Ausgabe 04/2022

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

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

4 191978 304903 04

Java aktuell

iJUG Verbund
www.ijug.eu

Java 18
Einblick in das aktuelle Release

Microservices
mit Kotlin, ohne Framework und mehr

JavaLand
Präsenz-Comeback und neues Zutrittssystem

Microservices

JavaLand!

on demand

Javaland 2022 verpasst?

Jetzt On-Demand-Ticket buchen und Vortragsaufzeichnungen anschauen!

Alle Angebote im On-Demand-Ticket-Shop

Community-Partner: iJUG Verbund

Präsentiert von: DOAG Heise Medien

Liebe Leserinnen und Leser der Java aktuell,

im März ist das aktuelle Release 18 des OpenJDK erschienen. Ab Seite 10 fasst Falk Sippach die Neuerungen für euch zusammen. Auf den neuesten Stand rund um Java, die Community und die Eclipse Foundation bringen euch wie gewohnt das Java-Tagebuch und die Eclipse Corner zu Beginn dieser Ausgabe.

Freut euch in dieser Ausgabe auf informative Fachartikel rund um das Thema Microservices – das aktuell vorherrschende Architekturmuster, mit der viele Entwicklerinnen und Entwickler konfrontiert werden. Den Anfang dazu macht Elmar Brauch, der uns die Entwicklung eines Kotlin-Microservice mit REST-API in Spring Boot vorstellt. In seinem Artikel ab Seite 23 nimmt Markus Karg uns mit auf eine kurze Exkursion in die Microservices-Entwicklung ohne Framework. Er erklärt, für welche Anwendungszwecke Java SE, APIs und bewusst ausgesuchte Implementierungen der bessere Ansatz sind. Mit Thomas Michael geht es wieder zurück zum Framework-basierten Ansatz mit Spring Boot und Docker Images. Er beschreibt in seinem Beitrag, wie ein Monolith von einer Microservices-Architektur abgelöst wird und welche Motivation diese Entscheidung beeinflusst.

Ihr habt die JavaLand 2022 verpasst? Auf Seite 36 haben wir alle Highlights vom Präsenz-Comeback der großen Java-Community-Konferenz im Phantasialand Brühl für euch zusammengefasst. Außerdem hieß es diesmal: Adé Plastikbadges! Die Zutrittsausweise der Teilnehmenden wurden nicht im Vorfeld ausgedruckt und versendet, sondern digital als Download zur Verfügung gestellt. Gemeinsam mit Ronan Le Tiec, der dieses Projekt betreut hat, werfen wir einen Blick hinter die Kulissen der initialen Anforderungen und wie diese umgesetzt wurden.

Dann wird es fantastisch: Uwe Sauerbrei und seine gute Fee Bitsi stellen ab Seite 44 die Open-Source-Wissensdatenbank Obsidian vor. Zum Abschluss dieser Ausgabe folgt der dritte und letzte Teil von Matthias Eschholds Artikelreihe zu Clean Architecture, die sich den Architeturabkürzungen, sogenannten Shortcuts, widmet.

Wir wünschen euch viel Spaß beim Lesen!

Eure

Lisa Damerow
Redaktionsleitung Java aktuell

Inhalt

OpenJDK 18: Das aktuelle Release im Detail

Kotlin-Microservices mit Spring Boot entwickeln

Das große Wiedersehen der Java-Community in Person

Die Open-Source-Wissensdatenbank Obsidian im Überblick

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java.

16. März 2022

Sieben Gründe für Java

Warum macht Java nach 26 Jahren immer noch Sinn? Diese Frage hat Bazlur Rahman unter anderem in der Mailing-Liste der JUG Leaders und auf Twitter gestellt, und die Antworten im Blog zusammengefasst *[1]*. Keine Ahnung, wie groß die Anzahl der Rückmeldungen war und ob sie als einigermaßen repräsentativ angesehen werden kann, ich kann der Zusammenfassung aber in weiten Teilen zustimmen. Stabilität – bei gleichzeitiger Innovationsstärke, die aktive und offenherzige Community, „mehr gelöste Probleme als irgendein anderes Ökosystem" sind einige der Punkte. Und Geertjan Wielenga schreibt: „Hier habe ich meine Freunde."

17. März 2022

Javaland 2022

Endlich: Nach der langen Corona-Dürre fand die JavaLand wieder als Präsenzveranstaltung statt. Insgesamt waren es noch spürbar weniger Teilnehmerinnen und Teilnehmer als bei der letzten Konferenz vor Ort, aber die hatten sichtlich wieder genauso viel Spaß wie „damals". Und nach der erfolgreichen Online-Ausgabe 2021 gab es auch diesmal wieder ein Online-Angebot mit Vorträgen im Live-Stream und Moderation aus dem JavaLand-Studio.

Ein Unterschied noch: Die Konferenz war etwas weniger international, auch noch eine Auswirkung der Pandemie. Viele potenzielle Referenten, zum Beispiel aus den USA, durften gar nicht reisen. Den meisten ist das allerdings vermutlich gar nicht negativ aufgefallen. In jedem Fall war es das erhoffte gelungene Revival nach längerer Zwangspause und wir können uns jetzt schon auf die nächste Ausgabe vom 21. bis 23. März 2023 freuen – ich bin sicher: wieder vor Ort im Phantasialand.

22. März 2022

Java 18 und 19

Der Java Release Train ist pünktlich an Station 18 angekommen. Details hatte ich ja in früheren Ausgaben beschrieben. Aber jetzt ist die Zeit, einen Ausblick auf Release 19 zu werfen, das ja auch nicht mehr weit weg ist. Geplant sind bislang: Die inzwischen vierte Inkubator-Version des Vector-API mit einigen Änderungen, es braucht offensichtlich ein bisschen zur Produktionsreife. Daneben eine Preview der leichtgewichtigen Virtual Threads aus „Project Loom" (ursprünglich auch Fibers genannt). Wer erinnert sich noch? Ganz früher (JDK 1.1) gab es ja schon mal die ebenfalls rein in der JVM simulierten „Green Threads", die mit 1.2 dann durch „richtige" Betriebssystem-Threads ersetzt wurden. In Version 1.1 konnte die JVM nur einen Betriebssystem-Thread nutzen, was bei den heutigen massiv parallelen Prozessoren natürlich ein absolutes Desaster wäre. In Zukunft werden sich dann beide Ebenen mit ihrer jeweils eigenen Parallelität ergänzen. Außerdem soll es einen Port des OpenJDK für Linux auf RISC-V Architekturen geben – der Port an sich ist anscheinend schon mehr oder weniger fertig, es geht nur noch um die Integration.

Vorgeschlagen für JDK 19 sind außerdem eine Preview des Foreign Function & Memory API („Project Panama") – ein API zum effizienten und sicheren Zugriff auf Code und Daten außerhalb der Java Runtime – und die, auch schon dritte, Preview von „Pattern Matching for switch". Viele Previews also, aber die lang erwarteten Beiträge aus den Projekten Loom und Panama dürften ein großer Gewinn werden und können so in Ruhe vor dem nächsten Long Term Support Release (21) heranreifen.

22. März 2022

Die JavaOne kehrt zurück

Auch die JavaOne soll diesen Herbst als Präsenzveranstaltung zurückkehren. Nicht wieder wie früher mal als eigenständige Konferenz, sondern als Teil der Oracle CloudWorld in Las Vegas, vom 16. bis 20. Oktober.

23. März 2022

Clojure 1.11

Nach drei Jahren gibt es ein neues Release der funktionalen JVM-Sprache Clojure. Die Neuerungen sind, dem Sprung von 1.10 auf 1.11 entsprechend, nicht allzu umfangreich *[2]*. Clojure 1.11 unterstützt weiterhin das JRE in der Version 8 als Laufzeitumgebung sowie die Versionen 11 und 17.

30. März 2022

Entscheidung zu MicroProfile Metrics

Das Metrics-Projekt von Eclipse MicroProfile hat jetzt eine Entscheidung darüber gefällt, welche der vor einigen Monaten diskutierten Optionen in Bezug auf die geforderte(n) Implementierung(en) umgesetzt wird: Das neue Metrics-API soll Implementierungs-agnostisch werden, das heißt, jeder Hersteller kann wählen, ob er unter der Haube Micrometer, OpenTelemetry, DropWizard Metrics oder etwas ganz anderes nutzt.

31. März 2022

Spring-Sicherheitslücken

Nach log4Shell kommt Spring4Shell. Die gerade bekannt gewordene „Remote Code Execution"-Sicherheitslücke betrifft Spring-

MVC- und Spring-WebFlux-Applikationen, die auf Java 9 oder höher laufen und als Web-Archiv in Tomcat deployt sind – also nicht als ausführbares jar-File mit eingebettetem Container laufen. Das ist zumindest der aktuelle Stand: Im spring.io-Blog wird extra darauf verwiesen, dass die Schwachstelle allgemeinerer Natur ist und möglicherweise weitere „Exploits" existieren.

Mehr oder weniger gleichzeitig ist eine weitere Schwachstelle in Spring bekannt geworden, die sich für DoS-Attacken eignet (CVE-2022-22950).

1. April 2022

Project „Greenfields"

Ich gebe zu, ich hatte kurz ein großes Fragezeichen auf der Stirn, als ich diese Mail im OpenJDK-Verteiler zu einem neuen Projekt „Greenfields" gelesen habe *[3]*. Ziel sei es, die Auswirkungen von „Project Loom" auf das Java-Ökosystem und die reale Umwelt zu messen (analog zur Textilindustrie, aus deren Vokabular sich das OpenJDK-Projekt ja reichlich bedient). Zu meiner Verteidigung sei gesagt, dass ich die Mail erst mit deutlicher Verzögerung gelesen habe und das ursprüngliche Datum nicht im Blick hatte. Die Kreativität der OpenJDK-Developer zeigte sich auch in den Antworten. Unter anderem wurde ein – nun ja: ressourcenschonendes – neues Sprach-Konstrukt „try without resources" vorgeschlagen.

7. April 2022

Jakarta EE Starter

Der Jakarta EE Starter ist da! Genau wie Spring und seit einiger Zeit auch MicroProfile hat nun auch Jakarta ein Tool, um das schnelle Aufsetzen eines Projekts mit den wesentlichen Bestandteilen und Sample-Code zu ermöglichen. Aktuell ist es jedoch erst mal nur ein Maven-Archetype, der unter *[4]* beschrieben wird. Eine interaktive Website zur Projektgenerierung analog zu start.spring.io u.a. ist noch in Arbeit.

16. April 2022

Spring4Shell und andere

Update zu Spring4Shell: Auch Payara und GlassFish sind betroffen. Einen aktuellen Stand der Analysen gibt es hier *[5]*. Ach ja, im JDK (15+) ist auch vor ein paar Tagen ein Fehler entdeckt worden, der für Angreifer das Fälschen digitaler Signaturen zum Kinderspiel macht. Gut beschrieben hier bei heise *[6]*.

Es wird nicht weniger werden. Besonders bedenklich aber: Zum schon Ende 2021 bekannt gewordenen log4shell kommen immer noch fast täglich neue Nachrichten. Etwa zu Patches, die neue Sicherheitslücken aufreißen (wie bei AWS, wo mit dem Fix der kritischen Lücke gleich vier neue, immer noch als hoch eingestufte Schwachstellen entstanden sind *[7]*. Und es befinden sich immer noch unzählige betroffene Applikationen „da draußen". Laut einer Analyse von Rezilion sind erst 40 % von fast 18.000 Open-Source-Paketen, die Log4j nutzen, gepatcht. Inzwischen ist log4shell nach einem Bericht von Sicherheitsexperten bei Trend Micro fester Bestandteil des berüchtigten Mirai BotNet geworden, das weltweit Router, Fernseher etc. infiziert, um sie für DDoS-Attacken einzusetzen.

19. April 2022

Eclipse Java Migration Toolkit

Unter Eclipse Adoptium, dem Top-Level-Projekt, das sich im Kern mit Releases auf Basis des OpenJDK und ihrer Qualitätssicherung beschäftigt, ist ein weiteres Projekt geplant: das „Eclipse Migration Toolkit for Java". Ziel ist es, die Migration zu neueren Java-Releases zu erleichtern, insbesondere zwischen Long-Term-Support-Releases (8 auf 11, 8 auf 17,11 auf 17 und demnächst dann auch auf 21 usw.). Das entstehende Toolkit soll sowohl statische als auch dynamische Analyse zur Laufzeit nutzen und die Resultate in verschiedenen menschen- und maschinen-lesbaren Formaten zur Verfügung stellen.

26. April 2022

GraalVM 22.1

Die neueste Version von Oracles GraalVM führt einen „Quick Build Mode" für native Images ein, der während der Entwicklung die Build-Zeiten durch weniger gründliche Optimierungen deutlich reduziert. Der Modus wird einfach durch den Switch „Ob" („native-image -Ob …") aktiviert und bringt – getestet mit dem Build des Graal-Compilers selbst – eine Verringerung der reinen Compile-Zeit um 81 % und der gesamten Build-Zeit um immer noch 43 %. Außerdem ist die GraalVM nun auch für Apple-Silicon-Prozessoren verfügbar, allerdings noch als Preview-Version. Darüber hinaus gibt es diverse Änderungen für verschiedene unterstützte Programmiersprachen und einzelne Werkzeuge.

28. April 2022

Adoptium Marketplace

Eclipse Adoptium möchte einen gemeinsamen Marktplatz schaffen, auf dem Hersteller ihre Angebote rund um das JDK veröffentlichen können. Die technischen und prozessualen Voraussetzungen sind wohl schon geschaffen, momentan wird noch mit den verschiedenen Herstellern gesprochen, um alle Informationen rechtzeitig zu einer geplanten Veröffentlichung am 25. Mai auf die Adoptium-Plattform zu bekommen.

30. April 2022

Java "State of the Ecosystem"

Seit 2020 gibt New Relic jährlich den „State of the Java Ecosystem Report" heraus (beruhend auf den mit ihrer „Observability Platform" integrierten Applikationen). Der 2022er Ausgabe zufolge liegt Java 11 bei der produktiven Nutzung inzwischen vorn (48 %), jedoch weiterhin

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

nur knapp vor Java 8 (46 %). Der Anteil von Java 17 ist noch unter einem halben Prozent, aber so lange gibt es das neueste LTS Release ja auch noch nicht. Bei den Distributionen ist der Anteil des Oracle JDK deutlich auf 34 % gesunken, während Amazon Corretto mit 22 % auf Platz 2 liegt, vor Eclipse Adoptium (11 %). Auf den weiteren Rängen liegen Azul (8 %), Red Hat (6 %), IcedTea (5 %), Ubuntu und BellSoft (je knapp 3 %).

3. Mai 2022

MicroProfile 5.1 oder 6.0?

Momentan läuft im MicroProfile-Projekt noch eine Diskussion, ob das für den Juni oder gegebenenfalls Juli geplante Release ein 6.0 oder doch „nur" ein 5.1 wird. Ein Major Release würde bedeuten, dass OpenTracing direkt aus dem „Umbrella" entfernt werden könnte, zugunsten von Telemetry, das dann zumindest die Tracing-Funktionalität beinhaltet. Und auch Metrics könnte dann in der neuen Version 5.0 enthalten sein. Bei MicroProfile 5.1 würde OpenTracing lediglich als „zu entfernen" angekündigt (beziehungsweise abgekündigt), während OpenTelemetry zumindest „schlafend" enthalten und per Switch aktivierbar sein könnte.

Die Diskussion läuft wohl darauf hinaus, dass es erst mal bei 5.1 bleibt, da ansonsten auch in deutlich kürzerer Zeit als anvisiert (einmal pro Jahr) die Releases 5, 6 und 7 aufeinander folgen würden. Das 6.0 Release könnte dann einige Monate später folgen.

Das andere größere Thema ist der Umgang mit den Java-Versionen. Soll bereits jetzt Java 11 als Minimum vorausgesetzt werden? Hier läuft es auf eine Erhöhung der Mindest-Java-Version nur in Abstimmung mit Jakarta hinaus, also konkret mit Jakarta EE 10. Das wäre dann erst mit MicroProfile 6.0 der Fall.

Jakarta EE 10

Das Alignment von MicroProfile 6 mit Jakarta würde zeitlich ganz gut passen, denn das EE 10 Release macht Fortschritte und scheint seinen Zeitplan Q2 2022 so gerade zu halten. Die meisten Einzelspezifikationen sind fertig oder gerade im Abstimmungsprozess (Abstimmung im Sinne einer finalen Entscheidung, oder englisch „ballot"). Zertifiziert wird die Referenzimplementierung GlassFish dann mit Java 11 und 17. Jeder Hersteller kann dann entscheiden, ob seine Implementierung 11, 17 oder beide unterstützt.

Ganz nebenbei steigt die Anzahl der Jakarta-Implementierungen an *[8]*. Und mindestens ein weiterer (AISWare Flying Server) ist in der Pipeline.

Ein Punkt, der wohl etwas Sorge bereitet – wenn auch nicht akut – ist die Abkündigung des SecurityManager im OpenJDK. Dessen Implementierung soll zunächst durch Code ersetzt werden, der einfach alle Aufrufe durchreicht, ohne tatsächlich Prüfungen durchzuführen. In der entsprechenden Projektbeschreibung (JEP 411 des OpenJDK) steht allerdings, dass das API später entfernt wird, sobald das Risiko von Inkompatibilitäten mit existierenden Applikationen oder Bibliotheken auf ein „akzeptables Maß" sinkt. Genau diese Definition von „akzeptabel" soll nun genauer zwischen den Projekten abgestimmt werden. Darüber hinaus sei „aus den kürzlichen Diskussionen klar geworden, dass eine Alternative zum SecurityManager von der Jakarta-EE-Platform-Development-Group angegangen werden müsste. Dies sei aber ein nicht-triviales Vorhaben und würde vielleicht niemals erfolgen *[9]*.

4. Mai 2022

Jakarta EE auf Reddit

Jakarta EE hat jetzt auch eine eigene Community auf Reddit *[10]*.

Referenzen:

[1] *https://foojay.io/today/7-reasons-why-after-26-years-java-still-makes-sense/*
[2] *https://clojure.org/news/2022/03/22/clojure-1-11-0*
[3] *https://mail.openjdk.java.net/pipermail/discuss/2022-April/006031.html*
[4] *https://start.jakarta.ee*
[5] *https://spring.io/blog/2022/04/13/spring-framework-data-binding-rules-vulnerability-cve-2022-22968*
[6] *https://www.heise.de/news/Bug-in-Java-macht-digitale-Signaturen-wertlos-6847744.html*
[7] *https://aws.amazon.com/de/security/security-bulletins/AWS-2022-006/*
[8] *https://jakarta.ee/compatibility/*
[9] *https://mail.openjdk.java.net/pipermail/security-dev/2022-April/030094.html*
[10] *https://www.reddit.com/r/JakartaEE/*

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

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich im DOAG e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit 2015 ist er stellvertretender Leiter der neugegründeten Java Community innerhalb der DOAG. Beruflich hat er seit dem Jahr 1999 als Entwickler und Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet und ist seit dem Jahr 2016 als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Markus' eclipse-Corner

Vor etwa einem Jahr hat die Jakarta EE Working Group der Eclipse Foundation den Plan vorgelegt, was Jakarta EE 10 beinhalten soll und wann das Ganze der Allgemeinheit zugänglich gemacht wird: im zweiten Quartal 2022. Als Co-Autor der JAX-RS-Spezifikation freut es mich sehr, dass wir unseren Teil dazu zeitgerecht beigetragen haben: JAX-RS 3.1, oder Jakarta REST, wie es neuerdings genannt wird, steht seit einigen Wochen bereit, sodass darauf aufbauende APIs oder Produkte wiederum in ihre finale Phase gehen können.

Allen voran seien hier RESTeasy und Jersey genannt, die beide bereits jetzt die neuen Features umsetzen, wie das Java-SE-Bootstrap-API zur Implementierung von Microservices in reinem Java SE 11, also ohne Frameworks oder Application Server. Siehe hierzu auch den Artikel auf Seite 23.

Ganz besonders freut mich das, da ein großer Teil der neuen Features in JAX-RS 3.1 aus den Händen der Anwender-Community, also von „uns", stammt – sowohl beim API als auch in den Produkten. In Jersey beispielsweise habe ich das meiste davon in meiner Freizeit implementiert und viel dabei gelernt. Danke an dieser Stelle übrigens an alle, die sich dieses Hobby mit mir teilen und sich für Open Source einsetzen, und besonderen Dank auch an den iJUG-Stipendiaten Jeremias Weber, der das JAX-RS-Team bei der Portierung des TCK unterstützt hat! Wenn es bei den anderen Teilen von Jakarta auch nur annähernd so gut läuft, wird Jakarta EE 10 tatsächlich zeitgerecht fertig!

Und natürlich nerve ich euch auch an dieser Stelle wieder mit dem Thema Engagement! „Wir", also die Leute „hinter" Jakarta, MicroProfile und Adoptium, würden uns wirklich sehr freuen, wenn „ihr", also die Anwender von Jakarta, MicroProfile und Adoptium, zukünftig eine noch größere Rolle in diesen Arbeitsgruppen spielen würdet. Das muss nicht zwingend die Portierung eines TCK oder die Implementierung eines neuen Features sein. Es langen ja auch schon die kleinen Sachen wie Tippfehler in Dokus korrigieren, Fehler früh und ordentlich melden oder uns einfach nur sagen, was ihr in Zukunft von uns erwartet. Jan Westerkamp, Heiko Rupp, Hendrik Ebbers und meine Wenigkeit vertreten den iJUG – und somit die Mehrzahl der Leser – in den entsprechenden Working Groups. Wir benötigen aber gelegentlich Feedback von euch, das Ganze in die richtige Richtung zu lenken und zukünftige Veranstaltungen, wie etwa die vom iJUG organisierte JakartaOne German, mit relevantem Content auch interessant zu gestalten. Daher heute mein ganz konkreter Aufruf: Sagt mir schnellstmöglich, was ihr von Jakarta EE 11 im Großen oder von JAX-RS im Kleinen erwartet, denn die Planung für Jakarta EE 11 und die konkrete Arbeit an JAX-RS 4.0 hat bereits begonnen! Egal, welchen Weg ihr wählt, ob ihr mir direkt schreibt oder an die Redaktion, ob über Twitter oder als Kommentar unter einem meiner YouTube-Videos: Hauptsache, eure Wünsche kommen bei mir an. Ich leite diese dann gerne in eurem Namen weiter!

Markus Karg
markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

OpenJDK 18 – Ist Java jetzt erwachsen?

Falk Sippach, embarc Software Consulting GmbH

Auch wenn die Versionsnummer anderes suggeriert, ist Java mit seiner über 25-jährigen Geschichte natürlich schon längst erwachsen. Das Ökosystem ist lebendig wie nie, auch wenn es bereits einige Male totgesagt wurde. Der 2018 eingeschlagene Weg mit den halbjährlichen Versionen hat sich etabliert. Zudem sind das Java Development Kit (JDK) und die Laufzeitumgebung (JVM) mittlerweile Open Source und es gibt viele verschiedene Anbieter, die sowohl freie als auch kommerzielle Distributionen herausbringen. In die regelmäßigen Releases schaffen es immer nur die Features hinein, die gerade fertig sind. Größere Themenbereiche, wie das Pattern Matching, erhalten schrittweise Einzug. Die Features werden typischerweise über JEPs (JDK Enhancement Proposals) entwickelt und je nach Umfang gibt es mehrere Inkubator- und/oder Preview-Phasen. Im März 2022 ist mit dem OpenJDK 18 ein halbes Jahr nach der letzten LTS-Version 17 das erste Zwischen-Release erschienen. Schauen wir uns an, was alles passiert ist.

Neben den drei „wiederkehrenden" Baustellen (Pattern Matching, Vector-API und Foreign Function & Memory API) birgt die Liste der JDK Enhancement Proposals *[1]* auch einige andere interessante Neuerungen:

- JEP 400: UTF-8 by Default
- JEP 408: Simple Web Server
- JEP 413: Code Snippets in Java API Documentation
- JEP 416: Reimplement Core Reflection with Method Handles
- JEP 417: Vector API (Third Incubator)
- JEP 418: Internet-Address Resolution SPI
- JEP 419: Foreign Function & Memory API (Second Incubator)
- JEP 420: Pattern Matching for switch (Second Preview)
- JEP 421: Deprecate Finalization for Removal

Wie so oft, sind nicht alle Themen für uns Java-Entwickler direkt relevant. Trotzdem werden wir uns viele der JEPs näher anschauen und bei den meisten auch tiefer in die Details gehen.

UTF-8 als Standard für Dateioperationen

Obwohl Java schon sehr früh auf die mächtigen UTF-Standards gesetzt hat, stolpern Java-Entwickler immer wieder über Encoding-Probleme bei Dateioperationen. Werden ohne explizite Nennung eines Zeichensatzes Dateien in einem Java-Programm unter einem Betriebssystem weggeschrieben, kommt aufgrund des implizit verwendeten Encodings beim Lesen auf einem anderen Betriebssystem gegebenenfalls nur Kauderwelsch zurück. Der Hintergrund ist, dass der Java-Standardzeichensatz je nach OS und Spracheinstellungen unterschiedlich ist. Linux und macOS verwenden das UTF-8-Format, während Windows Dateien mit Codepage 1252 (basierend auf ISO 8859-1) liest und schreibt.

Noch verwirrender wird die Geschichte, weil neuere Methoden in der Java-Klassenbibliothek den Standardzeichensatz nicht berücksichtigen und grundsätzlich UTF-8 verwenden, sofern kein Zeichensatz explizit angegeben wurde. Das passiert beispielsweise in der Klasse Files in den Methoden `writeString()` und `readString()`.

Es gab bisher zwei Möglichkeiten, mit dieser Thematik umzugehen:

- beim Aufruf aller Methoden immer explizit einen Zeichensatz angeben, wenn Zeichenfolgen von/nach Bytes umgewandelt werden
- per System Property „file.encoding" den Standardzeichensatz explizit setzen

Variante 1 führt zu viel Code-Duplikation und verhindert bei Verwendung des Stream-API den Einsatz von Methodenreferenzen. *Listing 1* zeigt dazu zwei Beispiele.

Mit dem JEP 400 (UTF-8 by Default) wird vieles besser. Einerseits wird das Standard-Encoding auf allen Betriebssystemen unabhängig von Gebiets- und Spracheinstellungen immer UTF-8 sein. Außerdem wird das bisher nicht ausreichend beschriebene System Property „file.encoding" nun besser dokumentiert und kann damit als offizielle, valide Lösungsvariante betrachtet werden. Vorsichtig sollte man hier trotzdem sein, weil einige neuere API-Methoden das eingestellte Standard-Encoding ignorieren und das Setzen des System-Property dann nicht die gewünschte Wirkung zeigt.

Laut API-Dokumentation *[2]* sind nur zwei mögliche Werte erlaubt, alles andere führt zu einem nicht spezifizierten Verhalten:

- UTF-8 (für die einheitliche Codierung)
- COMPAT (Kompatibilitätsmodus zur Simulation des Verhaltens vor Java 18)

Vermutlich wird „file.encoding" in zukünftigen Versionen deprecated und später komplett entfernt werden, um mögliche Fehlerquellen zu vermeiden. Das aktuell verwendete Encoding kann zur Laufzeit auf zwei Arten ausgelesen werden: durch den Aufruf von `Charset.defaultCharset()` oder direkt über das System Property „file.encoding". Zusätzlich gibt es noch das Property „native.encoding", das das vor Java 18 verwendete Standard-Encoding ausgibt (UTF-8 bei Linux beziehungsweise macOS und Cp1252 bei Windows).

```
try(FileWriter fileWriter = new FileWriter("output.txt", StandardCharsets.UTF_8) ){
  fileWriter.write("hallo äöüß");
}
char[] result = new char[10];
new FileReader("output.txt", StandardCharsets.UTF_8).read(result);

Stream<String> stream = Stream.empty();
// keine Möglichkeit für Angabe des Encodings
System.out.println(stream.map(URLDecoder::decode));
// Wrappen in einen Lambda-Ausdruck, um Endoding anzugeben
System.out.println(stream.map(s -> URLDecoder.decode(s, StandardCharsets.UTF_8)));
```

Listing 1

```
$ jwebserver
Binding to loopback by default. For all interfaces use "-b 0.0.0.0" or "-b ::".
Serving /home/sven and subdirectories on 127.0.0.1 port 8000
URL http://127.0.0.1:8000/
```

Listing 2

Zwar nicht Teil des JEP 400, aber zur Thematik passend, wurde im JDK 18 eine neue Methode im JDK eingeführt: `Charset.forName(String charsetName, Charset fallback)`. Diese gibt bei unbekannten/undefinierten Zeichensatz-Namen den angegebenen Fallback-Wert zurück, statt eine Exception zu werfen. Über den Java-Almanac *[3]* kann man sich im Übrigen sehr leicht alle Änderungen an der Java-Klassenbibliothek zwischen einzelnen Releases anzeigen lassen.

Webserver to go im JDK

Von anderen modernen Programmiersprachen (Python, Ruby, JavaScript, …) ist man es gewohnt, dass sie von Hause aus oder über Erweiterungen die Möglichkeit mitbringen, einen trivialen HTTP-Web-Server über die Konsole hochzufahren. Das ist praktisch, um einfache Webseiten und vor allem auch darin verlinkte CSS- und JavaScript-Dateien über einen Browser ausliefern zu können. Dadurch lassen sich zu Testzwecken die Sicherheitsmechanismen der Browser bei lokalen File-Zugriffen (file://) umgehen. Mit dem Kommando `jwebserver` ist nun auch ein Java-Webserver „im Lieferumfang" des JDK verfügbar und lässt sich ganz einfach über die Kommandozeile starten *(siehe Listing 2)*.

Dieser Webserver ist allerdings sehr rudimentär. Er unterstützt als Protokoll nur HTTP/1.1 und kein SSL (HTTPS). Bei den HTTP-Methoden sind nur GET und HEAD erlaubt. Einige Optionen sind allerdings anpassbar, wie *Listing 3* zeigt.

Das Kommando `jwebserver` ist eigentlich nur eine Art Wrapper um den Befehl:

```
java -m jdk.httpserver
```

Dadurch wird im Modul `jdk.httpserver` eine `main`-Methode aufgerufen, die die Kommandozeilenparameter ausliest und dann den HTTP-Server programmatisch startet. Das kann man auch in beliebigen Java-Programmen machen, wie in *Listing 4* dargestellt.

```
$ jwebserver -b 127.0.0.2 -p 8888 -d /out -o verbose
Serving /out and subdirectories on 127.0.0.2 port 8888
URL http://127.0.0.2:8888/}
```

Listing 3

Über dieses Java-API lässt sich der Server auch um eigene Handler für bestimmte Pfade und HTTP-Methoden (etwa POST statt GET) erweitern.

Code-Snippets in JavaDoc

Die Integration mehrzeiliger Code-Schnipsel in JavaDoc war bisher umständlich und hielt einige Fallstricke parat. Im JEP 413 wurde die Syntax um das `@snippet`-Tag erweitert. *Listing 5* zeigt ein Beispiel dazu. Mit dem `@highlight`-Tag können einzelne Worte/Passagen hervorgehoben werden.

Es können sogar Snippets aus anderen Text- oder Quellcodedateien eingebettet werden. *Listing 6* und *7* zeigen die Markierung des Codes in Form von Regionen und die Integration in die Dokumentation.

Adieu, finalize

Es hat sich nun seit einigen Jahren etabliert, dass aus dem JDK auch wieder Features entfernt oder zunächst für die zukünftige Löschung markiert werden. Das hilft dabei, unnötigen Ballast abzuwerfen.

```
HttpServer server =
  SimpleFileServer.createFileServer(
     new InetSocketAddress(8888), Path.of("out"), OutputLevel.INFO);
server.start();
```

Listing 4

```
/**
 * The following code shows how to use {@code Optional.isPresent}:
 * {@snippet :
 * if (v.isPresent()) {
 *       System.out.println("v: " + v.get()); // @highlight substring="println"
 * }
 * }
 */
```

Listing 5

```
/**
 * The following code shows how to use {@code Option-
al.isPresent}:
 * {@snippet file="ShowOptional.java"
region="example"}
 */
```

Listing 6

```
public class ShowOptional {
  void show(Optional<String> v) {
     // @start region="example"
     if (v.isPresent()) {
        System.out.println("v: " + v.get());
     }
     // @end
  }
}
```

Listing 7

Diesmal hat es mit „Deprecation for Removal" den Finalization-Mechanismus erwischt, immerhin ein Relikt aus Java 1.0.

