jax® HYBRID

Software-Architektur für Java-Entwickler

JAX-Dossier 2021

Microservices, Modulithen, Microfrontends, Domain-driven Design, DevOps, CQRS, Reactive, Datenstrukturen

Inhalt

Strategie

KISS of Death by Complexity? 3
von Uwe Friedrichsen

Qualität ist besser! 8
von Marco Schulz

DevOps

Architekturüberprüfung in der Pipeline 12
von Arne Limburg

Modulithen

Architekturpatterns in Modulithen
Teil 1: Ordnung ins Chaos bringen 15
von Arnold Franke

Teil 2: Wer mit wem reden darf 22
von Arnold Franke

Teil 3: Die Bude sauber halten 29
von Arnold Franke

Microfrontends

Architektur für agile Teams 35
von Manfred Steyer

Datenstruktur

Von reaktiven UIs und expliziten Datenflüssen 42
von Hendrik Müller

Verteilte Transaktionen in verteilten Systemen 45
von Michael Hofmann

Domain-driven Design

CQRS und DDD: Gemeinsam mehr erreichen 49
von Golo Roden

Strategisches Mapping mit Wardley Maps 55
von Tom Asel

Das Spiel mit dem Komplexitätsfeuer

KISS of Death by Complexity?

KISS ist das Motto dieser Ausgabe des Java Magazins: Keep it simple, stupid! Auf Deutsch: Mach es nicht unnötig kompliziert. Das klingt gut und erstrebenswert. Aber machen wir in der Praxis nicht eher das Gegenteil davon? Wenn man sich umschaut, dann steht uns die Komplexität bis zum Hals. Und anstatt etwas dagegen zu tun, häufen wir jeden Tag fleißig neue Konzepte, Tools und Technologien obendrauf, getrieben von immer neuen Hypes und Hoffnungen auf Patentrezepte. Zeit für eine kritische Betrachtung.

von Uwe Friedrichsen

Das mit der Komplexität glauben Sie mir nicht (Kasten: „Komplexität“)? Schauen wir doch einmal auf die Zutatenliste für eine relativ überschaubare internetfähige Anwendung. Die liest sich häufig in etwa so: Java/Kotlin, JavaScript, SPA, Angular/React/Vue, npm/Yarn, webpack, Grunt, React Native/Swift/Kotlin, PWA, Microservices, Maven/Gradle, Spring Boot/Micronaut/Quarkus, BFF, API Gateway, REST/GraphQL, IAM (Identity and Access Management), OAuth 2.0, JWT, Keycloak, Docker, Kubernetes, Rancher, Helm, Operators, Service Mesh (Istio/Linkerd), Consul/ZooKeeper/etcd, Prometheus/Graphite, ELK, RDBMS, NoSQL/NewSQL (MongoDB/Cassandra), Event Sourcing, CQRS, Kafka/RabbitMQ, Jenkins/CircleCI, Unit Testing (JUnit/MochaJS), Integration/Contract Testing (FitNesse/Jasmine/Pact), Security Testing (ZAB), Load Testing (JMeter/Gatling/Locust), Chaos Engineering, User Acceptance Testing, Git, Artifactory/Nexus, IaC (Terraform/AWS CloudFormation/Puppet/Ansible), OpenAPI, ...

Da fehlt noch eine dreistellige Anzahl an Bibliotheken und Frameworks, die typischerweise zum Einsatz kommt (keine Übertreibung!). Es fehlt noch eine Menge weiterer Tools und Technologien, die ich hier nicht erwähnt habe. Es fehlen Hunderte an Alternativen zu den Lösungen, die hier aufgezählt sind, die aber häufig alle in der IT eines größeren Unternehmens zum Einsatz kommen, weil jedes Projekt und jeder Software-Engineer andere Präferenzen hat.

Mit IAM und Security Testing wird das hochkritische und hochkomplexe Thema Security hier auch nur angedeutet. Nicht zuletzt fehlen all die anderen Konzepte, Tools und Technologien, die man benötigt, um eine zuverlässig funktionierende Betriebsinfrastruktur aufzubauen und zu betreiben – was problemlos noch einmal eine solche Liste ergeben würde.

Und das alles für eine überschaubare internetfähige Anwendung, sagen wir: ein nicht zu komplexes Kundenportal. Falls Sie glauben, ich übertreibe: Ich habe mehr als ein Projekt in der Größenordnung gesehen, in dem die Buzzword-Bingo-Liste in etwa so ausgesehen hat. Der Unfairness halber: Das Testen war häufig nicht so ausgeprägt. Ansonsten hat die Liste aber gepasst.

Nun betreibt ein Unternehmen in der Regel nicht nur eine Anwendung, sondern eine ganze Menge davon – häufig hunderte bis tausende Anwendungen. Diese Anwendungen basieren natürlich nicht alle auf den gleichen Konzepten, Tools und Technologien wie die oben skizzierte Anwendung. Entsprechend findet man ganz viele unterschiedliche Konzepte, Tools und Technologien in einem wilden Mix und ohne klare Abgrenzung, häufig zurückreichend bis in die 1970er Jahre.

Zusätzlich findet man auch sehr viel Standardsoftware. Das beschränkt sich nicht nur auf das unvermeidliche SAP mit seinen ganzen Modulen in unterschiedlichen Generationen und Ausbaustufen. Man findet Standardprodukte verschiedenster Art, Aufgabe und Größe, auf vielfältige Weise untereinander und mit den selbstentwickelten Systemen integriert.

Dazu passend findet man in der Regel auch verschiedenste Inkarnationen von Integrationslösungen – hier eine EAI-Lösung, dort einen ESB und eine API-Management-Lösung, alle bei Weitem nicht so flächendeckend und einheitlich genutzt wie bei der Einführung versprochen.

In Summe stapelt sich also sehr viel Komplexität in der IT. Und weil die meisten Beteiligten keine Ahnung mehr haben, wie man diesen Wildwuchs in den Griff bekommen könnte, suchen sie stattdessen angestrengt nach der nächsten Silberkugel, die das Problem auf einmal auf magische Weise lösen soll – und erhöhen die Komplexität damit um die nächste Schicht.

Unentbehrliche IT

Das könnte man jetzt je nach Gemütslage schmunzelnd oder kopfschüttelnd zur Kenntnis nehmen und danach wieder zur Tagesordnung übergehen, wenn IT mittlerweile nicht unverzichtbar geworden wäre.

In der Vergangenheit waren die meisten Produkte und Dienstleistungen des täglichen Lebens frei von IT. Einige von ihnen enthielten eine kundenspezifische Hardwarelösung, die einmal entworfen und getestet viele Jahre lang verwendet wurde.

Heute jedoch sind die meisten Produkte und Dienstleistungen zu großen Teilen softwaregestützt oder rein softwarebasiert (sprich als App oder Anwendung realisiert). Das ermöglicht deutlich anspruchsvollere Lösungen als früher, die auch wesentlich schneller verbessert und angepasst werden können. Nicht nur Mobiltelefone und Tablets, auch Autos, Waschmaschinen und Verstärker verlangen heute immer häufiger nach regelmäßigen Softwareupdates.

Das ist aber nur der sichtbare Teil der Produkte und Dienstleistungen. Die meisten davon verlassen sich auf Backend-Dienste, die in einem (Cloud-)Rechenzentrum gehostet werden und dort große Teile der Intelligenz implementieren – noch einmal ganz viel zusätzliche Software.

Ein Großteil der sogenannten B2C-(Business-to-Customer-)Interaktion erfolgt heute über Software oder wird zumindest von Software unterstützt – Tendenz steigend: Onlinehandel, Onlinebanking, Onlineversicherung, Onlinemedien und, und, und, bis hin zur Onlinestadtverwaltung.

Wenn wir uns die B2B-(Business-to-Business-)Interaktion anschauen, sehen wir uns einer noch stärker softwaregestützten Kommunikation gegenüber. Auch intern (B2E, Business to Employee) sind die meisten Unternehmen in hohem Maße von Software abhängig. Das beschränkt sich nicht nur auf die Büros. Software schleicht sich auch immer mehr in die Fertigungsstraßen und Werkstätten ein, wo die Produkte gebaut und gewartet werden. Zusammenfassend gilt: Software umgibt uns heutzutage. Software ist überall.

Das bedeutet aber auch, dass immer mehr Aspekte unseres täglichen Geschäfts- und Privatlebens auf funktionierende Software angewiesen sind. Software ist unentbehrlich geworden. Wenn die Software nicht funktioniert, wenn die entsprechenden Anwendungen ausfallen, haben wir ein Problem.

Ein selbstverstärkender Teufelskreis

Wir können also zwei Entwicklungen beobachten:

- IT wird immer komplexer.
- IT wird immer unentbehrlicher.

Wenn man diese zwei Entwicklungen nebeneinanderstellt, dann bilden sich zumindest auf meiner Stirn Sorgenfalten. IT muss immer zuverlässiger und robuster werden, weil sie immer unverzichtbarer wird. Auf der anderen Seite wächst ihre Komplexität unaufhörlich.

Das deckt sich auch mit den Beobachtungen, die ich immer wieder mache. Viele IT-Abteilungen bewegen sich bereits am Rande der Unwartbarkeit. Da gibt es kritische Anwendungen, die nicht mehr angefasst werden, weil niemand mehr weiß, was darin passiert. Da wird versucht, dringend benötigte Integrationen in die Core-Systeme per RPA (Robot Process Automation) zu realisieren, weil alle Angst haben vor dem, was passiert, wenn man den alten Code noch einmal anfassen würde – sofern man das System aus dem Quelltext überhaupt noch bauen könnte. Da laufen uralte, seit Jahren ungepatchte Betriebssystemversionen (zur großen Freude aller Hacker da draußen), weil die dringend benötigte Standardsoftware für neuere Versionen des Betriebssystems nicht mehr erhältlich ist. Und vieles mehr …

Da bekommt das vielzitierte „Never touch a running system“ einen ganz anderen, bitteren Beigeschmack. Das ist nicht nur riskant für die Unternehmen. Auch die betroffenen Mitarbeiter leiden darunter. Während sie versuchen, mit viel Energie das wacklige Konstrukt irgendwie aufrechtzuerhalten, werden die Fachbereiche immer unzufriedener, weil Änderungen immer länger dauern, und erhöhen entsprechend kontinuierlich den Veränderungsdruck.

Damit entziehen sie den betroffenen Mitarbeitern die dringend benötigte Zeit, zumindest die schlimmsten Sollbruchstellen reparieren zu können. Das resultiert häufig in einem sich selbstverstärkenden Teufelskreis:

1. Die IT wird immer komplexer, weshalb Änderungen immer länger dauern.
2. Die Fachbereiche werden immer unzufriedener, weshalb sie den Lieferdruck immer weiter erhöhen.
3. Änderungen müssen immer häufiger mit der heißen Nadel gestrickt werden, um dem Lieferdruck gerecht zu werden, was die Komplexität weiter erhöht.

Wenn dann zusätzlich wohlmeinende Kolleg*innen mit immer neuen Konzepten, Tools und Technologien um die Ecke kommen, weil sie von einer gut geölten Hypeverkaufsindustrie fleißig eingeflüstert bekommen, dass mit Technologie X garantiert alles „endlich richtig gut“ wird … Nun ja, dann wird meistens gar nichts gut, sondern der Komplexitätsturm nur um eine Etage erhöht.

Für nicht wenige fühlt sich das wie ein kafkaesker Alptraum an, der irgendwann auch auf die Gesundheit schlägt.

Komplexität

In diesem Artikel verwende ich den Begriff „Komplexität“ eher umgangssprachlich, also im Sinne von „nicht einfach“. In anderen Zusammenhängen werden die Begriffe „kompliziert“ und „komplex“ (auch von mir) strikt unterschieden. In diesem Artikel hingegen werden sie synonym verwendet.

Ganz viele Treiber

Aber eigentlich will das doch niemand. Niemand wollte immer mehr Komplexität, eigentlich wollen es alle doch viel lieber einfach … KISS halt. Das ist auch viel günstiger, schneller, besser zu ändern und macht mehr Spaß. Aber wenn es niemand je wollte, wie sind wir dann da hingekommen?

