen können. Bisher blieb jedoch offen, was wir an diesen Punkten genau testen. Im Folgenden möchten wir kurz den „Test-Rezept"-Ansatz (test recipe) aus dem Buch *The Art of Unit Testing [19]* vorstellen.

Die Idee hinter dem Test-Rezept-Ansatz ist es, kurz vor dem eigentlichen Code abzusprechen, was wir auf welcher Ebene testen und das als informalen Plan festzuhalten. Wenn wir das Beispiel des Ausleihens eines Buches aus den vorigen Teilen aufgreifen, könnte ein Test-Rezept wie folgt aussehen.

„Buch ausleihen"-Feature Test-Rezept:

> Acceptance – Login, geh in die Buchübersicht, leihe ein Buch aus, gehe ins eigene Profil, Buch sollte ausgeliehen sein.
> Module – Leihe ein Buch aus.
> Action (Query) – Überprüfe, dass nur verfügbare Bücher angezeigt werden.
> Action (Change) – Überprüfe, dass ein ausgeliehenes Buch nicht noch mal ausgeliehen werden kann.
> Calculation – Überprüfe, dass der Leihpreis richtig berechnet wird.

Der Plan sollte immer den Happy Path und alle Edge Cases beinhalten. Ziel ist es, als Team zu entscheiden, was am meisten Sinn macht, auf welcher Ebene zu testen. Dieser Plan ist dann auch nicht fest in Stein gemeißelt, sondern wird iterativ verbessert. Wir können ihn direkt im Ticket festhalten. Mit dem Plan können wir Test-Überschneidungen vermeiden und unsere Test-Zeit minimieren,

```
@Layer("System")
class RentingSytemTest {

  @Test
  @Description("Der Benutzer meldet sich an, geht in die Buchübersicht und leiht ein Buch aus. Danach überprüft er,
dass im eigene Profil das Buch auch als ausgeliehen markiert ist")
  @Severity(CRITICAL)
  @Persona("Alex Whistler")
  @Link(name = "Website", url = "https://dev.example.com/")
  @Feature("RNTR-111")
  @Story("RNTR-1112")
  @Scenario("1")
  void should_be_able_to_rent_book() {
     // ...
  }
}
```

Listing 1: Allure-Annotationen

Property	Originalerläuterung	Vorgehen mit TestDsl
Isolated	Tests should return the same results regardless of the order in which they are run.	✓ Pro Test wird eine neue Dsl mit neuem Floor initialisiert.
Fast	Tests should run quickly.	✓ Die Unit-Tests sind sehr schnell.
Writable	Tests should be cheap to write relative to the cost of the code being tested.	(✓) Wenn die Dsl erstellt ist, sind alle aufbauenden Tests schnell zu schreiben.
Readable	Tests should be comprehensible for readers, invoking the motivation for writing this particular test.	✓ Die Dsl macht Tests deutlich fokussierter und leichter zu lesen.
Structure-insensitive	Tests should not change their result if the structure of the code changes.	✓ ☺
Automated	Tests should run without human intervention.	✓
Deterministic	If nothing changes, the test result shouldn't change.	✓ Pro Test wird eine neue Dsl mit neuem Floor initialisiert.
Composable	If tests are isolated, then I can run 1 or 10 or 100 or 1,000,000 and get the same results.	✓ Die Test-Rezepte helfen uns bei der Überschneidungsfreiheit.
Behavioral	Tests should be sensitive to changes in the behavior of the code under test. If the behavior changes, the test result should change.	(✓) Kommt auf unser Vorgehen beim Erstellen der Test-Rezepte an. Nutzen wir beim Entwickeln TDD [23], dann haben wir per Definition für jede Verhaltensänderung einen Test.
Specific	If a test fails, the cause of the failure should be obvious.	(✓) Unsere Tests sind oft weiter weg vom Verhalten. Sie sind meist spezifisch genug, dass wir nicht lange suchen müssen.
Inspiring	Passing the tests should inspire confidence.	
Predictive	If the tests all pass, then the code under test should be suitable for production.	

Tabelle 1: Vorgehen+TestDsl Desiderata

während wir unsere Feature-Zeit maximieren.

Wenn wir den Rezept-Ansatz auf die bisher vorgestellte Struktur anwenden, ergibt sich ein klares Vorgehen mit überschneidungsfreien Tests *(siehe Abbildung 12)*.