Klassen können ähnlich einem Destruktor die `finalize()`-Methode implementieren. Der Aufruf erfolgt bei nicht mehr benutzten Objekten durch den Garbage Collector, direkt bevor der Speicher freigegeben wird. Damit lassen sich beispielsweise Ressourcen aufräumen. Leider gibt es viele Probleme bei dieser Vorgehensweise, sodass die Verwendung von `finalize()` schon lange nicht mehr empfohlen wird:

- potenziell wird `finalize()` erst sehr spät oder möglicherweise nie aufgerufen (keine Garantie über den Aufrufzeitpunkt)
- beim Wegräumen von vielen Objekten mit der `finalize()`-Methode kann es beim Full-GC zu spürbaren Latenzen kommen
- Aufruf von `finalize()` für jede Instanz einer Klasse, man kann es nicht auf bestimmte Objekte begrenzen
- diverse Sicherheitsrisiken, etwa durch Zugriff auf die wegzuräumende Referenz in `finalize()` oder beim Aufruf von `finalize()` bei unvollständig initialisierten Objekten (Exception im Konstruktor-Aufruf)
- im Gegensatz zum Konstruktor wird kein impliziter Aufruf von `finalize()`-Methoden in Super-Klassen erzwungen (wird zur Vermeidung von Memory Leaks dringend empfohlen, muss aber explizit gemacht werden)
- Thread-Sicherheit muss für das aufzuräumende Objekt sichergestellt werden, da `finalize()` von einem nicht spezifizierten Thread aufgerufen wird

Diese vielen verschiedenen Probleme und Stolperfallen, die im sehr empfehlenswerten Buch „Effective Java" detaillierter beschrieben sind, werden in näherer Zukunft zur Entfernung von `finalize` führen. Bereits im JDK 9 wurden viele JDK-Methoden (etwa in Object) als deprecated markiert. Um Anwendungen für den zukünftigen Ernstfall zu testen, kann mit der VM-Option `--finalization=disabled` der Mechanismus bereits jetzt probeweise deaktiviert werden.

Eine Alternative ist das in Java 7 eingeführte „try-with-resources"-Konstrukt, das für Implementierungen des Interface AutoClosable automatisch einen `finally`-Block generiert. Dort wird automatisch die `close()`-Methode aufgerufen. In diesem Callback werden Aufräumarbeiten durchgeführt, die garantiert und vor allem direkt ausgeführt werden. Das Objekt wird anschließend vom Garbage Collector entfernt.

In Java 9 wurde zudem das Cleaner-API eingeführt. Dabei können sogenannte „Cleaner Actions" registriert werden. Sie werden automatisch aufgerufen, wenn ein Objekt nicht mehr erreichbar ist (und nicht erst, wenn es aufgeräumt wird).

Pattern Matching in Java

Eine der größten Baustellen der letzten und auch zukünftigen Jahre ist das Pattern Matching, das im Rahmen von Project Amber *[4]* entwickelt und schrittweise ins OpenJDK eingeführt wird. Dazu waren zunächst diverse Vorarbeiten notwendig. Los ging es mit den Switch Expressions bereits im JDK 12. Ab Version 14 folgten TypePatterns („Pattern Matching for instanceof") und Records. Sealed Classes wurden im JDK 15 erstmals eingeführt und mit 17 kam „Pattern Matching for switch" als erste Preview hinzu. Der finale Stand dazu wird aber frühestens im JDK 19 erwartet. Das Pattern Matching insgesamt wird sowieso nicht so schnell abgeschlossen sein. Denn mit „Record Patterns" (JEP 405) steht bereits ein weiterer Teil in den Startlöchern, der voraussichtlich im OpenJDK 19 erstmals als Preview mit an Bord sein wird. Ursprünglich sollten im JEP 405 auch die Array Patterns enthalten sein. Aufgrund zu vieler ungeklärter Fragen wurde dieser Pattern-Typ aber zunächst zurückgestellt. Zudem sind weitere Arten von Patterns denkbar, etwa der Abgleich von Keys in Hash-Tabellen (java.util.Map) oder das Enthaltensein von Werten in Listen beziehungsweise Wertebereichen (Ranges).

Beim Pattern Matching geht es darum, bestehende Strukturen mit Mustern abzugleichen. Ein Pattern ist dabei eine Kombination aus einem Prädikat (das auf die Zielstruktur passt) und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen und damit extrahiert. Die Intention des Pattern Matching ist die Destrukturierung (Dekonstruktion) von Objekten, also das Aufspalten in die Bestandteile und das Zuweisen zu einzelnen Variablen für die weitere Bearbeitung.

Zum Beispiel spart die Spezialform des Pattern Matching beim `instanceof`-Operator unnötige Casts auf die zu prüfenden Ziel-Datentypen. Wenn `obj` ein String ist, dann kann direkt mit der neuen

```
if (obj instanceof String s) {
  // Let pattern matching do the work!
  ...
}
```

Listing 8

```
record Point(int i, int j) {}
enum Color { RED, GREEN, BLUE; }

static void typeTester(Object o) {
  switch (o) {
     case null -> System.out.println("null");
     case String s -> System.out.println("String");
     case Color c -> System.out.println("Color with " + c.values().length + " values");
     case Point p -> System.out.println("Record class: " + p.toString());
     case int[] ia -> System.out.println("Array of ints of length" + ia.length);
     default -> System.out.println("Something else");
     }
}
```

Listing 9

```
static void test(Object o) {
  switch (o) {
     case String s:
        if (s.length() == 1) { ... }
        else { ... }
        break;
        ...
     }
}

static void test(Object o) {
  switch (o) {
     case String s && (s.length() == 1) -> ...
     case String s -> ...
     ...
  }
}
```

Listing 10

```
static int coverage(Object o) {
  return switch (o) { // Error - not exhaustive
     case String s  -> s.length();
     case Integer i -> i;
  };
}
```

Listing 11

```
static int coverage(Object o) {
  return switch (o) {
     case String s  -> s.length();
     case Integer i -> i;
     default -> 0;
  };
}
```

Listing 12

```
sealed interface S permits A, B, C {}
final class A implements S {}
final class B implements S {}
record C(int i) implements S {}  // Implicitly final

static int testSealedExhaustive(S s) {
  return switch (s) {
     case A a -> 1;
     case B b -> 2;
     case C c -> 3;
  };
}
```

Listing 13

Variablen (`s`) weitergearbeitet werden. Das Ziel ist es, Redundanzen zu vermeiden und dadurch auch die Lesbarkeit zu erhöhen *(siehe Listing 8)*.

Der Unterschied zum zusätzlichen Cast mag marginal erscheinen. Für die Puristen unter den Java-Entwicklern spart das allerdings eine lästige Redundanz ein. Laut Java-Language-Architect Brian Goetz wurde die Sprache dadurch prägnanter und die Verwendung sicherer gemacht, da manuelle Typumwandlungen vermieden und stattdessen implizit durchgeführt werden.

Pattern Matching for switch (Second Preview)

Erstmals in der LTS-Version 17 neu hinzugekommen und im OpenJDK 18 als zweite Vorschau erneut dabei ist die Integration des Type Pattern (Prüfung mit instanceof) in die Switch Expression. Übrigens, diese Preview-Features sind standardmäßig deaktiviert und müssen sowohl beim Kompilieren als auch Starten mit `--enable-preview` explizit eingeschaltet werden. Und natürlich sollte man sie noch nicht produktiv einsetzen, da sie sich bis zur Finalisierung gegebenenfalls noch ändern können.

Listing 9 demonstriert den Funktionsumfang mit einem Beispiel aus der JEP-Dokumentation *[5]*. Der Selektionsausdruck `o` wird auf verschiedene Typen geprüft. Je nachdem, ob es sich um einen String, eine Farbe, einen Punkt oder ein `int`-Array handelt, wird der jeweilige `case`-Zweig aufgerufen. In einem normalen Switch darf im `case` nur gegen primitive Zahlen beziehungsweise deren Wrapper sowie Strings und Enums verglichen werden. Hier wird dagegen geschaut, ob `o` einem beliebigen Datentyp (inklusive Array) entspricht. Ist das der Fall, wird automatisch ein Cast ausgeführt und der Wert in einer Variablen gespeichert. Diese kann dann direkt weiterverarbeitet werden. Sollte `o` eine Null-Referenz sein, dann wird der optionale Zweig `case null` aufgerufen und nicht, wie bei `switch`-Statements sonst üblich, eine NullPointerException geworfen. Ein früher notwendiger, expliziter Null-Check vor dem Switch kann somit entfallen.

Die Type Patterns kann man mit Guarded Patterns noch verfeinern, indem zusätzlich zum Typ auf bestimmte Inhalte geprüft wird. Dazu wird ein boolescher Ausdruck mit einer logischen Und-Verknüpfung an das Type Pattern angehängt. Damit erspart man sich `if-else`-Abfragen im `case`-Zweig. *Listing 10* stellt beide Varianten gegenüber. Mit dem Guarded Pattern wirkt der Switch aufgeräumter und ist einfacher zu lesen.

Achtung, für das OpenJDK 19 sind bereits Änderungen zur Syntax der Guarded Patterns angekündigt. Mehr dazu dann in einem nachfolgenden Artikel zum nächsten Java Release.

```
static void error(Object o) {
  switch(o) {
     case CharSequence cs ->
        System.out.println("A sequence of length " + cs.length());
     // Error - pattern is dominated by previous pattern
     case String s ->
        System.out.println("A string: " + s);
     default -> {
        break;
     }
  }
}
```

Listing 14

Exhaustiveness

Im Gegensatz zu den Switch-Statements müssen die Switch-Expressions entweder sicherstellen, dass es für alle möglichen Fälle einen `case`-Zweig gibt, oder alternativ wird ein `default`-Block als Fallback vom Compiler verlangt. Diese Vollständigkeitsprüfung (Exhaustiveness) kann etwa bei Verwendung von Enum-Konstanten sichergestellt werden. Bei Enums muss es für jeden Wert einen `case`-Zweig geben. Während es sich dabei um die Konstanten eines Typs handelt, lassen sich mittlerweile durch Sealed Classes auch verschiedene Typen über das Type Pattern im Switch inklusive Vollständigkeitsprüfung verwenden.

Schauen wir uns in *Listing 11* ein Beispiel an, das nicht kompiliert. Es werden nicht alle Fälle, die der Parameter `o` annehmen kann, von den zwei `case`-Zweigen (String und Integer) abgedeckt.

Um den Compiler zufriedenzustellen, braucht es mindestens einen `default`-Zweig, der in dem Fall den Wert 0 zurückliefert. Damit ist bei der Switch-Expression immer sichergestellt, dass ein sinnvolles Ergebnis zurückgegeben wird *(siehe Listing 12)*.

Auf den `default`-Zweig kann nur verzichtet werden, wenn alle möglichen Werte behandelt sind. Das können nun also auch Instanzen aus einer Typhierarchie sein, die durch Sealed Classes beschränkt ist. In *Listing 13* leiten von dem Sealed Interface S drei Sub-Klassen ab, die selbst keine weiteren Sub-Typen erlauben, also Endknoten in der Hierarchie sind. Im Switch kann man genau auf die drei Sub-Klassen prüfen, der `default`-Zweig ist dann optional. Sollte in der späteren Entwicklung eine weitere Subklasse hinzukommen, käme es in allen Switch Expressions, in denen diese Sealed Classes verwendet werden, zu Compiler-Fehlern. Dadurch lassen sich leicht alle Vorkommen identifizieren und erweitern. Typische Fehlerquellen wie das Verschlucken fehlender `case`-Zweige durch den `default`-Block gehören damit der Vergangenheit an.

Im JDK 18 kamen im JEP 420 zwei kleinere Änderungen zum „Pattern Matching for switch (Second Preview)" hinzu: einerseits eine Verbesserung bei der Dominanzprüfung und andererseits ein Bugfix bei der Exhaustiveness-Prüfung im Zusammenhang mit Sealed Classes.

Bei der Dominanzprüfung zeigt der Compiler an, wenn es unerreichbare `case`-Zweige gibt. In *Listing 14* dominiert `CharSequence` den Typ String, somit kann `case String s` nie aufgerufen werden und wird mit einem Compiler-Fehler quittiert.

Ein Fall wurde anfänglich nicht bedacht. So konnte das Beispiel in *Listing 15* mit dem JDK 17 anstandslos übersetzt werden, auch wenn die String-Konstante in keinem Fall aufgerufen wird. Ab dem JDK 18 wird dieser nicht erreichbare Code zu einem Fehler führen.

Die zweite Verbesserung war ein Bugfix bei generischen Sealed Classes, die Typ-Parameter verwenden. Da wurde im JDK 17 ein bestimmter Fall bei der Vollständigkeitsprüfung noch nicht korrekt behandelt. *Listing 16* zeigt ein Beispiel, wie eine Hierarchie mit Typ-Parametern aussehen kann. In diesem Fall wird nur ein `case`-Zweig (für den Typ B) benötigt, da A (Subtyp von I<String>) nicht zum Typ der Variable I<Integer> i passt.

```
Object obj = ...
switch (obj) {
  case String s && s.length() < 5 ->
     System.out.println(s.toUpperCase());
  case "Test" -> System.out.println("constant string");
  ...
}
```

Listing 15

```
sealed interface I<T> permits A, B {}
final class A<X> implements I<String> {}
final class B<Y> implements I<Y> {}

static int testGenericSealedExhaustive(I<Integer> i) {
  return switch (i) {
     // Exhaustive as no A case possible!
     case B<Integer> bi -> 42;
  }
}
```

Listing 16

Fazit und Ausblick

Im Vergleich zu den letzten halbjährlichen Releases erscheint der Umfang des OpenJDK 18 mit „nur" 9 umgesetzten JEPs *[1]* deutlich überschaubarer. Nach dem LTS-Release von Java 11 war das allerdings ähnlich. Und vielleicht ist das einfach die Ruhe vor dem nächsten Sturm. Denn schon im kommenden Jahr (Herbst 2023) erwartet uns ja bereits die nächste LTS-Version.

Eine spannende Frage ist, wie viele Java-Projekte überhaupt schon die modernen Java-Versionen (11+) verwenden. Laut diversen, nicht repräsentativen Umfragen ist Java 8 immer noch stark vertreten. Der Schritt von 8 auf 11 kann aufwendig sein. Aus der Erfahrung heraus fallen die Updates danach leichter. Die geringste Hürde bei den Update-Strategien stellen halbjährliche Aktualisierungen auf die jeweils aktuelle Major-Version dar. Durch die kurze Zeitspanne zwischen den LTS-Versionen (Long Term Support) von nur noch zwei Jahren stellen Versionssprünge zum nächsten LTS-Release aber auch keine große Hürde mehr dar. Das Hauptproblem ist dann vermutlich eher die noch fehlende Unterstützung von Bibliotheks-, Framework- und Tool-Herstellern. Für die stellen die kurzen Release-Zyklen tatsächlich eine große Herausforderung dar.

Bei den JDK-Distributionen hat man mittlerweile auch viele verschiedene gute Alternativen. Technisch basieren insbesondere die freien, aber auch viele der kommerziellen Varianten sowieso auf dem OpenJDK von Oracle. Insbesondere das frühere AdoptOpenJDK (jetzt Temurin des Projekts Adoptium bei der Eclipse Foundation) *[6]* hat sehr viele unterstützende Firmen mit im Boot und auch eine stetig steigende Nutzerzahl. Interessant ist, dass die Temurin JDK Builds mittlerweile auch gegen das TCK geprüft werden. Das Java SE Technology Compatibility Kit (TCK) wird von Oracle angeboten, um Java Binaries als kompatibel zum Java-Standard zertifizieren zu lassen. Das TCK ist nicht frei verfügbar und muss bei Oracle lizenziert werden. Früher war das beim AdoptOpenJDK nicht der Fall. Dafür gab es dort schon immer eine große Menge von separaten Tests. In der Summe sind die Temurin Java Builds nun also noch besser getestet. Infrastrukturell wird ebenfalls ein sehr hoher Aufwand betrieben, um die verschiedensten, teilweise exotischen Va-

rianten aus den Kombinationen Version, Betriebssystem und Plattform anzubieten.

Das Java-Ökosystem ist also lebendiger denn je und weiterhin sehr innovativ. Konkurrenz wie beispielsweise Python (hauptsächlich wegen Machine/Deep Learning), JavaScript, Go und die C-basierten Sprachen beleben das Geschäft. Java, das besonders im Unternehmensanwendungsumfeld vertreten ist, wird aber weiterhin ein gehöriges Wort mitreden.

Das nächste LTS-Release (Java 23) steht schon Ende nächsten Jahres an. Neben den in diesem Artikel angesprochenen Dauer-Themen stehen noch viele spannende Initiativen aus den verschiedenen Inkubator-Projekten (Amber, Loom, Valhalla, Panama, …) in den Startlöchern. Sehr interessant sind zum Beispiel Looms Virtual Threads (früher Fibers genannt), die es voraussichtlich in den Funktionsumfang der anstehenden Version (OpenJDK 19 *[7]*) schaffen. Noch nicht eingeplant, aber nicht minder spannend sind die Bestrebungen des Projekts Valhalla, das Java-Typ-System (primitive vs. Referenz-Typen) zu vereinheitlichen (Einführung von Value und Primitive Types).

Schon seit einigen Releases dabei sind das Vector-API und das Foreign Function & Memory API. Das Vector-API wurde bereits in Java 16 und Java 17 als Incubator-Feature vorgestellt und geht nun in die dritte Runde (Third Incubator). Es geht dabei nicht um die bereits seit Java 1.0 existierende Klasse java.util.Vector. Vielmehr will man die modernen Möglichkeiten der SIMD-Rechnerarchitektur mit Vektorprozessoren ins JDK bringen. Single Instruction Multiple Data (SIMD) lässt viele Prozessoren gleichzeitig unterschiedliche Daten verarbeiten. Durch die Parallelisierung auf Hardware-Ebene verringert sich beim SIMD-Prinzip der Aufwand für rechenintensive Schleifen.

Wer schon sehr lange in der Java-Welt unterwegs ist, wird sich sicher noch an das Java Native Interface (JNI) erinnern. Damit kann man nativen C-Code aus Java heraus aufrufen. Der Ansatz ist aber relativ aufwendig und fragil. Das Foreign-Linker-API bietet einen statisch typisierten, rein Java-basierten Zugriff auf nativen Code. Zusammen mit dem Foreign-Memory-Access-API kann diese Schnittstelle den bisher fehleranfälligen und langsamen Prozess der Anbindung einer nativen Bibliothek beträchtlich vereinfachen. Mit Letzterer bekommen Java-Anwendungen die Möglichkeit, außerhalb des Heap zusätzlichen Speicher zu allokieren. Ziel der neuen APIs ist es, den Implementierungsaufwand um 90 % zu reduzieren und die Leistung um Faktor 4 bis 5 zu beschleunigen. Beide APIs sind seit dem JDK 14 beziehungsweise 16 im JDK enthalten und nun im JEP 419 wieder als Inkubator dabei. Wann sie finalisiert werden können, steht noch nicht fest. Inkubator-Stadium bedeutet, dass die Features noch erhebliche Änderungen durchlaufen können.

So wie es aussieht, wird uns als Java-Entwickler so schnell nicht langweilig werden. Die Zukunftsaussichten für die Sprache und das Ökosystem sind weiterhin sehr rosig.

Referenzen:

[1] *https://openjdk.java.net/projects/jdk/18/*
[2] *https://download.java.net/java/early_access/jdk18/docs/api/java.base/java/lang/System.html#file.encoding*
[3] *https://javaalmanac.io/jdk/18/apidiff/17/*
[4] *https://openjdk.java.net/projects/amber/*
[5] *https://openjdk.java.net/jeps/420*
[6] *https://adoptium.net/*
[7] *https://openjdk.java.net/projects/jdk/19/*

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

Falk Sippach ist bei der embarc Software Consulting GmbH als Softwarearchitekt, Berater und Trainer stets auf der Suche nach dem Funken Leidenschaft, den er bei seinen Teilnehmern, Kunden und Kollegen entfachen kann. Bereits seit über 15 Jahren unterstützt er in meist agilen Softwareentwicklungsprojekten im Java-Umfeld. Als aktiver Bestandteil der Community (Mitorganisator der JUG Darmstadt) teilt er zudem sein Wissen gern in Artikeln, Blog-Beiträgen sowie bei Vorträgen auf Konferenzen oder User-Group-Treffen und unterstützt bei der Organisation diverser Fachveranstaltungen. Falk twittert unter @sippsack.

Neuer Microservice, neuer Technologie-Stack mit Kotlin statt Java

Elmar Brauch, Deutsche Telekom IT GmbH

Ein angenehmer Aspekt von Microservices ist, dass ihre Entwicklung häufig auf der grünen Wiese startet. Ein neuer Microservice ist damit immer eine Chance, einen neuen Technologie-Stack auszuprobieren. Für eingefleischte Java-Entwickler/innen bietet sich dazu die Programmiersprache Kotlin an. Als JVM-basierte Programmiersprache kann Kotlin problemlos das bei Java-Entwickler/innen beliebte Spring-Framework verwenden. Ich zeige in diesem Artikel, wie ihr mit Spring Boot schnell einen Kotlin-Microservice mit REST-API aufsetzt.

Neuer Microservice – die Gelegenheit, mit Kotlin anzufangen…

In meinem beruflichen Umfeld entwickeln wir deutlich häufiger neue Microservices als neue Systeme mit klassischen Software-Architekturen. Da sich Microservices auf eine Geschäfts-Capability *[0]* konzentrieren, ist ihre Codebase im Vergleich zu herkömmlichen Systemen meist deutlich kleiner. Die Situation, einen neuen Microservice auf der grünen Wiese zu entwickeln, kommt relativ häufig vor. Daher ist dies immer eine Gelegenheit, einen neuen Technologie-Stack auszuprobieren, beispielsweise mit der Programmiersprache Kotlin anstelle von Java. Aufgrund der kleinen Codebase würde ein unpassender Technologie-Stack im schlimmsten Fall nur einen einzelnen kleinen Microservice betreffen.

Als Software-Entwickler arbeite ich hauptsächlich mit Java. Ein guter Online-Kurs *[2]* *[11]* überzeugte mich von der Programmiersprache Kotlin. So war mein erster Wow-Effekt, als ich realisierte, dass Kotlin in der JVM läuft und somit (fast) alle Java-Bibliotheken, also auch Spring, verwenden kann. Ab dann hat mich Kotlin damit beeindruckt, dass es als moderne Programmiersprache vieles kompakter und weniger fehleranfällig ausdrücken kann. Schaut euch zum Beispiel an, wie Kotlin NullPointerExceptions vermeidet *[3]*. Der nächste Microservice war meine Gelegenheit, Kotlin in der Praxis auszuprobieren. Das Spring-Framework ist in meinem Umfeld neben Java die zweite Konstante. Um weniger experimentierfreudige Kollegen und Kolleginnen mitzunehmen, habe ich Spring als Framework nicht ausgetauscht. In den nächsten Abschnitten schauen wir uns den Kotlin- und Spring-Technologie-Stack genauer an.

1. IDE auswählen: IntelliJ vs. Eclipse

Eigentlich entwickle ich am liebsten mit Eclipse. Da die Programmiersprache Kotlin aber von JetBrains designt und entwickelt wird, habe ich für die Entwicklung mit Kotlin die IntelliJ-IDEA ausprobiert, die ebenfalls von JetBrains entwickelt wird. Wie zu erwarten, funktioniert das sehr gut, da Programmiersprache und IDE von derselben Firma stammen. Die Kotlin-Website hat eine empfehlenswerte Anleitung zum Loslegen *[1]*.

Es gibt auch diverse Anleitungen, wie ihr mit Eclipse in Kotlin programmieren könnt. Dazu installiert ihr euch ein Kotlin-Plug-in über den Eclipse Marketplace und könnt dann in der Theorie loslegen. In der Praxis hatte ich leider immer wieder „komische" Fehler, sodass ich entschieden habe, die IntelliJ-Community-Edition zu verwenden. Ich teile hier keinen Link zu einer Anleitung, weil ich damit kein Glück hatte.

2. Kotlin-Projekt mit Spring Boot aufsetzen

Neue Spring-Projekte setzt ihr leicht mit Plug-ins in der Entwicklungsumgebung auf *[9]* oder ihr verwendet den Online Spring Initializr *[4]*. Der ist mindestens genauso leicht zu bedienen wie die IDE-Plug-ins. *Abbildung 1* zeigt den im Tutorial verwendeten Spring Initializr und die zu setzenden Parameter:

- Gradle als Build Tool
- Kotlin als Sprache
- Spring Boot Version 2.6.6 (oder höher)
- jar als Ziel-Paket meiner Anwendung
- Java-Version 11 für die unterliegende JVM (Programmiersprache ist Kotlin)
- Spring Web und Spring Boot DevTools als Dependencies

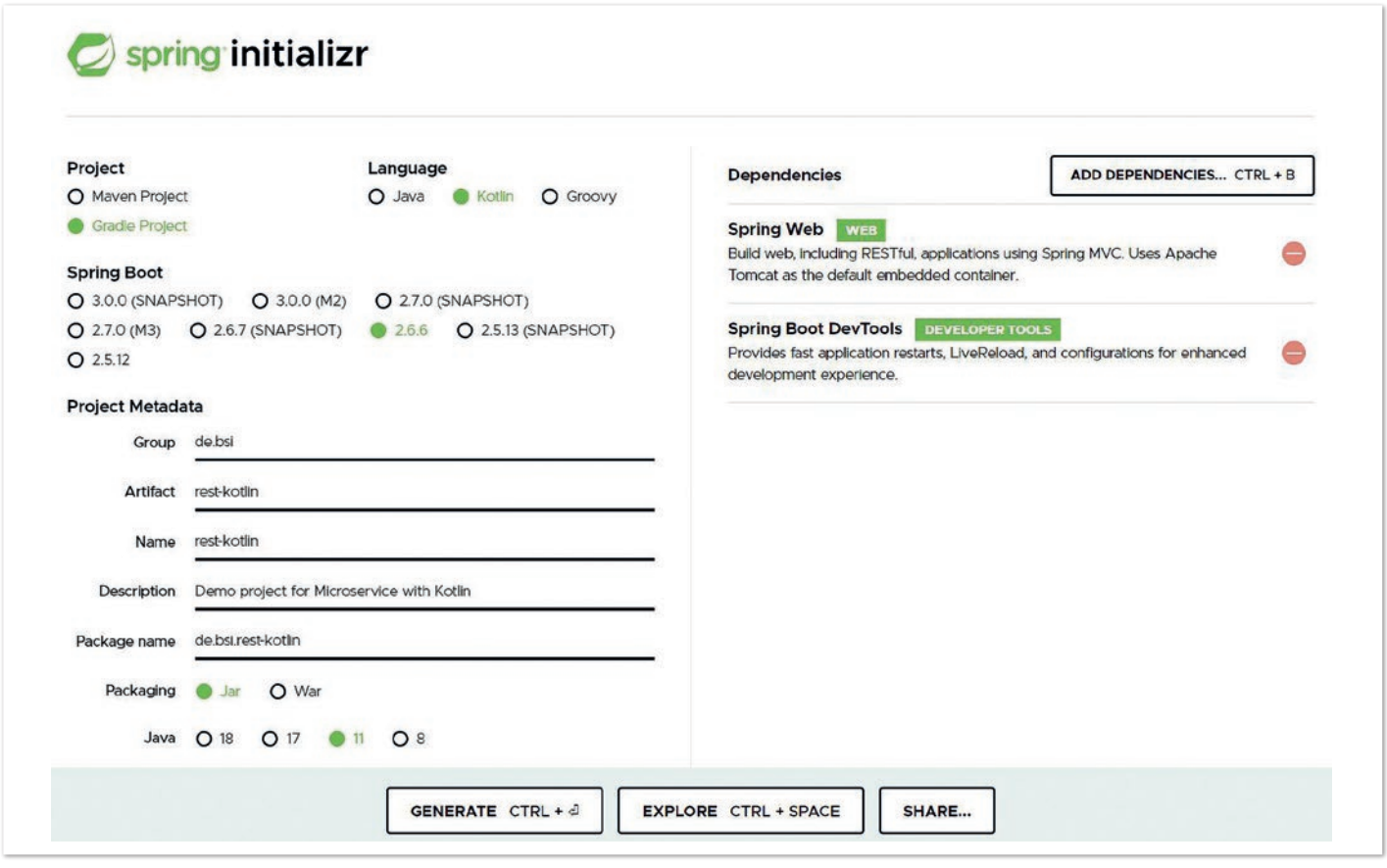

Abbildung 1: Spring Initializr [4]

Nach dem Klick auf den GENERATE-Knopf bekommt ihr ein zip-File, das euer Spring-Boot-Kotlin-Projekt enthält und in IntelliJ importiert beziehungsweise darin geöffnet werden kann.

3. Kotlin-Projekt starten

IntelliJ sollte dann automatisch den Gradle-Build starten und ihr solltet ein Projekt ohne Compile-Fehler haben (Infos zu Gradle im Spring-Boot-Kontext *[5]*). Danach könnt ihr die Spring-Boot-Anwendung starten, indem ihr die Kotlin-Datei mit der `static-void-main`-Methode ausführt (`Run ....main()`). In einem neu generierten Projekt sollte es im src/main-Verzeichnisbaum nur eine kt-Datei geben, die dann auch die Methode `main` enthält.

In der Konsole beziehungsweise im Run Tab in IntelliJ seht ihr dann die typischen Spring-Log-Einträge, also auch, unter welchem Port die Anwendung läuft und wie lange der Start gedauert hat *(siehe Listing 1)*. Ruft ihr die Anwendung jetzt im Browser unter *http://localhost:8080* auf, wird euch die „Whitelabel Error Page" der Anwendung angezeigt. Im nächsten Abschnitt schreiben wir Code, um das zu ändern.

Gradle Dependencies von Spring Boot

Zum Entwickeln des REST-API können wir in Kotlin die gleichen Spring-Annotationen verwenden, die wir auch in Java einsetzen. Das liegt daran, dass wir die gleichen Spring Dependencies in Gradle verwenden, die wir auch mit Java nutzen würden. Die build.gradle.kts wurde vom Spring Initializr generiert und hat in unserem Fall die in *Listing 2* gezeigten Dependencies. Zusätzlich zu den Spring-Bibliotheken gibt es noch spezielle Dependencies für Kotlin, die aber auch vom Spring Initializr konfiguriert wurden.

Rest-API in Kotlin

Das eigentliche REST-API wird von einem RestController bereitgestellt. Microservices haben typischerweise ein einfaches API, das sich auf eine Domäne konzentriert. Das hier gezeigte Item-API bietet daher auch nur drei Methoden an:

- `POST /item`: Erstellt ein neues Item
- `GET /item?itemName=Ball`: Sucht ein vorhandenes Item anhand seines Namens
- `DELETE /item/<itemId>`: Löscht ein Item anhand seiner ID

Items sind einfache Objekte mit drei Attributen. Sie werden im JSON-Format mit dem API ausgetauscht und sehen wie in *Listing 3* gezeigt aus.

Die Klasse `ItemController` fungiert in unserem Microservice als RestController und verarbeitet HTTP-Requests mit ihren Methoden. Die POST-Methode ist in *Listing 4* gezeigt.

- `@RestController` macht aus der Kotlin-Klasse einen Spring MVC Controller im Kontext des Design Patterns Model-View-Controller *[6]*. Der Controller wird vom standardmäßig konfigurierten Spring RequestMappingHandler erkannt und verarbeitet.
- `@RequestMapping`-Annotationen werden verwendet, um zu definieren, auf welche HTTP-Methoden, URL-Pfade, Datentypen usw. die Klasse und/oder die jeweilig annotierte Methode reagieren. Hier verarbeitet der Controller alle eingehenden Requests mit dem Pfad `/item`.
- `@PostMapping` entspricht dabei `@RequestMapping`, ergänzt um die zu verwendende HTTP-POST-Methode, also: `@RequestMapping(method = RequestMethod.POST)`. Die Annotation an der Methode übernimmt den definierten Pfad in der Klassen-Annotation als Präfix.
- `@RequestBody` gibt an, dass der Body des POST-Request im annotierten Parameter an die Methode übergeben wird. Der Request-Body ist im JSON-Format und wird von Spring automatisch mittels Jackson-Bibliothek in die Datenklasse Item ge-

```
2022-03-31 22:13:11.002  INFO 18444 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started
on port(s): 8080 (http) with context path ''
2022-03-31 22:13:11.017  INFO 18444 --- [  restartedMain] de.bsi.restkt.RestKtApplication : Started RestKtApplica-
tion in 3.031 seconds (JVM running for 3.617)
```

Listing 1: Log-Ausschnitt beim Start einer Spring-Anwendung

```
dependencies {
	implementation("org.springframework.boot:spring-boot-starter-web")
	implementation("com.fasterxml.jackson.module:jackson-module-kotlin")
	implementation("org.jetbrains.kotlin:kotlin-reflect")
	implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")
	developmentOnly("org.springframework.boot:spring-boot-devtools")
	testImplementation("org.springframework.boot:spring-boot-starter-test")
}
```

Listing 2: Gradle Dependencies für Spring und Kotlin

```
{
	"name": "Ball",
	"id": "1",
	"date": "2022-03-27T09:55:16.930+00:00"
}
```

Listing 3: Daten im JSON-Format

```
import org.springframework.http.HttpStatus
import org.springframework.http.ResponseEntity
import org.springframework.http.ResponseEntity.*
import org.springframework.web.bind.annotation.*

@RestController
@RequestMapping("/item")
class ItemController {

     val items = mutableListOf<Item>()

     @PostMapping
     fun addItem(@RequestBody item : Item): ResponseEntity<Unit> {
        items.add(item)
        return status(HttpStatus.CREATED).build()
     }
     ...
}
```

Listing 4: Klasse mit RestController und POST-Methoden-Mapping

parst. Weiter unten zeige ich den Code dieser Klasse – er ist vergleichbar mit einer einfachen POJO-Klasse in Java.