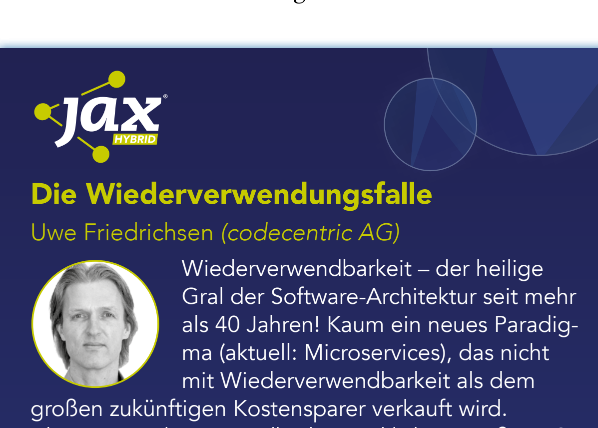
Treten wir also ein paar Schritte zurück und schauen ganz nüchtern auf die Situation. Welche Treiber haben in die Komplexität geführt und was können wir dagegen machen? Tun wir das, stellen wir – wenig überraschend – fest, dass auch die Treiber komplex sind und sich gegenseitig beeinflussen:

- Das beginnt damit, dass sich die Marktbedingungen geändert haben, man auf Unternehmensebene aber fälschlicherweise mit „mehr von dem alten Bewährten“ reagiert.
- Das wird massiv dadurch verstärkt, dass die meisten Leute – nicht nur, aber insbesondere auch Entscheider (häufig selbst innerhalb der IT) – nicht wirklich IT an sich und insbesondere die Besonderheiten von Software gegenüber anderen „Werkstoffen“ verstehen.
- Ebenso wenig verstehen sie, welche Rolle IT heute in der Wertschöpfung hat (Stichwort: Digitale Transformation) und versuchen entsprechend, an sinnlosen Stellen den letzten Cent aus der IT weg zu sparen, da sie lediglich als Cost Center betrachtet werden.
- Es geht weiter mit nicht verstandenen und stumpf, aber falsch kopierten „Agile“- oder „DevOps“-Cargo-Kulten, die Komplexität erhöhen, ohne Mehrwert zu liefern.
- Da werden Architekturen von Hyperscalern sinnbefreit kopiert, auch wenn man eigentlich ganz andere Anforderungen und Herausforderungen als die Hyperscaler hat.
- Da werden regelmäßig großen Architekturneubauinitiativen gestartet, mit denen die IT „endlich mal komplett auf Linie gebracht“ werden soll, die aber immer auf halbem Weg an der Realität scheitern.
- Ideologische OSS-Debatten gepaart mit obskuren Vendor-Lock-in-Diskussionen bei offensichtlich nicht verstandenen TCO-Modellen machen es auch nicht leichter, sondern führen meist nur zu mehr nicht erforderlicher Komplexität. Es ließen sich komplette Artikel über die Unsinnigkeit von Aussagen wie „OSS kostet nichts“ oder „Mit OSS habe ich keinen Lock-in“ schreiben – das würde den Rahmen dieses Texts bei Weitem sprengen.
- Ebenso wenig verbessert sich die Situation durch in Entwicklerkreisen häufig zu beobachtendes eskapistisches Qualitätstheater. Das entsteht u. a. dadurch, dass die meisten Unternehmen Entwickler systematisch aller intrinsischen Motivationsfaktoren beraubt haben. Sie stürzen sich deshalb zur Erhaltung ihres Selbstwertgefühls (verständlicherweise) gerne auf Ersatzbestätigungen – leider häufig mit fast schon religiösem Eifer. Das fühlt sich dann zwar gut für die betreffende Person an, schießt aber meistens deutlich über das Ziel hinaus. Am Ende ist die Komplexität deutlich stärker gewachsen als der zusätzliche Nutzen.
- Nicht zuletzt verstärken wir unsere Probleme in der IT mit unserem schon fast krankhaften Jugendwahn. „Neu ist gut, alt ist schlecht“ wenden wir leider auch regelmäßig auf das Wissen an, das wert wäre, als IT Body of Knowledge konserviert zu werden – eben so, wie es echte Ingenieursdisziplinen machen. Stattdessen werfen wir das mühsam erworbene Wissen regelmäßig kollektiv wieder weg, um der nächsten Silberkugel am Horizont hinterherzurennen – weil sie neuer und deshalb garantiert besser sein muss. Willkommen bei der nächsten Komplexitätsschicht.

Jeder dieser Treiber (und noch ein paar hier nicht genannte) ist mindestens einen ganzen Artikel wert. Aus Platzgründen beschränke ich mich hier darauf, nur einen der Treiber zu skizzieren.

Beispiel: Kopierte Hyperscaler-Architekturen

Diverse Hyperscaler (Amazon, Netflix und Co.) stießen auf Schwierigkeiten, die sehr spezifisch für sie waren. Sie hatten Probleme bzgl. Liefergeschwindigkeit und Skalierung, die vor ihnen in dieser Form noch niemand hatte. Aus diesem Grund haben sie viele Teile ihrer IT komplett neu denken und konzipieren müssen. Dabei entstand u. a. das Konzept der Microservices, weil es ihnen in Kombination mit einer ganzen Reihe weiterer Maß-



Die Wiederverwendungsfalle

Uwe Friedrichsen *(codecentric AG)*

Wiederverwendbarkeit – der heilige Gral der Software-Architektur seit mehr als 40 Jahren! Kaum ein neues Paradigma (aktuell: Microservices), das nicht mit Wiederverwendbarkeit als dem großen zukünftigen Kostensparer verkauft wird. Aber ist Wiederverwendbarkeit wirklich so großartig? Ist es das, was wir in unseren Designs immer anstreben sollten? Oder könnte sie an manchen Stellen gar kontraproduktiv sein? In diesem Vortrag werden wir mit einigen Mythen in Bezug auf Wiederverwendbarkeit aufräumen: Wir werden lernen, warum die meisten Argumentationen fehlerhaft sind, dass Wiederverwendbarkeit einen hohen Preis hat, warum sie sich nur selten bezahlt macht, wann sie ein falscher Freund ist und mehr. Außerdem lernen wir zu unterscheiden, wann man auf Wiederverwendbarkeit setzen sollte und wann nicht – und was man dann besser tun sollte. Nach diesem Vortrag werden Sie ein besseres Verständnis dafür haben, wann Sie Wiederverwendbarkeit in Ihren Designs anstreben sollten, wann nicht und was Sie stattdessen tun sollten.

nahmen half, neue Ideen deutlich schneller zu erproben und ihre Skalierungsprobleme besser zu adressieren.

Sie haben aber auch gelernt, dass Microservices einen hohen Preis haben. Sie mussten auf Anwendungsebene mit der gesamten Komplexität verteilter Systeme umgehen. Das bedeutete im Prinzip, dass sie lernen mussten, Anwendungen ganz anders zu designen, zu implementieren, zu testen, zu deployen und zu betreiben. Insbesondere für den zuverlässigen Betrieb haben sie in der Regel hunderte oder tausende an Personenjahren investiert, um die dafür erforderliche Infrastruktur aufzubauen und zu betreiben. Trotzdem war es ihnen der Aufwand wert, weil in ihrer speziellen Situation die Vorteile die Nachteile überwogen haben.

Dann wurden Microservices populär. Die Engineers der Hyperscaler begannen, über ihre Konzepte zu erzählen. Sie waren sicherlich auch berechtigterweise ein wenig stolz auf das, was sie geschaffen haben. Das hörten dann die IT-Mitarbeiter traditioneller Unternehmen: schneller liefern, skalierbar und ziemlich cool. Das wollten sie auch. Schneller war ja eh ein leidiges Thema – die Fachbereiche saßen ihnen dauernd im Nacken. Und nur noch Services, die gerade mal so groß sind, dass man sie als einzelne Person verstehen kann? Mit einem Blick auf die angehäufte Komplexität im eigenen Unternehmen klang das wie die langersehnte Silberkugel.

Als dann noch jemand „wiederverwendbar“ sagte (das seit Jahrzehnten in vergleichbaren Kontexten immer wieder gemachte und nie eingelöste Versprechen, mit dem sich alle erforderlichen Investitionen in kürzester Zeit amortisieren würden), gab es kein Halten mehr. Es gab praktisch kein neues Projekt ohne Microservices-Architektur mehr.

Leider wurde nie die Frage gestellt, ob Microservices überhaupt ein Problem lösen, das man selbst hat. Letztlich zahlt man für sie einen hohen Preis (siehe oben). Da sollte man erwarten, dass es eine sorgfältige Abwägung gibt, ob es das wert ist.

Gab es aber nach meiner Erfahrung leider nie. Da wurde sehr viel über Skalierbarkeit geredet, auch wenn praktisch kein Unternehmen Skalierungsanforderungen hat, die sich nicht auch mit deutlich einfacheren Architekturen lösen lässt. Da wurde viel über „einfacher“ geredet, ohne zu bedenken, dass solche Architekturen in Summe deutlich komplexer sind. Da wurde leichtere Technologiemigration ins Spiel gebracht, ohne zu bedenken, dass die bestehenden Migrationsprobleme zum geringsten Teil in den verwendeten Architekturstilen begründet liegen. Und so weiter.

Was bleibt, wenn man ganz ehrlich ist? Für die meisten Unternehmen fast nichts außer sehr viel zusätzlicher Komplexität durch unreflektiertes Kopieren der Konzepte der Hyperscaler. Die eigenen Probleme hätte man mit deutlich einfacheren Mitteln lösen können.

Das bedeutet übrigens nicht, dass Microservices grundsätzlich schlecht wären. Ganz im Gegenteil: Sie sind ein großartiger Architekturstil ... für eine spezielle Art von Problemen. In allen anderen Fällen bleibt am Ende nur sehr viel zusätzliche Komplexität, ohne dass irgendein Problem wirklich gelöst wäre.

Den Kreis durchbrechen

Das war jetzt ein Treiber, kurz skizziert. Daneben gibt es wie oben beschrieben ganz viele weitere Treiber, die sich gegenseitig beeinflussen und verstärken, und nicht selten stapeln wir selbst – mit den besten Absichten – weitere Komplexität auf. Was können wir tun, damit es besser wird?

Ich glaube nicht, dass es eine einfache Antwort auf diese Frage gibt. Gäbe es sie, hätten wir sie (hoffentlich) schon lange gefunden und umgesetzt. Was ich stattdessen anbieten möchte, sind ein paar Empfehlungen, die kein Allheilmittel sind, aus meiner Erfahrung aber einen guten Startpunkt bieten:

- *Fragen „Warum?“*: Wenn jemand mit einer kompliziert aussehenden Lösung um die Ecke kommt, „Warum?“ fragen. Warum diese Lösung? Was wollen wir erreichen? Wie trägt die Lösung dazu bei, das Problem zu lösen? Ginge es auch anders? Ginge es auch einfacher? Die Frage „Warum?“ schafft Fokus. Viel zu häufig tun wir Dinge mehr oder minder reflexartig, ohne zu fragen, warum wir sie machen. Ein kritisches „Warum?“ zur rechten Zeit kann viele unnötig komplexe Irrwege verhindern.
- *Besser beraten*: Ganz viel Komplexität entsteht dadurch, dass Entscheider die Konsequenzen ihrer Entscheidungen nicht richtig bewerten können, sei es durch fehlendes Wissen bzgl. diverser Facetten der IT, sei es durch Empfehlungen fragwürdiger „Experten“. Hier ist es unsere Aufgabe, diesen Entscheidern Alternativen anzubieten und deren Vor- und Nachteile klar und objektiv herauszuarbeiten und zu kommunizieren – in der Sprachwelt der Entscheider und nicht in kryptischem IT-Speak. Aus meiner Erfahrung wirkt das echte Wunder.
- *Ganzheitlich denken*: Eine IT-Lösung betrifft immer viele Stakeholder und das, was für eine Gruppe ein Vorteil ist, kann für eine andere Gruppe ein Nachteil sein. Deshalb ist es wichtig, Lösungsideen nicht nur aus Entwicklersicht zu bewerten, sondern auch die anderen betroffenen Gruppen mit zu betrachten, z. B. Betrieb, Endnutzer, Fachbereich und so weiter. Häufig stellt man bei einer solchen ganzheitlichen Betrachtung fest, dass die lokal attraktiv erscheinende Lösung in Summe aber gravierende Nachteile und häufig viel zusätzliche Komplexität mit sich bringt.
- *Optionen kennen*: Um verschiedene Lösungsideen gegeneinander abwägen zu können, muss man erst einmal mehr als eine Option kennen. Wenn ich mich immer nur in der gleichen vertrauten Ecke bewege, dann werde ich immer nur die eine Lösungsidee haben. Die Tools variieren vielleicht im Detail, aber eigentlich ist es immer die gleiche Lösung. Deshalb ist es wichtig, unterschiedliche Optionen zu kennen, ohne dem Drang zu erliegen, sie auch alle einsetzen zu wollen. So kann man etwa einen Java/OSS-Stack verwenden – kennen wahrscheinlich alle von uns. Aber wie sieht es z. B. mit Low-Code-Lösungen aus? Wann

wären die eine sinnvolle Option? Wann nicht? Oder einen Managed Service nutzen, statt die Lösung selbst zu bauen? Das kann ich aber nur in die Überlegungen einbeziehen, wenn ich mich auch einmal unvoreingenommen mit den Optionen beschäftigt habe.

- *Hypes kritisch hinterfragen*: Das sollte eigentlich klar sein. Allerdings kämpfen wir hier gegen eine mit ganz vielen Millionen Euro gut geschmierte Marketingindustrie, die uns stets das jeweils neueste Wundermittel schmackhaft machen will. Da werden ganz viele Versprechen gemacht, da buhlt die auf Hochglanz polierte Website mit dem einfachen „Getting started" um unsere Gunst, da erzählt uns der Developer Advocate perfekt verpackt von einer wunderbaren Zukunft. Eine ganze Industrie will, dass wir auf den Hypezug aufspringen. Da ist es häufig schwierig, die notwendige kritische Distanz zu wahren, um Nutzen und Preis bezogen auf das konkrete Problem nüchtern herauszuarbeiten – müssen wir aber, wenn es nicht immer komplexer werden soll.
- *Ockhams Rasiermesser ansetzen*: Wenn wir mehrere Optionen zur Lösung eines Problems haben und die einfachste der Lösungen keine offensichtlichen Nachteile hat, sollten wir diese wählen. Menschen haben einen sogenannten „Complexity Bias", sprich: Wir neigen dazu, komplexere Lösungen einfachen Lösungen vorzuziehen. In der IT scheint dieser Bias sehr ausgeprägt zu sein und sorgt häufig für zusätzliche Komplexität. Deshalb bei der nächsten Lösungsidee vielleicht mal das Rasiermesser ansetzen und fragen, ob es möglicherweise eine einfachere Lösung für das Problem gibt.