Port-Contract-Tests und wie wir Akzeptanz-Tests mit einer High-Level TestDsl wartbar machen können, haben wir in Teil 2 gesehen. Die Modul- und Action-/Calculation-Tests wurden umfangreich in diesem Teil beschrieben. Einzig die modulübergreifenden Tests kennen wir noch nicht. Diese brauchen wir in den Fällen, in denen ein Feature sich über zwei Module zieht. Das ist aber nicht der Regelfall. Gleiches gilt für Port-Contract-Tests. Diese müssen wir nur anpassen, wenn ein neuer Port entsteht.

Dieser Test-Rezept-Ansatz lässt sich auch sehr gut mit dem Test-Report-Framework *Allure [20]* kombinieren. Es gibt uns die Möglichkeit, sehr klare Links zwischen automatisiertem Test und unserem Ticket-Tracking-Tool herzustellen *(siehe Listing 1)*.

Mit Hilfe der Annotationen kann Allure sehr gut durchsuchbare Reports erzeugen, in denen wir schnell sehen können, auf welchen Ebene wir was getestet haben und wie diese Tests sich ergänzen.

Fazit

In dieser Artikelserie haben wir uns sehr umfangreich mit dem Thema Strukturzementierung durch Tests auseinandergesetzt. Während wir in den ersten zwei Teilen technische Konzepte beleuchtet haben, insbesondere die TestDsl, haben wir uns diesmal mit eher mentalen Konzepten beschäftigt: einem Vorgehen das Testen und Strukturieren ganzheitlich betrachtet. Wir haben dabei eine Struktur entwickelt, die gleichzeitig starr und wandelbar ist. Starr dort, wo wir testen wollen. Wandelbar, wo wir Änderungen der Fachlichkeit zulassen wollen. Alles mit dem Ziel, Strukturzementierung zu verhindern. Das ist allerdings nur eine Eigenschaft von guten Tests. Mit der *Test Desiderata [21]* können wir den Ansatz umfangreich bewerten.

Die *Test Desiderata* von Kent Beck ist eine Liste aller wünschenswerter Eigenschaften für Tests. Man kann sich die Eigenschaften wie Schieberegler vorstellen – mal mehr von diesem, mal weniger von jenem. Wir nutzen diese Liste, da sie die Eigenschaften anderer Ansätze wie „4 Säulen guter Unit Tests" *[2]*, „Unit Tests are FIRST" *[22]* oder der Erweiterung „FIRSTRCC" *[11]* bereits inkludiert und noch ein paar Dimensionen hinzufügt.

Wenn wir die Desiderata auf die TestDsl anwenden, dann kommt folgende (positive) Bewertung heraus *(siehe Tabelle 1)*.

Inspiring und predictive habe ich bewusst leer gelassen. Die Bewertung davon würde ich gerne in eure Hände geben, liebe Leser. Weckt das geschilderte Vorgehen zusammen mit der TestDsl bei euch Vertrauen? Würdet ihr das System in Produktion geben, wenn alle Tests auf allen Ebenen grün sind?

Quellen

[1] M. Fowler, „Domain Specific Language". 2008. Verfügbar unter: *https://martinfowler.com/bliki/DomainSpecificLanguage.html*

[2] V. Khorikov, „Unit Testing Principles, Practices, and Patterns". 2020. Verfügbar unter: *https://www.manning.com/books/unit-testing*

[3] D. L. Parnas, „On the criteria to be used in decomposing systems into modules". 1972. Verfügbar unter: *https://dl.acm.org/doi/10.1145/361598.361623*

[4] M. Feathers, „Unit tests are tests of modularity". 2019. Verfügbar unter: *https://blogs.itemis.com/en/unit-tests-are-tests-of-modularity*

[5] G. James, „The Tao Of Programming". 1987. Verfügbar unter: *http://www.canonical.org/~kragen/tao-of-programming.html*

[6] S. Roberts und M. Zaikin, „Quiz yourself: Apply encapsulation principles to a class". 2020. Verfügbar unter: *https://blogs.oracle.com/javamagazine/post/quiz-yourself-apply-encapsulation-principles-to-a-class*

[7] P. Gafert, „ArchUnit". 2024. Verfügbar unter: *https://www.archunit.org/getting-started*

[8] Project Jigsaw, „Java Platform Module System (JSR 376)". 2017. Verfügbar unter: *https://openjdk.org/projects/jigsaw/spec/*

[9] S. Brown, „Modular Monoliths". 2015. Verfügbar unter: *http://static.codingthearchitecture.com/presentations/softwarecircus2015-modular-monoliths.pdf*