- `ResponseEntity` ist eine Klasse des Spring-Frameworks, mit der eine HTTP- Response gebaut werden kann. Im Beispiel wird einfach der HTTP-Code 201 (CREATED) gesetzt und ein leerer HTTP Response Body mit den Standard-Headern zurückgeschickt. Status, Body und Header können generell mit dieser Klasse gesetzt werden, sodass unser Controller mit entsprechenden HTTP Responses antwortet.
- Zum Speichern der Item-Objekte verwende ich eine veränderbare Liste, die ich mit der Kotlin-Methode `mutableListOf<Item>()` erzeugt habe. Unter der Haube verwendet Kotlin hier die Klasse `ArrayList` von Java.

HTTP-Methoden hinzufügen

Spring unterstützt in Rest-Controllern alle HTTP-Methoden:

- GET mittels Annotation `@GetMapping`
- POST mittels Annotation `@PostMapping`
- PUT mittels Annotation `@PutMapping`
- PATCH mittels Annotation `@PatchMapping`
- HEAD mittels `@RequestMapping(method = RequestMethod.HEAD)`
- DELETE mittels Annotation `@DeleteMapping`
- OPTIONS mittels `@RequestMapping(method = RequestMethod.OPTIONS)`
- TRACE mittels `@RequestMapping(method = RequestMethod.TRACE)`

Die jeweilige Annotation wird an die zugehörigen `public`-Methoden geschrieben. *Listing 5* zeigt zwei weitere Beispiele für eine GET- und eine DELETE-Methode.

- `@RequestParam` ist eine weitere Spring-Annotation, die verwendet wird, um Query-Parameter aus der URL in einen Methoden-Parameter zu überführen. Wird unser REST-Service mit der URL *http://localhost:8080/item?itemName=Ball* aufgerufen, so nimmt der String-Parameter `itemName` automatisch den Wert „Ball" an. Wenn der Parametername mit dem Namen des Request-Parameters in der URL nicht übereinstimmt, bekommen wir automatisch einen Bad-Request-Fehler mit HTTP-Code 400.
- Mit `ok(it)` wird hier eine HTTP-Response mit dem Code 200 erstellt. Im Body dieser Response beziehungsweise in der Variable it befindet sich die gefundene Item-Instanz. `it` ist in Kotlin ein Keyword zur Vereinfachung von Lambda-Ausdrücken.

```
@GetMapping
fun getItemByName(@RequestParam(required = true) itemName: String): ResponseEntity<Item> =
     items.stream()
        .filter { it.name.equals(itemName, true) }
        .findFirst().map { ok(it) }
        .orElse(notFound().build())

@DeleteMapping("/{itemId}")
fun deleteItemById(@PathVariable itemId: String?): ResponseEntity<Unit> =
  if (items.removeIf{it.id == itemId}) noContent().build()
  else notFound().build()
```

Listing 5: GET- und DELETE-Methoden-Mapping

```
@PostMapping
fun addItem(@RequestBody item : Item): ResponseEntity<Unit>

@PostMapping
public ResponseEntity<Void> addItem(@RequestBody Item newItem)
```

Listing 6: Methodendeklaration in Kotlin und Java

```
data class Item (val name: String, val id: String, val date: Date = Date())
```

Listing 7: Kotlin data class

- Eine weitere Variante der Parameter-Übertragung von der URL in die `public`-Methode (`@PathVariable`) wird in der mit `@DeleteMapping` annotierten Methode gezeigt. In `@DeleteMapping` wird zusätzlich ein Teil des URL-Pfades definiert, nämlich `/{itemId}`. Im vorherigen Abschnitt haben wir gesehen, dass man auch in der `@RequestMapping`-Annotation den URL-Pfad spezifizieren kann. Wenn dies auf Klassenebene geschieht, so gilt es für alle Methoden. Eine valide URL für unsere DELETE-Methode würde so aussehen: *http://localhost:8080/item/123*. Die in geschweiften Klammern notierten Teile der URL werden an Parameter mit der Annotation `@PathVariable` übergeben. In unserem Beispiel hätte der String-Parameter `itemId` also den Wert „123".
- Die hier gezeigten Annotationen lassen sich beliebig kombinieren, sodass man das REST-API wie benötigt implementieren kann.

Gegenüberstellung Kotlin und Java

Methodendeklarationen sehen in Kotlin anders aus als in Java. Details dazu findet ihr in der Kotlin-Dokumentation *[7]*. In *Listing 6* zeige ich eine direkte 1:1-Gegenüberstellung.

- Der Kotlin-Code ist kompakter, wenn der Methoden-Rückgabewert in einem Statement berechnet und einfach per Operator zurückgegeben wird. So spart ihr geschweifte Klammern, `return`-Statement und die Deklaration des Methoden-Rückgabetyps. Das habe ich so in `getItemByName` und `deleteItemById` gemacht, siehe *Listing 5*.
- In Kotlin können auch `if`-Ausdrücke einen Rückgabewert haben. In `deleteItemById` wird das Ergebnis des `if`-Ausdrucks direkt als Methodenergebnis zurückgegeben. Das ist hier eine ResponseEntity mit dem HTTP-Code 204 (No Content) oder 404 (Not Found). Java entwickelt sich ebenfalls in diese Richtung, wie man anhand der in Java 14 eingeführten Neuerungen am `switch`-Statement sieht *[10]*.
- Die Item-Klasse war in Java eine reine Datenklassen mit Getter- und Setter-Methoden für alle Attribute – also voll von Boilerplate-Code. Kotlin bietet `data class`-Klassen an, die es uns ermöglichen, alles in eine einzige Zeile zu schreiben und somit sämtlichen Boilerplate-Code zu vermeiden, siehe *Listing 7*.

Kotlin hat noch viele weitere tolle Features, die ich hier aber nicht im Detail erklären kann, da es hier in erster Linie um das REST-API

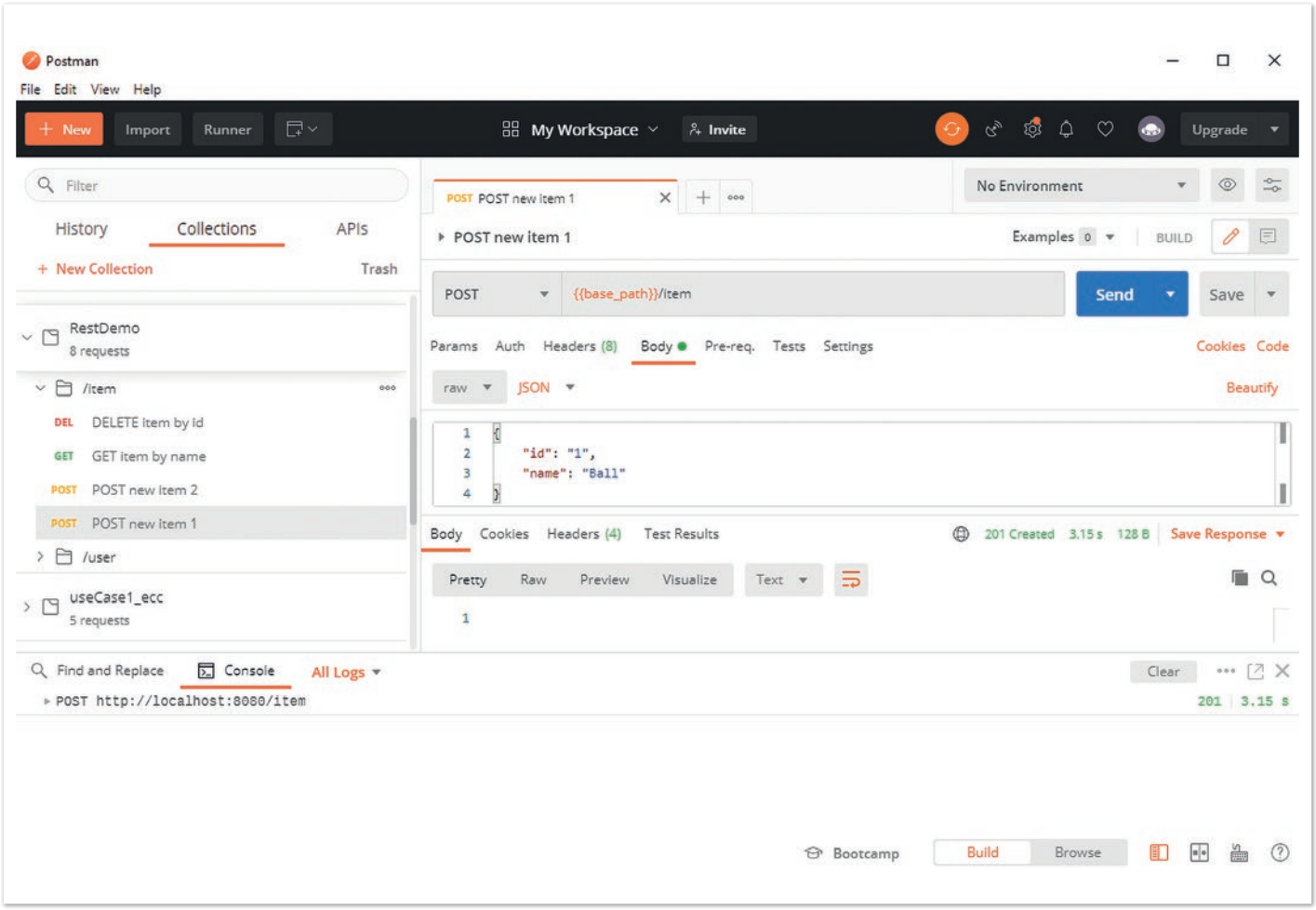

Abbildung 2: Postman

eines Microservice mit Spring-Mitteln geht. Einen vergleichbaren Service in der Programmiersprache Java stelle ich in meinem Blog vor *[6]*.

Item-API ausprobieren

Zum Demonstrieren und Ausprobieren von REST-APIs verwende ich gerne Postman. In GitHub *[8]* findet ihr eine Postman-Testsuite, in der ich alle Methoden vorbereitet habe. Diese Postman-Testsuite funktioniert sowohl bei der Kotlin- als auch bei der Java-Variante *[6]* meines REST-API. Die Spring-Boot-Anwendung muss vorher gestartet werden, das funktioniert so wie weiter oben beschrieben. *Abbildung 2* zeigt einen HTTP-POST-Request mit Postman.

Fazit

Neben des hier gezeigten REST-API bestehen Microservices meist noch aus weiteren Bestandteilen. So gibt es eine Geschäftslogik, die die Geschäfts-Capability des Microservice umsetzt und dazu gegebenenfalls andere Microservices, Cloud-Services oder eine Datenbank benutzt. In diesem Artikel legte ich den Fokus aber auf das REST-API des in Kotlin geschriebenen Microservice. Den kompletten Code zu diesem Artikel findet ihr in GitHub *[8]* und eine ausführlichere Einführung von mir bei Udemy *[11]*.

Microservices in Kotlin zu schreiben, sollte Java-Programmierer/innen im Vergleich zu anderen Programmiersprachen relativ leichtfallen, da beide Sprachen auf der JVM basieren. Mit Kotlin könnt ihr bestehende Java-Bibliotheken über euer Build-Tool einbinden, so wie hier Spring als Dependency geladen wurde. Damit wird der Einstieg erheblich erleichtert, weil ihr in der für euch neuen Programmiersprache Kotlin etablierte und vertraute Frameworks wie Spring verwenden könnt. Um produktiv zu arbeiten, müsst ihr also „nur" eine neue Programmiersprache lernen und nicht noch zusätzlich neue Frameworks.

Quellen

[0] *https://www.martinfowler.com/microservices*
[1] *https://kotlinlang.org/docs/jvm-get-started.html*
[2] *https://www.coursera.org/learn/kotlin-for-java-developers*
[3] *https://kotlinlang.org/docs/null-safety.html*
[4] *https://start.spring.io/*
[5] *https://docs.gradle.org/current/samples/sample_building_spring_boot_web_applications.html*
[6] *https://agile-coding.blogspot.com/2020/10/rest-json-apis-in-java-leicht-gemacht.html*
[7] *https://kotlinlang.org/docs/functions.html*
[8] *https://github.com/elmar-brauch/rest-controller-kotlin*
[9] *https://agile-coding.blogspot.com/2020/09/microservices-mit-spring-boot-erstellen.html*
[10] *https://agile-coding.blogspot.com/2021/09/java-12-bis-15-features.html*
[11] *https://www.udemy.com/course/api-und-kotlin/?referralCode=DDCE6B8332EAB4AEC664*

Elmar Brauch
Deutsche Telekom IT GmbH
elmar.brauch@t-online.de

Elmar Brauch arbeitet als Software-Entwickler und -Architekt bei der Deutschen Telekom in Darmstadt. In vielen verschiedenen Projekten hat er Software entwickelt, Architektur designt und Prozesse automatisiert. Das dabei gesammelte Wissen teilt er mit seinen Kolleg/innen und anderen Software-Entwickler/innen in seinem Blog https://agile-coding.blogspot.com und bei Udemy https://www.udemy.com/user/elmar-brauch/.

Rahmenlos – Java-SE-basierte Microservices ohne Framework

Markus Karg

Microservice-Frameworks nehmen einem viel Arbeit und Verantwortung ab, entziehen aber auch Einflussmöglichkeit. Nicht in jedem Fall sind sie daher die optimale Lösung. Für viele Anwendungszwecke ist es der bessere Ansatz, sich direkt auf Java SE, einzeln selektierte APIs und bewusst ausgesuchte Implementierungen zu stützen.

Microservices sind derzeit der allgemein vorherrschende Architekturstil. Zu ihrer Umsetzung greifen die meisten Entwickler zu spezialisierten Frameworks wie Quarkus oder Helidon oder zu abgespeckten Application-Servern wie Payara Micro. Für viele Anwendungszwecke ist der Overhead dieser Produkte, vor allem aber der notwendige Lernaufwand für deren Konfiguration und Administration, weder nötig noch gewünscht. Zudem bergen diese Produkte die Gefahr, sich dauerhaft an eine bestimmte Implementierung zu binden – mit allen Vor- und Nachteilen, Stichwort „Vendor Lock-in" *[1]*.

In manchen Fällen wünscht man sich daher, den Dienst als reine Java-SE-Anwendung zu implementieren: Die Plattform ist bekannt und beherrscht, es existiert mittlerweile eine große Zahl an breit unter-

stützten Distributionen [2] und bei Stützung auf die unter Jakarta EE bekannt gewordene Sammlung an API-Spezifikationen lässt sich eine Anwendung jederzeit kurzfristig und ohne Programmänderung auf andere Implementierungen umziehen – beispielsweise, wenn ein Bug im HTTP-Server, in der JAX-RS-Implementierung oder im JSON-Parser einen Showstopper für die Auslieferung darstellt und dessen Hersteller keine Anstalten macht, ihn zeitnah zu fixen. Und wäre es nicht schön, einfach eine Anwendung in der IDE starten und debuggen zu können, ohne zig Hersteller-Plug-ins und Framework-Zusatztools zu benötigen? Genau das geht mit Jakarta EE, denn die Plattform, die vor allem durch Application-Server bekannt wurde, hat sich mittlerweile Cloud- und Microservice-tauglich gemacht und dient beispielsweise als Unterbau für Spring 6 [3].

Das Programmiermodell von Jakarta EE, ehemals Java EE, war bisher darauf ausgerichtet, dass ein Java-Container (nicht zu verwechseln mit einem Docker-Container) durch einen Dienstleister (beispielsweise einen Application-Server) bereitgestellt und verwaltet wird. Die Lebenszeit der Anwendung wird durch diesen Container kontrolliert. Grundsätzlich nichts Schlechtes, doch dummerweise sind diese Dienstleister allesamt sehr schwergewichtig und ein grundlegender Aspekt war nicht standardisiert: Wie starte ich eigentlich einen solchen Java-Container? Manche Application-Server konnten dies über ein bestimmtes API aus Java heraus tun, andere benötigten spezielle Tools oder boten schlicht überhaupt keine Kontrolle darüber (Server an = Container an = Application läuft). Manche konnten es nur, indem man spezielle Befehle an der Kommandozeile herausgab, die jedoch herstellerspezifisch waren: Payara Micro startete man beispielsweise anders als WildFly. An diesem Problem haben leider auch die modernen Frameworks wie Quarkus oder Helidon nichts geändert. Der technische Weg, wie die Anwendung gestartet wird, das sogenannte Bootstrapping, ist in jedem Framework anders. Für Java-Entwickler, die die Phrase „void main(String[] args)“ beziehungsweise den Befehl „java -jar MeineApp.jar“ seit Tag eins auswendig kennen, eigentlich vollständig unverständlich.

Glücklicherweise gibt es seit Jakarta EE 10 eine portable und simple Lösung, genau diesen Weg auch für Server-Anwendungen zu gehen: Das Java-SE-Bootstrap-API von JAX-RS 3.1, das Bestandteil von Jakarta EE ist, in Kombination mit dem Java-SE-Bootstrap-API

```
<dependencies>
  <dependency>
     <groupId>jakarta.ws.rs</groupId>
     <artifactId>jakarta.ws.rs-api</artifactId>
     <version>3.1.0</version>
  </dependency>

  <dependency>
     <dependency>
        <groupId>org.glassfish.jersey.inject</groupId>
        <artifactId>jersey-cdi2-se</artifactId>
        <version>3.1.0</version>
        <scope>runtime</scope>
     </dependency>

     <dependency>
        <groupId>org.glassfish.jersey.containers</groupId>
        <artifactId>jersey-container-grizzly2-http</artifactId>
        <version>3.1.0</version>
        <scope>runtime</scope>
     </dependency>
</dependencies>
```

Listing 1: Die Anwendung wird gegen das JAX-RS API gebaut, zur Ausführung wird Jersey als deren Implementierung genutzt.

```
public class EchoServer {
  public static void main(final String[] arguments) throws InterruptedException {
     SeBootstrap.start(EchoApplication.class);
     Thread.currentThread().join();
  }

  @ApplicationPath("echo")
  @Path("")
  public static class EchoApplication extends Application {
     @Override
     public Set<Class<?>> getClasses() {
        return Set.of(EchoApplication.class);
     }

     @GET
     public String helloWorld() {
        return „Hello World“;
     }
  }
}
```

Listing 2: Ein simpler RESTful WebService, der vollständig portabel ist, da er ausschließlich auf Java SE und Jakarta-EE-APIs zurückgreift.

von CDI 2, das schon länger existierte, allerdings kaum bekannt war, stellen im Kern das bereit, was eine moderne Web-Server-Anwendung zum Booten braucht. Um sich dieser zu bedienen, genügt es, zwei Abhängigkeiten in den Klassenpfad zu laden *(siehe Listing 1)*: Das API an sich, um die Anwendung dagegen kompilieren zu können, und auch eine (frei wählbare!) Implementierung, um es zur Laufzeit zu nutzen. In diesem Artikel kommt hierzu beispielsweise als Implementierung von JAX-RS etwa das ehemals von Oracle entwickelte und mittlerweile durch die Eclipse Foundation gepflegte Produkt „Jersey" zum Einsatz, es kann jedoch auch jedes andere Produkt genutzt werden, das vollständig kompatibel zu JAX-RS 3.1 (oder höher) ist, wie beispielsweise RESTeasy. Der Quellcode der Anwendung ist relativ simpel und in *Listing 2* zu finden.

Es findet hier keinerlei „Magie" statt, wie man sie üblicherweise mit Java EE in Verbindung brachte. Im Gegenteil, der gesamte Ablauf, vor allem das Bootstrapping, ist klar ersichtlich und kann sogar in der IDE ohne spezielle Plug-ins debuggt werden – in-process, also ohne IPC- oder gar Netzwerk-Verbindung dazwischen! Dieses Mini-Echo-Programm benötigt CDI noch nicht einmal, daher habe ich sowohl das Dependency-Injection-API als auch dessen Implementierung (beispielsweise „Weld") sogar einfach mal weggelassen und spare mir dessen eher zeitaufwendige Initialisierung!

Nicht jede Web-Anwendung ist ja zwingend eine Zwölf-Faktoren-Anwendung *[4]*, ebenso wenig zwingend eine Cloud-Anwendung und auch nicht zwingend ein Microservice. Noch nicht einmal Security oder gar TLS wird von wirklich jeder Anwendung benötigt. Oftmals genügen ein oder zwei Bibliotheken, beispielsweise, wie oben bereits erwähnt und im Beispiel schon gezeigt, JAX-RS und CDI, für den tatsächlichen Anwendungszweck. Die APIs, die Jakarta EE standardisiert, müssen durch Bibliotheken implementiert werden. Bei Application-Servern und Frameworks trifft der Anbieter die Entscheidung, welche Bibliothek für welches API angeboten wird. Framework-freie Anwendungen lassen dem Entwickler (genauer gesagt, dem Deployer) die freie Auswahl. Was der Entwickler aber auf jeden Fall tun muss, ist, die Wahl zu treffen, welche APIs er einsetzen möchte. Jakarta EE ist eine sehr umfangreiche Sammlung an APIs. Eine Liste mit den einzelnen Bestandteilen zur Auswahl ist auf der Jakarta-EE-Website *[5]* unter dem Abschnitt „Individual Specifications" zu finden und erster Anlaufpunkt für diese Überlegung.

Hier wird ein weiterer Vorteil dieser Implementierungsweise deutlich: Was man nicht braucht, lässt man einfach bereits zur Entwicklungszeit komplett weg. Manche Frameworks sind hier teilweise unflexibel. Von Application-Servern gar nicht erst zu sprechen. Dort konnte man entweder gar nichts weglassen, nur teilweise (etwa durch Wahl von Payara Micro statt Payara Full) oder es nur abschalten – „da" war es trotzdem, und was „da" ist, kann kaputtgehen oder bietet Angriffsfläche oder wird versehentlich wieder eingeschaltet...

Natürlich würde man in der echten Welt mindestens zwei Dinge ändern wollen, selbst beim allertrivialsten Dienst: den genutzten TCP-Port und das TSL-Zertifikat. Dazu ist der Code minimal zu ändern, da das Beispielprogramm noch gar kein TLS verwendet. Natürlich kann auf Wunsch hierfür ein spezielles API genutzt werden, wie beispielsweise MicroProfile Config. Ob sich das lohnt, kommt auf das Projekt an. Auch hier erkennt man wieder das Grundmuster dieses Designs: Entscheidungsfreiheit. Wenn es nur darum geht, den Default-Port nicht mehr zu benutzen, kann JAX-RS auch automatisch einen freien Port suchen (dazu gibt es die Konstante FREE_PORT mit dem Wert 0). Typischerweise will man aber den Port manuell festlegen. Der Weg dazu geht am einfachsten über Umgebungsvariablen. Diese werden von Docker, Kubernetes oder anderen Ausführungsumgebungen „von außen" gesetzt und sind in Java trivial per Startargument abzufragen. Das TSL-Zertifikat zu setzen, geschieht prinzipiell über den gleichen Weg, wobei es aber von der Java-Anwendung selbst gar nicht benötigt wird. JSSE hat schließlich für so ziemlich alles wichtige Java-Properties, beispielsweise javax.net.ssl.keyStore für eine Zertifikatsdatei im PKCS-Format, sodass auch diese Eigenschaften „von außen" gesetzt und geändert werden können, ohne den Programmcode zu ändern – ein Reboot genügt, und dann ist der Verzicht auf den schwergewichtigen Unterbau in etwa einer Sekunde erledigt. *Listing 3* zeigt die benötigten Änderungen.

Der Start und die Konfiguration der Anwendung erfolgen mit Java-Bordmitteln, wobei die von Docker oder Kubernetes gesetzten Variablen in Properties gewandelt werden:

```
java -jar EchoServer.jar
       $PORT
       -Djavax.net.ssl.keyStore=$KEYFILE
       -Djavax.net.ssl.password=$KEYPASSWORD
       -Djavax.net.ssl.trustStore=$TRUSTFILE
```

Damit diese Codezeile funktioniert, muss natürlich erst einmal ein solches self-contained executable JAR („uber-jar") erzeugt werden, in unserem Fall die Datei EchoServer.jar. Hierzu kann beispielsweise das Maven-Shade-Plug-in dienen. Es erzeugt ein neues Jar, indem es den gesamten Inhalt aller im Klassenpfad vorkommenden JARs in dieses kopiert. Dies macht die Auslieferung der Anwendung sehr simpel, da sie nur noch aus einem einzigen JAR besteht. Dabei gibt es aber einige Fallstricke zu beachten.

```
public static void main(String[] arguments) throws InterruptedException,
                                                    NoSuchAlgorithmException {
  var port = Integer.parseInt(arguments[0]);
  var sslContext = SSLContext.currentThread();
  var configuration = Configuration.builder()
                  .protocol("HTTPS")
                  .port(port)
                  .sslContext(sslContext)
                  .build();
  SeBootstrap.start(EchoApplication.class, configuration);
  Thread.currentThread().join();
}
```

Listing 3: TLS wird auf einem von außen gesetzten Port betrieben

Beispielsweise nutzten einige Jakarta-EE-APIs, aber auch einige Bibliotheken, das Service-Loader-API *[6]* von Java SE (das ist übrigens extrem praktisch, da man damit eine eigene Plug-in-Architektur mit reinen Java-Bordmitteln bauen kann). Dies führt jedoch bekanntlich dazu, dass mehrere JARs gleichlautende Dateien im Klassenpfad liegen haben, wenn sie den gleichen Service implementieren (was ja der Sinn einer Plug-in-Architektur ist) – beispielsweise alle Jersey-Module. Somit würde das fertige uber-jar aber nur eine einzige solche Service-Registrierung enthalten, die anderen Jersey-Modulen würden dann fehlen. Um also wirklich alle benötigten Services in das uber-jar zu packen, muss dem Shader mitgeteilt werden, dass er sich entsprechend darum zu kümmern hat, die Service-Dateien entsprechend zu konsolidieren. Das erledigt der folgende POM-Eintrag: `<transformer implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer" />`

Ebenso werden sehr viele Klassen, die dank der Bibliotheken im Klassenpfad liegen, niemals verwendet – der übliche Nachteil von Bibliotheken: Sie sind meist sehr umfassend. Um ungenutzten Kram zur Laufzeit wegzulassen, somit die Startzeit der Anwendung und ihren Fußabdruck zu reduzieren, kann das Shade-Plug-in die Anwendung auf die wirklich notwendigen Klassen und Ressourcen reduzieren. Hierzu kommt die Option minimizeJar zum Einsatz. Sie auf true zu stellen, ist prinzipiell eine sehr gute Idee:
`<minimizeJar>true</minimizeJar>`

Leider gibt es aber auch hier wieder Hürden, die es zu umschiffen gilt: Nicht alle Beziehungen zwischen Klassen sind statisch ermittelbar. Wenn beispielsweise eine Klasse über den Wert einer String-Variable referenziert wird, ist sie für Optimierungstools schlicht nicht sichtbar und fehlt daher (inklusive allem, was sie wiederum selbst benötigt) im fertigen uber-jar. Hier ist teilweise Recherche und manuelle Konfiguration nötigt, um „unsichtbare" Abhängigkeiten explizit bekannt zu geben. Wie dies im Falle von Weld (CDI-Implementierung) sinngemäß (und stark vereinfacht) aussehen kann, ist in *Listing 4* zu erkennen.

Ein anderer Fallstrick ist, dass das Shading zwangsweise die Modulgrenzen des Java-Modulsystems sprengt, da ja nur noch ein JAR übrig bleibt. Wer das Modulsystem zwingend benötigt, sollte stattdessen alternative Optimierungsmöglichkeiten nutzen.

```
<configuration>
  <minimizeJar>true</minimizeJar>
  <filters>
     ...
     <filter>
        <artifact>*:weld-core-impl</artifact>
        <includes>
           <include>**/*</include>
        </includes>
     </filter>
     ...
  </filters>
</configuration>
```

Listing 4: Weld macht dem Klassenpfad-Optimierer einen Strich durch die Rechnung, daher ist manuelle Konfiguration notwendig. Komplettes Nicht-Optimieren ist die einfachste, aber auch am wenigsten effektive Lösungsstrategie

```
<plugin>
  <artifactId>maven-exec-plugin</artifactId>
  <configuration>
     <mainClass>EchoServer</mainClass>
  </configuration>
</plugin>
```

Listing 5: Konfiguration des Maven-Exec-Plug-ins

Alternativ kann man natürlich gerne die JARs einzeln in den Klassenpfad setzen, was in den meisten Fällen aber keine echten Vorteile mit sich bringt. Maven kann auch hier unterstützen, indem beispielsweise das Maven-Exec-Plug-in *[7]* zum Einsatz kommt, das einen Start per mvn exec:java erlaubt, um zumindest das manuelle Eintippen des (irgendwann sehr, sehr langen) Klassenpfads zu verhindern.

Im Rhythmus Anforderung, API, Produktauswahl, POM-Dependency wird nun Feature für Feature hinzugefügt, bis der technische Unterbau, der „Boilerplate"-Code, fertig ist – sofern es „fertig" überhaupt gibt, da mit jedem Änderungswunsch eine Technologie hinzukommen oder gerne auch mal wegfallen kann (etwa, wenn man auf JPA verzichtet, da man von SQL auf No-SQL umsteigt).

Aufgrund der modularen Struktur vor Jersey sind im Beispielprojekt übrigens faktisch mindestens drei Dependencies anzugeben; je nach gewünschten oder benötigten Jersey-Features können es sogar noch deutlich mehr werden. Zur Verkürzung des POM ist es sinnvoll, zunächst dessen BOM mit dem Scope „import" zu importieren. Das Projekt hat zu diesem Zeitpunkt bereits drei Importe, obwohl es kaum über den Sinngehalt von „Hallo Welt" hinausgeht. Und es werden in der Realität bald deutlich mehr: Soll das mittlerweile ubiquitäre JSON mithilfe von JSOB-B zum Einsatz kommen, muss der Klassenpfad wieder sowohl um ein API (JSON-B) als auch um eine Implementierung (beispielsweise Yasson) ergänzt werden und eine Bindung an Jersey zum Einsatz kommen. Soll CDI genutzt werden, wird auch dessen API und eine Implementierung (etwa Weld) hinzugefügt. Mittlerweile ist das Projekt dann bereits bei einem POM angelangt, das sich in diesem Heft nicht mehr auf einer Seite abdrucken lässt.

Der Nachteil dieses Framework-losen Ansatzes ist somit offensichtlich: Es ist viel Recherche und Know-how notwendig, um den Klassenpfad korrekt zu gestalten. Welche Jakarta APIs sind nötig? Welchen Versionsstand haben diese? Welche Implementierung ist für den vorliegen Anwendungsfall vorteilhaft? Möchte ich eine von Oracle gepflegte Implementierung oder lieber eine, bei der Red Hat einen Wartungsvertrag anbietet? Und die Fragen können sehr, sehr ins Detail gehen. Bei Jersey beispielsweise gibt es so viele Module, dass man nicht nur zwischen XML und JSON wählen kann (was sich in der Anwendung begründet), sondern auch beispielsweise Implementierungsdetails tauschen kann. In unserem Fall haben wir zum Beispiel auf den HTTP-Server Grizzly v2 gesetzt. Dieser ist extrem skalierbar, aber auch relativ umfangreich. Wenn die Zahl der Clients für unseren Dienst eher gering ist und wir lieber den Fußabdruck der Anwendung reduzieren möchten, ergibt eventuell Netty mehr Sinn. Durch reine Änderung des POM lässt sich dies bewerkstelligen, ohne die Java-Anwendung anzufassen.

Und genau hier liegt der unschlagbare Vorteil dieser No-Framework-Konstruktionsweise: Wir müssen unser Programm niemals ändern

```
java -XX:ArchiveClassesAtExit=EchoServer.jsa -jar EchoServer.jar
java -XX:SharedArchiveFile=EchoServer.jsa -jar EchoServer.jar
```

Listing 6: Dynamic AppCDS beschleunigt den zweiten Start der Anwendung ähnlich stark wie die Nutzung von native-image, benötigt aber keinerlei Vorarbeiten zur Build-Zeit.