Wie geschrieben, das sind keine Allheilmittel und sie sind häufig auch nicht so einfach umsetzbar, wie es auf den ersten Blick erscheinen mag. Mit etwas Augenmaß angewandt (bitte keinen Dogmatismus, das stiftet immer mehr Schaden als Nutzen) können diese Empfehlungen aber durchaus helfen, eine Menge unnötiger Komplexität zu vermeiden.

Der Fairness halber aber noch eine kurze Warnung: So eine Denkweise ist nicht populär. Das führt zu unspektakulären Lösungen. Damit kann man sich keine Denkmäler bauen. Das läuft gegen das verbreitete Verständnis „guter" (komplexer) Lösungen – von der Hypeindustrie, die von ständig neuer Komplexität lebt, einmal ganz zu schweigen. Sie werden daher bei aller vordergründigen Zustimmung damit rechnen müssen, keine offenen Türen einzurennen, sondern auf viele versteckte und offene Widerstände zu stoßen.

Fazit

Fassen wir kurz zusammen: Wir können beobachten, dass IT gleichzeitig immer komplexer und immer unentbehrlicher wird. Das führt zu immer mehr Problemen in der Entwicklung und dem Betrieb von Anwendungen. Gleichzeitig wird der Lieferdruck durch die Fachbereiche immer stärker, was zu einem selbstverstärkenden Teufelskreis führt, der für die beteiligten Personen sehr negative Folgen bis hin zu Gesundheitsproblemen hat.

Die Gründe für diese sich immer weiter auftürmende Komplexität sind ebenfalls komplex und reichen von falschen Reaktionen auf veränderte Markterfordernisse bis hin zu nicht zielführenden Kompensationshandlungen auf Entwicklerebene. Häufig führt ein fehlendes Verständnis bzgl. der Konsequenzen von Entscheidungen dazu, dass trotz bester Absichten immer mehr Komplexität angehäuft wird.

Ich habe ein paar Empfehlungen gegeben, die mit Augenmaß angewandt helfen können, unnötige Komplexität besser zu erkennen und zu vermeiden. Diese sind aber kein Patentrezept und aufgrund der geringen Wertschätzung von einfachen Lösungen in der IT-Branche rennt man damit vermutlich keine offenen Türen ein.

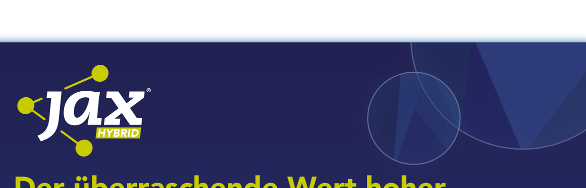
Trotzdem gilt: Zu viel Komplexität macht keinen Spaß. Deshalb sollten wir alle daran arbeiten, etwas gegen zu viel Komplexität zu tun – denn Arbeit ohne Spaß? Das will doch wirklich niemand!

Falls Sie mehr Informationen zu den Aussagen aus diesem Artikel suchen, können Sie die Blogserie lesen, die ich dazu verfasst habe [1].

Uwe Friedrichsen ist ein langjähriger Reisender in der IT-Welt. Als CTO der codecentric AG ist er stets auf der Suche nach innovativen Ideen und Konzepten. Seine aktuellen Schwerpunktthemen sind (verteiltes) Systemdesign und die IT von (über)morgen. Er teilt und diskutiert seine Ideen regelmäßig auf Konferenzen, als Autor von Artikeln, Blogposts, Tweets und im direkten Gespräch.

Links & Literatur

[1] Friedrichsen, Uwe: „Simplify! – Part 1": https://www.ufried.com/blog/simplify_1/



Der überraschende Wert hoher Geschwindigkeit in der IT

Uwe Friedrichsen *(codecentric AG)*

Untertägige Lead Times? Eine neue Idee in nur wenigen Stunden zu Ihren Anwendern bringen? Irrelevant für Sie? Deployment einmal pro Woche ist gut genug? Das denken immer noch viele Unternehmen in Deutschland. In dieser Session werden wir diskutieren, warum diese Vorstellung falsch ist. Wir werden erörtern, dass untertägige Lead Times eigentlich grundsätzlich unvermeidlich sind, selbst wenn Sie nicht in einem hochdynamischen Markt leben. Wir werden sehen, wie Sie viel Geld sparen, flexibler werden, Ihre Organisation und Prozesse vereinfachen, Stress (!) abbauen und vieles mehr können, einfach nur indem Sie sich für untertägige Lead Times entscheiden. Lassen Sie sich überraschen und bringen Sie sich in Position für einen Geschwindigkeitsschub.

Nichtfunktionale Anforderungen: Struktur, Organisation und Wartung

Qualität ist besser!

Um der Definition von Codequalität gerecht zu werden, genügt es nicht, sich einzig auf den Quelltext zu konzentrieren. Auch Struktur, Organisation und Wartung spielen eine wichtige Rolle.

von Marco Schulz

Aus Erfahrung wissen die meisten von uns, wie schwierig es ist, auszudrücken, was wir unter Qualität verstehen. Warum ist das so? Es gibt viele verschiedene Ansichten zum Thema Qualität, und jede von ihnen hat ihre Bedeutung. Im Kontext eines Projekts muss die Definition von Qualität mit den Bedürfnissen und dem Budget des Projekts übereinstimmen. Der Versuch, Perfektionismus zu erreichen, kann kontraproduktiv sein, wenn ein Projekt erfolgreich beendet werden soll. Wir werden unsere Überlegungen auf der Grundlage eines 1976 von Barry W. Boehm verfassten Aufsatzes [1] beginnen. Boehm beleuchtet die verschiedenen Aspekte der Softwarequalität und den zugehörigen Kontext. Schauen wir uns das etwas genauer an.

Wenn wir über Qualität sprechen, sollten wir uns auf drei Bereiche konzentrieren: Codestruktur, Korrektheit der Implementierung und Wartbarkeit. Viele Manager kümmern sich nur um die ersten beiden Aspekte, der Bereich Wartung wird von ihnen meist vollständig vernachlässigt. Das ist gefährlich, da das Risiko einer Investition für Unternehmen in die individuelle Entwicklung einer Anwendung nicht eingegangen wird, um diese nur für kurze Zeit in Produktion zu nutzen. Abhängig von der Komplexität der Anwendung kann der Preis für die Erstellung leicht mehrere Hunderttausend Euros erreichen. Da ist es mehr als verständlich, dass der erwartete Geschäftsnutzen solcher Aktivitäten als hoch eingeschätzt wird. Eine Lebensdauer von zehn Jahren und mehr in der Produktion ist üblich. Um die Vorteile auch künftig zu sichern, sind Anpassungen obligatorisch. Das impliziert einen starken Fokus auf die Wartung. Sauberer Code bedeutet nicht, dass sich Ihre Anwendung einfach ändern kann. Ein sehr leicht verständlicher Artikel [2], der dieses Thema berührt, wurde von Dan Abramov geschrieben. Bevor wir weiter darauf eingehen, wie Wartung definiert werden kann, werden wir den ersten Punkt diskutieren: die Struktur.

Einrüstarbeiten

Ein häufig unterschätzter Aspekt in Entwicklungsabteilungen ist ein fehlender Standard für Projektstrukturen. Eine klare Definition, wo Dateien abgelegt werden müssen, hilft Teammitgliedern, sogenannte Points of Interest (POI) schnell zu identifizieren. Eine solche Metastruktur für Java-Projekte wird beispielsweise vom Build-Werkzeug Maven definiert. Vor mehr als einem Jahrzehnt haben Unternehmen den Einsatz von Maven in ihren Projekten ausprobiert und das Tool an die vorhandene Verzeichnisstruktur angepasst. Das führte zu umfangreichen Wartungsarbeiten, da im Laufe der Zeit immer mehr Infrastrukturtools für die Softwareentwicklung hinzugekommen sind, wie beispielsweise SonarQube zur Codeanalyse. Solche Tools arbeiten nach dem von Maven definierten Standard. Das bedeutet, dass jede Anpassung des Standards an eigene Strukturen den Erfolg der Integration neuer Tools oder des Austauschs eines vorhandenen Tools gegen ein anderes beeinflusst.

Ein weiterer zu betrachtender Aspekt ist die unternehmensweit definierte Metaarchitektur. Wenn möglich, sollte jedes Projekt der gleichen Metaarchitektur folgen. Das reduziert die Zeit, die ein neuer Entwickler benötigt, um einem bestehenden Team beizutreten und produktiv zu werden. Eine solche Metaarchitektur muss für Adaptionen offen sein, was in zwei einfachen Schritten erreicht werden kann:

- Kümmern Sie sich nicht um zu viele Details.
- Folgen Sie dem KISS-Prinzip (Keep it simple, stupid!).

Ein klassisches Muster, das gegen das KISS-Prinzip verstößt, besteht darin, dass Standards stark angepasst werden. Ein sehr gutes Beispiel für die fatalen

Auswirkungen einer starken Anpassung beschreibt George Schlossnagle [3]. In Kapitel 21 seines Buchs erklärt er die Probleme, die für ein Team entstanden, als der ursprüngliche PHP-Kern angepasst wurde, anstatt dem empfohlenen Weg zu folgen und Erweiterungen zu nutzen. Das führte dazu, dass jedes Update der PHP-Version manuell bearbeitet werden musste, um die eigenen Entwicklungen in den neuen Kern aufzunehmen. In diesem Zusammenhang bilden die besprochenen Maßnahmen hinsichtlich Struktur, Architektur und KISS bereits drei Qualitätsziele, die einfach umzusetzen sind.

Das auf GitHub gepostete Open-Source-Projekt TP-CORE [4] befasst sich mit den oben genannten Punkten: Struktur, Architektur und KISS. Dort wird gezeigt, wie Sie den beschriebenen Ansatz in die Praxis umsetzen können. Diese kleine Java-Bibliothek hält die Maven-Konvention mit ihrer Verzeichnisstruktur streng ein. Das vereinfacht unter anderem auch die Komplexität der zu schreibenden Build-Logik. Zur schnellen Kompatibilitätserkennung werden Releases durch semantische Versionierung [5] definiert. Als Architektur wurde das Schichtenmodell gewählt, das wir nachfolgend in den wichtigsten Punkten erläutern. Eine ausführliche Beschreibung findet sich unter [6].

Die wichtigsten Architekturentscheidungen basieren auf folgenden Überlegungen: Jede Ebene wird durch ein eigenes Paket definiert und die Dateien folgen ebenfalls einer strengen Regel. Es wird kein spezielles Prä- oder Postfix verwendet. Die Logger-Funktionalität wird beispielsweise von einer Schnittstelle namens *Logger* und der entsprechenden Implementierung *LogbackLogger* deklariert. Die APIs können im Paket business und in den Implementierungsklassen im Paket application gefunden werden. Namen wie *ILogger* und *LoggerImpl* sollten vermieden werden. Stellen Sie sich ein Projekt vor, das vor zehn Jahren gestartet wurde und dessen *LoggerImpl* auf Log4j basiert. Jetzt entsteht eine neue Anforderung und das Loglevel muss zur Laufzeit aktualisiert werden. Um diese Herausforderung zu lösen, könnte die Log4j-Bibliothek durch Logback ersetzt werden. Jetzt ist es verständlich, warum es eine gute Idee ist, die Implementierungsklasse wie die Schnittstelle zu benennen und das mit den Implementierungsdetails zu kombinieren – es erleichtert die Wartung erheblich! Gleiche Konventionen finden Sie auch im Java-Standard-API. Das Interface *List* wird von einer *ArrayList* implementiert. Offensichtlich ist die Schnittstelle nicht als *IList* und die Implementierung nicht als *ListImpl* benannt.

Zusammenfassend wurde hier ein vollständiger Regelsatz definiert, der auch messbar ist und unser Verständnis der strukturellen Qualität beschreibt. Erfahrungsgemäß sollte diese Beschreibung kurz sein. Wenn andere Menschen Ihre Absichten leicht nachvollziehen können, akzeptieren sie eine Empfehlung eher, da sie unter den genannten Umständen einleuchtend ist. Darüber hinaus erkennt der Architekt Regelverstöße weitaus schneller.

Erfolgsmesslatte

Der schwierigste Teil ist es, sauberen Code über einen langen Zeitraum zu erhalten und vor Erosion zu schützen. Einige Ratschläge sind an sich nicht schlecht, aber im Kontext Ihres Projekts möglicherweise nicht so nützlich. Daher habe ich nachfolgend einige Grundregeln zusammengestellt, die in der Praxis meist schon angewendet werden.

Meiner Meinung nach wäre die wichtigste Regel, immer die Compilerwarnung zu aktivieren, egal welche Programmiersprache Sie verwenden. Alle Compilerwarnungen müssen behoben werden, wenn eine Version vorbereitet wird. Organisationen wie die NASA, die sich mit kritischer Software befassen, wenden diese Regel in ihren Projekten strikt an, was zu äußerst erfolgreichen Resultaten führt.