[10] J. Shore, „Testing Without Mocks: A Pattern Language". 2023. Verfügbar unter: *https://www.jamesshore.com/v2/projects/nullables/testing-without-mocks*

[11] U. Enzler, „Die agile Testpyramide vNext - Agile Bodensee 2017". 2017. Verfügbar unter: *https://www.youtube.com/watch?v=-35GpOJnjmM*

[12] „Design for testing". Verfügbar unter: *https://en.wikipedia.org/wiki/Design_for_testing*

[13] M. Fowler, „Presentation Domain Data Layering". 2015. Verfügbar unter: *https://martinfowler.com/bliki/PresentationDomainDataLayering.html*

[14] A. Cockburn, „Hexagonal architecture". 2005. Verfügbar unter: *https://alistair.cockburn.us/hexagonal-architecture/*

[15] D. O. Hebb, „Hebbsche Lernregel". 1949. Verfügbar unter: *https://de.wikipedia.org/wiki/Hebbsche_Lernregel*

[16] J. B. Rainsberger, „Beware the Integrated Tests Scam (was Integrated Tests Are a Scam)". 2009. Verfügbar unter: *https://blog.thecodewhisperer.com/permalink/integrated-tests-are-a-scam*

[17] E. Normand, „Grokking Simplicity: Taming complex software with functional thinking". 2021. Verfügbar unter: *https://grokkingsimplicity.com/*

[18] K. Beck, „Stackoverflow Antwort auf: How deep are your unit tests?" 2008. Verfügbar unter: *https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/153565#153565*

[19] R. Osherove, „Art of Unit Testing, 3rd Edition". 2024. Verfügbar unter: *https://www.artofunittesting.com/*

[20] Qameta Software, „Allure JUnit". 2024. Verfügbar unter: *https://allurereport.org/docs/junit5/*

[21] K. Beck, „Test Desiderata". 2019. Verfügbar unter: *https://medium.com/@kentbeck_7670/test-desiderata-94150638a4b3*

[22] T. Ottinger und J. Langr, „Unit Tests Are FIRST: Fast, Isolated, Repeatable, Self-Verifying, and Timely". 2021. Verfügbar unter: *https://medium.com/pragmatic-programmers/unit-tests-are-first-fast-isolated-repeatable-self-verifying-and-timely-a83e8070698e*

[23] M. Fowler, „Test Driven Development". 2023. Verfügbar unter: *https://martinfowler.com/bliki/TestDrivenDevelopment.html*

[24] RabbitMQ. Verfügbar unter: *https://www.rabbitmq.com/*

Richard Gross

richard.gross@maibornwolff.de

Richard Gross ist IT-Archäologe und arbeitet seit 2013 für den Bereich IT-Sanierung. Seine Schwerpunkte sind hexagonale Architekturen, Hypermedia-driven APIs sowie die ausdrucksstarke und eindeutige Modellierung der Domäne als Code. Daher ist er von vielen eher jüngeren Programmiersprachen wie Kotlin oder F# begeistert, die ihn bei der Modellierung unterstützten. Außerdem hat er auch lange Zeit das F&E Projekt CodeCharta begleitet, mit dem auch Nicht-Entwickler ein Verständnis für die Qualität ihrer Software bekommen können. Privat saniert Richard auch; sein Eigenheim.

Mitglieder des iJUG

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

iJUG Verbund
www.ijug.eu

Impressum

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

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

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

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

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

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

Bildnachweis:
Titel: Bild © Sentavio
https://stock.adobe.com
S. 10 + 11: Sloth McSloth
https://stock.adobe.com
S. 14 + 15: Bild © macrovector
https://freepik.com
S. 20 + 21: Bild © Humpty22
https://pixabay.com
S. 26 + 27: Bild © Green Creator
https://stock.adobe.com
S. 34 + 35: Bild © Thanaphon
https://stock.adobe.com
S. 42 + 43: Bild © Designed by freepik
https://freepik.com
S. 48 + 49: Bild © vectorjuice
https://freepik.com
S. 56 + 57: Bild © Designed by freepik
https://freepik.com

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

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

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

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

Inserentenverzeichnis

DOAG e. V. U 2, S. 8, S. 47
iJUG e.V. S. 9, S. 13, S. 33, S. 41
JavaLand GmbH U 4

EARLY BIRD BIS 04.02.2025

JavaLand

AM NÜRBURGRING

JAVALAND

1. - 3. APRIL 2025

JAVALAND.EU

#JAVALAND

Präsentiert von: iJUG Verbund, Heise Medien, DOAG

Veranstalter: JavaLand