– zumindest nicht, solange es nur um den Tausch des technischen Unterbaus geht. Keine Codeänderung. Kein Neukompilieren. Lediglich den Klassenpfad ändern und neu starten. Nach nur einer Minute läuft das Programm nun auf einem gänzlich anders optimierten HTTP-Server. Und das geht natürlich nicht nur mit den Modulen von Jersey, sondern eben auch mit sämtlichen Implementierungen von Jakarta-EE-Bestandteilen. Wer kein Eclipse Yasson will, sondern JSON-B lieber per Apache Johnzon nutzen möchte: Eine einzige Dependency tauschen, Neustart, fertig. Nach kurzer Zeit wechselt man einzelne Module oder zieht modulweise auf neue Versionen hoch, ohne jemals darüber nachzudenken, wie viele Monate ein Framework oder ein Application-Server braucht, um all seine Bestandteile zu aktualisieren und durch die QA zu schicken. Wir tun das nun quasi bei jedem Build-Lauf (dank Maven-Versions-Ranges) und deployen somit täglich die jeweils neuesten Bugfixes. Der Prozess wird erheblich agiler als mit Frameworks oder Application-Servern.

Aber ist die Anwendung nicht viel schlechter als bei einem Framework? Gegenfrage: Weshalb denn? Frameworks und Application-Server basieren zumeist sowieso auf exakt jenen Bestandteilen, die das „No-Framework-Design" ebenfalls einsetzt. Wer sich etwas anstrengt, bekommt dank AppCDS *[8]* und Maven-Shade-Plug-ins-Optimizer oder OpenJDKs „jlink" bereits auf der Hot Spot VM ähnlich kurze Startzeiten und kleine Fußabdrücke, wie anderswo per GraalVM's „native-image" *(siehe Listing 6)*. Man muss schon lange suchen, was da wirklich messbar schlechter sein soll. Aber wie gesagt: Es kommt immer auf die Anwendung und den Einsatzzweck an!

Fazit

Eine portabel geschriebene Anwendung, die sich rein auf bestimmte APIs aus dem Jakarta-EE-Reigen stützt, funktioniert sowohl auf Application-Servern, in Frameworks, als auch nativ auf Java SE. Sie hat einen schlanken Footprint und startet extrem schnell. Oftmals ist der letztgenannte Ansatz der einfachste und ressourcensparendste. Die Entscheidung für oder gegen dieses Design ist eine Einzelfallentscheidung des jeweiligen Projekts.

Referenzen:

[1] Erklärung des „Vendor Lock-in": *https://de.wikipedia.org/wiki/Lock-in-Effekt*

[2] Liste mit frei verfügbaren OpenJDK-Distributionen unterschiedlicher Anbieter: *https://adoptium-marketplace.netlify.app/*

[3] Spring 6 nutzt Jakarta EE 9: *https://spring.io/blog/2021/09/02/a-java-17-and-jakarta-ee-9-baseline-for-spring-framework-6*

[4] Definition: Zwölf-Faktoren-Anwendung: *https://12factor.net/de/*

[5] Liste der individuell nutzbaren APIs von Jakarta EE: *https://jakarta.ee/specifications/*

[6] Das Service-Loader-API ermöglicht die Implementierung von Plug-in-Architekturen in Java SE: *https://alvdavi.github.io/genshen/docs/api/java.base/java/util/ServiceLoader.html*

[7] Start einer Java-Anwendung per Maven-Exec-Plug-in, ohne den Klassenpfad manuell anzugeben: *https://www.mojohaus.org/exec-maven-plugin/java-mojo.html*

[8] Application Class Data Sharing ermöglicht extrem kurze Startzeiten der HotSpot JVM: *https://docs.oracle.com/en/java/javase/18/vm/class-data-sharing.html#GUID-2942983A-E83C-4DA3-A60C-60411D731D5A –*

Markus Karg

markus@headcrashing.eu

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

Just another Application with Microservices

Thomas Michael, GOD mbH

In diesem Artikel wird eine Anwendung vorgestellt, die aus mehreren Microservices besteht. Jeder Microservice basiert auf Java mit Spring Boot, orchestriert mit Docker Images über OpenShift, lose gekoppelt über REST und per Service Discovery mit Spring Cloud Netflix. Durch Spring Cloud Sleuth gibt es für jeden Request eine Microservice-übergreifende ID, die per Zipkin zusammengefasst wird.

Wem das zu langweilig ist oder wer alles schon kennt, der darf gerne zum nächsten Artikel weiterblättern. Wer jedoch bleibt und weiterliest, erfährt etwas über die Motivation für die Microservices-Architektur – Spoiler: ein Monolith – und darüber, wie man alles aufbaut und implementiert.

Motivation

Es existiert ein langjähriger Monolith, basierend auf mittlerweile Java 8, ausgeliefert als (sehr) großes War-File auf einem IBM Liberty Server. Die Schnittstelle zur Außenwelt, hier das Intranet, ist eine SOAP-Schnittstelle *[1]*. Es gibt Dutzende von verschiedenen Anwendungen/Clients/Servern, die die SOAP-Schnittstelle über ein bereitgestelltes Jar einbinden, wie in *Abbildung 1* angedeutet.

Der genaue Überblick darüber, wer alles ein Jar benutzt, ist leider schwer zu rekonstruieren, da es keine Kontrolle über die Verteilung der Jars zum Einbinden der SOAP-Schnittstelle gab und gibt. Dies verhindert auch das problemlose Entfernen alter Funktionen und Methoden. Die Datenbank ist sehr groß mit Dutzenden Schemata und Tabellen. Updates und neue Features gibt nur noch alle paar Monate, nach mehrwöchigen Tests und QS-Phasen.

Und jetzt löst eine Microservices-Architektur den Monolithen ab und alle sind glücklich, wie in *Abbildung 2* visualisiert!? Das mag in der Theorie und in Büchern einfach sein, aber in der Praxis ist dies meist ein größeres Projekt, das sich über Jahre hinzieht – und wir sind mitten dabei.

Theorie

Die Idee ist, dass für jeden uns bekannten Client aus den SOAP-Zugriffen REST-Zugriffe direkt auf die Datenbank werden. Client für Client, Call by Call! Es gibt einen zentralen Microservice, nachfolgend Controller genannt, mit einem möglichst generischen REST-API, um die Daten von der Datenbank bereitzustellen.

Für jeden Client, also jede Anwendung, wird ein eigener Microservice implementiert, der für diesen Client ein eigenes REST-API zur Verfügung stellt. Dieses Vorgehen kapselt die Anforderungen und Bedürfnisse jedes Clients gegenüber den anderen Clients untereinander. Dies ist ähnlich dem Backend-for-Frontend-Pattern *[2]*, weshalb die verschiedenen Clients nachfolgend Client-BFF oder auch nur BFF genannt werden. Dadurch ergibt sich der in *Abbildung 3* gezeigte Aufbau für die Microservices.

Der Datenbankzugriff der BFFs erfolgt ausschließlich über das generisch gehaltene REST-API des Controllers. Nicht selten sind mehrere Controller-Aufrufe für eine Anfrage des Clients notwendig, um Daten aggregiert zurückzugeben. Der Zugriff erfolgt zentralisiert über ein gemeinsames Gateway, das an das jeweilige BFF weiterleitet.

Ich bin ich und wer bist du? Kommunikation zwischen den Clients

Wie findet ein BFF den Controller? In unterschiedlichen Umgebungen? Was muss dazu implementiert werden? Ist das aufwendig? Wie sieht so ein Gateway aus?

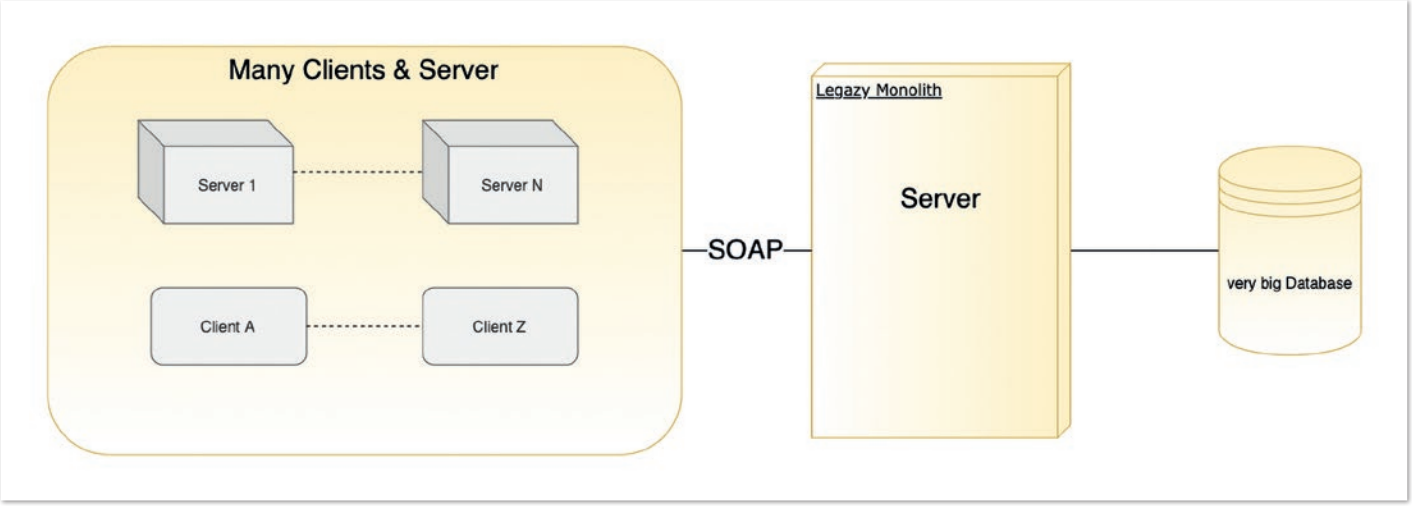

Abbildung 1: Motivation – Legacy-System (© [Thomas Michael, per draw.io erstellt])

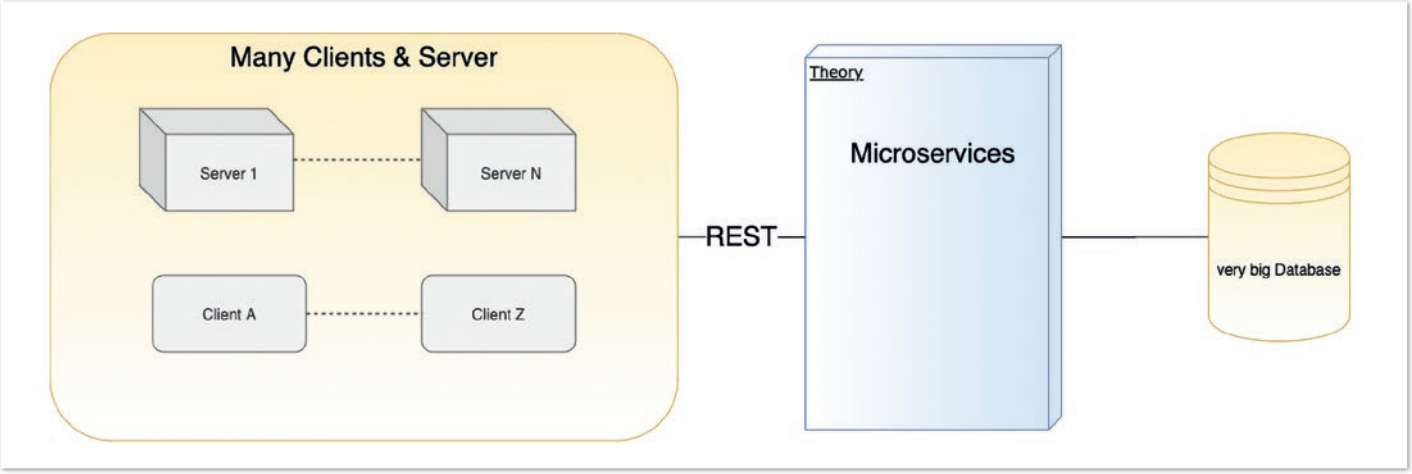

Abbildung 2: Microservices statt eines Servers (© [Thomas Michael, per draw.io erstellt])

Mit dem Spring-Cloud-Netflix-Framework *[3]* sind die Antworten sehr einfach und die Implementierung sehr kurz. Das Framework kümmert sich um die Kommunikation zwischen den Microservices. Dazu wird ein weiterer Microservice für die Namesauflösung (Service Discovery) benötigt, dies ist realisiert über einen Eureka-Server aus dem Spring-Cloud-Netflix-Framework. Ein Gateway für den gemeinsamen Einstieg lässt sich sehr einfach, mit dem Spring-Cloud-Gateway-Framework *[4]*, als eigener Microservice implementieren.

Praxis, die Konfiguration

Der Eureka-Server ist ein neuer Spring-Boot-Microservice und benötigt in der pom.xml den Eintrag aus *Listing 1*. Die `main`-Klasse zum Starten wird erweitert um `@EnableEurekaServer`, sodass sich *Listing 2* ergibt. Zum Schluss wird in den Properties *(siehe Listing 3)* ein Port gesetzt und dafür gesorgt, dass der Client sich nicht selbst bei sich registriert. Mehr Informationen dazu unter *[5]*.

Sobald der Eureka-Server lauffähig ist, müssen alle anderen Microservices zu Eureka-Clients konfiguriert werden. Auch dies ist wieder nur eine zusätzliche Annotation für den Eureka-Client *(siehe Listing 4)*, die Eureka-Client-Bibliothek dazu in der pom.xml hinzufügen *(siehe Listing 5)*, eine Anpassung in der Konfiguration, um den Server zu finden und den eigenen Namen festzulegen *(siehe Listing 6)*.

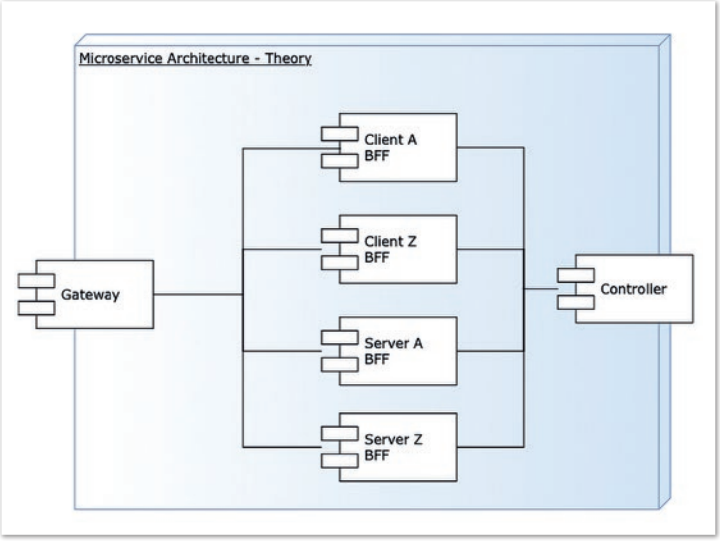

Abbildung 3: Aufbau der Microservices zueinander (© [Thomas Michael, per draw.io erstellt])

Das Gateway wird als eigener Eureka-Client implementiert. Von Spring gibt es dafür die Bibliothek spring-cloud-starter-gateway. In *Listing 7* ist der Eintrag für die pom.xml gezeigt. Dazu benötigt das Gateway das Routing zu den BFFs und kann dafür den Namen

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Listing 1: Pom-Konfiguration für Eureka-Server

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication
{
  public static void main(String[] args)
  {
     SpringApplication.run(EurekaServerApplication.class, args);
  }

}
```

Listing 2: Java-Konfiguration für den Eureka-Server

```
spring:
  application:
     name: my-eureka-server
server:
  port: ${SERVER_PORT:9092}
eureka:
  client:
     registerWithEureka: false
     fetchRegistry: false
```

Listing 3: Properties-Konfiguration für Eureka-Server

```
@SpringBootApplication
@EnableEurekaClient
public class ClientABff
{
  public static void main(String[] args)
  {
     SpringApplication.run(ClientABff.class, args);
  }
}
```

Listing 4: Java-Konfiguration für Eureka-Client

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Listing 5: Pom-Konfiguration für Eureka-Client

```
eureka.client.serviceUrl.defaultZone: ${EUREKA_URI:http://localhost:9092/eureka}
eureka.instance.hostname=clientA
```

Listing 6: Properties-Konfiguration für Eureka-Client

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Listing 7: Pom-Konfiguration für Gateway

nutzen, den die BFFs dem Eureka-Server mitgeteilt haben *(siehe Listing 8)*. In dem Beispiel werden zwei Routen für zwei verschiedene BFFs angelegt. Der Controller soll nicht direkt über das Gateway erreichbar sein, daher gibt es keine Routen-Konfiguration für den Controller.

Damit ergibt sich *Abbildung 4*: Jeder BBF ist ein Eureka-Client, auch der Controller und das Gateway.

Praxis, die Benutzung

Nachdem alles konfiguriert ist, wie sieht die Nutzung aus? Wie kann ein BFF den Controller aufrufen? Die Architektur in den BFFs ist jeweils die Gleiche wie in *Abbildung 5* dargestellt.

```
spring:
  application:
     name: my-api-gateway
  cloud:
     gateway:
        routes:
        - id: clientA
          uri: lb://clientA
          predicates:
          - Path=/clientA/**
        - id: clientB
          uri: lb://clientB
          predicates:
          - Path=/clientB/**
```

Listing 8: Routing vom Gateway zu den Clients

Der Controller beinhaltet das REST-API und leitet die Aufrufe an den Service weiter. Der Service führt nun 1-n Aufrufe gegen den Controller aus, um die gewünschten Daten zu holen und zu aggregieren. Dabei nutzt das BFF das WebFlux-Framework *[6]*, um die Daten asynchron zu holen. In *Listing 9* ist ein Beispielaufruf, der die Eingabe-Parameter, eine Root ID und eine Child ID an den Controller weiterreicht, dargestellt.

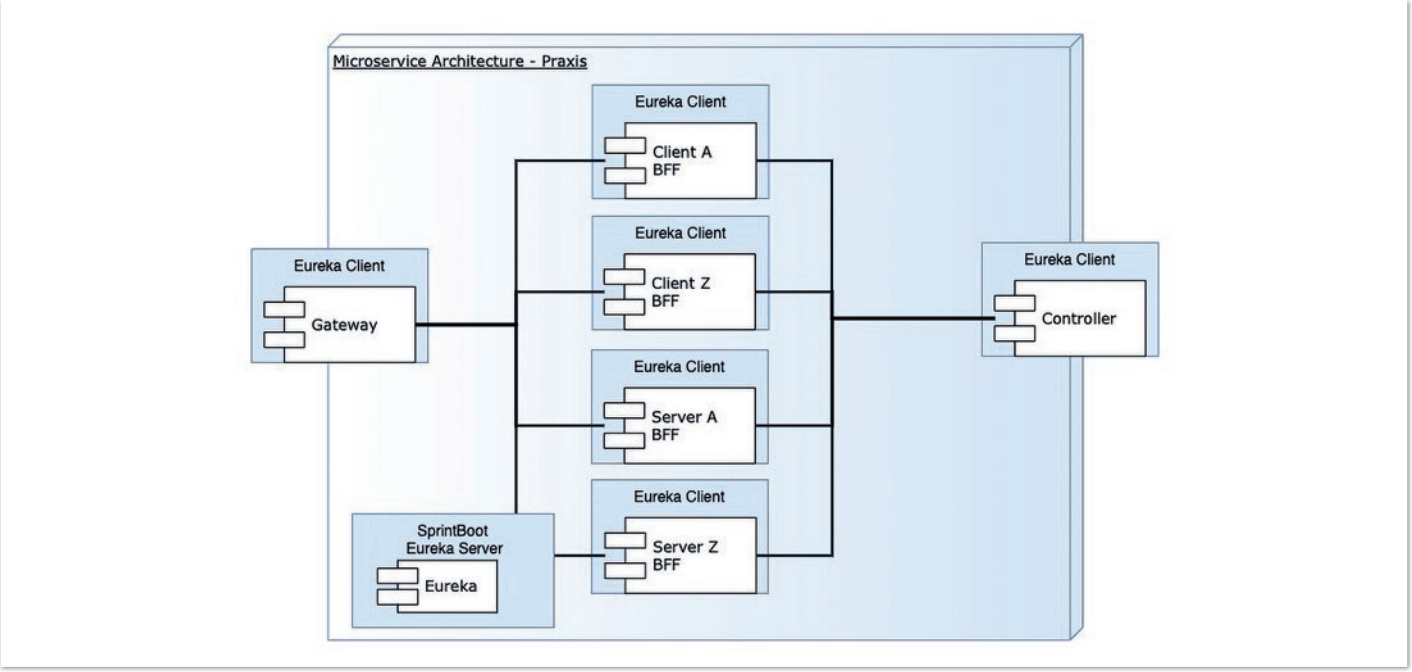

Abbildung 4: Microservices in der Praxis (© [Thomas Michael, per draw.io erstellt])

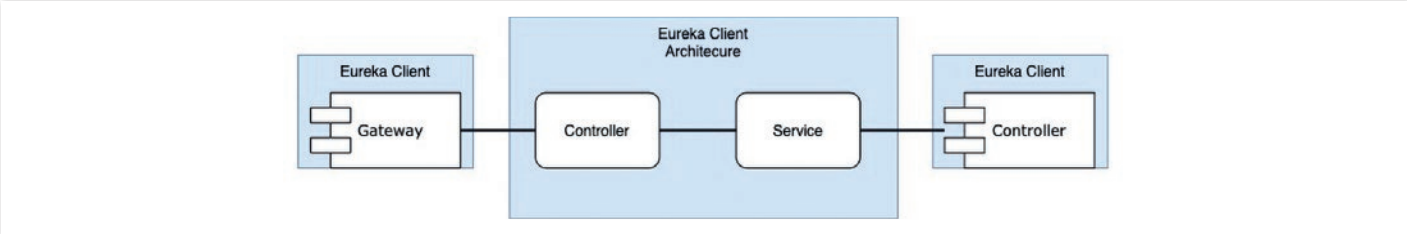

Abbildung 5: BFF von innen (© [Thomas Michael, per draw.io erstellt])

```
private static final String ROOT_WITH_CHILD = "api/v2/someRoot/{rootId}/child/{childid}";

  @Autowired
  private WebClient.Builder restWebClient;

  public Mono<RootWithChild> findRootByIdAndChildId(long rootId, long childId)
  {
     return restWebClient.build().get().uri(builder -> builder.path(ROOT_WITH_CHILD)//
        .build(rootId, childId))//
        .retrieve()//
        .bodyToMono(RootWithChild.class);
  }
```

Listing 9: BFF ruft den Controller auf

Die Magie verbirgt sich hinter dem `WebClient.Builder`. Über eine Konfiguration, die für alle BFFs gleich sein kann, wird der Controller bekannt gegeben.

Dies funktioniert, weil der Controller ein REST-API mit `api/v2/someRoot/{rootId}/child/{childid}` hat und der Controller dem BFF bekannt ist. In *Listing 10* ist eine Konfiguration des BFF für den Controller gezeigt. Eureka kennt den Controller als „Controller", wie in *Listing 11* gezeigt. Jeder BFF bekommt die Konfiguration aus *Listing 9* und erkennt so den Controller.

Wer hat was wann getan?

Wie bekommt man heraus, welches BFF welche Funktionen des Controllers aufgerufen hat, wie hängen die Aufrufe zusammen und alles mit möglichst wenig Aufwand?

Auch für die Antworten auf diese Fragen gibt es wieder ein Spring-Framework: Spring Cloud Sleuth *[7]* in Verbindung mit Zipkin *[8]* zur Visualisierung der Logs. Alle Microservices bekommen als Bibliothek Sleuth und Zipkin, wie in Listing *[12]* gezeigt. Beim Aufruf eines Microservice sorgt das Framework für eine einheitliche Span ID und Trace ID, sodass zusammenhängende Aufrufe sehr leicht identifiziert werden können. Zusätzlich kann per Docker ein Zipkin-Server gestartet werden *(siehe Listing 13)*, dann sieht man dort unter `http:localhost:9411` eine Oberfläche mit den zusammenhängenden Aufrufen.

Out of the box sind im Logging-File nun auch Span ID, Trace ID und der Applikationsname vorhanden. Wer sich das Logging-File selber konfigurieren möchte, kann dies über die Variablen `%X{traceId}` und `%X{spanId}` auch selbst gestalten. Sobald Sleuth dabei ist, sind die Variablen gültig.

Ab in die Cloud

Jeder Microservice wird in einem Docker-Container für die Cloud bereitstellt. Dazu reicht ein generischer Container, wie in *Listing 14* dargestellt, da jeder Microservice als Jar eigenständig gebaut wird und läuft.

```
@Configuration
public class WebClientConfiguration
{
  private static final String REST_CONTROLLER_URL = "http://controller/";

  @Bean
  @LoadBalanced
  public WebClient.Builder restWebClientBuilder()
  {
     return WebClient.builder().baseUrl(REST_CONTROLLER_URL);
  }
}
```

Listing 10: BFF ruft den Controller auf

```
eureka.instance.hostname=controller
```

Listing 11: Auszug aus der application.properties des Controllers

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
  </dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

Listing 12: Sleuth und Zipkin in den Poms

```
docker run -d -p 9411:9411 openzipkin/zipkin
```

Listing 13: Starten des Zipkin-Servers

```
FROM openjdk:8-jdk-alpine
RUN addgroup -S somename && adduser -S somename -G somename
USER somename: somename
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Listing 14: 08/15-Dockerfile für Jars

Mit diesem generischen Dockerfile können sehr einfach alle Microservices als Docker-Container gebaut *(siehe Listing 15)* und einer beliebigen Docker Registry *(siehe Listing 16)* bereitgestellt werden. Dies sollte im gleich Pfad wie das Dockerfile ausgeführt werden oder statt dem . am Ende den Pfad zum Dockerfile angeben. Das Pushen funktioniert nur nach vorherigem `docker login` an der entsprechenden Registry *[9]*.

Sobald die Docker-Container in der Registry verfügbar sind, können sie in der Cloud genutzt werden. Als Beispiel dient hier OKD *[10]*, die OpenShift Cloud von Red Hat, die auf ein Kubernetes Cluster aufsetzt. Damit die Docker-Container in OKD laufen und von außen gefunden werden, werden drei Dinge konfiguriert:

- Deployments
- Services
- Routes

```
docker build -t <name-des-services>:<version> .
```

Listing 15: Bauen eines Docker-Containers

```
docker push <name-des-services>:<version>
```

Listing 16: Hochladen des Containers in eine Registry

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: eureka
  namespace: some-namespace-in-okd
spec:
  selector:
     matchLabels:
        app: eureka
  replicas: 1
  template:
     metadata:
        labels:
           app: eureka
     spec:
        containers:
           - name: eureka
             image: <name-des-services>:<version>
             ports:
             - containerPort: 9092
```

Listing 17: Deployment-YAML für OKD.

```
apiVersion: v1
kind: Service
metadata:
  name: eureka-service
  namespace: some-namespace-in-okd
spec:
  selector:
     app: eureka
  ports:
     - protocol: TCP
       port: 80
       targetPort: 9092
```

Listing 18: Eureka-Service-YAML für OKD.

Mit einem Deployment wird der zu verwendende Docker-Container konfiguriert und es wird festgelegt, wie er heißen soll und viele Instanzen gestartet werden sollen, siehe *Listing 17* für unseren Eureka.

Zuerst wird der Eureka-Service deployt, da dieser dann allen anderen Services bekannt gemacht werden muss.

Sobald ein Deployment erfolgreich ist, ist ein Pod mit dem Docker-Container gestartet. Bei der Pods-Übersicht sieht man auch die ersten Fehler, falls etwa der Docker-Container nicht gefunden wurde. Ansonsten kommuniziert die Cloud mit dem Anwender über die Deployment-YAML, die um weitere Felder angereichert wurde. Unter anderem auch um Message-Felder mit Hinweisen auf Erfolg oder Misserfolg.

Für Eureka wird zusätzlich eine Route für den Zugriff und zum Verbinden der Clients benötigt und dies bedeutet, es wird auch ein Service benötigt. Der Service kann direkt über das UI angelegt werden mit folgender YAML-Datei, siehe *Listing 18*.

Den Erfolg sieht man sofort im UI auf dem Pods- Reiter. Falls dort nicht der Pod aus dem Deployment vorher erscheint, hat man einen Konfigurationsfehler.

Für die Route gibt es direkt ein UI, das ohne YAML auskommt, siehe *Abbildung 6*. Die Route bekommt einen Namen, man wählt den Service, der veröffentlicht werden soll, und den Port. Am Ende kann man die Route noch verschlüsseln – das lassen wir hier aus Gründen der Vereinfachung aber außen vor. Es wird jedoch dringend empfohlen, öffentlich nur verschlüsselte Routen zu erstellen!

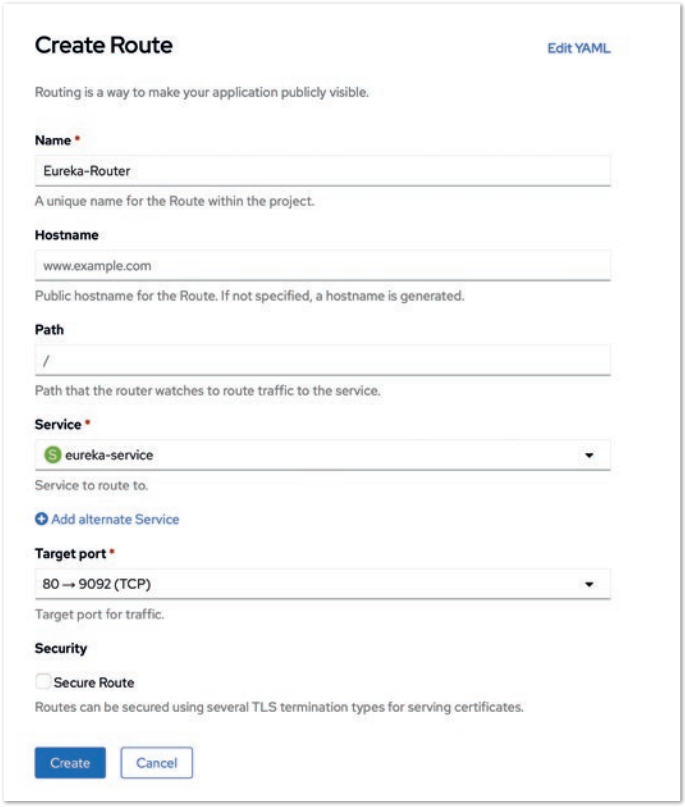

Abbildung 6: Konfiguration der Route für Eureka (© [Thomas Michael, Screenshot OKD])