Codierungskonventionen zu Klassenbenennung, Zeilenlänge und API-Dokumentation wie Javadoc können von Tools wie Checkstyle einfach definiert und eingehalten werden. Dieser Prozess kann während des Builds vollautomatisch ausgeführt werden. Aber Achtung: Selbst wenn die Codeprüfungen ohne Warnungen bestanden werden, bedeutet das nicht, dass alles optimal funktioniert. Javadoc ist beispielsweise problematisch. Mit einem automatisierten Checkstyle kann sichergestellt werden, dass die API-Dokumentation vorhanden ist, obwohl wir keine Ahnung von der Qualität der Beschreibungen haben.

An dieser Stelle sollte es nicht erforderlich sein, die Vorteile von Tests zu erörtern. Lassen Sie mich lieber

Frontend: State Management in der Praxis

Karsten Sitterberg (Freiberufler)

Wird ein gewohnter Frontend-Stack wie JSF oder Spring MVC durch eine Browseranwendung mit clientseitigem Zustand abgelöst, ändert sich nicht nur die Technologie: Hatte bisher der Server stets eine vollständige Repräsentation des Anwendungszustands, werden nun lediglich einzelne API-Aufrufe getätigt. Welchen Zustand sich das Frontend daraus zusammensetzt, welche weiteren APIs integriert werden und wann Aktualisierungen erfolgen, obliegt allein dem Frontend. Um das Potenzial der Architektur auszuschöpfen, gilt es, die Herausforderungen und Lösungsoptionen zu verstehen. In diesem Vortrag wird vermittelt, wieso auch bei modularen Anwendungen ein zentraler Zustand Vorteile bieten kann. Es wird diskutiert, wie typische Muster bei SPA-Architekturen aussehen und wie sie sich auf Aspekte wie Wartbarkeit und Testbarkeit auswirken. Als Referenztechnologien werden Angular und React herangezogen, um anhand von Beispielen die konkrete Umsetzung zu demonstrieren.

einen Überblick über die Testabdeckung geben. Der Industriestandard von 85 Prozent Testabdeckung sollte eingehalten werden, da eine Abdeckung von weniger als 85 Prozent die komplexen Teile Ihrer Anwendung nicht erreicht. Eine hundertprozentige Abdeckung belastet lediglich Ihr Budget, ohne dass sich daraus weitere Vorteile ergeben. Ein Paradebeispiel dafür ist das TP-CORE-Projekt, dessen Testabdeckung meist zwischen 92 und 95 Prozent liegt. Das wurde getan, um zu verdeutlichen, ab wann sich mit vertretbarem Aufwand keine aussagekräftigen Tests mehr schreiben lassen. Auch Mock-Objekte würden keine weitere Aussagekraft bei den Tests bringen.

Wie bereits erläutert, enthält die Businessschicht nur Schnittstellen, die das API definieren. Diese Ebene ist ausdrücklich von der Code Coverage ausgeschlossen. Ein anderes Paket heißt internal und enthält versteckte Implementierungen wie den SAX *DocumentHandler*. Aufgrund der Abhängigkeiten, an die der *DocumentHandler* gebunden ist, ist es selbst mit Mocks sehr schwierig, diese Klasse direkt zu testen. Das ist an sich auch völlig unproblematisch, da ihr Zweck nur der interne Gebrauch ist. Darüber hinaus wird die Klasse von der Implementierung des *DocumentHandler* implizit getestet. Um eine höhere Abdeckung zu erreichen, könnte es auch eine Option sein, alle internen Implementierungen von Überprüfungen auszuschließen. Es ist jedoch immer eine gute Idee, die implizite Abdeckung dieser Klassen zu beobachten, um Aspekte zu erkennen, die sonst möglicherweise nicht erkannt werden.

Neben den Unit-Tests auf niedriger Ebene sollten auch automatisierte Abnahmetests [7] durchgeführt werden. Wenn Sie diese Punkte genau beachten, können Sie eine Vielzahl von Problemen vermeiden. Aber vertrauen Sie vollautomatischen Prüfungen niemals blind: Regelmäßig wiederholte manuelle Codeinspektionen sind immer obligatorisch, insbesondere bei der Arbeit mit externen Anbietern. In unserem Vortrag auf der JCON 2019 [8] haben wir gezeigt, wie einfach die Testabdeckung gefälscht werden kann. Um andere Schwachstellen zu erkennen, können Sie zusätzliche Checker wie etwa SpotBugs ausführen. Tests zeigen nicht an, dass eine Anwendung fehlerfrei ist, sie zeigen jedoch ein definiertes Verhalten für implementierte Funktionen an.

SCM-Suites wie GitLab oder Microsoft Azure unterstützen seit einiger Zeit Pull-Anfragen, die vor langer Zeit auf GitHub eingeführt wurden. Diese Workflows sind per se nichts Neues. IBM Synergy verwendete eine ähnliche Strategie. Ein Build Manager war dafür verantwortlich, die Änderungen der Entwickler in der Codebasis zusammenzuführen. Das führte dazu, dass alle vom Entwickler durchgeführten Revisionen vom Build Manager in das Repository aufgenommen wurden. Der Build Manager verfügte nicht über ausreichend fundierte Kenntnisse, um die Implementierungsqualität beurteilen zu können. Es wurde schnell zur üblichen Praxis, die Commits durchzuwinken und lediglich darauf zu achten, dass der Build nicht kaputt geht und die Kompilierung immer ein Artefakt erzeugt.

Unternehmen haben als neue Strategie Pull-Request entdeckt, um sie in ihren Projekten einzusetzen. Den dahinterstehenden Dictatorship-Workflow, wie er zu Recht in Open-Source-Projekten angewendet wird, hinterfragen sie hingegen nicht auf Sinnhaftigkeit. Denn das sagt implizit, dass dem eigenen Entwicklungsteam weder Vertrauen entgegengebracht noch Kompetenz zugetraut wird. Heutzutage treffen Manager häufig die Entscheidung, Pull-Requests als Qualitätsschranke zu verwenden. Nach meiner Erfahrung verlangsamt das die Produktivität, da es einige Zeit dauert, bis die Änderungen in der Codebasis verfügbar sind. Das Verständnis von Branch- und Merge-Mechanismen hilft Ihnen bei der Entscheidung für die Nutzung eines einfachen Branch-Modells, wie beispielsweise Release Branch Lines. Auf diesen Branches arbeiten dann Tools wie SonarQube, um das geforderte Qualitätsziel zu sichern. Wenn ein Projekt einen orchestrierten Build mit einer definierten Reihenfolge für die Erstellung von Artefakten benötigt, haben Sie einen starken Hinweis auf eine notwendige Refaktorierung.

Die Kopplung zwischen Klassen und Modulen wird oft unterschätzt. Es ist sehr schwierig, eine automatisierte Visualisierung für die Bindungen von Modulen zu haben. Sie werden sehr schnell feststellen, welchen Effekt es hat, wenn eine starke Kopplung aufgrund zunehmender Komplexität Ihre Build-Logik unnötig verkompliziert.

Checkliste

- Befolgen Sie etablierte Standards
- KISS – Keep it simple, stupid!
- Gleiche Verzeichnisstruktur für verschiedene Projekte
- Einfache Metaarchitektur, die weitgehend in anderen Projekten wiederverwendet werden kann
- Codierungsstyles definieren und befolgen
- Es werden keine Compilerwarnungen akzeptiert
- Testabdeckung um die 85 Prozent
- Vermeiden Sie Bibliotheken von Drittanbietern soweit möglich
- Verwenden Sie nicht mehr als eine Technologie für ein bestimmtes Problem (z. B. JSON)
- Decken Sie Fremdcode durch ein Entwurfsmuster ab
- Vermeiden Sie starke Objekt-Modul-Kopplungen

Wiederholungstäter

Seien Sie versichert, nichts ist so sicher wie die Änderung! Es ist eine Herausforderung, eine Anwendung für Anpassungen offen zu halten. Einige der vorherigen Empfehlungen haben implizite Auswirkungen auf die zukünftige Wartbarkeit. Eine gute Source-Qualität vereinfacht das Unterfangen durchaus. Es gibt jedoch keine Garantien. Im schlimmsten Fall ist das Ende des Produktlebenszyklus (EOL) erreicht, wenn obligatori-

sche Verbesserungen oder Änderungen beispielsweise wegen einer erodierten Codebasis nicht mehr realisiert werden können. Das tritt weit häufiger auf, als man annimmt.

Wie bereits erwähnt, bringt eine lose Kopplung zahlreiche Vorteile für die Wartung und Wiederverwendung mit sich. Dieses Ziel zu erreichen ist nicht so schwierig, wie es auf den ersten Blick erscheinen mag. Versuchen Sie zunächst, die Verwendung von Drittanbieterbibliotheken so weit wie möglich zu vermeiden. Nur um zu überprüfen, ob ein String leer oder null ist, ist es nicht notwendig, sich von einer externen Bibliothek abhängig zu machen. Diese wenigen Zeilen sind schnell selbst formuliert. Ein zweiter wichtiger Punkt, der in Bezug auf externe Bibliotheken berücksichtigt werden muss: Nur eine Bibliothek zur Lösung eines Problems. Wenn beispielsweise im Projekt JSON verwendet wird, entscheiden Sie sich für eine Implementierung und integrieren Sie nicht verschiedene Artefakte mit gleicher Funktionalität. Das wirkt sich auch stark auf die Sicherheit aus: Ein Artefakt von Drittanbietern, dessen Verwendung wir vermeiden können, kann keine Sicherheitslücken verursachen.

Zudem ist es nach einer Entscheidung für eine externe Bibliothek sehr hilfreich, diese für das gesamte Projekt zu kapseln. Das gelingt über die Verwendung von Entwurfsmustern wie Proxy, Fassade oder Wrapper und gestattet einen einfacheren Austausch, da die Codeänderungen so nicht gleich über die gesamte Codebasis verteilt werden. Sie müssen also nicht alles auf einmal ändern, wenn Sie den Anweisungen zum Benennen der Implementierungsklasse und Bereitstellen einer Schnittstelle folgen. Obwohl ein SCM auf Zusammenarbeit ausgelegt ist, gibt es Einschränkungen, wenn mehr als eine Person dieselbe Datei bearbeitet. Durch die Verwendung eines Entwurfsmusters zum Verstecken von Information können Sie Ihre Änderungen iterativ aktualisieren.

Fazit

Wie wir gesehen haben, ist eine nichtfunktionale Anforderung nicht so schwer zu beschreiben. Mit einer kurzen Checkliste (Kasten: „Checkliste") können Sie die wichtigsten Aspekte für Ihr Projekt klar definieren. Es ist nicht erforderlich, alle Punkte für jeden Code-Commit im Repository zu überprüfen. Das würde höchstwahrscheinlich nur die Kosten erhöhen und führt nicht zu mehr Vorteilen. Eine vollständige Überprüfung etwa wenige Tage vor einem Release stellt in einem agilen Kontext eine effektive Lösung dar. POIs zur Qualitätssicherung sind die Überarbeitungen der Codebasis für ein Release. Das gibt Ihnen eine vergleichbare Statistik und hilft, die Zuverlässigkeit von Schätzungen zu erhöhen. Schlussendlich sollte das Hauptanliegen sein, ein hohes Maß an Automatisierung in der Infrastruktur zu erreichen; Continuous Integration z. B. ist äußerst hilfreich, befreit Sie jedoch nicht von manuellen Code Reviews.

Marco Schulz studierte an der HS Merseburg Diplominformatik. Sein persönlicher Schwerpunkt liegt auf Softwarearchitekturen, der Automatisierung des Softwareentwicklungsprozesses und dem Softwarekonfigurationsmanagement. Seit über fünfzehn Jahren entwickelt er für namhafte Unternehmen auf unterschiedlichen Plattformen umfangreiche Webapplikationen. Er ist freier Consultant, Trainer und Autor verschiedener Fachartikel.

https://enRebaja.wordpress.com

marco.schulz@outlook.com

@ElmarDott

Frontend-Architektur für Kubernetes

Thomas Kruse *(trion development GmbH)*

Kubernetes ist das Betriebssystem für die Cloud und für Microservices. Viele Projekte und Architekten sehen sich damit konfrontiert, dass das Management von oben oder die Entwickler von unten Kubernetes als Marschrichtung positionieren. Doch wie passen bisherige Anwendungen dazu? Ist Kubernetes der neue Application Server, und die liebevoll gepflegte JSF-Anwendung wird sich auch in Kubernetes wie zu Hause fühlen? Funktionieren damit Self Healing und Load Balancing? Wie sieht es mit SPAs auf Basis von Angular oder React aus? Von so einem modernen Stack darf man ja erwarten, dass Anwendungen reibungslos Kubernetes-ready sind, oder? In diesem Vortrag werden wichtige Diskussionspunkte für Auswirkungen von Kubernetes auf Frontend-Architekturen vorgestellt. Neben Antworten auf die häufigsten Fragen bei einer Migration zu Kubernetes werden die Möglichkeiten, die sich durch Kubernetes bieten, an praktischen Beispielen im Rahmen von Build, Test und Deployment gezeigt.