```
containers:
  - name: clientA
     image: '<name-des-services>:<version>'
     ports:
         - containerPort: 8080
           protocol: TCP
     env:
         - name: EUREKA_URI
           value: 'http://eureka-service-some-namespace-in-okd/eureka'
         - name: spring.zipkin.baseUrl
           value: 'http://zipkin-server:9411'
```

Listing 19: Ausschnitt der Deployment-YAML für ClientA.

Mit der Route wird eine URL erstellt; diese wird in den anderen Services benötigt, damit sich jeder Microservice am Eureka registrieren kann. Nur so funktioniert die geplante Architektur! In *Listing 19* ist dargestellt, wie in einer Deployment-YAML die Parameter in den Docker-Container kommen. Es werden die URL für den Eureka-Server und die URL für den Zipkin-Server überschrieben.

Fazit

Mit geringem Aufwand lassen sich aus Anwendungen, die auf Spring Boot basieren, einfache Docker-Container erstellen. Diese lassen sich ihrerseits sehr einfach in einer Cloud deployen. Die Kommunikation zwischen den Containern ist mithilfe von Eureka und REST auch recht simpel gehalten.

Alles in allem helfen mir die genutzten Frameworks, mich wieder mehr auf die Kundenanforderungen zu konzentrieren und mich weniger um Infrastruktur etc. zu kümmern, weil es läuft!

Quellen

[1] *https://de.wikipedia.org/wiki/SOA*
[2] *https://microservices.io/patterns/apigateway.html*
[3] *https://spring.io/projects/spring-cloud-netflix*
[4] *https://spring.io/projects/spring-cloud-gateway*
[5] *https://spring.io/guides/gs/service-registration-and-discovery/*
[6] *https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html*
[7] *https://spring.io/projects/spring-cloud-sleuth*
[8] *https://zipkin.io*
[9] *https://docs.docker.com/registry/*
[10] *https://www.okd.io*

Thomas Michael
GOD mbH
Thomas.Michael@god.de

Thomas Michael ist Softwareentwickler aus Leidenschaft. Nach dem Informatikstudium widmete er sich ganz der Software-Entwicklung in Braunschweig und hat damit sein Hobby zum Beruf gemacht. Nach dem ersten Kontakt mit der Cloud lassen ihn die Möglichkeiten nicht mehr los. Als AWS Developer arbeitet er bei der GOD mbH in Braunschweig an der erfolgreichen Umsetzung von Projekten mit Cloud-Technologien für interne und externe Kunden. Privat arbeitet er an kleineren Spring-Boot- oder Angular-Anwendungen, die immer öfter durch Serverless-Lambda ersetzt werden.

JavaLand 2022 feiert erfolgreiches Comeback als Präsenzveranstaltung

Lisa Damerow, DOAG Dienstleistungen GmbH

Die Vorfreude war riesig, jedoch gleichzeitig auch die Sorge, dass die JavaLand 2022 doch noch in letzter Minute abgesagt werden könnte. Doch am Ende wurde alles gut: Die große Java-Community-Konferenz fand vom 15. bis 17. März 2022 nach zwei langen Jahren endlich wieder in Präsenz im Phantasialand Brühl unter der behördlich angeordneten 3G-Regelung statt.

Mehr als 1.200 Java-Fans pilgerten zum Freizeitpark, um gemeinsam zu lernen und zu netzwerken, was das Zeug hält. Die Sehnsucht war groß, aber es hat sich gelohnt! Es gab wieder reichlich zu entdecken und zu erleben: Mehr als 130 Speaker hielten über 130 Vorträge an zwei Konferenztagen. Am Donnerstag rundete ein Schulungstag die JavaLand ab. Dort konnten sich die die Teilnehmer/innen auf 9 spannende ganztägige Workshops freuen, in denen sie in kleinen Gruppen Neues lernen und bereits vorhandenes Wissen intensiv vertiefen konnten.

Auch auf das JavaLand-Studio, das Java-Fans schon in der Online-Edition 2021 lieben gelernt haben, mussten sie nicht verzichten. Live on Stage moderierte das Team, bestehend aus Hendrik Ebbers, Sandra Parsick, Falk Sippach, Melanie Andrisek und Alexander Neumann im Raum Wintergarten und sorgte für bestes Entertainment. Daheimgebliebene konnten das gesamte Programm im Raum Wintergarten, inklusive der Vorträge und des Studios, von Zuhause aus per Livestream mitverfolgen. Sowohl auf der JavaLand-Website als auch über heise online gab es die Möglichkeit dazu.

Ein Highlight für viele Besucher war zweifelsohne erneut die Abendveranstaltung „OpenPark" am Ende des ersten Konferenztags, an dem der Freizeitpark in seinen schönsten Farben erstrahlte. Neben köstlichen Speisen und Getränken standen den Teilnehmer/innen auch einige Fahrgeschäfte des Phantasialands exklusiv zur Verfügung. Trotz des frostigen Märzabends wurden diese intensiv genutzt!

Besonders die insgesamt 26 Aussteller freuten sich, Teilnehmer/innen endlich wieder vor Ort an ihren Ständen zum persönlichen Austausch begrüßen zu dürfen. Sowohl die großen Namen der Szene als auch aufsteigende Newcomer waren mit dabei und luden zu Gesprächen ein, verlosten Preise und verteilten Goodies an interessierte Besucher/innen.

Wer die JavaLand 2022 verpasst hat, aber dennoch nicht auf die Vorträge verzichten möchte, kann sich noch bis zum 28. Februar 2023 das JavaLand On-Demand-Ticket sichern und so bis zum 31. März 2023 Zugriff auf alle Vortragsaufzeichnungen und Unterlagen erhalten. Angemeldete Teilnehmer/innen, die vor Ort dabei waren, müssen kein zusätzliches Ticket erwerben; ihnen stehen die Aufzeichnungen und Folien ohnehin zur Verfügung. In unserem Web-Archiv gibt es zudem noch vieles mehr zu entdecken – unter anderem die Daily Conference News sowie Social-Media-Eindrücke.

Save the Date: Die JavaLand 2023 wird vom 21. bis 23. März 2023 stattfinden – hoffentlich wieder im Phantasialand! Jatumba – wir freuen uns auf euch.

DAS CLOUD NATIVE FESTIVAL

DAS EVENT DER DEUTSCHSPRACHIGEN CLOUD NATIVE COMMUNITY

on demand

DAS CLOUD NATIVE FESTIVAL VERPASST?

JETZT ON-DEMAND-TICKET BUCHEN UND VORTRAGSAUFZEICHNUNGEN ANSCHAUEN!

Alle Angebote im On-Demand-Ticket-Shop

Ein digitales Ticketing-System für die JavaLand 2022

Ronan Le Tiec, escape GmbH

Endlich fand die JavaLand 2022 wieder in gewohnter Form statt. Doch in puncto Eventmanagement-Tools gab es einige Veränderungen – vor allem unter der Haube mit neuen Software-Bestandteilen. Wir nehmen das digitale Ticketing-System unter die Lupe, das die Organisatoren pünktlich zur JavaLand 2022 in Betrieb genommen haben.

In den Vorjahren fanden die rund 2.500 Besucher der JavaLand ihre Konferenztickets im Normalfall in der Post. Ziel war es, die Tageskasse vor Ort zu entlasten. Größtenteils gelang es. Trotzdem verzeichnete die Kasse in der Regel zu Beginn der Veranstaltung ein hohes Aufkommen. In der Schlange vor den Toren des Phantasialands gesellten sich viele Personen zu den Spontanbesuchern ohne Tickets – Badge vergessen oder verloren, Ticket nicht (rechtzeitig) eingetroffen, Lieferanschrift nicht mehr gültig, Rechnung noch nicht beglichen, Ersatzbesucher für eine verhinderte Person – das waren einige der Gründe fürs Anstehen.

Vorher/nachher

Mit der Auflage 2022 verzichtete die seit 2014 jährlich stattfindende JavaLand erstmalig auf den Ausdruck und Versand von Plastik-Zutrittsausweisen im Kreditkartenformat. Stattdessen erfolgte der Einlass komplett digital: Ein paar Wochen vor Veranstaltung konnten die Besucher ihr individuelles Ticket ganz bequem in ihrem persönlichen Event-Cockpit als Bild oder PDF aufrufen beziehungsweise herunterladen.

Wer das Ticket als Namensschild nutzen wollte, konnte die PDF-Datei als A4 ausdrucken, dieses nach doppelter Faltung in eine passende Hülle einführen und an einem Schlüsselband mit Karabiner um den Hals tragen. Für die Zutrittskontrolle selbst reichte es durchaus aus, das PDF beziehungsweise das Bild des QR-Codes auf einem mobilen Gerät seiner Wahl vorzuzeigen.

An den Eingängen der Veranstaltungslocation erledigte das mit Handscannern gewappnete JavaLand-Personal den Rest. Bis zu 30 Personen pro Minute wurden zu Stoßzeiten eingeschleust.

Web-Applikation für die Zutrittskontrolle

Die für die Zutrittskontrolle entwickelte Web-Applikation besteht aus einer JavaScript-Datei von knapp 300 Zeilen und nutzt die jQuery-Library. Die Software kommt im Frontend mit einer einzigen Seite aus, die sich im Look-and-Feel des Portals shop.doag.org präsentiert. Sie gehört zum Portfolio von shop.doag.org und nutzt weitere existierende Module.

Die Seite ist denkbar einfach aufgebaut *(siehe Abbildung 1)*. Trotzdem oder ausgerechnet deswegen sind auf konzeptioneller Ebene viele Gedanken eingeflossen, die für eine reibungslose und abgestimmte Logistik vor Ort sorgen.

Die Organisatoren der JavaLand sahen unterschiedliche Scanning-Stationen vor, die hauptsächlich zwei Zwecke erfüllen sollen:

- Zum einen sollten mehrere Stationen an den Eingängen in unterschiedlichen Szenarien und mit unterschiedlichen Produkten die Gültigkeit der Tickets überprüfen.
- Zum anderen sollten weitere Stationen die Ausgabe von Giveaways und Getränke-Marken im Park übernehmen.

Für diese Aufgaben kommt die Applikation mit einer Seite aus.

- Im zentralen Content-Bereich befindet sich ein Input-Feld mit Return-Button. Parallel zum Betrieb mit QR-Scanner sollte auch als Fallbacklösung eine manuelle Eingabe über eine auf dem Ticket abgebildete, menschenlesbare ID möglich sein.
- Nach getätigter Abfrage erscheint die Antwort darunter.

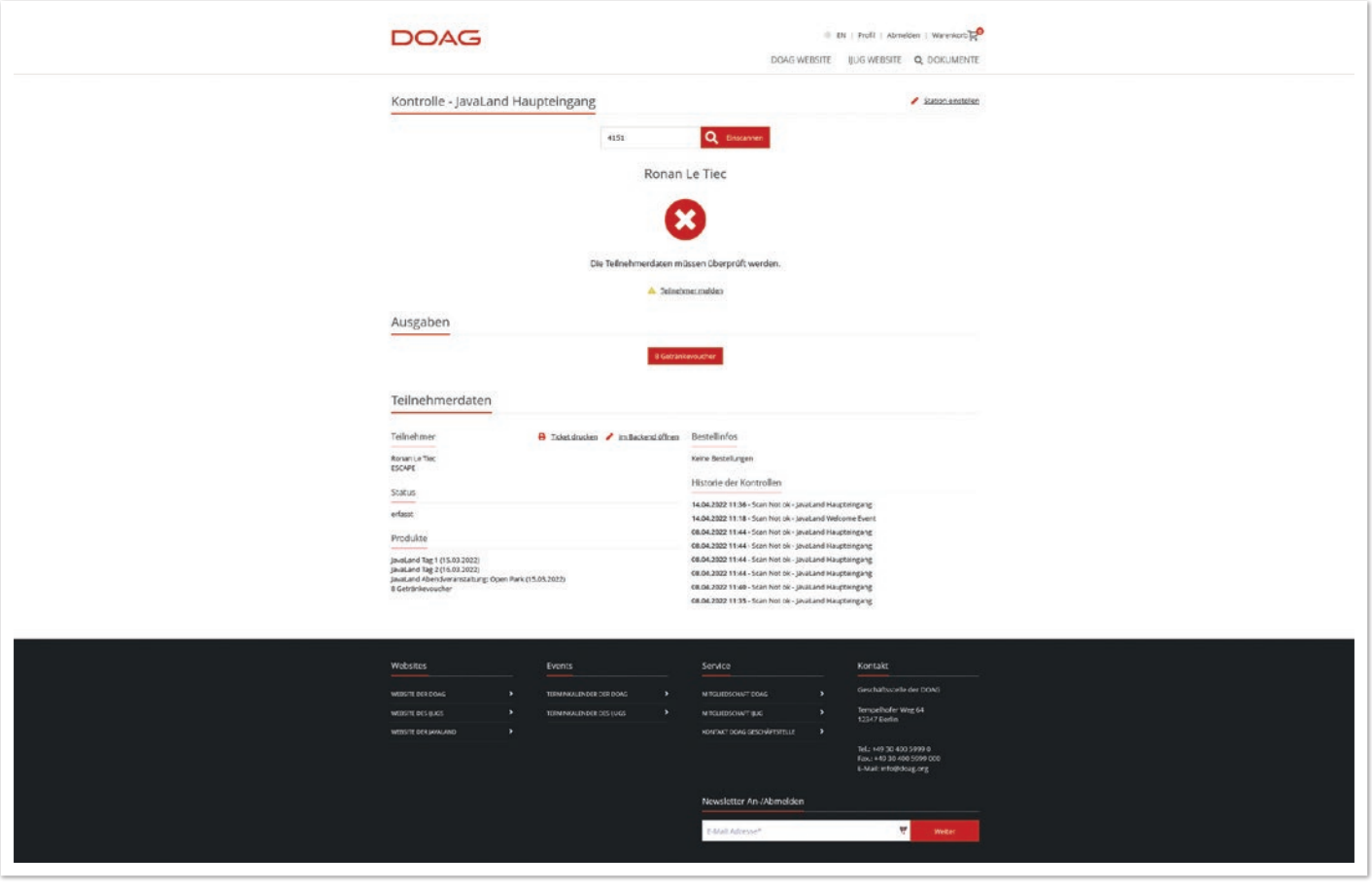

Abbildung 1: So sieht die Maske aus, mit der die JavaLand-Mitwirkenden an der Zutrittskontrolle arbeiten.

- Unter „Teilnehmerdaten" werden alle Informationen und Aktionen zusammengefasst, die für die Arbeit an den unterschiedlichen Stationen nötig sind.

Darüber hinaus sollten die von den Scan-Stationen vernommenen Aktivitäten aufgezeichnet werden. Hier sollten die Daten über folgende Aspekte Auskunft geben können:

- Verdacht des Ticketmissbrauchs aufgrund von erhöhten Scanning-Aktivitäten überwachen
- Aussagekräftige Zahlen über die Teilnehmerströme nahezu in Real-Time
  - Optimierung der Ressourcenplanung an den Scanning-Stationen
  - Mitteilung über Catering-Zahlen
  - Unterstützung der Referentenbetreuung in den Vortragsräumen

Einfache, intuitive Bedienung

Die rund 50 Helfer der JavaLand konnten dementsprechend mithilfe ihrer mitgeteilten Benutzerdaten die Web-Applikation aufrufen. Da an den Zutrittsstationen vorrangig ehrenamtliche Helfer unterstützten, die im Normalfall nicht mit internen Vorgängen vertraut sind, legte das Team bei der Konzeption großen Wert auf eine einfache und intuitive Bedienung: Die Scan-Stationen sollten ohne Vorkenntnisse benutzt werden können.

Im oberen Content-Bereich entschied man sich für ein naheliegendes Ampelsystem: Bei Grün wird Einlass gewährt, bei Rot wiederum „kein Zutritt" mit dem Hinweis „Bitte begeben Sie sich zur Tageskasse". Für Sonderfälle, die es auf Veranstaltungen immer gibt, wurde eine gelbe Ampel mit Kommentarfeld vorgesehen, das zur Kommunikation zwischen den unterschiedlichen Stationen und der Tageskasse genutzt werden konnte. Weiter unten blendet die Applikation diverse Detailinformationen in zwei gleichwertigen Spalten ein.

Performance & Aktualität

An den Kontrollstationen spielt der Faktor Zeit eine entscheidende Rolle: Viele Besucher kommen kurz vor Beginn der Veranstaltung an und möchten trotzdem pünktlich im ersten Vortrag sitzen. Hier ist ein Bottleneck vorprogrammiert.

Deswegen sollte die Applikation an den Scan-Stationen ein zügiges Tempo vorlegen. So legten wir bei der Entwicklung viel Wert auf Performance und effizient geschriebene Datenbank-Abfragen. Mit Fokus auf Aktualität und Schnelligkeit entschieden wir uns dazu, bestimmte Elemente der Seite wie Header und Footer zu cachen. Alle Informationen, die für die Zutrittskontrolle notwendig waren, sollte die Applikation wiederum frisch vom Server per AJAX-Call einholen: Informationen zum Teilnehmer, zur entsprechenden Buchung, zum Zahlungsstatus und den dazugehörenden Produkten, inklusive Leistungszeitraum.

Auf der anderen Seite stellte sich die Frage der Sicherheit. Hier erweiterten wir das bereits existierende „Rollen & Rechte"-Konzept um die entsprechende Rolle „Zutrittskontrolle" und sorgten für einen passwortgeschützten Zugriff auf die Seite.

Web-App aus den Hosentaschen bedienen

Nun müssen wir uns vorstellen, dass das Zutrittspersonal nicht unbedingt an einem bequem und schön eingerichteten Arbeitsplatz sitzt. Vielmehr arbeiten die Mitwirkenden an provisorisch aufgebauten Stehtischen, an denen Laptop und Scanner aufgestellt sind. Die Helfer stehen Mitte März in den frühen Morgenstunden in der Kälte. Die Zutrittskontrolle sollte auch mit Handschuhen oder aus der Wärme der Jackentasche aus funktionieren: Wir wählten Scanner, die im Tastatur-Wedge-Modus arbeiten können (Daten in virtuelle Tastatureingaben umwandeln) und über einen Dauermodus sowie einen Standfuß verfügten – die Applikation sollte diese Form der Arbeit unterstützen.

Triviale, aber wichtige Details für eine gute und effiziente Arbeit unter solchen Bedingungen:

- Die Benutzer sollten sich nicht darum bemühen müssen, das Input-Feld anzuklicken, bevor sie den QR-Scanner betätigen. Vielmehr sollte hier die Applikation den Datenfluss des Scanners abfangen und in das Input-Feld eintragen.
- Aus demselben Grund sollte bei einer Scanner-Nutzung kein manuelles Bestätigen nötig sein, um die Abfrage abzufeuern.

Ein gut funktionierender QR-Code ist ein Code mit einer guten Auflösung. Dafür sollte er so wenig Details und Informationen wie möglich abbilden. Wir entschieden uns, die Anfragen auf die Domain *https://qr.doag.org* auszulagern.

Wer bereits QR-Scanner in einer Web-App genutzt hat, weiß, dass es eine kleine Herausforderung darstellen kann: Der QR-Scanner findet im Tastatur-Wedge-Modus Anwendung. Betrachtet man unseren QR-Code als String, so schickt das Gerät jedes Zeichen als Tastatureingabe an den Computer. Das hat Vor- und Nachteile.

Ein Plus ist, dass man kein Plug-in oder sonstige Apps braucht, um den QR-Scanner zu verwenden. Man muss aber wiederum zwischen einer von Menschen generierten und einer QR-generierten Tastatureingabe differenzieren. Nach welchen Kriterien kann das erkannt werden? Ein wichtiger Faktor ist die Geschwindigkeit der Eingabe.

Wir haben zwei unterschiedliche Scanner ausgewählt – ein Modell aufgrund der handlichen Größe zum Mitnehmen, ein Modell für den Dauermodus und den Standfuß. Je nach QR-Scanner variiert die Pause zwischen zwei Zeichen-Eingaben zwischen 20 Millisekunden und 150 Millisekunden. Die Eingabegeschwindigkeit ist sogar je nach Modell einstellbar. Wenn ein Mensch es schafft, so schnell zu tippen, dann mit Sicherheit nur kurzfristig. Dementsprechend haben wir ein valides Kriterium für die Unterscheidung der Tastatur-Eingaben.

In unserer Web-Applikation fangen wir jede Tastatureingabe ab, ganz ungeachtet dessen, wo der Cursor-Fokus liegt. Dadurch muss der Benutzer nicht explizit ins Input-Feld klicken.

Bei jeder Eingabe tragen wir das entsprechende Zeichen in eine zentrale Variable ein. In den meisten Fällen wird das Standardverhalten nicht unterbunden. So merkt der Benutzer nicht, dass seine Eingabe abgefangen wird. Wir nutzen ein Time-out von 200 Millisekunden. Werden diese zwischen zwei Eingaben überschritten, beginnt die Analyse der Zeichenkette, die wir zwischengespeichert haben.

Nun zurück zu unseren QR-Codes: Diese haben eine wohlgeformte URL, mit der die Software arbeiten kann. Wenn ein Mensch indes etwas eintippt, dann ignoriert das die Applikation. Die Zeichenkette

//digitale zukunft gemeinsam gestalten

Bewirb dich jetzt!

escape-germany.de/career

Tekkie inside?

Für die Entwicklung unserer hauseigenen Software cellms® suchen wir nach Verstärkung. Bist du dabei?

e s c a p e – wir schaffen Lösungen

Werde Teil von escape und bringe dich je nach Vorlieben, Stärken und Fähigkeiten ein. Wir leben Digitalisierung und Software-Entwicklung ist unsere Leidenschaft. Wir programmieren unsere eigenen Tools und kommen beim Coden in den Flow. Ob Architektur, Technologie-Stack, Prozess oder User Experience – dein Input zählt. Wenn du gern anpackst, kannst du bei escape richtig etwas bewegen.

e s c a p e

wird in diesem Fall nicht als Befehl anerkannt *(siehe dazu Listing 1)*.

Flexibilität & Mobilität als Antwort zum Flaschenhals

Wir sprachen das Thema Flaschenhals bei der Zutrittskontrolle bereits an: Viele Personen müssen in kürzester Zeit eingeschleust werden. Mit performanten Abfragen und Caching kann die App alle zwei Sekunden einen Besucher scannen und im Idealfall bei positiver Rückmeldung durchwinken. Wenn jedoch einige voll besetzte Shuttle-Busse zeitgleich vor dem Eingang halten, kann eine performante Software nur bedingt weiterhelfen.

Um diese Besucherpeaks abzufangen, ist eine punktuelle Aufstockung des Personals nötig. Die App skalierte zwar, aber sie brauchte weitere Eigenschaften und Fähigkeiten, um eine adäquate Antwort zu bieten: Flexibilität und Mobilität waren hier die Stichwörter.

Ein Crew-Mitglied sollte von einer Station mit wenig Zulauf oder einer Aufgabe mit niedriger Priorität abgezogen werden können, um an einer Kontrollstation punktuell zu unterstützen. Deswegen waren neben den stationären Geräten auch mobile Endgeräte eingeplant – ganz nach dem verbreiteten Prinzip des BYOD („Bring your own Device").

Aus diesem Grund ist die Web-Applikation in der Desktop-Ansicht sehr schlicht gestaltet: Hier verzichteten wir bewusst auf eine aufwendige Gestaltung und wählten einen Mobile-first-Ansatz.

Konfiguration der Stationen per QR-Code

Genauso einfach sollte auch die Konfiguration der Stationen für das mobile Personal sein. Hat ein Benutzer die Rolle „Zutrittskontrolle" inne, ist er in der Lage, seine zugeteilte Scan-Station in den App-Einstellungen über eine Dropdown-Liste vorzunehmen. Aber es sollte noch einfacher sein: An den Orten, an denen gescannt werden sollte, fanden die Nutzer einen Aushang mit QR-Code. Nach Anmeldung auf die Website mit der entsprechenden Rolle konnten sie den QR-Code nach dem gleichen Prinzip wie bei den Tickets einscannen. Werfen wir wieder einen Blick in den Code *(siehe Listing 2)*. Wir waren bei der wohlgeformten URL stehengeblieben, die unsere QR-Codes generieren. Ein direkter Aufruf der URL kam bei unseren Überlegungen nicht infrage. Er hätte ein Neuladen der Seite mit sich gezogen. Aus Performance-Gründen galt es, andere Wege zu finden.

Deswegen erfolgt die Analyse der QR-Code-URL clientseitig. Ein regulärer Ausdruck übernimmt die Analyse der URL. Eine Weiterverarbeitung erfolgt nur unter der Bedingung, dass diese wohlgeformt ist. Ist dies der Fall, so extrahiert die Software folgende Informationen:

- Typ der Anfrage: Handelt es sich um eine Scan-Station oder um einen Teilnehmer?
- Wie lautet die passende ID?

Erkennt die Applikation eine Teilnehmerabfrage, so ruft sie die zentrale Funktion zur Analyse der Teilnehmerinformationen auf. Dabei trägt sie die Teilnehmer-ID in das Input-Feld ein. Per AJAX werden alle Informationen nachgeladen.

Wird eine Scan-Station erkannt, ruft die Software die zentrale Funktion zur Einstellung der Station auf. Parallel dazu wird die Station-Aktualisierung in einem Cookie festgeschrieben, sodass sich der Browser die Station-Auswahl auch nach einer Arbeitsunterbrechung merkt.

Auch hier achtete das Team penibel auf eine für den Nutzer einfache Bedienung: Die Stationen kann nur der Organisator im Backend einrichten. Dort werden auch die entsprechenden QR-Codes generiert und heruntergeladen. Die zahlreichen Helfer kriegen diese Arbeitsschritte nicht mit und können sich auf ihre Aufgabe konzentrieren.

```
this.initBarcodeScanner = function()
{
  self.scannerInput = "";
  $(document).on('keydown', self.handleKeyPress);
}

this.handleKeyPress = function(event)
{
  // clear the time out and start it again
  clearTimeout(self.inputKeyTimeout);
  self.inputKeyTimeout = setTimeout(self.analyzeKeyInput, 200);

  // we save the key pressed in a central buffer if it's a
  // single key (and not for instance 'Enter')
  event = event.originalEvent;
  if(event.key.length == 1)
  {
     self.scannerInput += event.key;
  }

  var _focused = $(document.activeElement);
  var _inputting = _focused.get(0).tagName.toLowerCase()==="textarea" || _focused.get(0).tagName.
toLowerCase()==="input";

  // we prevent some default behaviour in certain conditions.
  if (!_inputting && (self.scannerInput.length > 2) || _inputting && event.key == "Enter")
  {
     event.preventDefault();
     return;
  }
};
```

Listing 1: So analysiert die Applikation die Tastatureingaben

```
this.analyzeKeyInput = function()
{
  let regex = /^http[s]?:\/\/qr\.[^\/]+\/(p|cs)([0-9]+)$/g;

  // do we have a qr-URL?
  if (self.scannerInput.match(regex))
  {
     // extract the request type and id
     var result = regex.exec(self.scannerInput);
     var type = result[1];
     var id = result[2];

     //reset the participant input
     self.element.find('[name=participantIdSelection]').val("");

     // trigger the update of the participant infos or station
     if(type == "p")
     {
        self.updateParticipant(id);
     }
     else if(type == "cs")
     {
        self.updateStation(id);
     }
  }
  self.scannerInput = "";
}
```

Listing 2: Die Analyse unserer URL anhand eines regulären Ausdrucks

Ausgabestationen

Nach dem gleichen Prinzip funktioniert die Ausgabe von Give-aways. Hierfür erstellen und konfigurieren die Organisatoren Produkte und Unterprodukte im Backend und weisen diese den bestimmten Teilnehmern zu. Die Mitarbeiter der Ausgabestationen sehen für jedes definierte Produkt einen Button, der entweder rot oder ausgegraut ist, je nachdem, ob ein Anspruch besteht und/oder die Ausgabe stattgefunden hat.

Teilnehmer scannt sein Badge selbst – na und?

Eine kleine, aber wichtige Anekdote zum Schluss: Heutzutage hat jeder einen mobilen Scanner in der Hosen- oder Handtasche. Deswegen stellte sich folgende Frage: Wie gehen wir mit Scan-Vorgängen um, die nicht im Kontext unserer Web-Applikation erfolgen?

Hier ist die Antwort: Ein Teilnehmer scannt sein eigenes Ticket mit dem Smartphone. Die Scanner-App öffnet den Browser und die QR-URL wird geladen. Die URL hat zu diesem Zeitpunkt folgende Form: *https://qr.doag.org/p123*

Im ersten Schritt erfolgt eine Weiterleitung zur Domain unserer Kontroll-App. Wir befinden uns nun serverseitig im Kontext der Webseite, immer noch im QR-Request-Handling. Unsere URL sieht dann folgendermaßen aus: *https://shop.doag.org/qr/p123*

Der Server prüft anschließend den Login-Zustand des Nutzers. Ist dieser nicht eingeloggt, leitet die Teilnehmerabfrage zur entsprechenden Event-Seite weiter. Ein Teilnehmer erhält also über das Scannen des eigenen Badges Informationen zur Veranstaltung.

Ist der Nutzer eingeloggt, dann prüft die Applikation seine Rechte. Wenn er die Zutrittskontrolle einsehen darf, wird er zu der Seite weitergeleitet. In diesem Schritt wird die URL um zusätzliche Hash-Parameter ergänzt. So erfährt unsere Web-Applikation, welche Teilnehmer-Daten aufgerufen oder welche Station eingestellt werden soll. Hier ein Beispiel dafür: *https://shop.doag.org/urlzutrittskontrolle/#participantId.123*

```
1. https://qr.doag.org/p123
2. https://shop.doag.org/qr/p123
3. https://shop.doag.org/urlzutrittskontrolle/#partici
pantId.123
```

Listing 3: Die Verwandlung einer URL

Fazit

Manchmal ist die größte Herausforderung in einem Projekt, die Dinge einfach zu gestalten. Diese kleine Applikation ist ein Paradebeispiel dafür. Ein gut lesbarer, wiederverwendbarer Code ist wichtig. Genauso wichtig ist es als Entwickler, gute Kenntnisse über den Kontext zu haben, in dem die Applikation angewandt werden soll. In unserem Fall hoffen wir, dass wir zur Customer Experience der JavaLand beitragen konnten.

Ronan Le Tiec
escape GmbH
ronan.letiec@escape-germany.de

Ronan ist Senior Full-Stack Web Developer und arbeitet seit zwölf Jahren bei der Digitalagentur escape. Er war an der Entwicklung des firmeneigenen Werkzeugkastens beteiligt. Der gebürtige Franzose hat eine kleine Schwäche für Gestaltung, worüber Backend-affine Kollegen sehr glücklich sind. Sein Schwerpunkt liegt inzwischen in der agilen Anforderungsanalyse und -Planung in Kundenprojekten. Ronan hat in Frankreich, Deutschland und Spanien Computer Sciences Engineering studiert und fühlt sich sowohl beim Coden als auch in sozialen Interaktionen in mehreren Sprachen wohl.

Obsidian und Feenstaub

Uwe Sauerbrei, Sauerbrei IT-Consult

Entwickler sind stets auf der Suche nach mehr Wissen und einer Lösung, wohin damit, um es immer verfügbar zu haben. Manchmal sind sie so verzweifelt, dass sie eine fantasievolle Inspiration benötigen. Hier wird gezeigt, dass auch kommerzielle Feen ein großes Herz für Open Source besitzen.

Es war einmal ein einfacher Softwareentwickler, der in seinem Leben schon eine Reihe von Projekten mehr oder minder erfolgreich bewältigt hatte. Eigentlich war er mit seinem Leben durchaus zufrieden. Doch so, wie er wusste, dass er Zeit seiner beruflichen Laufbahn nie ausgelernt haben würde, war ihm auch klar, dass es stets viel mehr an interessanten Dingen zu entdecken gab, als die 24 Stunden eines Tages erlaubten. So stapelten sich auf Tischen und Regalen Bücher und Zeitschriften, die mit vielen bunten Stickern beklebt und markiert waren.

Der Entwickler nahm die neueste Ausgabe einer Zeitschrift, die er heute im Briefkasten vorgefunden hatte, in die Hand, als ein kleiner Flyer herausfiel. Es war einer dieser Hochglanzprospekte, die wohl mit Kursen und noch mehr Büchern und Zeitschriften warben. So dachte er zumindest, als ihm die ungewöhnliche Überschrift auffiel: *„IT-Fee, zu Ihren Diensten. Ihr Wunsch ist unser Business!"*

Darunter das Bild einer kleinen Fee, die, gewappnet mit einem Terminal in der einen und einem Zauberstab in der anderen Hand, kess in die Kamera blinzelte.

„Interessanter Marketing-Gag", murmelte der Entwickler, als er ein sanftes Klopfen auf der Schulter verspürte. Erschrocken zuckte er zusammen, drehte sich um und erblickte die Fee aus dem Prospekt, wie sie, über das kleine Gesicht strahlend, vor ihm auf und ab flog. „Du kommst aus Hamburg, ich sag mal Moin", trällerte sie fröhlich. „Ich bin Bitsi und kann Dir helfen!" Noch bevor er überhaupt reagieren und berechtigterweise darauf hinweisen konnte, dass es Feen nicht gab, sprach sie bereits weiter. „Ich weiß, was Du sagen willst und Zeit ist Geld, also lass uns den Teil mit ‚das kann es nicht geben', ‚ich muss wohl träumen', ‚ist das versteckte Kamera' gleich überspringen und zum Thema kommen."

Der Entwickler fühlte sich etwas überrumpelt, kam aber zu dem Schluss, dass es nicht schaden könnte, erst einmal mitzuspielen. „Du kannst mir helfen? Wobei?", fragte er vorsichtig. „Ganz einfach", sie drehte eine Pirouette im Raum und versprühte dabei eine Art grünen Staub, der sehr verdächtig nach nicht druckbarem Zeichensalat aussah. „Du möchtest all deine Notizen und Aufzeichnungen an einem zentralen Platz speichern, ohne sie in gedankliche Schubladen zu stecken, und sie intuitiv finden, wenn du sie brauchst. Dazu soll es noch einfach zu bedienen und zukunftssicher speicherbar sein."

Der Entwickler war erstaunt ob der Tatsache, dass er sich tatsächlich oft mit diesem Problem auseinandergesetzt und einen langen Weg hinter sich gebracht hatte, der irgendwann mit Evernote *[1]* begann und über OneNote *[2]*, BoostNote *[3]* bei seinem derzeitigen Tool Joplin *[4]* endete. „Ich möchte Dir heute Obsidian *[5]* vorstellen und nein", sie zeigte mit ihrem Zauberstab auf seine Nase, „es geht nicht um Gesteinsglas! *[6]*" Sie flog eine kleine Runde durch das Arbeitszimmer und nahm auf seiner Schulter Platz. „Mein Job ist der Vertrieb, wir schauen es uns einfach mal zusammen an!"

Es war eine bizarre Situation, aber in der Software-Branche ist man einiges gewohnt und so harrte er interessiert der Dinge, die da jetzt

Abbildung 1: Auswahl beim Start von Obsidian

Abbildung 2: Struktur der Datenfiles

kommen mochten. Zurückgelehnt in seinen ergonomisch perfekt austarierten (und sehr teuren) Bürostuhl konnte er beobachten, wie sich vor seinen Augen eine virtuelle Projektion entwickelte und eine Stimme aus dem Off zu erzählen begann.

Obsidian wurde im Jahr 2020 aus der Taufe gehoben und von Beginn an unter drei Gesichtspunkten als Eckpfeiler entwickelt.

Lokale Speicherung in einem einfachen Format

Öffnet man die Anwendung, wird man aufgefordert, einen neuen Vault zu erstellen oder einen existierenden Ordner zu öffnen *(siehe Abbildung 1)*. Vault ist hierbei tatsächlich nur ein Synonym für einen Ordner im Filesystem.

Alle Files, Ordner, Bilder und sonstige Artefakte werden direkt im Filesystem abgelegt und entsprechen 1:1 der angezeigten Struktur in Obsidian. Die Konfigurationsdateien liegen in einem versteckten Unterordner und werden dort im JSON-Format gespeichert *(siehe Abbildung 2)*.

Das Herz der Anwendungen sind jedoch die Notes, die als Markdown abgelegt werden. Über einen Shortcut oder die Maus können neue Notes an- und abgelegt werden. Je nach persönlicher Vorliebe in Ordnern, ungeordnet oder in einem Mix aus beidem. Für die Bearbeitung gibt es einen Editor, eine Live-Preview und natürlich eine Ansichtsview. Da es sich um pures Markdown-Format handelt, kann jede Datei auch mit einem beliebigen Texteditor bearbeitet werden. Daneben ist es auch jederzeit möglich, alle Notizen mit einem anderen Tool zu visualisieren, das Markdown versteht.

Kleiner Markdown Primer

Abbildung 3 zeigt natürlich nur einen kleinen Ausschnitt der Möglichkeiten, die die Formatierungssyntax bietet. Sie können gerne

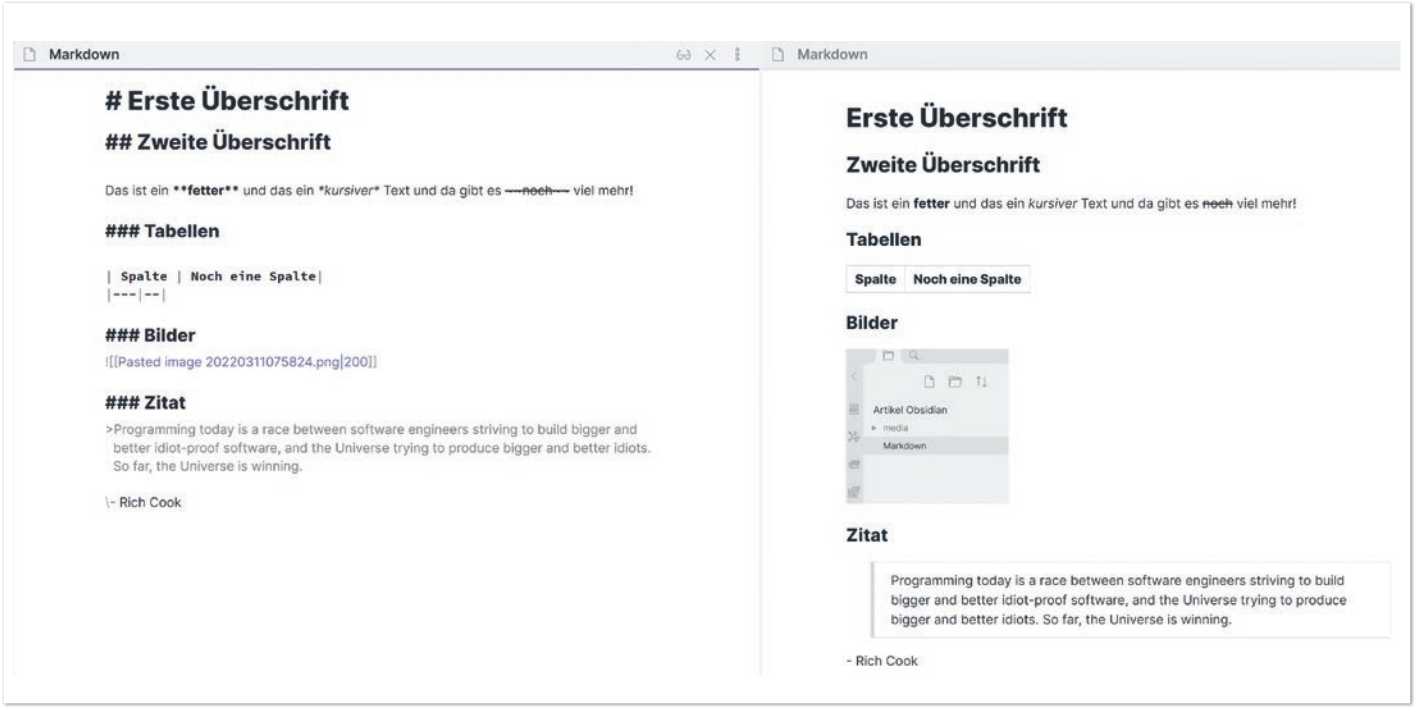

Abbildung 3: Beispiele für Markdown-Syntax

unseren international ausgezeichneten Markdown-Kurs (noch diese Woche mit 33 % Rabatt – Bezahlung in Fairy-Coins) buchen. Unsere Fee wird Ihnen gerne behilflich sein. Oder Sie besuchen die Hilfeseiten *[7]* von Obsidian.

„Wie synchronisiere ich die Daten auf einen anderen Rechner?", unterbrach der Entwickler den Vortrag. Die Stimme aus dem Off verstummte und es hörte sich an, als atme sie tief durch. „Das ist eine aufgezeichnete Präsentation und Unterbrechungen sind nicht vorgesehen. Aber da Sie ein Neukunde sind, mache ich mal eine Ausnahme!" Die Stimme fuhr fort.

Der lokale Zugriff wirft natürlich sofort die Frage nach einer Synchronisierung der Notizen auf. Hier bietet es sich an, einen Dienst zu nutzen, der Verzeichnisse automatisch synchronisiert. Als Beispiel kann hier HiDrive *[8]* angeführt werden. Auch ein Service, der Verzeichnisse im Hintergrund aktualisiert, ist möglich. Obsidian bietet, neben Installationen für Windows, Linux und macOS, auch Apps für Android und iOS an. Wer sich die eigene Synchronisierung ersparen will, kann auf einen kostenpflichtigen Service zurückgreifen, der die Daten via Clouddienst aktualisiert. Dabei werden sowohl der gesamte Datenverkehr als auch alle Notizen verschlüsselt, wobei der Key im Besitz des Anwenders bleibt.

„Deine Notizen gehören nur dir!"

Eine Outline *(siehe Abbildung 4)* hilft bei der Orientierung (wie man auf dem Screenshot sehen kann, wurde dieser Artikel mit Obsidian geschrieben) und jeder Eintrag kann sowohl als Favorit markiert oder an beliebigen Positionen mit Tags versehen werden.

Besondere Bedeutung von Links

Viele der etablierten Notizverwaltungssysteme erfordern, dass man sich beim Erstellen einer Notiz bereits festlegt, wo sie gespeichert wird. Häufig genug stellt man im Nachhinein fest, dass die Aufzeichnungen auch an einem anderen Platz eine sehr gute Figur machen würden. Hier kommen die Links ins Spiel.

Nehmen wir obiges Beispiel *(siehe Abbildung 5)*. Auf einer Seite mit Zitaten von Albert Einstein soll ein Zitat in eine andere Notiz, die passende Zitate über Menschen sammelt, eingebettet werden. Dazu kann mit der Syntax: `![[Link Ziel^kryptischer code]]` eine Referenz auf den anderen Textblock erzeugt werden. Obsidian unterstützt das mit einem Menü, das, nachdem man die Klammern eingegeben hat, aufpoppt *(siehe Abbildung 6)*. So wächst mit der Anzahl von Notizen, Referenzen, Bildern auch die logische (natürliche) Verbindung zwischen allen Artefakten. Angelehnt ist das alles an die Funktionsweise des Gehirns, was wohl auch der Grund dafür ist,

Abbildung 4: Darstellung der Outline

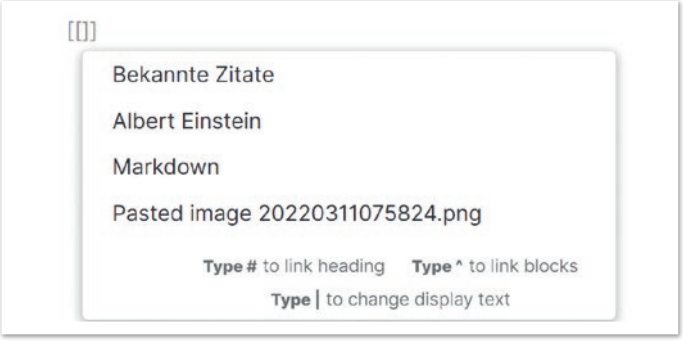

Abbildung 6: Erstellen von (Deep-) Links

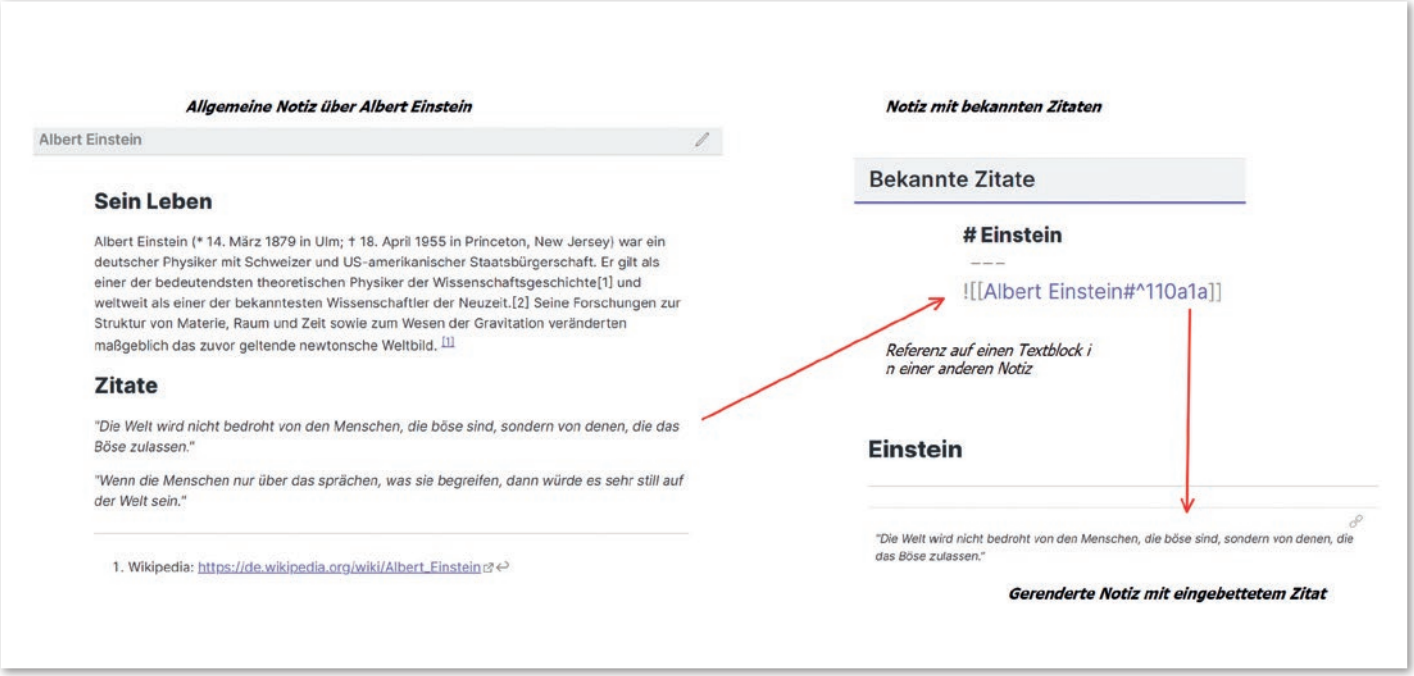

Abbildung 5: Nutzung von Links

Abbildung 7: Graphische Darstellung von Links

dass die Software oft auch als *Second Brain* bezeichnet wird. Um hier auch mal ein Buzzword in den Raum zu werfen: Links sind „first-class citizens"!

Abbildung 7 zeigt, wie so eine Darstellung aussehen kann. Die linke Note ist eine sogenannte moc (map of content), die als fachliche oder logische Klammer genutzt werden kann, um gleichartige Notes zu verbinden. Auf der rechten Seite sieht man den gesamten Graphen inklusive der Richtungspfeile. Daneben gibt es tabellarische Übersichten zu Links, Backlinks und Notes, die noch keine Verbindung haben.

Die graphische Ansicht kann detailliert angepasst werden *(siehe Abbildung 8)* und wenn es noch immer nicht reicht und genügend CSS-Kenntnisse vorhanden sind, gibt es keine Grenzen mehr. Generell erlaubt die Anwendung, die fast komplett mit TypeScript geschrieben ist und auf dem Electron-Framework aufbaut, beliebige Anpassungen, die als Skripte abgelegt und verwendet werden können.

Im Laufe der Zeit entstehen so Netzstrukturen, die gefiltert, markiert und dynamisch angepasst werden können. Im Internet gibt es dazu eindrucksvolle Bilder, bei deren Anblick tatsächlich der Eindruck entsteht, es mit einem neuronalen Netz zu tun zu haben. Es kann ein lokaler Graph erzeugt werden, der nur die Verbindungen der aktuellen Notiz anzeigt oder eine graphische Ansicht des gesamten Vault.

Offen für Erweiterungen

Kommen wir mal zu einem weiteren Eckpfeiler des Erfolgs von Obsidian. Wie kann man ein Projekt, das von zwei Entwicklern initiiert wurde, sehr schnell sehr erfolgreich machen? Ganz einfach, man bindet ganz viele andere Entwickler ein. Was bedeutet das konkret?

- offene Schnittstellen für Plug-ins, Erweiterungen und Anpassungen (es existieren derzeit rund 500 Community Plug-ins)
- freie Belegung von Keyboard-Shortcuts (zum Beispiel Vim-Mode)
- Import von Themes, respektive eigener Gestaltung (siehe Kenntnisse zu CSS)

Die Möglichkeiten zur Anpassung der Anwendung sind fast unendlich (also nahe dran, *siehe Abbildung 9*) und natürlich kann man bei den meisten Skins auch einen Dark-Mode auswählen.

Bei den Plug-ins wird zwischen Core- und Community-Plug-ins entschieden. Ein Teil der Ersteren ist bereits aktiviert, Letztere sind initial komplett deaktiviert (Safe-Mode). Vermutlich wird man sich auf Dauer dem Berg an zusätzlicher Funktionalität der (extrem) kreativen Community nicht entziehen können und durch den Schatz verfügbarer Funktionalitäten browsen. Allein die Aufzählung der beliebtesten Plug-ins würde ein ganzes Kapitel füllen. Um mal einen kleinen Ausblick zu geben, hier eine kurze Liste:

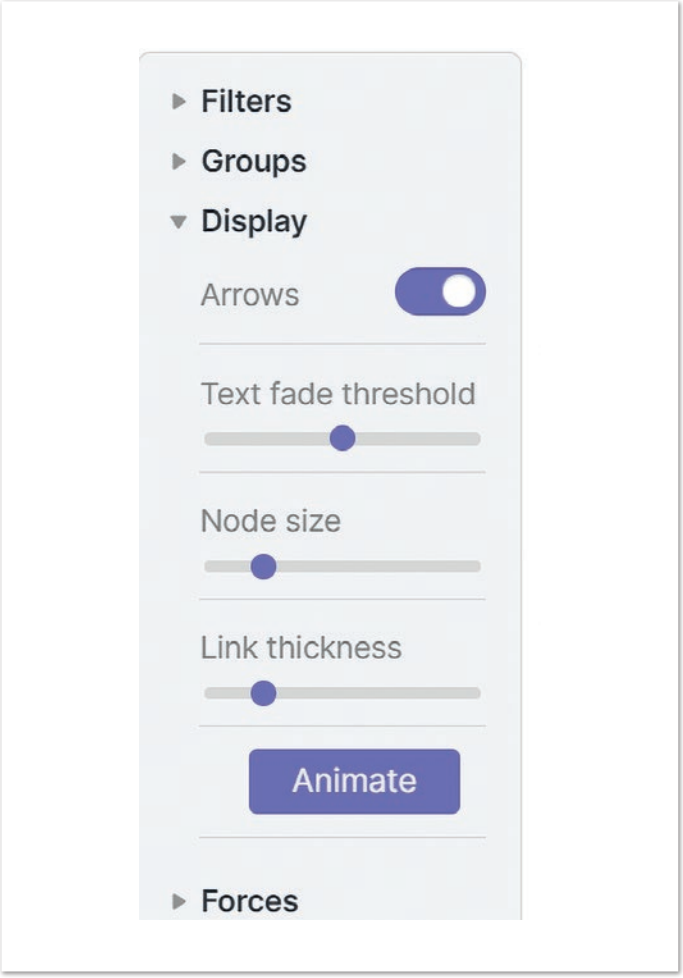

Abbildung 8: Konfigurationseinstellungen der graphischen Ansicht

Abbildung 9: Themes für jeden Geschmack

- **Dataview**: eine mächtige, SQL ähnliche Syntax, die alle Notes eines Vault per Query analysiert und das Ergebnis in einer Tabelle darstellt
- **Obsidian-Icons**: erlaubt eine Integration von Remix oder FontAwesome Icons
- **Templater**: eine mächtige Templatesyntax, die für verschiedene Darstellungen von Notes genutzt werden kann
- **Workspaces**: generiert einen Workspace aus der derzeitigen Ansicht, den man direkt wieder laden kann

Der Einsatz der Plug-ins orientiert sich natürlich auch an der eigenen Arbeitsweise. Arbeitet man mit täglichen Notizen, kann man Daily Notes oder den Zettelkasten nutzen. Architekten können auf *PlantUML* oder *Mermaid* zurückgreifen und *LaTeX* ist natürlich auch kein Problem.

Suchen und Finden

Notizen können sortiert sein oder auch nicht, man kann sie taggen oder auch nicht. Wichtig ist, dass man sie (schnell) findet! Auch hier lag der Fokus auf Schnelligkeit und Flexibilität. Einige IDEs machen es ja vor. Einfach tippen und schauen, was passiert, und meistens klappt es genauso!

Möchte man etwas gezielter vorgehen, dürfen es auch spezielle Tags sein und mit `ctrl + shift+ f` (Windows) öffnet sich obiger Dialog, es wird im gesamten Vault gesucht. Es gibt eine History und diverse Einstellungen für das Feintuning (etwa match case, Sortierung). Auch hier gibt es die praktische Funktion, dass ganze Blöcke an Ergebnissen zugeklappt (collapsed) werden können. Ist übrigens auf verschiedenen Ebenen in der Lese-Ansicht aktivierbar und gestaltet den Text übersichtlicher. Das Ergebnis der Suche kann auch gleich in eine andere Notiz eingebettet werden, aber das ist ein anderes Thema.

Fazit

Obsidian ist eine noch recht junge Software, konzipiert und vorangetrieben von zwei sehr erfahrenen Köpfen und einer schnell wachsenden Community, die in jeglicher technischen Tiefe und fachlichen Breite zuarbeitet. Sieht man seine Notizen als lebendes Objekt, ent-

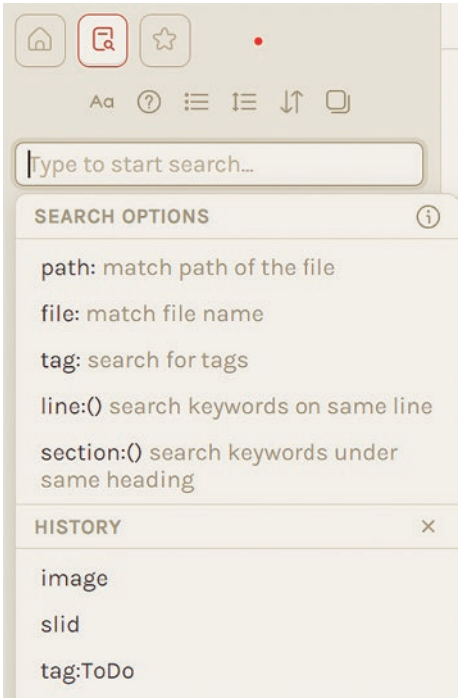

Abbildung 10: Wer suchet, der findet

wickelt sich mit der Zeit ein System, dass den eigenen Denk- und Verarbeitungsprozess abbilden und unterstützen kann. Zugegeben, es erfordert ein wenig Loslassen und Vertrauen, dass nicht alles sofort fertig und perfekt sein muss. Aber dann macht es auch viel Spaß. Notizen werden in Sekunden erstellt, mit entsprechenden Templates versehen und automatisch in den Workflow eingebunden. Wer sich, gezwungen oder freiwillig, mit dem Thema Wissensmanagement beschäftigt, sollte dem Tool eine Chance geben.

Bitsi, die während des Vortrags auf der Schulter des Entwicklers Platz genommen und ihre Nägel poliert hatte, schwang sich in die Luft, drehte eine Sonderzeichen versprühende Runde im Raum, stoppte vor der Nase ihres Klienten. „Es gibt hier noch eine ganze Menge mehr zu erzählen, aber alles zu seiner Zeit. Übrigens kostet Obsidian für den privaten Gebrauch nichts, man darf es aber gerne fördern. Ich muss ja auch irgendwie entlohnt werden." Sie zwinkerte schelmisch. „Für heute verabschiede ich mich von dir, aber du weißt ja nun, wie du mich finden kannst!", sagte sie und verschwand in einer bunten Wolke besonders schöner, nicht druckbarer Zeichen.

Quellen

[1] Evernote: *https://evernote.com/intl/de*
[2] MS OneNote: *https://www.microsoft.com/de-de/microsoft-365/onenote/digital-note-taking-app*
[3] BoostNote: *https://boostnote.io/*
[4] Joplin: *https://joplinapp.org/*
[5] Obsidian Notes: *https://obsidian.md/*
[6] Obsidian Gestein: *https://de.wikipedia.org/wiki/Obsidian*
[7] Obsidian Help: *https://help.obsidian.md/How+to/Format+your+notes*
[8] HiDrive: *https://www.strato.de/cloud-speicher/*

Uwe Sauerbrei
Sauerbrei IT-Consult
info@uwe-sauerbrei.de

Uwe Sauerbrei arbeitet als freiberuflicher Entwickler in Hamburg, organisiert seit vielen Jahren die JUG HH und ist Gründer von Kids4IT.

Flexible Anwendungsarchitektur mit der Clean Architecture Teil 3: Pragmatismus durch Shortcuts

Matthias Eschhold, Novatec Consulting GmbH

Der dritte und letzte Teil der Artikelserie „Flexible Anwendungsarchitektur" hat den Anspruch, ein Bewusstsein für Architekturabkürzungen aufzubauen, und soll dabei helfen, deren Einsatz im Projektalltag zu bewerten. Das Ziel ist es, die Flexibilität der erstellten Architektur zu erhalten. Wir überschreiten immer mehr die Grenze zwischen Theorie und Praxis und Sie werden meine persönlichen Lösungsansätze kennenlernen.

Geht es darum, die Clean Architecture als neues und unter Entwickler/innen teilweise unbekanntes Architekturmuster einzuführen, finden sich nicht nur Befürworter/innen. Die häufigsten Widerstände lassen sich zusammenfassen mit der Empfindung, dass die Architektur zu komplex ist und dies unnötig für die geforderten Anwendungsfälle erscheint. Die Seite der Gegner/innen empfindet entscheidende Elemente des Architekturmusters als Overhead in der Implementierung. Dazu zählen zum Beispiel die grundlegende Einführung von Interfaces, die starke Separierung von Schnittstellen gemäß dem Interface Segregation Principle sowie die Notwendigkeit der Implementierung von Mappings.

Es gibt hartgesottene Gegner/innen, die auch nach jahrelanger Tätigkeit in einem Big Ball of Mud nur schwer zu überzeugen sind. Ein gemeinsamer Verständnisaufbau im Team, das Erkennen von Vorteilen anhand einer konkreten Implementierung sowie ein paar Unterstützer/innen sind bedeutende Faktoren für eine erfolgreiche Einführung der Clean Architecture. Um Unterstützer/innen zu finden, sie nicht zu erschrecken und letztendlich auch die Gegner/innen zu überzeugen, können Architekturabkürzungen angewendet werden. Diese werden weiter als **Shortcuts** bezeichnet.

Psychologische Erfolgsfaktoren für die Einführung der Clean Architecture

Aus der Praxis gesprochen erachte ich es als eines der wichtigsten Elemente, dass zwischen Entwickler/innen und Product Owner/in ein **Commitment** für die Anwendung der Clean Architecture hergestellt wird. Dies bedeutet, dass jeder Betroffene die Möglichkeit erhalten muss, die Clean Architecture aus seiner Stakeholder-Perspektive zu verstehen. Es ist für die Entwickler/innen und Projektverantwortlichen wichtig zu verstehen, dass die Clean Architecture einen höheren initialen Aufwand bedeutet. Dass dies der Fall ist, aber auch welche Vorteile mit Fokus auf die Flexibilität erreicht werden können, wurde in Teil 1 und 2 ausführlich behandelt.

Dieses Commitment ist die Basis, um verantwortungsbewusst mit Shortcuts umzugehen. Um im ersten Schritt insbesondere die Unterstützer/innen zu gewinnen, habe ich mir bereits den Fauxpas erlaubt, den Shortcut „Using Domain Entities as Input or Output Models" sowohl in der Praxis als auch in dieser Artikelserie einzusetzen *[1]*. In den Teilen 1 und 2 wurde bereits erwähnt, dass die Clean Architecture mit der Full-Mapping-Strategie noch einen Schritt weiter gehen kann. Verwendet wurde bisher jedoch nur die Two-Way-Mapping-Strategie, bei der die Entitäten als Eingabeparameter und Rückgabewerte von Use Cases verwendet werden und die Adapter auf die Entitäten der Domäne mappen.

Zu meiner Verteidigung: Oft erreichen Sie nur erfolgreich Veränderung, wenn diese in homöopathischen Dosen durchgeführt wird. Betrachten wir die Ziele, die wir in der bisher beschriebenen Variante der Clean Architecture erreichen können, stellt dies eine signifikante Verbesserung zur weit verbreiteten Schichtenarchitektur dar. Darüber hinaus haben wir trotz Shortcut einen sauberen Start mit klaren Qualitätszielen. Dies ermöglicht zum einen ein einfaches Refactoring von einer Two-Way-Mapping auf eine Full-Mapping-Strategie und unterstützt zum anderen immer noch den zweiten psychologischen Erfolgsfaktor.

Ein **sauberer Start** und der **disziplinierte Erhalt** der Architektur, insbesondere in der frühen Phase des Lebenszyklus eines Systems, in der schnell viele Features in das Produkt wandern, ist elementar für die erfolgreiche Etablierung der Clean Architecture. *Hombergs* beschreibt diesen psychologisch wichtigen Aspekt anhand der „Broken Windows Theory" nach *Philip Zimbardo*. Diese Studie hat anhand eines geparkten Fahrzeugs untersucht, welche unterschiedlichen Verhaltensweisen die Tatsache, ob das Fahrzeug in Ordnung oder beschädigt ist, im Menschen hervorruft.

Hombergs fasst die Studie von *Zimbardo* wie folgt zusammen. Wenn Dinge ungepflegt erscheinen, anfangen kaputt zu gehen oder bereits beschädigt sind, dann erscheint es für den Menschen als in Ordnung, die Dinge nicht zu verbessern, selbst keine Sorgfalt mehr walten zu lassen oder sogar die Dinge weiter zu beschädigen. Es ist folglich von entscheidender Bedeutung, diesen Effekt bei der Anwendung von Shortcuts zu berücksichtigen. Das Ziel ist es, die Wartbarkeit und Flexibilität der Architektur zu erhalten. Die Gefahr, den psychologischen Effekt der „Broken Windows Theory" durch zu viele und zu starke Shortcuts auszulösen, muss bei allen Architekturentscheidungen in diesem Zusammenhang berücksichtigt werden *[1]*.

Die Fahrzeugbewertung stellt die Architektur vor neue Herausforderungen

In den ersten Sprints wurde der Baustein Fahrzeug implementiert. Dieser Baustein verwaltet Stamm- und Bewegungsdaten des Fahrzeugs. Fachlich entsteht die Vision von „Know Your Car", bei der Fahrzeugbesitzer/innen die komplette Historie zu ihrem Fahrzeug verwalten können. Dazu gehören etwa durchgeführte Wartungen und Reparaturen sowie weitere Informationen, die von dem/der Fahrzeugbesitzer/in erfasst werden. Das zu betrachtende System wird von gewerblichen Nutzern, wie Autohäusern, eingesetzt. Diese sollen durch „Know Your Car" sowie „Rate Your Car" bei der Kundenbindung und bei Vertriebsaktivitäten unterstützt werden.

„Rate Your Car" beinhaltet eine schnelle und exakte Fahrzeugbewertung. Nutzergruppen und weitere Details hinsichtlich der Fahrzeugbewertung wurden in Teil 2 ausführlich anhand des *Service Strategy Pattern* beschrieben. Der allgemeine Zustand eines Fahrzeugs ist eine relevante Größe für die Fahrzeugbewertung. Der/die Besitzer/in, eventuell ein/eine Fahrzeugliebhaber/in, pflegt das Fahrzeug intensiv und regelmäßig. Diese Informationen erlauben die Schlussfolgerung auf einen guten Fahrzeugzustand.

Stamm- und Bewegungsdaten, wie Serienausstattung oder Kilometerstand, aus „Know Your Car" sind ebenfalls relevante Einga-

begrößen für die Fahrzeugbewertung. Die von dem/der Fahrzeugbesitzer/in selbst erfassten Informationen sind insbesondere für die Schnellbewertung relevant. Die exakte Bewertung stützt sich auf Informationen, erfasst von einem/einer Automobilverkäufer/in, kombiniert mit weiteren allgemeinen und herstellerbezogenen Bewertungsfaktoren. Die Basis beider Bewertungen ist der durchschnittliche Marktpreis. Dieser wird anhand eines öffentlichen Web-Service ermittelt. Die Schnellbewertung wird selbst implementiert. Die exakte Fahrzeugbewertung kann Grundlage eines rechtlich bindenden Angebots sein, weshalb hierfür die Anbindung des zentralen Bewertungssystems eine Randbedingung darstellt. *Abbildung 1* zeigt den Systemkontext von „Alles rund ums Fahrzeug", der den beschriebenen Sachverhalt, ergänzt um den externen Service zur Abfrage von Fahrzeugstammdaten aus Teil 2, dokumentiert.

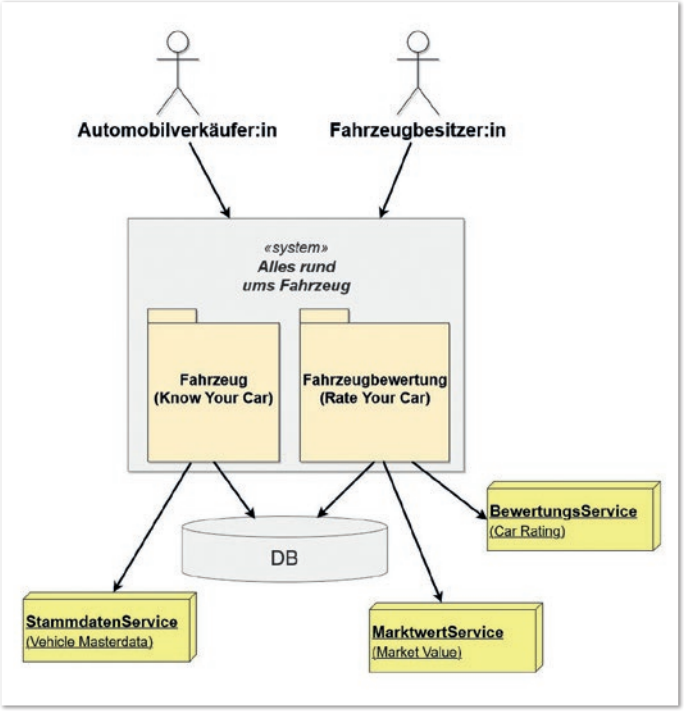

Abbildung 1: Systemkontext von „Alles rund ums Fahrzeug" (© Matthias Eschhold)

Output Adapter Command Facade

Die Anbindung des zentralen Bewertungsservice soll als Beispiel dafür dienen, wieso sehr hohe Komplexität ein Argument für den Einsatz der Full-Mapping-Strategie ist. Bei der Anbindung des zentralen Bewertungsservice wird darüber hinaus das *Output Adapter Command Facade Pattern* eingesetzt *(siehe Abbildung 2)*. Dies hat das Ziel, sehr komplexe Kommunikation mit externen IT-Systemen zu vereinfachen. Diese entsteht zum Beispiel, wenn Fachlichkeit und Technik nur sehr schwer voneinander separiert werden können.

Die *Output Adapter Command Facade* basiert auf den Grundideen des *Facade* und des *Command Pattern*. Eine Fassade ermöglicht den vereinfachten Zugriff auf eines oder mehrere komplex zu nutzenden Systeme beziehungsweise Subsysteme. Kommandos kapseln Befehle und bieten die Funktionalität, Befehle rückgängig zu machen (Undo) oder wiederherzustellen (Redo) *[2]*.

Das *Output Adapter Command Facade Pattern* ist am einfachsten am konkreten Beispiel zu verstehen. Das zentrale Bewertungssystem ist schon älter und das API baut auf einem historisch gewachsenen Kern auf. Dies führt dazu, dass das Schnittstellendesign aus Sicht eines Konsumenten nicht optimal ist. Viele Konsumenten mit stark variierenden Anforderungen erschweren das Schnittstellendesign zusätzlich.

Aus Sicht von „Alles rund ums Fahrzeug", ist die Erstellung einer Fahrzeugbewertung eine fachlich atomare Funktion. In der Kommunikation mit dem zentralen Bewertungssystem sind dies allerdings mehrere technische Funktionen. Dabei vermischen sich fachliche Funktionen mit den Aspekten der Authentifizierung und des Transaktionsmanagements. Das Sequenzdiagramm in *Abbildung 3* zeigt diesen Sachverhalt im Detail.

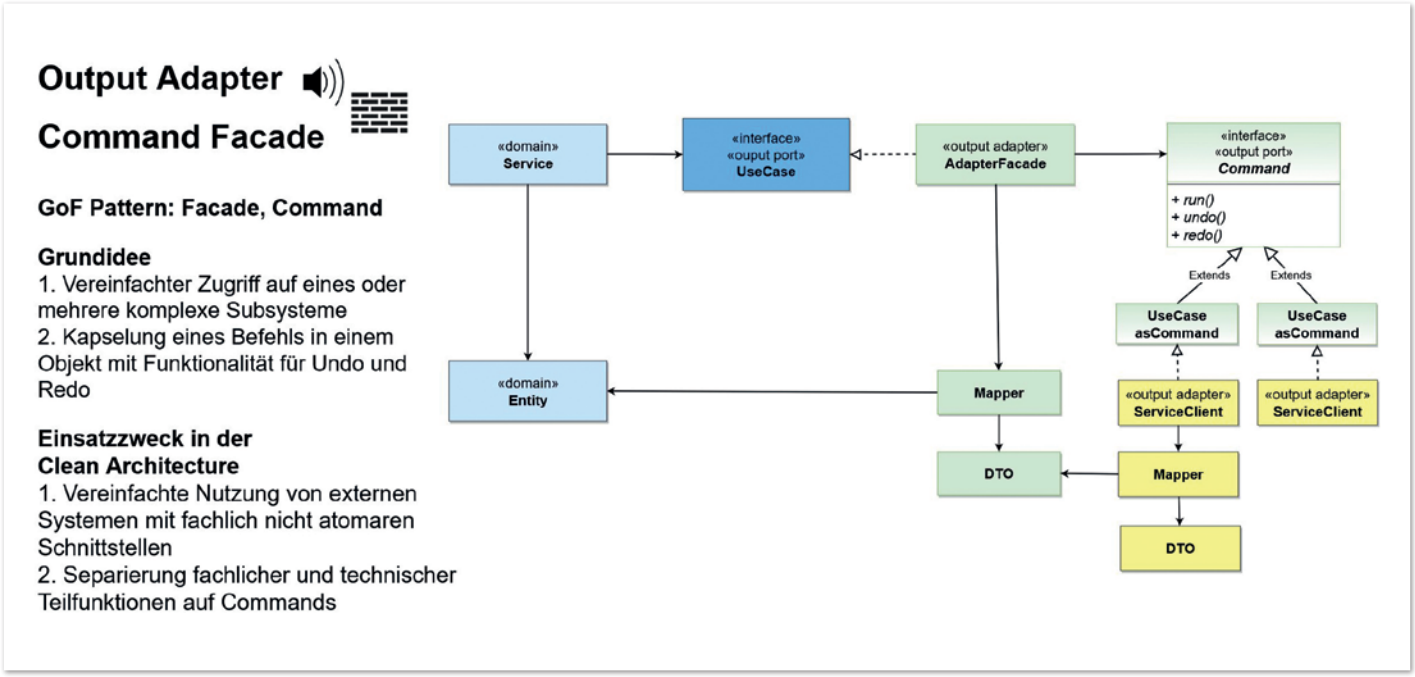

Abbildung 2: Steckbrief Output Adapter Command Facade (© Matthias Eschhold)

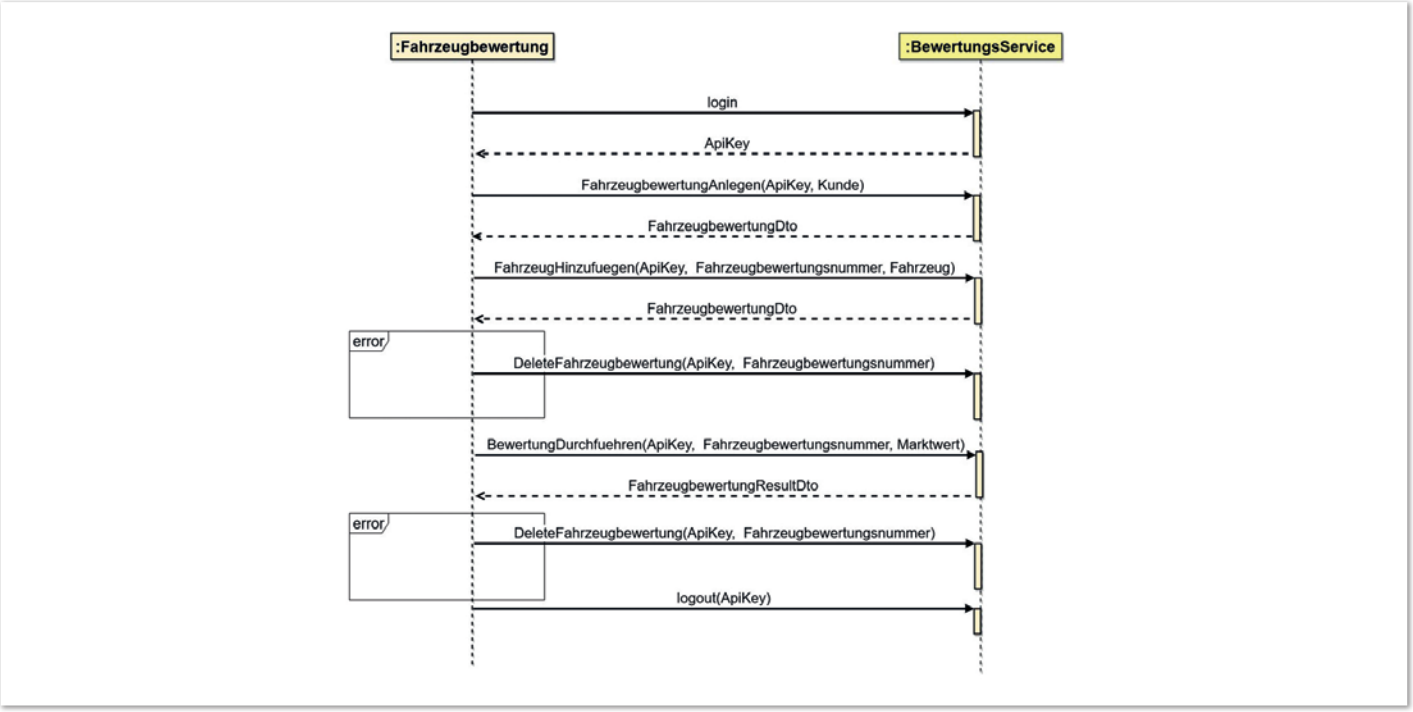

Abbildung 3: Interaktion zwischen der Fahrzeugbewertung und dem zentralen Bewertungsservice (© Matthias Eschhold)

Abbildung 4 zeigt die *Output Adapter Command Facade* angewendet auf die Fahrzeugbewertung. Fachliche Funktionen mit Rollback sowie Login und Logout werden in Kommando-Objekte gekapselt. Die *FahrzeugbewertungsAdapterFacade* steuert den Ablauf der Kommandokette. *Listing 1* zeigt den vereinfachten Ablauf der Kommandokette in der *FahrzeugbewertungsAdapterFacade*. *Listing 2* zeigt den *FahrzeugbewertungsService* als Nutzer der *FahrzeugbewertungsAdapterFacade* [4]. Hier wird der übergreifende Anwendungsfall realisiert,

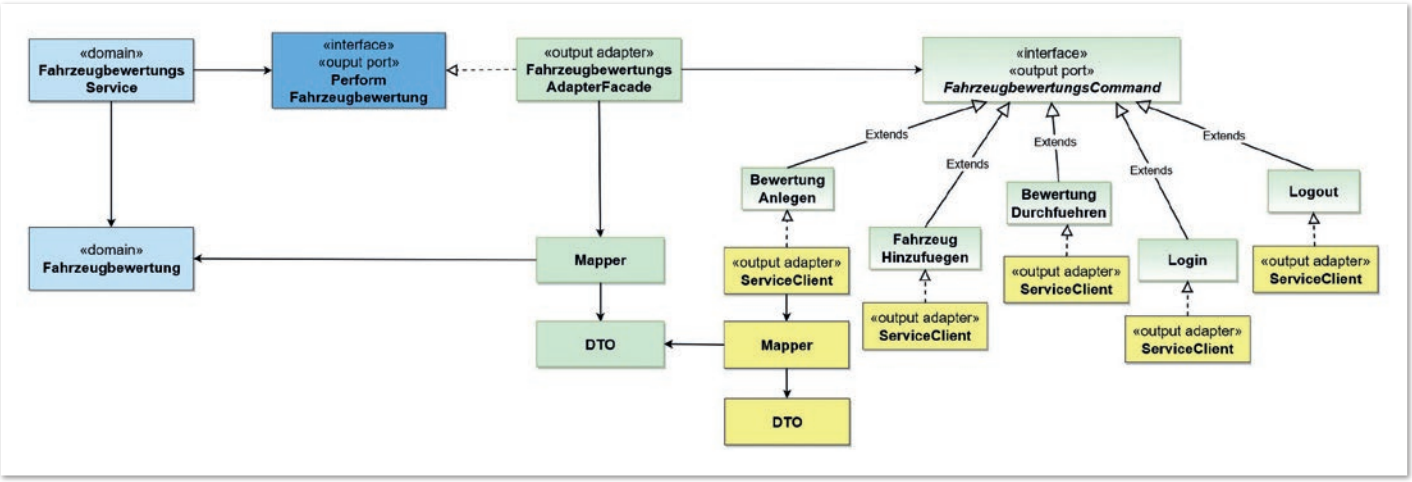

Abbildung 4: Das Output Adapter Command Facade Pattern eingesetzt bei der Integration des zentralen Bewertungsservice (© Matthias Eschhold)