Links & Literatur

[1] https://csse.usc.edu/TECHRPTS/1976/usccse76-501/usccse76-501.pdf

[2] https://overreacted.io/goodbye-clean-code/

[3] Schlossnagle, George: „Advanced PHP Programming: Developing Large-Scale Web Applications with PHP 5", Pearson Education, 2007

[4] https://github.com/ElmarDott/TP-CORE

[5] https://semver.org/

[6] https://enrebaja.wordpress.com/together-platform/

[7] https://dzone.com/articles/acceptance-tests-in-java-with-jgiven

[8] https://www.youtube.com/watch?v=VOwVfG0tbpM

Gecheckt

Architekturüberprüfung in der Pipeline

von Arne Limburg

Eine saubere Architektur zu definieren, ist das eine, sich im Projektalltag auch in stressigen Zeiten daran zu halten, meist eine ganz andere Geschichte. Und wenn es sich dann auch noch um Vorgaben handelt, die nicht direkt den eigenen Service betreffen, sondern seine Kommunikation mit anderen, ist die Motivation, sie einzuhalten, meist noch geringer. Wie kann man also solche Makroarchitekturvorgaben umgehen?

Eine gute Architektur ist wichtig, um eine Applikation langfristig wartbar und weiterentwickelbar zu halten, darin sind sich wohl alle einig. Aber wie stellt man sicher, dass die einmal überlegten Architekturvorgaben, wie z. B. eine Schichtenarchitektur oder – moderner – eine hexagonale Architektur (aka Clean Architecture) im Projekt auch dauerhaft eingehalten werden?

Die Applikationsarchitektur in der Pipeline

Normalerweise verstoßen Entwickler nicht absichtlich gegen Architekturvorgaben. Meistens passiert es unbemerkt beim Erstellen von neuem Code oder bei Refactorings. Daher ist es sinnvoll, dass diese Verstöße dem Entwickler so schnell wie möglich angezeigt werden, damit er darauf reagieren kann.

Aber was bedeutet „so schnell wie möglich“? Im Idealfall heißt das wohl, dass die Anzeige direkt in der IDE erfolgt. Da die verwendete IDE und auch deren Konfiguration sich häufig von Entwickler zu Entwickler unterscheiden, sollte die Überprüfung zusätzlich im Build-Prozess – und zwar lokal und in der Build Pipeline – erfolgen. Es gibt verschiedene Tools, mit denen man Architekturregeln beim lokalen Bauen oder in der Build Pipeline überprüfen kann. Die verbreitetsten sind wohl Sonarqube [1] und ArchUnit [2]. Mit beiden lassen sich Zugriffsregeln zwischen Klassen und Packages überprüfen, zirkuläre Abhängigkeiten erkennen und Namensregeln für gewisse Arten von Klassen definieren. Beide lassen sich sowohl in der IDE (Sonarqube via Plugin, ArchUnit als Unit-Test) als auch in der Build Pipeline ausführen, sodass ein schnelles Feedback für den Entwickler gegeben ist.

Vorgaben der Makroarchitektur

Sobald mehrere Services in einem System miteinander kommunizieren, wie es z. B. (aber nicht nur) bei Microservices der Fall ist, werden weitere Architekturvorgaben benötigt. Die beziehen sich nicht auf den inneren Aufbau der einzelnen Services, sondern regeln die Kommunikation der Services untereinander.

Dabei gibt es gewisse Entscheidungen, die für alle Services gemeinsam getroffen werden müssen. Sie können sowohl technologischer als auch architektonischer Natur sein. Zu behandeln sind Themen wie Service Discovery, Authentication, Tracing, ggf. ein gemeinsames Logformat, um das Logging zu zentralisieren, Health Checks, Testing-Strategien (z. B. über Consumer-driven Contract Testing) oder auch eine gemeinsame Dokumentation des API (z. B. über OpenAPI). Sind in den entsprechenden Bereichen einmal Architekturvorgaben entstanden, stellt sich natürlich auch hier die Frage, wie eine Einhaltung dieser Vorgaben sichergestellt werden kann.

Einhalten der Makroarchitekturvorgaben in der Pipeline

Im Gegensatz zu den oben erwähnten Tools zur Einhaltung von Architekturvorgaben innerhalb eines Service gestaltet sich die Überprüfung der genannten Makroarchitekturvorgaben deutlich komplizierter. Meistens läuft es darauf hinaus, das spezielle Integrationstests ge-

schrieben werden müssen, die die jeweiligen Vorgaben testen. Das kann z. B. ein Test sein, der überprüft, ob ein ausgehender REST Call auch tatsächlich die korrekten Tracing-Header enthält oder ob die Logausgaben während eines Calls auch tatsächlich den Vorgaben entsprechen.

Für andere Makroarchitekturvorgaben sind spezielle Schritte im Build notwendig. So ist es bei der klassischen Java-Integration von Pact [3] (einem Tool für Consumer-driven Contracts) so, dass zwar die Contracts beim normalen Build entstehen, sofern man Consumer-Contract-Tests geschrieben hat, das Hochladen der Contracts aber in einem separaten Schritt erfolgen muss (z. B. über das Pact-Maven-Plug-in [4]). Zusätzlich kann in einer Continuous Deployment Pipeline über das von Pact mitgelieferte Tool can-i-deploy [5] vor dem Deployment sichergestellt werden, dass der auf eine Stage zu deployende Stand kompatibel zu den anderen Services ist, die aktuell auf der entsprechenden Stage sind.

Architekturvorgaben, wie z. B. das Veröffentlichen der OpenAPI-Dokumentation an einer zentralen Stelle, können natürlich auch als expliziter Schritt in der Pipeline erfolgen. Wir können bereits beim Betrachten dieser wenigen Beispiele feststellen, dass die Einhaltung von Makroarchitekturvorgaben häufig nicht so einfach ist, wie bei der oben beschriebenen Applikationsarchitektur. Das liegt allerdings eher in der Natur der Makroarchitekturvorgaben, weil sie eben nicht nur den eigenen Code betreffen, sondern auch aufrufende und aufgerufene Services und die Kommunikation dazwischen.

Ein weiteres Problem bei der Überprüfung von Vorgaben in der Pipeline ist die Tatsache, dass das Team selbst auch die Art der Überprüfung und damit die Interpretation der Vorgaben vornimmt. Da es sich bei der Makroarchitektur aber um serviceübergreifende und damit in der Regel auch um teamübergreifende Vorgaben handelt, sollten alle Teams die gleiche Sicht auf die Vorgaben haben, und natürlich sollten sich auch alle Teams auf die gleiche Art daran halten. Darüber hinaus müssen natürlich auch alle Teams ein Interesse daran haben, sich an die Vorgaben zu halten.

Überprüfen der Makroarchitektur beim Deployment oder zur Laufzeit

Um das sicherzustellen, muss es eine Instanz außerhalb der einzelnen Service-Teams geben, die die Makroarchitekturvorgaben festlegt und weiterentwickelt. Das kann ein Architekturgremium sein, in das jedes Service-Team eine oder mehrere Personen entsendet oder ein Systemarchitekt, der die Vorgaben initial entwickelt und weiterpflegt. Das Ganze erfolgt dann (hoffentlich) in Abstimmung mit den jeweiligen Service-Teams, sodass auch dort ein Verständnis für die Vorgaben gegeben ist.

Dennoch zeigt die Erfahrung, dass solche Regeln dauerhaft nur von allen Teams eingehalten werden, wenn sie auch überprüft werden. In der Rechtswissenschaft spricht man von Rechtswirksamkeit [5]. Demnach ist eine Rechtsnorm nur dann wirksam, wenn sie auch durchsetzbar ist. In einer Makroarchitektur, in der es unabhängige Services gibt, kann die Durchsetzbarkeit von Regeln ein Problem sein. Die einfachste Variante der Durchsetzung solcher Regeln wäre es, eine bestimmte Deployment Pipeline (die die Regeln überprüft) vorzuschreiben und deren Nutzung sicherzustellen. Das widerspräche aber dem Paradigma der unabhängigen Teams. Wenn ein Team seinen Service nämlich unabhängig entwickeln, testen und deployen können soll, sollte es auch die Macht über die eigene Deployment Pipeline haben. Die Einhaltung der Makroarchitekturregeln läge dann allerdings wieder in der Verantwortung (und damit auch im Ermessen) des Teams.

Was also benötigt wird, ist eine weitere Instanz, die vor, während oder nach dem Deployment überprüft, ob alle Makroarchitekturregeln eingehalten wurden.

Glücklicherweise bietet der Kubernetes-Stack inklusive der verfügbaren Service Meshes verschiedene Hooks, um solche Überprüfungen vorzunehmen.

Wenn man zur Architekturüberprüfung nur informiert werden möchte, sobald eine neue Instanz eines Service (in Kubernetes-Sprache: ein Pod) auf einem Kubernetes-Cluster deployt wird, bietet sich das Event API von Kubernetes an. Kubernetes Events können konsumiert werden, indem man per REST einen GET-Request gegen

jax HYBRID

Infrastructure as Code – muss man nicht testen, Hauptsache es läuft?

Sandra Parsick *(Freiberuflerin)*

Mittlerweile wird die Infrastruktur immer mehr mithilfe von Code (Provisionierungsskripte, Dockerfiles, (Shell-)Skripte etc.) beschrieben und automatisiert. Klassische Betriebsabteilungen mutieren auf einen Schlag zu Entwicklungsabteilungen und müssen programmieren, um an ihre Infrastruktur zu kommen. Doch ist auch allen Beteiligten klar, dass sie zu professionellen Programmierern geworden sind? Wenn man sich Entwicklungsprozess und Code anschaut, erinnern beide stark an die Fricklermentalität der 2000er: Juhuu, es läuft irgendwie, kein VCS, keine Qualitätssicherung mit Test oder Review. Wenn es sich stark nach normaler Softwareentwicklung anfühlt, warum dann auch nicht die Best Practices und Lessons Learned der letzten 30 Jahren auf Infrastructure as Code adaptieren und somit die Qualität erhöhen? Müssen die frisch gebackenen OpsDevs die alten Fehler der Devs wiederholen? Kann man Infrastrukturcode vielleicht sogar testgetrieben entwickeln? Dieser Vortrag lädt zu einer Zeitreise ein, welche Best Practices in der Softwareentwicklung zur einer höheren Qualität geführt haben und wie diese Erkenntnisse helfen können, die Entwicklung von Infrastrukturcode qualitativ hochwertiger zu machen.

das Kubernetes API auf dem folgenden Pfad ausführt: /*api/v1/events?watch=1* (siehe auch [6]). Man wird dann über alle Änderungen am Cluster informiert, also z. B. auch, dass ein neuer Pod gestartet wurde. Empfängt man ein solches Event, können gewisse Vorgaben am gestarteten Pod überprüft werden: Stellt er eine OpenAPI-Definition bereit? Existieren die zugehörigen Consumer und Provider Contracts beim Consumer-driven Contract Testing usw.? Nicht eingehaltene Regeln könnten in einem zentralen Tool angezeigt werden und so ggf. einen Gamification-Effekt erzielen. Kein Team möchte, dass an zentraler Stelle Regelverstöße des eigenen Service zu sehen sind.

Möchte man einen Schritt weiter gehen und bei gewissen Regelverstößen das Starten des Service komplett verhindern, muss man sich in den Deployment-Prozess einhängen. Auch dafür bietet Kubernetes Möglichkeiten, nämlich via Sidecar Injection [7]. Dabei handelt es sich um dasselbe API, das auch von Service Meshes verwendet wird. Der Deployment Descriptor wird dabei so angepasst, dass innerhalb des Containers beim Start ein Befehl ausgeführt wird. Schlägt dieser fehl, scheitert das Deployment. Indem man dort eigene Befehle einhängt, die gewisse Checks durchführen, könnte man z. B. sicherstellen, dass der gerade startende Service tatsächlich eine OpenAPI-Definition an definierter Stelle zur Verfügung stellt. Weiterhin können auch Dinge wie die Verwendung eines definierten Basis-Docker-Image überprüft werden, oder es kann mittels des bereits erwähnten Tools can-i-deploy (das dann natürlich im besagten Basis-Image installiert sein müsste) festgestellt werden, ob alle Consumer-driven Contracts korrekt validiert sind.

Einige Regeln können von außen erst zur Laufzeit überprüft werden, wie z. B. das korrekte Logformat oder das Setzen des Tracing-Headers. Aber über einen Service Mesh gibt es auch hier die Möglichkeit, die Einhaltung dieser Regeln zur Laufzeit sicherzustellen. Im Service Mesh Istio [8] können z. B. HTTP-Filter konfiguriert werden, die Requests ohne Tracing-Header komplett blockieren. Eine Anwendung, die sich dann nicht an den Tracing-Standard hält, könnte so überhaupt nicht kommunizieren.

Fazit

Um sicher zu sein, dass sich alle Entwickler jederzeit an die Architekturvorgaben halten, ist es sinnvoll, sie im Continuous-Deployment-Prozess zu verankern. Für Mikroarchitekturvorgaben reicht dabei eine Implementierung in der Pipeline, da alle Informationen, die benötigt werden, in der Pipeline zur Verfügung stehen. Bei Makroarchitekturvorgaben ist die Überprüfung komplizierter. Hier ist es nur in Teilen möglich, die Vorgaben in der Pipeline sicherzustellen. Weitere Überprüfungen können im Deployment-Prozess oder zur Laufzeit erfolgen.