```
public Fahrzeugbewertung create(Fahrzeugbesitzer besitzer, Fahrzeug fahrzeug, Marktwert marktwert) {
  String apiKey = loginCommand.run(readCredentialsCommand.run());
  InitialRatingDto createdRatingDto = fahrzeugbewertungAnlegenCommand.run(apiKey, fahrzeugbesitzer);
  try {
     fahrzeugHinzufuegenCommand.run(apiKey, createdRatingDto.getRatingId(), fahrzeug, marktwert);
     CarRatingDto ratingDto = bewertungDurchfuehren.run(apiKey, created-RatingDto.getRatingId());
     return fahrzeugbewertungToCarRatingDtoMapper.map(ratingDto);
  } catch (Exception e) {
     fahrzeugbewertungAnlegenCommand.rollback(apiKey, createdRating-Dto.getRatingId());
     throw e;
  } finally {
     logoutCommand.run(apiKey);
  }
}
```

Listing 1: Ablauf der Kommandokette in der FahrzeugbewertungsAdapterFacade

```
public Fahrzeugbewertung calculate(Fahrzeugbesitzer besitzer, Fahrgestellnummer fahrgestellnummer) {
  Fahrzeug fahrzeug = readFahrzeug.read(fahrgestellnummer);
  Marktwert marktwert = calcuateMarktpreis.calcuate(fahrzeug);
  return adapterFacade.createBewertung(besitzer, fahrzeug, marktwert);
}
```

Listing 2: Nutzung der FahrzeugbewertungsAdapterFacade im FahrzeugbewertungsService

mit weiteren fachlichen Funktionen, wie etwa der Ermittlung des Marktwerts.

Durch die Trennung von fachlichen und technischen Funktionen sowie der Befehlskette von weiteren Aspekten des übergreifenden Anwendungsfalls wird die Testbarkeit und Erweiterbarkeit sehr gut unterstützt. Ändern sich einzelne Befehle, können diese isoliert angepasst und getestet werden. Ein neues *Command* kann ohne Modifikation bestehender Commands integriert werden. Auch die Befehlskette und die Basisfunktionalität können isoliert voneinander entwickelt und getestet werden. Und wir wissen bereits, eine gute Testbarkeit ermöglicht eine schnelle Reaktionszeit auf Veränderung.

Beziehungen zwischen fachlichen Bausteinen

Die Fahrzeugbewertung wirft weiter die Frage auf, wie wir die bereits beschriebenen fachlichen Abhängigkeiten zwischen den Softwarebausteinen „Fahrzeug" und „Fahrzeugbewertung" gestalten können. Um Flexibilität zu erhalten, ist das primäre Ziel, die fachlichen Abhängigkeiten möglichst lose gekoppelt abzubilden.

Im Kontext der Clean Architecture mit fachlicher Modularisierung bedeutet dies:

- Die Domäne muss unabhängig von der Infrastruktur bleiben
- Die Funktionen und Entitäten von fachlichen Bausteinen dürfen sich nicht direkt referenzieren

Fachliche Bausteine treiben unsere Architektur primär. Dies ist sichtbar am Beispiel der Bausteine „Fahrzeug" und „Fahrzeugbewertung". Fachliche Bausteine können als Teil ihrer Funktionalität exakt gleiche technische Funktionen benötigen. Diese für die fachlichen Bausteine unterstützenden Funktionen werden als eigene Bausteine in Form eines Shared Output Adapter oder Supporting Service realisiert.

Benötigt ein fachlicher Baustein die Kernfunktionalität eines anderen fachlichen Bausteins, darf diese gemäß der Grundidee des Ports-und-Adapters-Musters nur über den eingehenden Use Case in die Domäne konsumiert werden. Die originäre Motivation des

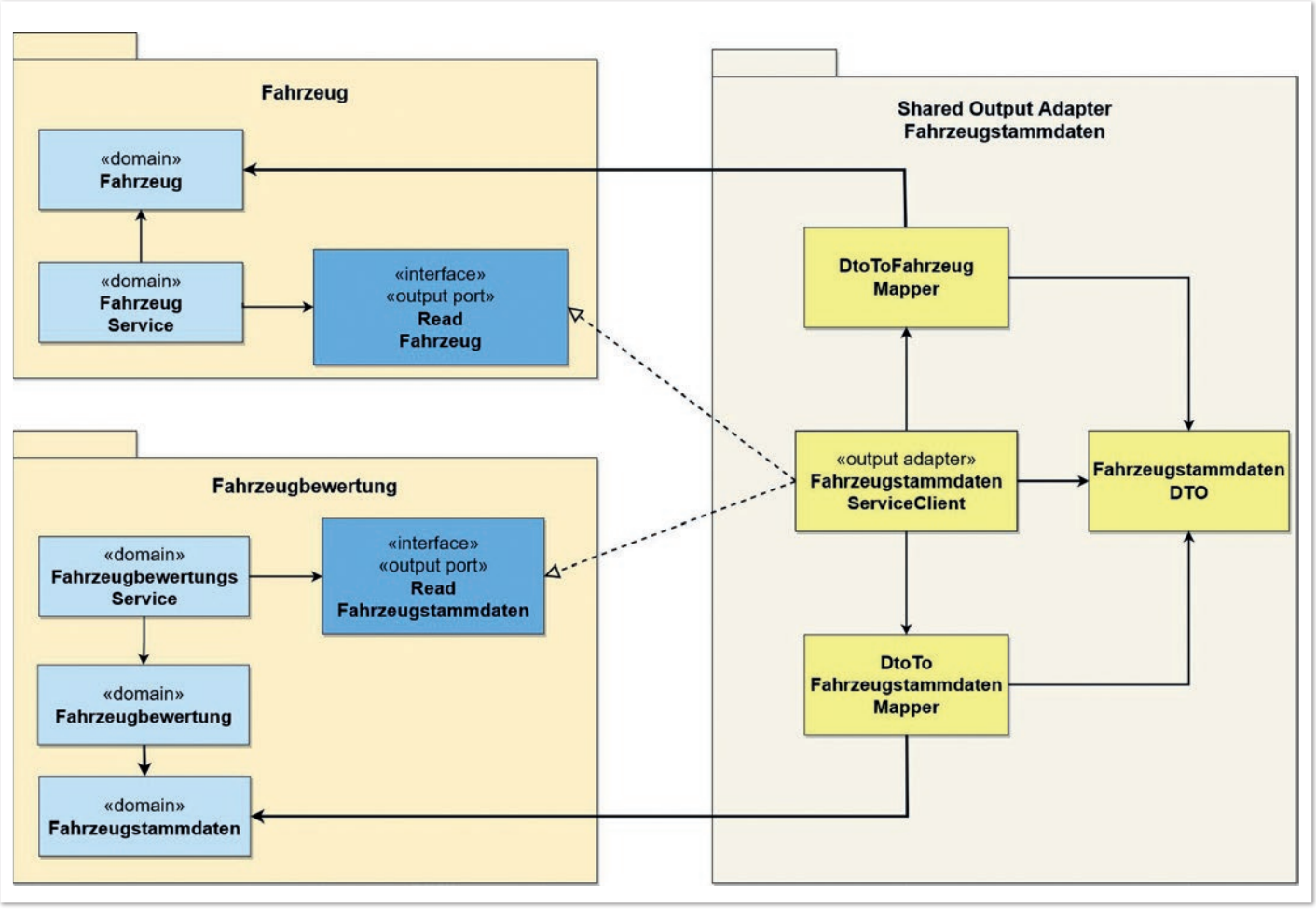

Abbildung 5: Das Shared Output Adapter Pattern am Beispiel der Fahrzeugstammdaten (© Matthias Eschhold)

Ports-und-Adapters-Musters besagt, dass die Domäne auf gleiche Weise, unabhängig vom Konsumenten, nutzbar sein muss *[3]*. Dies berücksichtigen das Application Service sowie das Adapter.OutUseCase.In Pattern für die Gestaltung von Beziehungen zwischen fachlichen Bausteinen.

Shared Output Adapter

Nehmen wir an, die Fahrzeugstammdaten sind für die Fahrzeugbewertung ausreichend. Ist dies der Fall, können die Bausteine „Fahrzeug" und „Fahrzeugbewertung" unabhängig voneinander ihre Aufgabe erledigen, indem beide Bausteine den externen Service zur Abfrage der Stammdaten anbinden. Innerhalb eines Software-artefakts und innerhalb eines Teams ist der Wunsch nachvollziehbar, die gleiche Integration eines externen Service nicht mehrfach zu implementieren. Ein Kompromiss zwischen diesen Zielsetzungen repräsentiert das Shared Output Adapter Pattern *(siehe Abbildung 5)*. Damit beide Softwarebausteine weiterhin keine technischen Abhängigkeiten zueinander aufbauen, muss der Shared Output Adapter für jeden Konsumenten einen ausgehen-

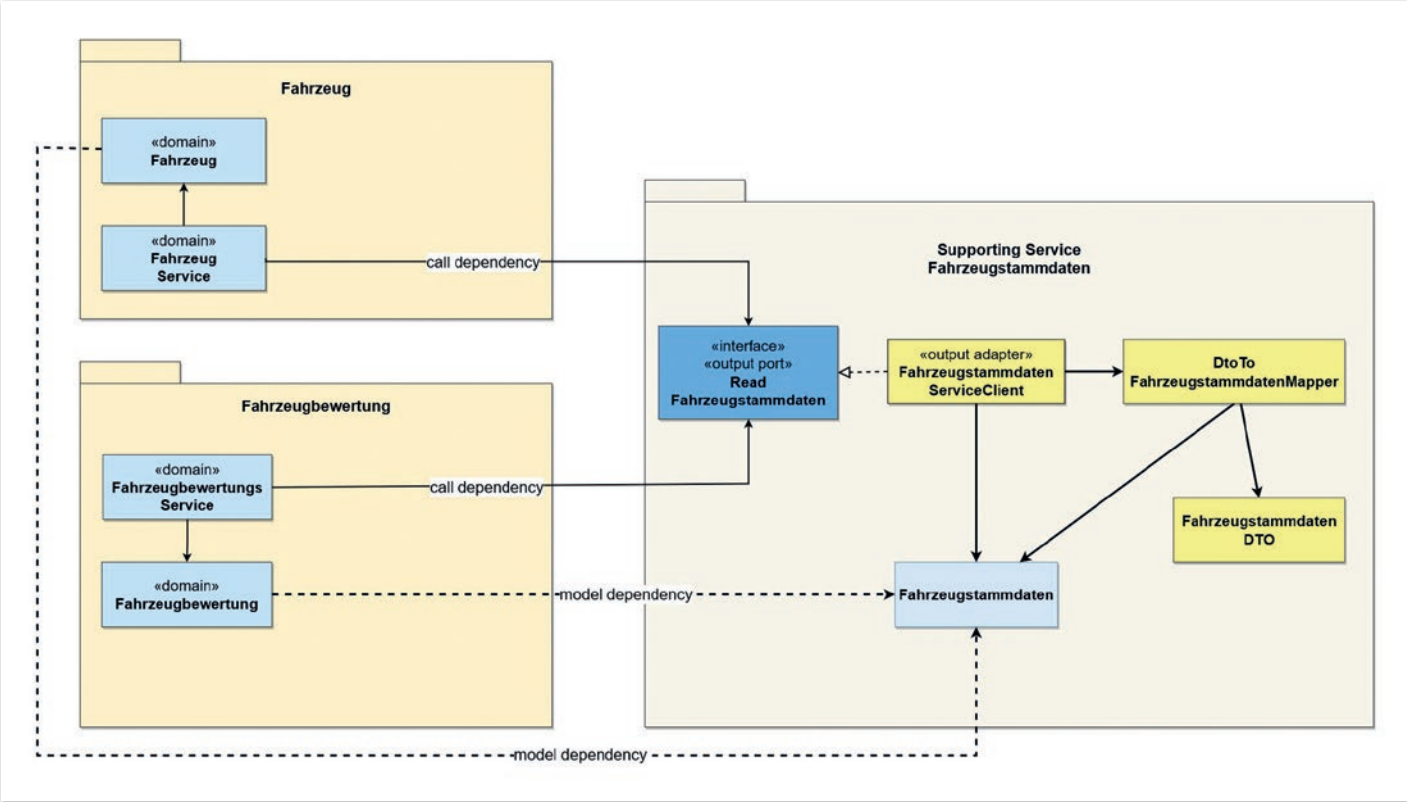

Abbildung 6: Das Supporting Service Pattern am Beispiel der Fahrzeugstammdaten (© Matthias Eschhold)

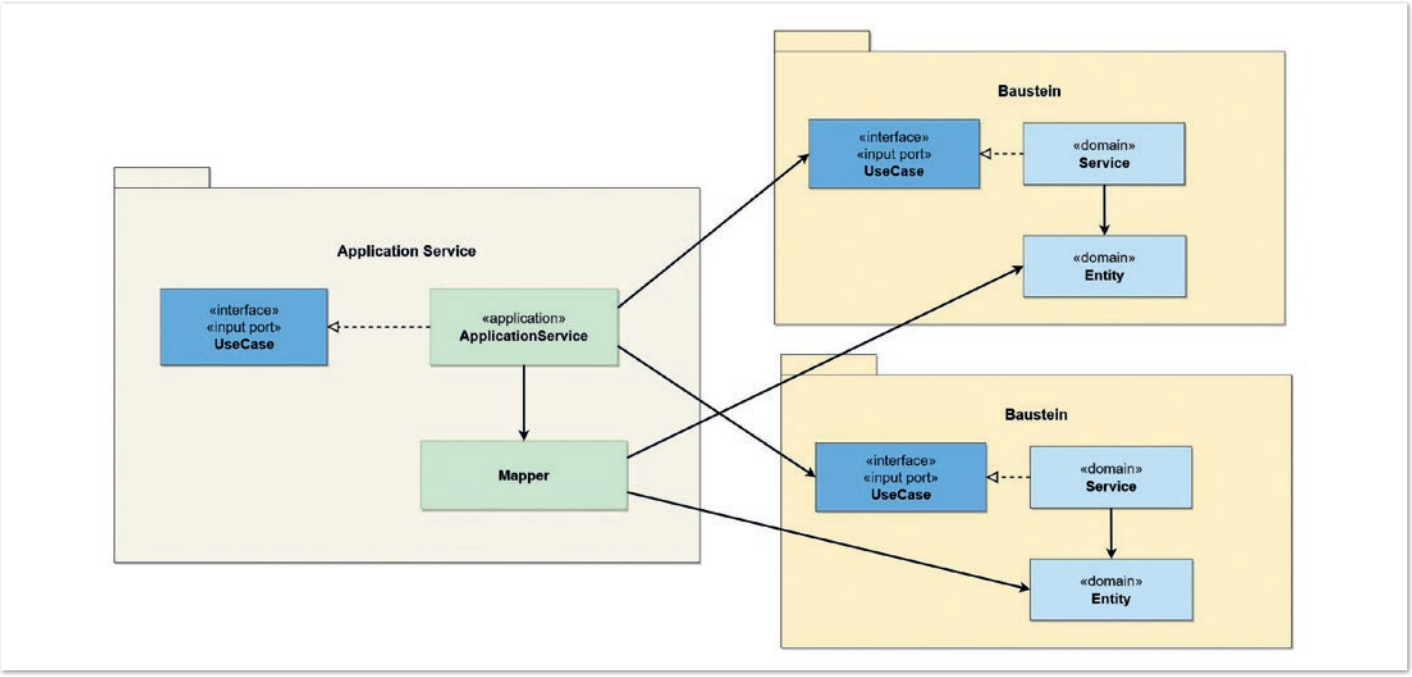

Abbildung 7: Allgemeine Beschreibung des Application Service Pattern (© Matthias Eschhold)

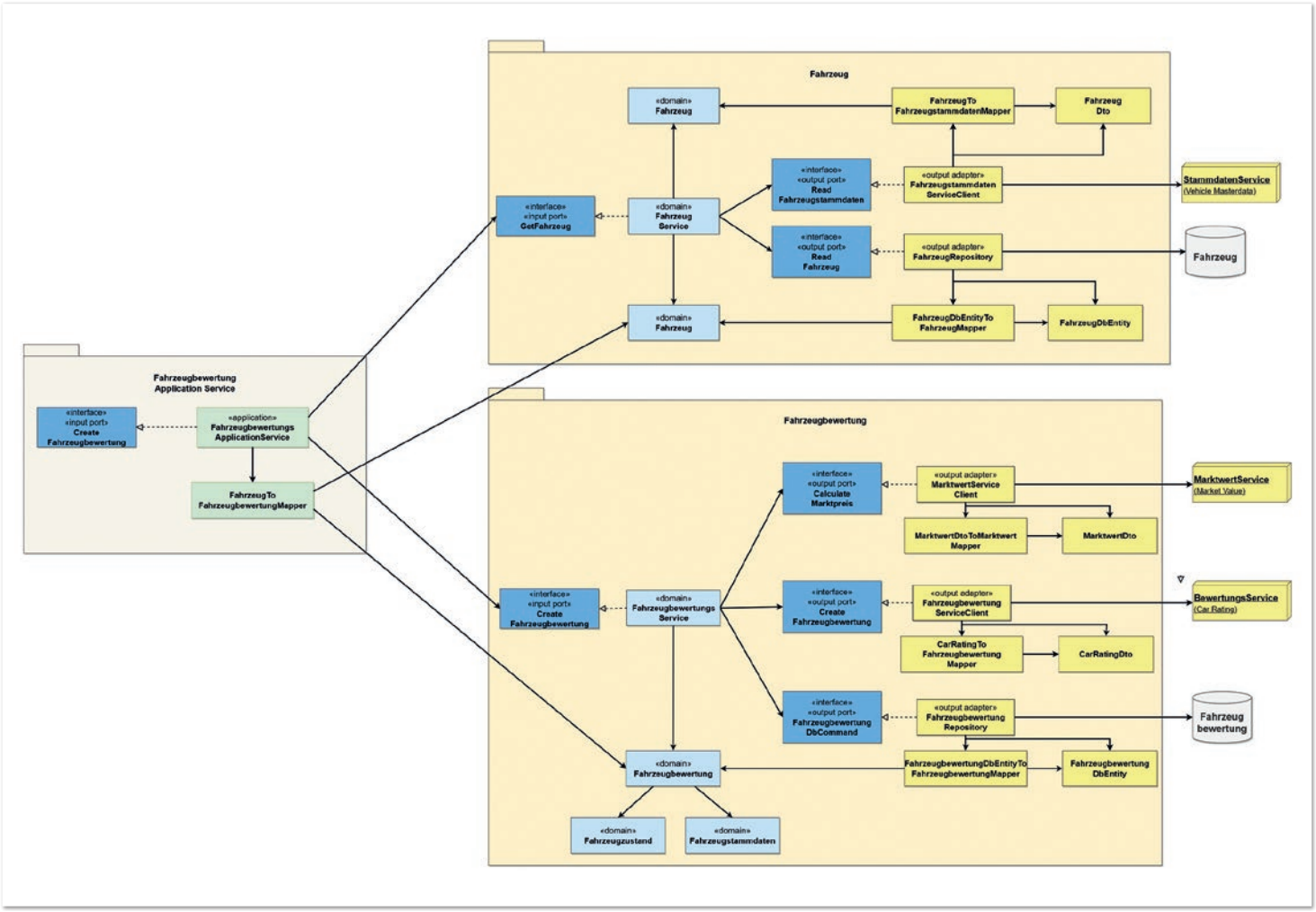

Abbildung 8: Das Application Service Pattern am Beispiel der Fahrzeugbewertung (© Matthias Eschhold)

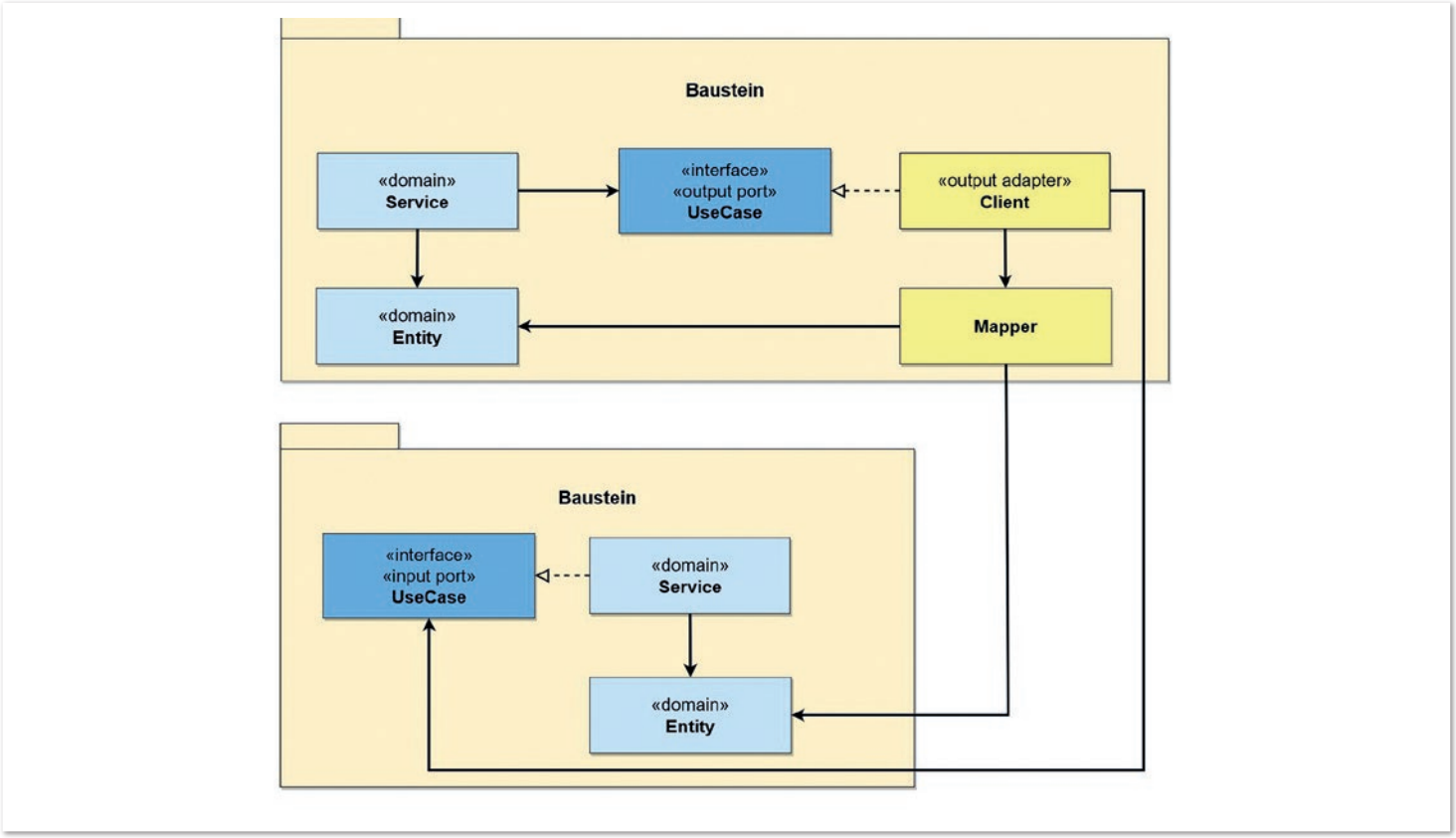

Abbildung 9: Allgemeine Beschreibung des Adapter.Out-UseCase.In Pattern (© Matthias Eschhold)

den Use Case des Konsumenten und ein Mapping auf die Entitäten des Konsumenten implementieren. Strukturell bleibt dadurch die Unabhängigkeit zwischen den Bausteinen „Fahrzeug" und „Fahrzeugbewertung" erhalten. Dennoch kann Koordinationsaufwand entstehen, wenn aufgrund einer Weiterentwicklung die Belange mehrerer Konsumenten vereinbart werden müssen. Ist dies zu häufig der Fall, ist dieses Muster nicht passend für mindestens einen der Konsumenten.

Der *Shared Output Adapter* sollte eine Ausnahme und nicht die Regel darstellen. Diese Ausnahme muss bewusst getroffen werden. Das Fenster wird hier sehr schnell auf Basis des allgemeinen Strebens nach Wiederverwendung zerbrochen. Dies führt dazu, dass der *Shared Output Adapter* unbeherrschbar wird. Um Entwicklungsaufwand zu reduzieren, wird auf Konsumentenorientierung und Mappings verzichtet. Die Folge ist eine unerwünschte hohe Kopplung zwischen den fachlichen Bausteinen aufgrund der Generalisierung.

Supporting Service

Ist Generalisierung der Wunsch und wird dies als valide Lösungsoption im Projektkontext bewertet, empfehle ich folgenden Denkansatz mit

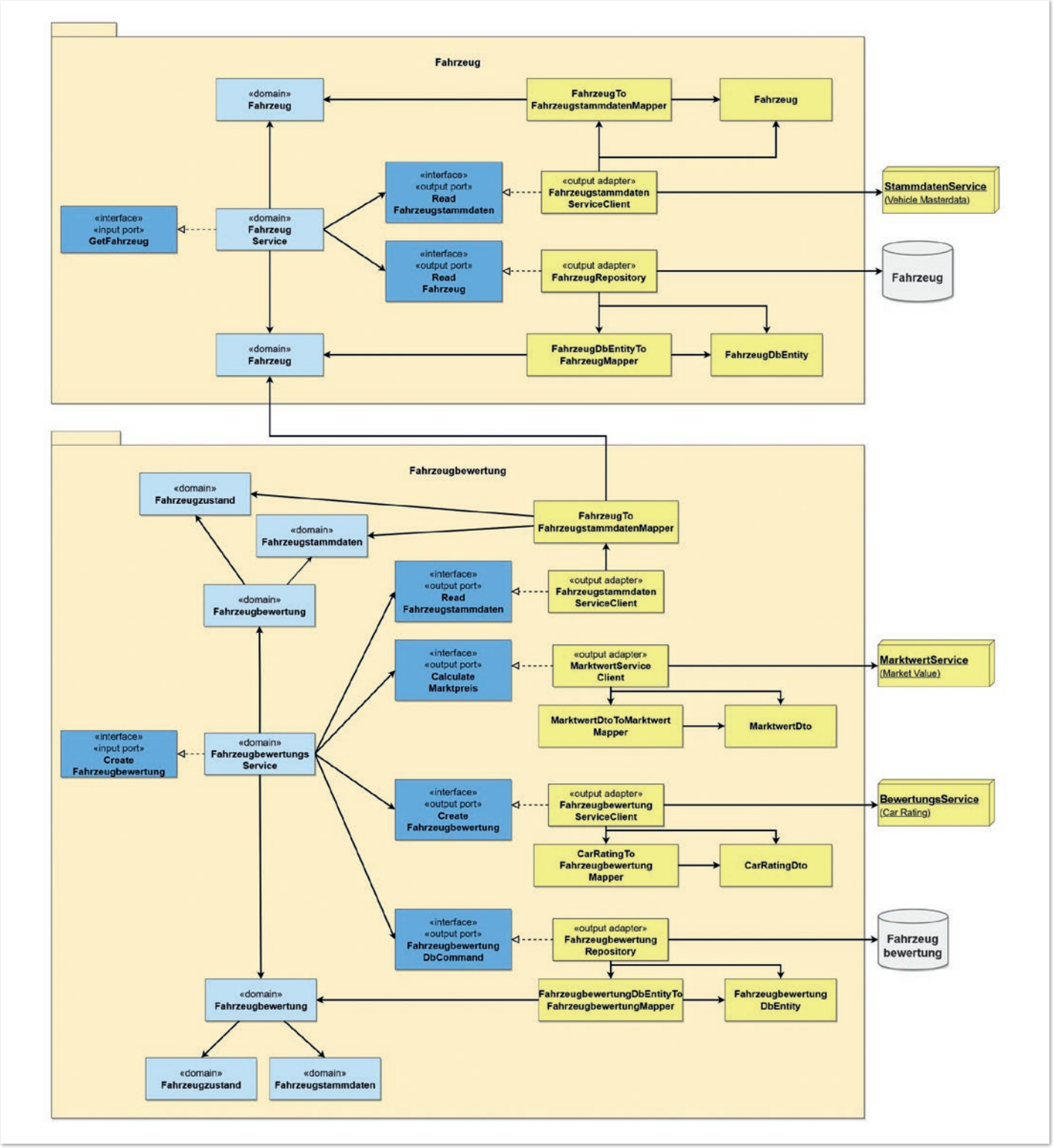

Abbildung 10: Das Adapter.Out-UseCase.In Pattern am Beispiel der Fahrzeugbewertung (© Matthias Eschhold)

dem Begriff *Supporting Service* *(siehe Abbildung 6)*. Die Definition dieses Musters ist eine generische Schnittstelle mit Entität, die von mehreren fachlichen Bausteinen gemeinsam verwendet wird. Die Ziele für die Beziehungen zwischen Bausteinen sind weiterhin erfüllt. Dennoch stellt dies eine sehr hohe Kopplung dar. Deshalb muss auch dies bewusst entschieden werden. Dabei hilft es, einen Begriff wie *Supporting Service* zu verwenden, der diesen Sachverhalt verständlich abbildet.

Die Funktionalität „E-Mail versenden" ist ein weiteres typisches Beispiel für einen *Supporting Service* oder *Shared Output Adapter*. Wie der *Shared Output Adapter* ist der *Supporting Service* mit Vorsicht zu verwenden. Dadurch kann schnell und unbemerkt unerwünschte Kopplung entstehen. Auch gilt, penibel darauf zu achten, dass fachliche Kernfunktionen nicht aufgrund des Wunsches nach Wiederverwendung zu unterstützenden Funktionen werden.

Application Service

An dieser Stelle wird die Diskussion „Rich vs. Anemic Domain Model" *[5]* sowie „Application Service vs. Domain Service" *[6]* nicht geführt. Ein Domain Service entspricht dem Klassenstereotyp *Service*. Abhängig davon, wie die Vorlieben zur Frage „Rich vs. Anemic Domain Model" im Projekt sind und welche fachliche Komplexität gegeben ist, existieren ein Service und eine Entität oder nur eine Entität mit Verhalten. Die im Folgenden beschriebene Logik sollte unabhängig davon immer in einen übergeordneten Application Service ausgelagert werden. *Abbildung 7* zeigt das *Application Service Pattern* mit dem neuen Klassenstereotyp *ApplicationService*. Dieser hat die Verantwortung, durch Orchestrierung die fachlichen Abhängigkeiten zwischen den Bausteinen aufzulösen. Dabei implementiert der *ApplicationService* einen eingehenden Use Case und mappt zwischen den Entitäten der fachlichen Bausteine.

Das *Application Service Pattern* bedingt ein Umdenken in der bisherigen Paketstruktur. Es liegt nahe, hierfür einen weiteren Ring in die Architektur einzuziehen. Die konkrete Abbildung ist jedoch abhängig von der Variante der Paketstrukturierung. Dieses Thema wird hier nicht weiter vertieft. Die beschriebenen Klassenstereotype sind auf die Variante architektonisch ausdrucksstark ausgelegt *[1]*. *Abbildung 8* zeigt eine detaillierte Architektur der Bausteine „Fahrzeug" und „Fahrzeugbewertung".

Adapter.Out–UseCase.In

Fans, die die bisherige Paketstruktur bevorzugen, finden eine Alternative im *Adapter.Out–UseCase.In Pattern*. Der Konsument implementiert für die Integration des Anbieters einen ausgehenden Adapter, so wie dies die Clean Architecture auch für die Integration von Infrastrukturkomponenten vorsieht. In diesem Adapter wird der eingehende Use Case des Anbieters referenziert. Ein Mapping zwischen den Entitäten des Konsumenten und des Anbieters erfolgt ebenfalls im Adapter. Der neue Klassenstereotyp *Client* stellt die Adapter-Implementierung für eine Beziehung zwischen fachlichen Bausteinen dar. *Abbildung 9* verdeutlicht die Unabhängigkeit der fachlichen Kernfunktionalitäten und -entitäten von Konsument und Anbieter. Die Kopplung ist beschränkt auf den Adapter. Dies ermöglicht eine schnelle Anpassungsfähigkeit.

Anhand des ausgehenden Use Case *ReadFahrzeugstammdaten*, dem *FahrzeugstammdatenClient* sowie des eingehenden Use Case *Read-*

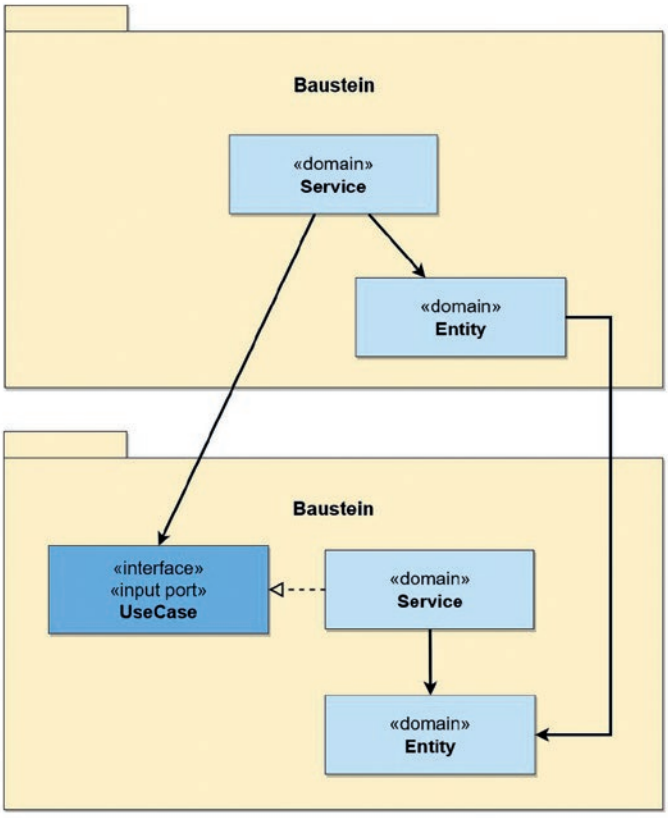

Abbildung 11: Allgemeine Beschreibung des Service-UseCase.In Pattern (© Matthias Eschhold)

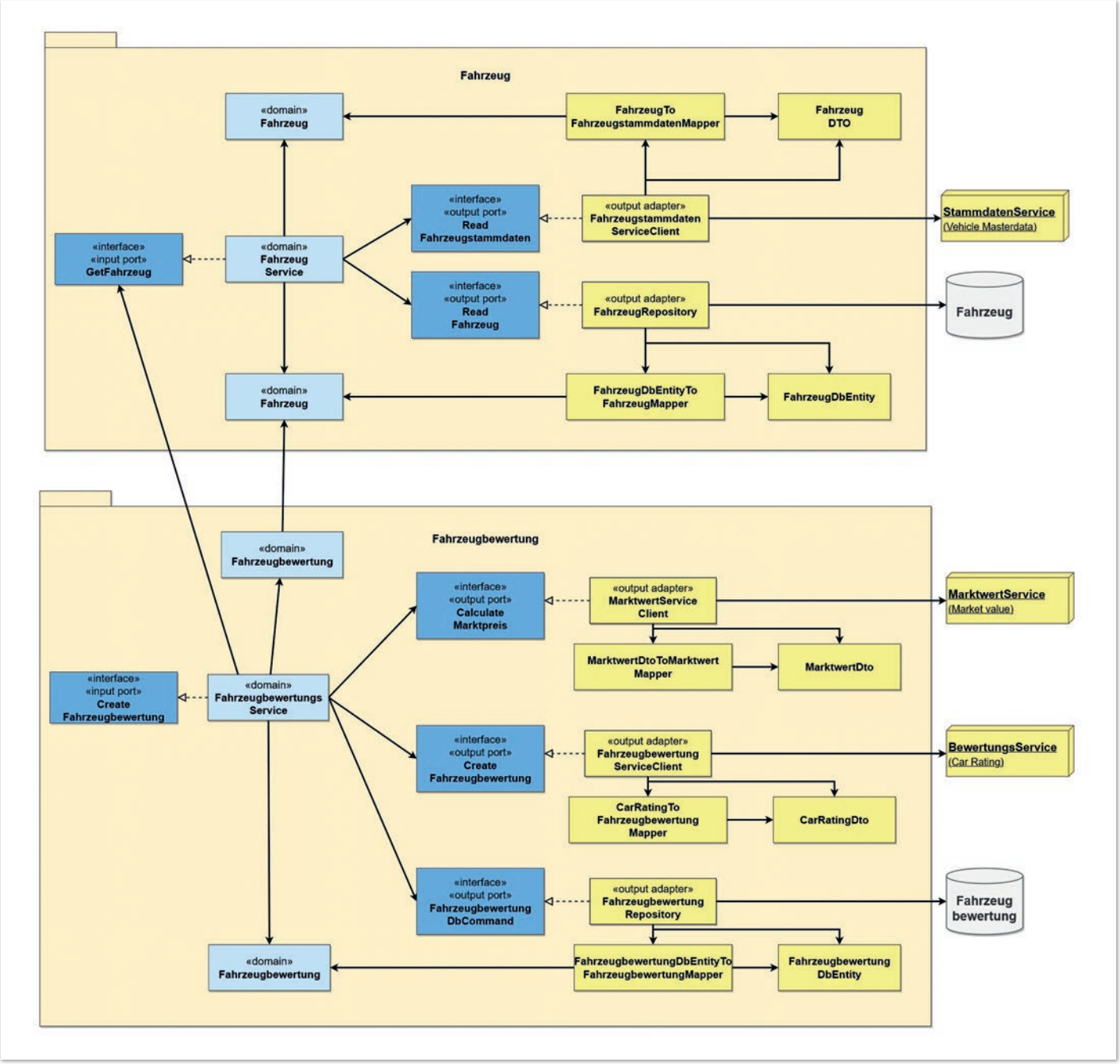

Abbildung 12: Das Service-UseCase.In Pattern am Beispiel der Fahrzeugbewertung (© Matthias Eschhold)

Fahrzeug wird das Muster am Beispiel der Fahrzeugbewertung realisiert *(siehe Abbildung 10)*.

Entstehen viele bidirektionale Beziehungen zwischen zwei Bausteinen, deutet dies auf einen verbesserungsfähigen fachlichen Schnitt hin. Sowohl beim *Application Service Pattern* als auch beim *Adapter. Out–UseCase.In Pattern* sind unidirektionale Beziehungen anzustreben. Ausnahmen bestätigen die Regel. Eine entsprechende Häufigkeit wird jedoch bedenklich.

Shortcut Shared Entities

In beiden Fällen der Beziehungsgestaltung zwischen Bausteinen teilen sich die Bausteine keine Entitäten. Nur auf diesem Weg vermeiden wir die direkte Abhängigkeit und bezahlen dies mit einer Adapter-Implementierung oder der strukturellen Komplexität eines *Application Service Pattern*.

Dieser Aspekt ruft den Widerstand bei Gegner/innen und eventuell auch Unterstützer/innen hervor. Nach meiner Erfahrung wird auf den Shortcut Shared Entities gerne zurückgefallen. Als Architekt halte ich es für wichtig, im Team die Stabilität der geteilten Entitäten einzuschätzen. Dies hilft zu beurteilen, welche negativen Auswirkungen diese Kopplung verursachen kann und welchen Aufwand ein Rückbau bedeutet. Hierfür sind keine exakten Zahlen notwendig. Eine offene Diskussion auf Basis des Commitment im Team führen zu angemessenen Lösungen. Dennoch ist die klare Empfehlung, die „Broken Windows Theory" im Hinterkopf zu behalten.

Entitäten können geteilt werden, indem sie der Konsument nicht mappt und somit die Entitäten des Anbieters akzeptiert. Änderungen beim Anbieter verursachen folglich Anpassungsbedarf beim Konsumenten, an den Codestellen, an denen die Entitäten verwendet werden. Der Adapter wird obsolet, weshalb dieser Ansatz *Ser-*

vice–UseCase.In Pattern getauft wurde *(siehe Abbildung 11)*.

Diese Muster, angewendet auf die Fahrzeugbewertung, zeigt *Abbildung 12*. Im großen Bild wird die Kopplung durch die Assoziationen zwischen dem *FahrzeugbewertungsService* und *GetFahrzeug* sowie zwischen den Entitäten *Fahrzeugbewertung* und *Fahrzeug* sichtbar.

Die zweite Möglichkeit, Entitäten zu teilen, sind Module, in denen die Entitäten abgelegt werden. Alle Nutzer der Entitäten können diese auch verändern. Der gegenseitige Einfluss und die Gefahr von Seiteneffekten nehmen zu. Aus der Perspektive der „Broken Windows Theory", lädt das Package *SharedEntities* *(siehe Abbildung 13)* grundsätzlich jeden Baustein dazu ein, diese Entitäten zu nutzen oder neue Entitäten darin abzulegen, auch wenn kein fachlicher Zusammenhang oder wirkliche Notwendigkeit hierfür besteht. Um dies zu verdeutlichen, wird in *Abbildung 13* der Baustein „Fahrzeugbesitzer" eingeführt.

Bewertung der Muster

Abbildung 14 zeigt eine Architektur, wenn die „Broken Windows Theory" doch zugeschlagen hat und zu viele Shortcuts zu einer hohen Kopplung sowie Komplexität geführt haben. Ich bezeichne dies als „Big Ball of Mud 2.0", da wir trotz Abkehr von der geschichteten und datenzentrierten Architektur unübersichtliche Beziehungsstrukturen, Chaos und Starrheit erschaffen haben.

Die in *Abbildung 14* aufgeführten Muster Shared Entities und Supporting Service verursachen die größte Kopplung zwischen fachlichen Bausteinen. Bei einfacher und stabiler Fachlichkeit können jedoch auch wertvolle Synergien erzielt werden.

Das *Service-UseCase.In Pattern* koppelt zwei Bausteine ebenfalls sehr stark aneinander. Die Kopplung bezieht sich aber erst mal nur auf zwei Bausteine im Vergleich zum *Shared Entities* oder *Supporting Service Pattern*. Indirekte Kopplungen entstehen, wenn

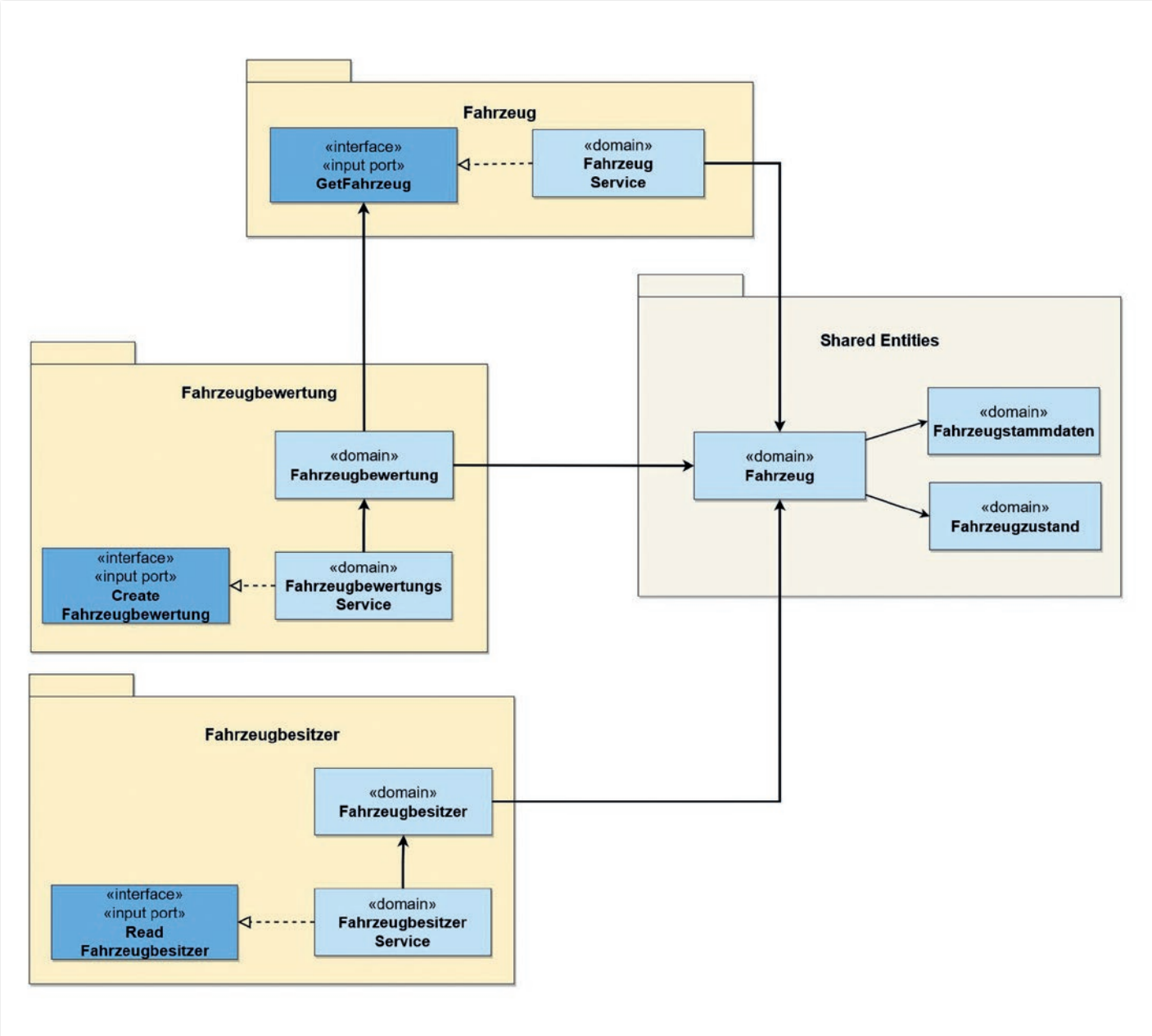

Abbildung 13: Das Shared Entities Pattern am Beispiel der Bausteine Fahrzeug, Fahrzeugbewertung und Fahrzeugbesitzer (© Matthias Eschhold)

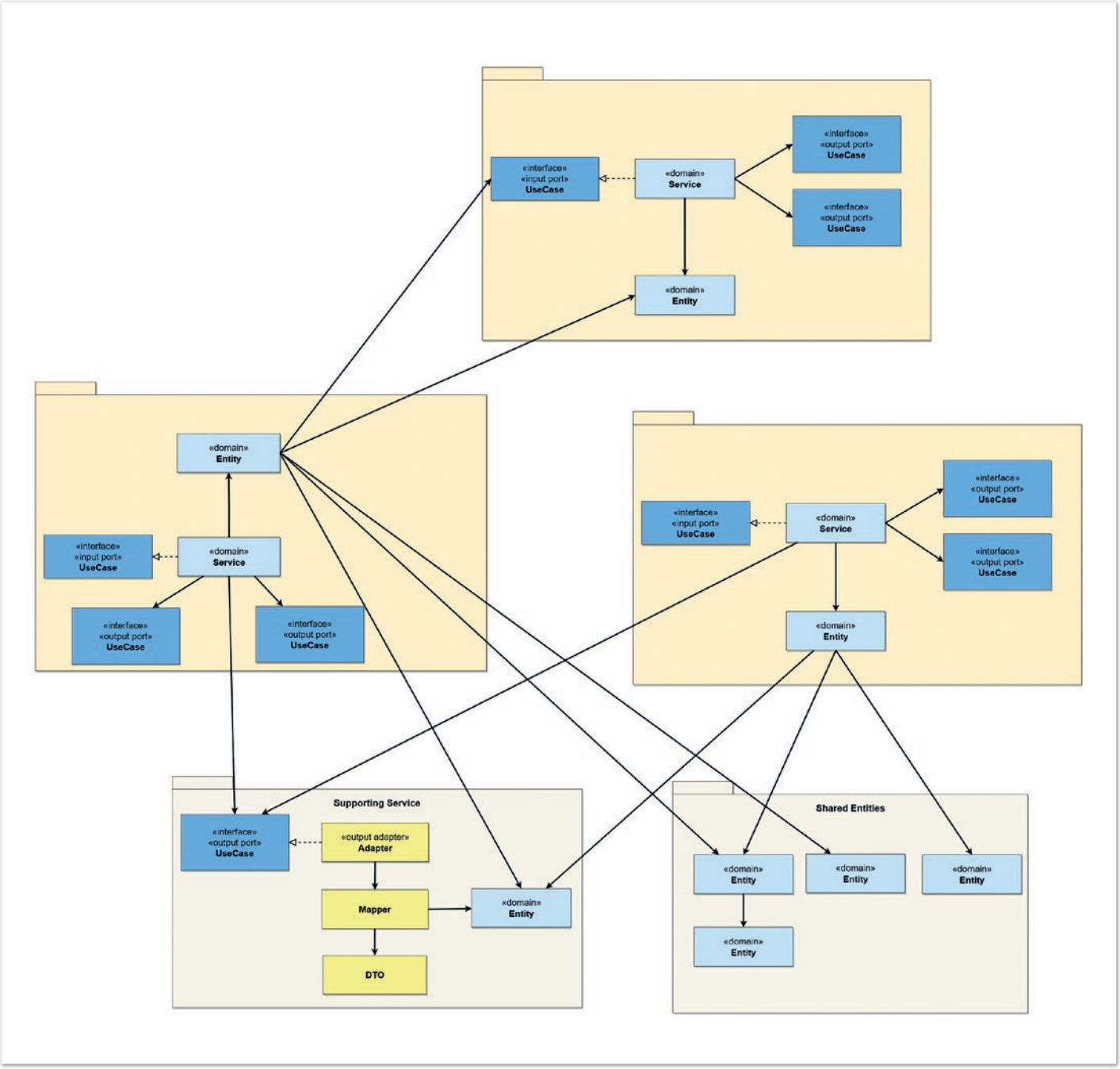

Abbildung 14: Big Ball of Mud 2.0 aufgrund zu vieler Shortcuts (© Matthias Eschhold)

mehrere Konsumenten den Anbieter über das *Service-UseCase.In Pattern* nutzen.

Das *Application Service* und *Adapter.Out-UseCase.In Pattern* verursachen die geringste Kopplung, jedoch aus Sicht der Gegner/innen Overhead in der Implementierung. Im direkten Vergleich gewinnt für mich das *Adapter.Out-UseCase.In Pattern*, weil es einfacher zu einem Event-getriebenen Ansatz umgebaut werden kann und auch eine Zerlegung der Bausteine in unterschiedliche Softwareartefakte besser unterstützt wird.

Was noch offen blieb

Alle vorgestellten Lösungsmuster wurden beschrieben mit Wirkungskreis auf ein Entwicklungsteam. Entscheidungen, insbesondere der Einsatz von Shortcuts, die das Teilen von Entitäten und Funktionen zur Folge haben, sind anders zu bewerten, wenn dies mehrere Entwicklungsteams betrifft. Kennern des Domain-driven Design sind mit Sicherheit einige Parallelen in den Lösungsansätzen aufgefallen. Die Anwendung des Aggregate Design kann die Clean Architecture noch weiter optimieren. Denn grundsätzlich teilen sich das taktische Domain-driven Design und die hier vorgeschlagene Umsetzung der Clean Architecture mit fachlicher Modularisierung das Problem der Eventual Consistency. Mein Tipp zum Umgang mit Eventual Consistency: *Adapter.Out-UseCase.In* oder das *Application Service Pattern*. Auf Basis des Erlernten kann an dieser Stelle auch über die dritte Lösungsvariante, einen Event-getriebenen Ansatz *[7]*, nachgedacht werden.

Was ebenfalls noch offen blieb, ist die Unterstützung der Entwickler/innen bei der Anwendung der Clean Architecture. Mappings können mithilfe von Third-Party-Bibliotheken sehr effizient unterstützt werden. Oft ist dies allerdings mit einer Lernkurve verbun-

den. Automatisiertes Feedback hinsichtlich der Einhaltung von Architekturregeln vereinfacht die Anwendung der Clean Architecture zusätzlich. Insbesondere unterstützt dies die schnelle Korrektur und hilft beim Erhalt der Flexibilität über den Lebenszyklus des Systems hinweg.

Quellen

[1] Tom Hombergs (2019): Get Your Hands Dirty on Clean Architecture. Packt Publishing, Birmingham.

[2] Matthias Geirhos (2015): Entwurfsmuster - Das umfassende Handbuch. Rheinerk Verlag. Bonn.

[3] Alistair Cockburn (2021): Hexagonal architecture. Online. Abgerufen am: 10.12.2021. *https://alistair.cockburn.us/hexagonal-architecture/*

[4] *https://github.com/MatthiasEschhold/clean-architecture-and-flexibility-patterns*

[5] *https://martinfowler.com/bliki/AnemicDomainModel.html*

[6] *https://enterprisecraftsmanship.com/posts/domain-vs-application-services/*

[7] *https://www.baeldung.com/spring-events*

Matthias Eschhold
Novatec Consulting GmbH
matthias.eschhold@novatec-gmbh.de

Warum ist es wichtig, dass Softwarearchitektur aus fachlicher Sicht strukturiert ist? Seit 2014 beantwortet Matthias Eschhold in seinem beruflichen Alltag diese und weitere Fragen zum Architekturentwurf sowie zu Architekturmitteln, Entwurfsprinzipien und -mustern. Sein schönstes Erfolgserlebnis dabei: Zu sehen, dass die getroffenen Maßnahmen und Entscheidungen wirken und Ziele wie Wartbarkeit und Verständlichkeit erreicht werden. Darüber hinaus vermittelt Matthias sein Wissen leidenschaftlich als Trainer für Softwarearchitektur.

Mitglieder des iJUG

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

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.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Björn Bröhl. Die DOAG Deutsche ORACLE-Anwendergruppe 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, Melanie Feldmann, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

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

Bildnachweis:
Titel: Bild © vladgrin
https://123rf.com
S. 10: Bild © Vectorideas
https://stock.adobe.com
S. 17: Bild © Alex
https://stock.adobe.com
S. 23: Bild © Kras99
https://stock.adobe.com
S. 28: Bild © ArtemisDiana
https://stock.adobe.com
S. 36+37: Bild © Malchev
https://stock.adobe.com
S. 45: Bild © DOAG
www.doag.org
S. 46: Bild © Onidji
https://stock.adobe.com
S. 52+53: Bild © Yashkovskiy
https://1stock.adobe.com
S. 60+61: Bild © royyimzy
https://1stock.adobe.com

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org

Mediadaten und Preise:
www.doag.org/go/mediadaten

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

Inserentenverzeichnis

Programm jetzt online!

DOAG 2022

Konferenz + Ausstellung

In Nürnberg

20.-23. SEPT.

Die Oracle-ANWENDERKONFERENZ

anwenderkonferenz.doag.org

Eventpartner: AOUG AUSTRIAN ORACLE USER GROUP · SOUG swiss oracle user group · iJUG Verbund