Glücklicherweise bieten Kubernetes und Co. in Verbindung mit Service Meshes genügend Eingriffsmöglichkeiten, um auch solche Tests zu implementieren. In diesem Sinne – haltet die Architektur sauber.

Arne Limburg ist Softwarearchitekt bei der open knowledge GmbH in Oldenburg. Er verfügt über langjährige Erfahrung als Entwickler, Architekt und Consultant im Java-Umfeld und ist auch seit der ersten Stunde im Android-Umfeld aktiv.

Links & Literatur

[1] https://www.sonarqube.org

[2] https://www.archunit.org

[3] https://docs.pact.io

[4] https://github.com/DiUS/pact-jvm/tree/master/provider/pact-jvm-provider-maven

[5] https://de.wikipedia.org/wiki/Wirksamkeit_(Recht)#Wirksamkeit_von_Rechtsnormen

[6] https://kubernetes.io/docs/reference/using-api/api-concepts/#efficient-detection-of-changes

[7] https://medium.com/dowjones/how-did-that-sidecar-get-there-4dcd73f1a0a4

[8] https://istio.io

Praktische Einführung in gute Tests – für Einsteiger #folienarm

Thomas Much *(Freiberufler)*

Vorträge drehen sich meist um neue Tools und Bibliotheken. Das ist gut und wichtig, damit wir nicht stehenbleiben. Aber was ist mit den Grundlagen, die zu diesen Neuentwicklungen geführt haben? Gerade beim Thema „Testautomatisierung" ist mir aufgefallen, dass Neu- und Quereinsteiger*innen oft das vorhandene (gute oder schlechte) Testvorgehen hinnehmen, nachahmen und zu wenig hinterfragen. In dieser Live-Coding-Session schauen wir uns daher folgende Fragen an: Wie strukturiere ich meine Tests? Wie mache ich meine Tests lesbar, verständlich und wartbar? Wie teste ich etwas, das noch gar nicht da ist? Und wie hilft mir das alles für einfache Design-Refactorings? Ideologie lassen wir außen vor. Ich möchte Erfahrungen aus der Praxis zeigen, natürlich mit ein paar Anleihen aus TDD, Clean Code und auch vom Domain-driven Design. Testprofis werden nicht viel Neues entdecken. Aber wer sich beim Schreiben von Tests unsicher fühlt, bekommt pragmatische, praxiserprobte Anregungen, sodass die Tests uns im Entwicklungsteam unterstützen und nicht nur den Chef durch das Erfüllen irgendeiner Testabdeckungsmetrik zufriedenstellen. Wir schauen uns vor allem Unit-Tests in Java an, die Ideen können aber auf andere Testarten und Programmiersprachen übertragen werden. Das Live-Coding ist #folienarm – mit ein paar wenigen, einleitenden Slides.

Architekturpatterns in Modulithen – Teil 1

Ordnung ins Chaos bringen

„Wir haben diesen Legacy-Monolithen, den wollen wir in Microservices aufbrechen". So einen Satz hört man als Berater in der Softwarebranche oft. Auf die Frage „Warum" erhält man oft die Antwort „Modularisierung". Denn es herrscht die weitverbreitete Ansicht, dass Monolithen grundsätzlich aus schlecht strukturiertem Legacy-Code bestehen und sich Monolithen und Modularisierung gegenseitig ausschließen. Dass dem nicht so ist, zeigt die Architekturform der Modulithen. In dieser Artikelserie wird sie beleuchtet und beschrieben, mit welchen Patterns ein Modulith gelingen kann und welche Antipatterns man dabei vermeiden sollte.

von Arnold Franke

Wer in den letzten sieben Jahren Softwarekonferenzen besucht hat, der musste ganz schön Slalom laufen, wenn er dem Thema Microservices [1] aus dem Weg gehen wollte. Aufgrund ihrer vielversprechenden Eigenschaften wurde diese Form der Systemarchitektur stark gehypt und von vielen Entwicklern, Architekten und Firmen als Heilsbringer angesehen. Mit etwas Abstand betrachtet, handelt es sich dabei aber um keine Silver Bullet, sondern nur um eine von vielen Möglichkeiten, ein Softwaresystem zu strukturieren – mit eigenen Vor- und Nachteilen (**Abb. 1**). Eine Landschaft aus wirklich entkoppelten Microservices zu bauen, die alle Vorteile dieser Architekturform mitnimmt, ist alles andere als trivial, und man ist selbst dann nicht gegen strukturelle Stolperfallen wie Conway's Law [2] gefeit. Wenn man Microservices ungünstig schneidet und verknüpft, dann besteht genauso die Gefahr, am Ende einen großen Deployment-Monolithen aus Legacy Code zu erhalten.

Den Hype überlebt haben die „guten alten" Monolithen. Große Deployment-Einheiten mit einer riesigen Codebase, die sich den Ruf von schlechter Wartbarkeit und Erweiterbarkeit eingehandelt haben. Einerseits überlebten sie, weil man einige davon aufgrund gewachsener Komplexität und schlecht auflösbarer Abhängigkeiten nur schwer wieder losbekommt. Andererseits, weil sie in Form der modularen Monolithen – Modulithen – eine Renaissance in der Entwicklercommunity erleben. Modulithen vereinigen einige der Vorteile der Microservices-Architektur, zum Beispiel modulare Struktur mit gekapselten Verantwortlichkeiten und übersichtlichen Abhängigkeiten, während sie auf einige der Nachteile, wie beispielsweise komplexe Infrastruktur und Kommunikations-Overhead, verzichten. Das macht die Modulithen nicht zu einer den Microservices überlegenen Architekturform (Kasten: „Was können Microservices, was ein Modulith auch kann?"). Sie sind

Architekturpatterns in Modulithen

Teil 1: Ordnung ins Chaos bringen
Teil 2: Wer mit wem reden darf
Teil 3: Die Bude sauber halten

Was können Microservices, was ein Modulith auch kann?

- Technische Trennung von fachlichen Kontexten
- Kleine, weitgehend unabhängig voneinander entwickelbare Module
- Beherrschbare Komplexität, einfache Erweiterbarkeit
- Klare Abhängigkeiten mit expliziten Schnittstellen, geringes Risiko für unerwartete Nebeneffekte
- Continuous Integration, Continuous Delivery

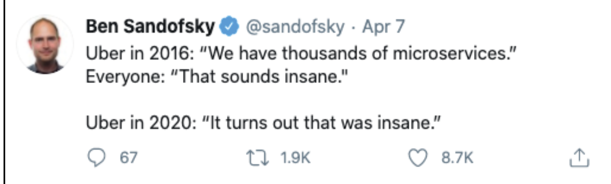


Abb. 1: Vier Jahre später ...

nur ein weiterer valider Ansatz zur Strukturierung eines Softwaresystems, der in bestimmten Kontexten sinnvoller ist als andere.

Die Wahl der Architekturform hängt letztendlich davon ab, auf welchen Aspekt man optimieren möchte. Individuelle Skalierung? Entwicklungsgeschwindigkeit? Resilience? Komplexität der Infrastruktur? Das sind alles Faktoren, die bei dieser Entscheidung eine Rolle spielen.

Dabei ist die Idee des modularen Monolithen durchaus keine neue. Sie ist aber nur erfolgreich und nachhaltig umsetzbar, wenn man es schafft, durch nachhaltige Anwendungsarchitektur die Komplexität einer großen Codebase im Zaum zu halten und Verständlichkeit und Wartbarkeit des Codes zu wahren. Damit habe ich mich in den letzten sieben Jahren in mehreren Softwareprojekten beschäftigt und möchte in dieser Serie einige Patterns und Antipatterns teilen, um anderen Modulithen-Bauern leichter zum Erfolg zu verhelfen.

Den Monolithen schneiden

Egal ob man einen historisch gewachsenen Monolithen besser strukturieren will oder ein großes Projekt auf der grünen Wiese startet – eine der ersten Entscheidungen, die man treffen muss, ist: „Wie schneide ich meinen Code in Module?" Dazu muss man sich bewusst machen, was ein gutes Modul ausmacht: Einer der wichtigsten Aspekte einer modularen Architektur ist die Separation of Concerns. Ein Modul hat eine klare Verantwortlichkeit und ist nur für diese zuständig. Es kümmert sich dabei nicht um Verantwortlichkeiten anderer Module (kein Featureneid [3]). Es kapselt seine Verantwortlichkeit und alle dazu benötigten internen Aspekte wie die Implementierung von Businesslogik und die Persistenz von Daten, sodass sie nicht von außen manipulierbar sind.

Um mit der Außenwelt und anderen Modulen zu kommunizieren, stellt ein Modul Schnittstellen für die notwendigen Operationen bereit, die kontrollierten Zugriff auf seine Verantwortlichkeit erlauben. Es ist möglichst entkoppelt von anderen Modulen und hat so wenige Abhängigkeiten wie möglich. Intern weist es eine hohe Kohäsion auf, d. h., dass es keine „Inseln" enthält, die mit dem Rest des Moduls nichts zu tun haben.

Dieses Paradigma der geringen Kopplung und hohen Kohäsion schließt es bereits aus, Module nach rein technischen Schichten horizontal zu schneiden. Stellen wir uns z. B. ein Modul vor, das als einzige Verantwortlichkeit Persistenz hat. Darin wären Interfaces/Klassen, die jeweils auf „ihre" Tabelle in der Datenbank zugreifen. Diese Klassen würden sich nie gegenseitig aufrufen, sondern jeweils nur Datenbankzugriffe vornehmen – also geringe Kohäsion innerhalb des Moduls. Die Kopplung mit „Modulen" anderer Schichten dagegen wäre vermutlich beträchtlich, da jede datenverarbeitende Logik der Anwendung auf dieses eine Persistenzmodul zugreifen müsste. Damit hätten wir gleichzeitig ein weiteres Antipattern geschaffen, den Dependency Magnet – ein Modul, das Abhängigkeiten zu allen anderen Teilen der Anwendung hat. Wenn man seine ganze Anwendung mit solchen horizontalen Schnitten strukturiert, dann hat man am Ende eine reine Schichtenarchitektur ohne Module und mit einem unübersichtlichen und immer größer werdenden Abhängigkeitsknäuel (**Abb. 2**).

Ein besseres Vorgehen zur Bildung von Modulen sind daher vertikale Schnitte nach fachlichen Verantwortlichkeiten (**Abb. 3**). Jedes Modul erhält die Verantwortung für einen klar abgetrennten fachlichen Subkontext und beinhaltet alle Aspekte, die dieser Kontext zum Funktionieren braucht. Es hat die Hoheit über die Daten und die Logik dieser einen Teilmenge der Fachlichkeit und entscheidet, auf welche Weise diese nach außen bereitgestellt werden. Wenn wir uns zum Beispiel das Verwaltungssystem eines Kinos [4] vorstellen, dann haben wir ein Modul für den fachlichen Subkontext „Reservierung", das die Daten aller Sitzplatzreservierungen hält und die fachliche Logik für das Reservieren und Stornieren von Sitz-

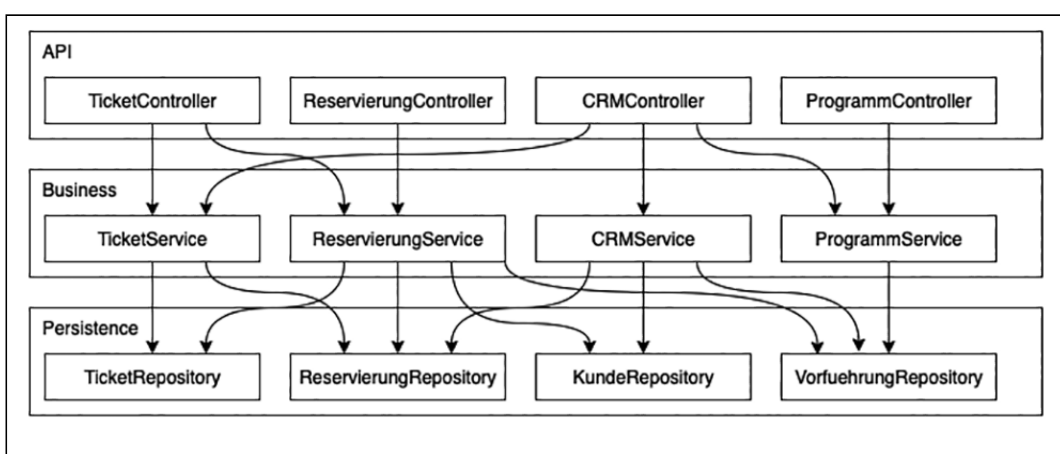


Abb. 2: Reine Schichtenarchitektur

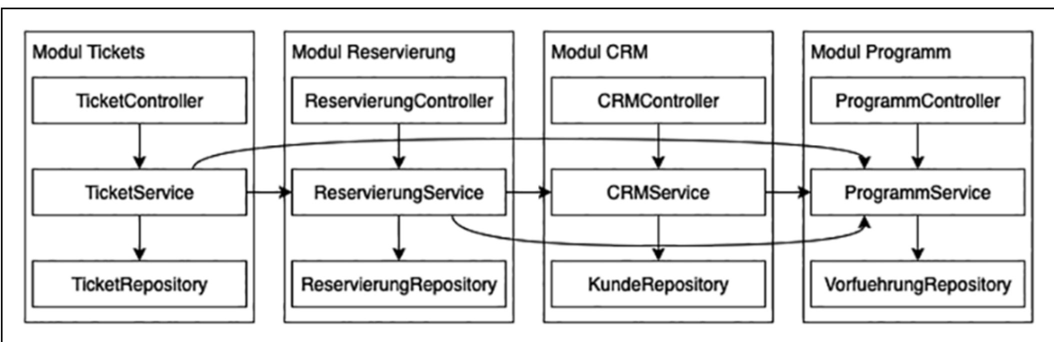


Abb. 3: Strukturierung nach fachlichen Modulen

plätzen implementiert. Ein anderes Modul „Ticket" hat die Hoheit über den Verkauf von Karten. Es darf keine Reservierungen selbst speichern oder ändern, kann aber dem Reservierungsmodul über eine Schnittstelle mitteilen, dass eine reservierte Karte abgeholt wurde, damit das Reservierungsmodul unter Berücksichtigung aller Reservierungsbusinessregeln die entsprechenden Änderungen an der Reservierung vornehmen kann. Diese Art von Modulschnitt hat noch weitere Vorteile:

- Die Codestruktur bewegt sich nahe an der Fachlichkeit, was für ein einheitlicheres Verständnis zwischen Entwicklern und Fachseite sorgt und die Kommunikation erleichtert.
- Abhängigkeiten zwischen fachlichen Kontexten finden sich 1:1 in den Abhängigkeiten zwischen den Modulen wieder.
- Da neue Features und Erweiterungen meist fachlich getrieben sind, fällt es leicht, die anzupassenden Module zu identifizieren und das Ausmaß der Änderungen darauf zu beschränken. Potenzielle Auswirkungen auf andere fachliche Aspekte werden schnell erkannt, und unerwünschte Nebenwirkungen sind leichter vermeidbar.
- Wenn man es zu einem späteren Zeitpunkt für sinnvoll hält, aus einem fachlichen Subkontext einen eigenen Service zu schneiden, dann ist es einfacher, den Code für den neuen Service zu extrahieren.
- Testbarkeit: Es ist möglich, die fachliche Logik eines Moduls getrennt von anderen Modulen und zusammenhängend durch alle Schichten durchzutesten – von der Schnittstelle bis zur Datenbank.

Um die fachlichen Subkontexte zu identifizieren und gegeneinander abzugrenzen, bietet sich das Konzept der Bounded Contexts [5] aus dem Domain-driven Design an. Ein Bounded Context vereinigt die Sprache, Regeln und Events einer Subdomain und grenzt sie gegen andere Subdomains ab. Methodisch kann man die Definition von Bounded Contexts durch Event Storming oder Context Mapping herauskristallisieren.

Nachdem man seine potenziellen Module identifiziert hat, sollte man sie im Code materialisieren. In einer Java-Anwendung ist der einfachste Weg dafür die Package-Struktur. Jedes Modul besteht aus einem Top-Level Package, in dem man allen Code unterbringt, der zum fachlichen Kontext des Moduls gehört. Im weiteren Verlauf der Artikelserie und in Beispielen wird exemplarisch diese Methode verwendet. Es gibt auch andere Möglichkeiten, Module innerhalb einer Java Codebase abzubilden, worauf ich in einem späteren Abschnitt noch eingehen werde.

Modulinterne Struktur

Wenn wir nun ein Modul definiert, ein Package dafür erstellt und allen Code, der zum Subkontext gehört, hi-

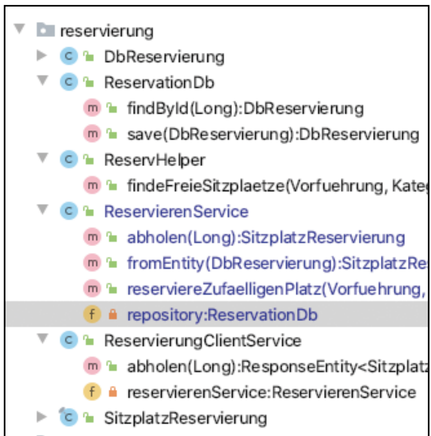


Abb. 4: Wie bei Hempels unterm Sofa

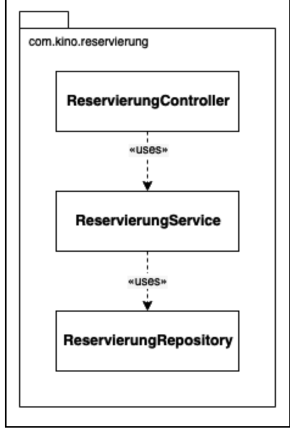
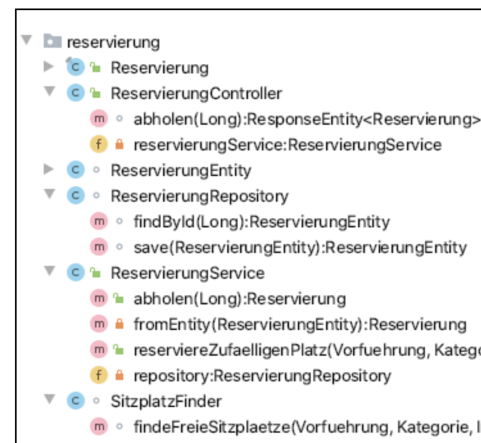


Abb. 5: Die Klassen halten sich an konsistente Namenskonventionen …



… und durch saubere Access Modifier können andere Module nur auf die definierten Schnittstellen des Moduls zugreifen (grünes Schloss)

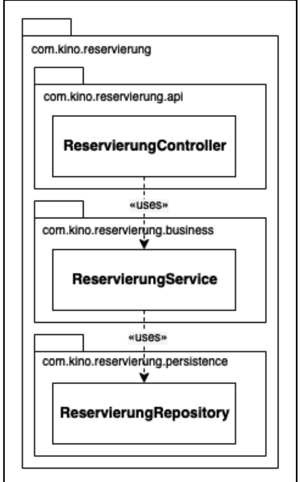


Abb. 6: Übersichtliche Strukturierung …

```
reservierung
  api
    ReservierungController
      abholen(Long):ResponseEntity<Reservierung>
      reservierungService:ReservierungService
  business
    Reservierung
    ReservierungService
      abholen(Long):Reservierung
      fromEntity(ReservierungEntity):Reservierung
      reserviereZufaelligenPlatz(Vorfuehrung, Katego
      repository:ReservierungRepository
    SitzplatzFinder
      findeFreieSitzplaetze(Vorfuehrung, Kategorie, I
  persistence
    ReservierungEntity
    ReservierungRepository
      findById(Long):ReservierungEntity
      save(ReservierungEntity):ReservierungEntity
```

… auf Kosten der Kapselung interner Logik

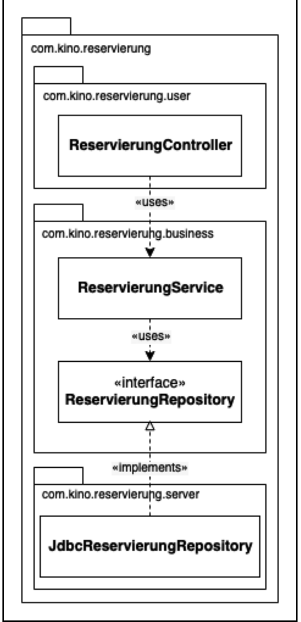
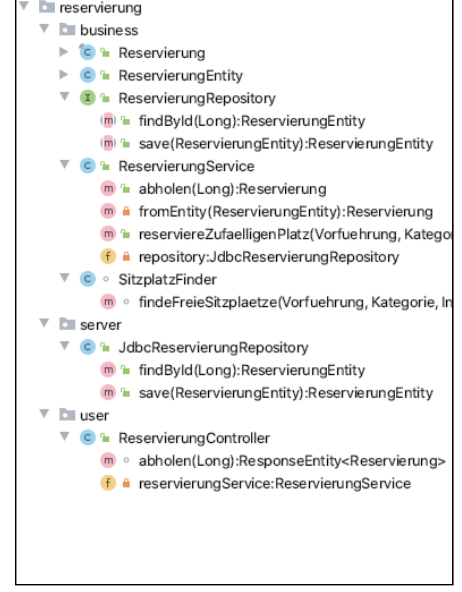


Abb. 7: Die Businesslogik ist gekapselt …



… aber modulinterne Methoden müssen als public deklariert werden

neingeworfen haben, dann kann das am Beispiel unseres Reservierungsmoduls erst einmal so aussehen, wie in **Abbildung 4** gezeigt. Wir sehen eine Menge Code, der sich anscheinend um fachliche Logik rund um die Reservierung kümmert. So weit, so gut. Leider ist es aber noch etwas unübersichtlich. Die Struktur ist undurchsichtig, die Namensgebung inkonsistent und auch die Schnittstelle nach außen ist nicht klar definiert, weil alle Klassen und Methoden den *public*-Modifier haben und damit von außen benutzbar sind. Um den Code modulintern besser zu strukturieren, gibt es unterschiedliche Patterns mit jeweils eigenen Vor- und Nachteilen:

Single Package Modul (Abb. 5): Bei diesem Ansatz verzichten wir auf weitere Unterteilungen innerhalb des Modul-Packages. Das kann bei großen Modulen der Übersichtlichkeit schaden. Dem kann man durch klare Namenskonventionen, die die Zuständigkeiten der einzelnen Klassen verdeutlichen, etwas entgegenwirken. Ein Riesenvorteil ist, dass hier die Access Modifier in Java voll genutzt werden können. Alle Klassen im Modul sind *package private*, von außen nicht sichtbar und stellen nur *package private*-Methoden bereit. Klasseninterne Methoden sind natürlich *private*. Nur der Business-Service, der die Fachlichkeit des Moduls nach außen bereitstellt, ist *public* und exponiert *public*-Methoden. Auch die zugehörigen Businessobjekte, die nach außen gereicht werden, sind *public*. So wird eine klar definierte Schnittstelle bereitgestellt, die von Clients des Moduls nicht umgangen werden kann..

Slices before Layers (Abb. 6): Dieser Ansatz geht davon aus, dass zuerst vertikale Slices gebildet werden (das sind bereits die Modulschnitte) und diese Slices danach in horizontale, technische Schichten weiter unterteilt werden. Diese Unterteilung findet innerhalb des Moduls durch Unterpackages statt, üblicherweise Persistence, Business, API etc. Dadurch werden die Schichten innerhalb eines Moduls leichter erkennbar und unidirektionale Zugriffe zwischen den Schichten von oben nach unten leichter kontrollierbar. Das dient vor allem der Organisation des Codes und verbessert die Übersichtlichkeit. Allerdings geht diese Aufteilung zu Lasten der Kapselung, denn das Modul muss interne Klassen und Methoden als *public* exponieren, da sonst die Packages der Schichten nicht aufeinander zugreifen können. Dadurch ist es schwieriger, eine klar definierte Schnittstelle bereitzustellen.

Hexagonal Architecture (Abb. 7): Diese auch Ports & Adapters [6] genannte Architekturform ist eigentlich für die Strukturierung einer ganzen Anwendung gedacht, lässt sich aber auch auf den Code innerhalb eines Moduls übertragen. Sie basiert auf dem Prinzip, dass die Domain-Logik in einer zentralen Businesskomponente gekapselt ist und Interaktionen mit der Außenwelt außerhalb davon liegen. Die Außenwelt sind in diesem Fall die User Side (APIs für Clients, GUI etc.) und die Server Side (Datenbank, Dateisystem, andere Server). Dabei hängt immer die äußere Interaktion von der zentralen Businesskomponente ab und nie umgekehrt. Im Fall eines Modulithen können sowohl User Side als auch Server Side Interaktionen mit anderen Modulen innerhalb desselben Modulithen beinhalten.

Der Vorteil ist, dass alle Businessregeln – auch Zugriffe auf Daten und die Art, wie Daten bereitgestellt und manipuliert werden – innerhalb der zentralen Businesskomponente definiert und gekapselt sind. Wie die

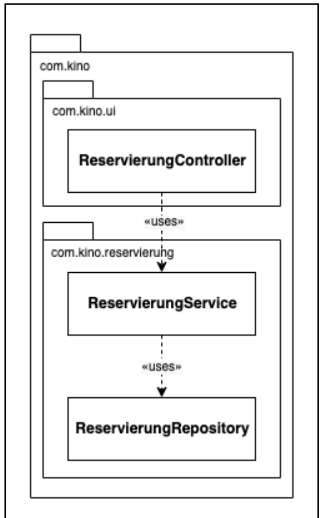


Abb. 8: Die UI-Komponente kann die gleichen Schnittstellen des Moduls benutzen wie andere Module

reservierung
Reservierung
ReservierungEntity
ReservierungRepository
findById(Long):ReservierungEntity
save(ReservierungEntity):ReservierungEntity
ReservierungService
abholen(Long):void
fromEntity(ReservierungEntity):Reservierung
reserviereZufaelligenPlatz(Vorfuehrung, Kateg
repository:ReservierungRepository
SitzplatzFinder
findeFreieSitzplaetze(Vorfuehrung, Kategorie,
ui
ReservierungController
abholen(Long):ResponseEntity<HttpStatus>
reservierungService:ReservierungService

Die Kapselung bleibt erhalten

User Side und Server Side diese Regeln implementieren, ist davon entkoppelt, sie können aber nicht von ihnen abweichen. Dadurch kann man sich bei der Entwicklung gezielt auf einen der Aspekte Businesslogik, Clientinteraktion und Serverinteraktion fokussieren und diese auch entkoppelt voneinander testen.

Eine potenzielle Schwäche ist auch hier: Die Schnittstellen, die die zentrale Businesskomponente innerhalb des Moduls der User Side und Server Side bereitstellt, müssen *public* sein und sind dadurch auch für andere Module sichtbar, obwohl sie nicht unbedingt dafür gedacht sind.

UI Component (Abb. 8): Bei diesem Ansatz wird die dem Client zugewandte Schicht (GUI oder API) zu einer eigenen großen, modulübergreifenden Komponente mit eigenem Top-Level Package. Sie benutzt das API aller Module, um dem Client eine einheitliche Repräsentation der Anwendung bereitzustellen. Die Module bestehen ansonsten wie im Single-Package-Ansatz aus jeweils einem einzelnen Package, das allen Code enthält, der zum Kontext des Moduls gehört, mit Ausnahme des Client API. Jedes Modul stellt dabei ein public API bereit, das idealerweise sowohl von der UI Component als auch von anderen Modulen benutzt werden kann.

Vorteile davon sind, dass die Anwendung dem Client gegenüber wie aus einem Guss präsentiert werden kann und gleichzeitig die Java Access Modifier zur Kapselung der Module verwendet werden können. Jedes Modul bestimmt vollkommen selbst, wie es verwendet wird, indem es außer dem bewusst bereitgestellten API alles als *package private* deklariert.

Dass die Clientschicht nicht in dieser Kapselung enthalten ist, kann in manchen Szenarien als Nachteil gesehen werden.

Welche der Möglichkeiten die beste ist, ist keine exakte Wissenschaft. In der Entwicklercommunity gibt es für jede der Varianten Befürworter und gute Argumente (beispielsweise [7]). Letztendlich entscheiden die Rahmenbedingungen darüber, welche Vorteile einem Projekt am meisten helfen und auf welche Aspekte hin man seinen Code optimieren möchte. Ist das Ziel ein Serviceschnitt? Dann ist ein Ansatz zu wählen, bei dem man einfach ein Modul, so wie es ist, ausschneiden und in ein frisches Projekt einfügen kann, wie z. B. der Single-Package-Ansatz.

Arbeitet ein großes Team mit unterschiedlichen Erfahrungslevels an dem Projekt? Dann ist der UI-Component-Ansatz vermutlich der beste Weg, da er die beste Kapselung der Module durch Access Modifier erreicht, sodass es schwerer wird, aus Versehen an der Architektur und den Businessregeln vorbei zu entwickeln. Dadurch hätte bestimmt schon der eine oder andere Big Ball of Mud vermieden werden können. Möchte man durch die Strukturierung des Codes in erster Linie Übersichtlichkeit und Codeorganisation und erst in zweiter Linie Kapselung erreichen? Dann bietet sich die Strukturierung nach Slices before Layers an.

Modularisierungstools im Java-Ökosystem

- OSGi [11]
- Java Platform Module System (Jigsaw) [12]
- JBoss Modules [13]
- Maven Multi-Module Projekte [8]
- Gradle Multi-Project Builds [9]

Enorm wichtig ist, dass man sich für eine der Varianten entscheidet und diese dann im gesamten Modulithen einheitlich verwendet. Die unterschiedlichen Varianten spielen ihre Stärken erst dann aus, wenn man sich darauf verlassen kann, dass sie für das gesamte Projekt gelten. Springt man stattdessen zwischen den Varianten, holt man sich alle deren Nachteile in den Code, ohne wirklich von ihren Vorteilen profitieren zu können. Zusätzlich stiftet man eine Menge Verwirrung für jeden, der sich an den Code heranwagt.

Technische Querschnittsmodule

Ein Antipattern, das in fast jedem größeren Projekt zu finden ist, ist das Util-Package. Oft wird es auch „helper", „common" oder „base" genannt, oder es existiert nicht einmal offiziell, da sein Inhalt stattdessen im Basis-Package der Anwendung ausgebreitet ist. Darin befindet sich dann Code, der von mehreren Teilen der Anwendung verwendet wird und deshalb nirgendwo anders zuzuordnen war. Und technische Utilities, die zwar nützlich sind, aber nichts miteinander zu tun haben. Und dann noch technische Querschnittsthemen, die alle Module betreffen und deshalb einfach in das „Package für alles" mit reingepackt wurden. Und alles andere, über das man halt gerade nicht nachdenken wollte. Ein solches Package erfüllt nicht die Eigenschaften eines Moduls und sollte in einem Modulithen vermieden werden:

- Es hat keine klare Verantwortlichkeit, sondern mehrere unzusammenhängende Zuständigkeiten, was der Name bereits widerspiegelt.
- Es hat eine niedrige Kohäsion, da die verschiedenen Aspekte meist nichts miteinander zu tun haben, sodass es nicht sinnvoll ist, sie zusammen zu gruppieren.
- Es hat eine hohe Kopplung, da jeder Aspekt von irgendeinem Modul gebraucht wird und das Util-Package manchmal sogar zusätzlich alle anderen Teile der Anwendung kennen muss. Oft schließt es damit einen Abhängigkeitskreis über alle Module und wird so zum schlimmsten Dependency Magnet.

Stattdessen sollte man technische und fachliche Querschnittsthemen in einem Modulithen ähnlich behandeln wie fachliche Kontexte. Nehmen wir als Beispiel das Verschicken von E-Mails. Wenn mehrere Module E-Mails versenden müssen, um ihre Fachlichkeit zu erfüllen, dann ist es gut, wenn es ein technisches Modul gibt, das sich um den technischen Aspekt „E-Mails versenden" kümmert und um nichts anderes. Dieses bietet dann eine von allen Modulen verwendbare Schnittstelle an, die unabhängig von deren Fachlichkeit funktioniert. Allgemeine Logik wie die Komposition der Mail oder Metainformationen der Mail ist in dem E-Mail-Modul gekapselt und es ist das einzige Modul, das die externe Abhängigkeit zum E-Mail-Server kennt. Dabei kennt das E-Mail-Modul keine fachlichen Details wie den Inhalt der Mails. Diese bleiben in der Hoheit der fachlichen Module, die die E-Mails verschicken möchten. Im Beispiel unseres Kinos könnte das Reservierungsmodul nach einer erfolgreichen Reservierung das E-Mail-Modul aufrufen, um dem Kunden eine Mail mit der Reservierungsnummer als Inhalt zu senden, während das Kundenmodul über dieselbe Schnittstelle einen Newsletter mit dem aktuellen Programm verschicken könnte.

Genauso erhalten andere Querschnittsthemen, wie z. B. Securityaspekte, Auditlog, Scheduling oder Reporting ihre eigene Kapselung durch ein eigenes Modul. Als Ergebnis entsteht eine schöne Landschaft aus fachlichen und technischen Modulen, die ihre Zuständigkeiten sicher kapseln und deren Abhängigkeiten untereinander leichter beherrschbar sind. Das Util-Package ist verschwunden oder zumindest so geschrumpft, dass es nur noch Klassen enthält, die die Bezeichnung „util" auch verdient haben, wie z. B. ein *DateUtil* mit statischen Methoden, die maßgeschneiderte Datumsoperationen bereitstellen.

Toolunterstützte Modulbildung

Bisher ging der Artikel von der einfachsten und gebräuchlichsten Modulbildungsmethode aus, nämlich der Abbildung von Modulen durch Top-Level Packages im Java-Code. Alternativ (Kasten: „Modularisierungstools im Java-Ökosystem") bieten auch Build-Tools wie Maven [8] und Gradle Features [9] wie Multi-Module Builds, mit denen man zumindest Code in Module aufteilen und zyklenfreie Abhängigkeiten zwischen den Modulen sicherstellen kann [10]. Es gibt aber auch Technologien wie OSGi [11] oder Jigsaw [12], das native Modulsystem der Java-Plattform, die sich als aus-

Microservice API Patterns

Dr. Daniel Lübke *(selbstständig)*

Die APIs von Microservices-Implementierungen sind ein zentraler Architekturaspekt. Viele Qualitätseigenschaften des Microservices sind eng mit seiner API verbunden, wie z. B. Geschwindigkeit von Anfragen, Wartbarkeit und Wiederverwendbarkeit in anderen fachlichen Kontexten. In vielen Projekten wurden solche APIs entwickelt und Softwarearchitekten und -entwickler haben mittlerweile jahrelange Erfahrung damit gesammelt. Das „Microservice API Patterns"-Projekt hat viele der verwendeten Patterns zusammengetragen und auf seiner Homepage https://microservice-api-patterns.org frei zur Verfügung gestellt. Diese Patterns sollen die Grundlage für eine gemeinsame Sprache im API-Design werden und noch nicht so erfahrenen Softwarearchitekten und -entwicklern Hilfestellung beim Entwurf ihrer APIs geben. Dieser Vortrag gibt einen Überblick über die Patterns und zeigt anhand eines Beispiels, wie diese eingesetzt werden können.

gewachsene Modularisierungslösungen verstehen und versuchen, das Problem der Modulbildung mitsamt Abhängigkeiten und Schnittstellen expliziter zu lösen.

Durch den Einsatz eines solchen Werkzeugs gewinnt man explizit definierte Module mit explizit definierten öffentlichen Schnittstellen. Das macht es viel einfacher, gesetzte Regeln von Kapselung, Interaktion und Abhängigkeiten der Module einzuhalten. Man muss sich nicht mehr allein auf die Java Access Modifier und Packagestruktur verlassen und ist innerhalb der Module freier mit deren Einsatz und der Codeorganisation durch Subpackages.

Der Preis dafür ist allerdings hoch. Man fügt seinem ohnehin schon komplexen Monolithen mit der Modularisierungsschicht eine weitere Abstraktionsebene und Komplexitätsdimension hinzu, die jeder Entwickler im Team kennen und beherrschen muss Jigsaw [14]. Oft tauchen dadurch auch neue Probleme auf, wie beispielsweise erschwertes Zusammenspiel mit externen Abhängigkeiten, Versions- und Namenskonflikte zwischen Modulen, Inkompatibilitäten zu alten Modulen oder Sprachversionen etc. Das kann zu Frustmomenten und unschönen Workarounds führen, die im schlimmsten Fall mehr schaden können als das Modulsystem genutzt hat.

Die Entscheidung zur Verwendung eines solchen Modularisierungstools muss sehr bewusst getroffen werden. Ich empfehle den Einsatz nur, wenn man damit ein konkretes Problem lösen kann, wie z. B. die Trennung der Module als Deployment-Einheiten oder wenn das Team auf herkömmliche Weise die Modularisierung nicht in den Griff bekommt. Proaktiv auf ein Modulsystem zu setzen wäre ein Fall von YAGNI: „You ain't gonna need it."

Fazit

Wir haben festgestellt, dass Modulithen eine lohnende Form der Softwarearchitektur sein können. Geschickt geschnittene Module sorgen für Struktur und Übersichtlichkeit in einer großen Codebase, womit sie beherrschbar und testbar wird. Das Innenleben der Module zu gestalten, ist eine Wissenschaft für sich, aber es gibt genügend Möglichkeiten zur Auswahl.

Aber wie kommunizieren Module? Wie managt man die Abhängigkeiten vieler Module, ohne dass sie sich zu einem Knäuel verheddern? Die Antworten auf diese und weitere Fragen wird der nächste Teil dieser Serie geben.

Arnold Franke wirkt als Entwickler und Berater für die Kunden der synyx GmbH & Co KG in Karlsruhe. Dabei baut er als Komplexitätsvermeider und Software Craftsman nachhaltige, pragmatische Lösungen mit sauberem Code.

In 3 Patterns zum skalierbaren Datenaustausch

Thorben Janssen *(Freiberufler)*

Wenn Microservices skalierbar und unabhängig sein sollen, wird der Datenaustausch zwischen ihnen schnell zur Herausforderung. Verteilte Transaktionen schaffen eine enge Kopplung und sind nicht länger eine Option. Das Gleiche gilt für synchrone Serviceaufrufe. Wir benötigen neue Ansätze, mit denen wir asynchron Daten austauschen und ihre Konsistenz sichern können. In vielen Architekturen werden dafür 3 Patterns eingesetzt: das Outbox-Pattern, das View-Database-Pattern und das SAGA-Pattern. Mit Hilfe des Outbox-Patterns werden Events in Kafka und Daten in der serviceeigenen Datenbank gespeichert. Darauf aufbauend können andere Services entweder die Daten in ihrer eigenen View-Datenbank speichern oder an einer komplexen SAGA zur Absicherung verteilter Schreiboperationen teilnehmen. Wie das alles genau funktioniert und worauf dabei zu achten ist, zeige ich in diesem Vortrag.

Links & Literatur

[1] https://martinfowler.com/articles/microservices.html

[2] https://www.thoughtworks.com/insights/articles/demystifying-conways-law

[3] Fowler, Martin: „Refactoring", Improving the Design of Existing Code, Addison Wesley, 2018

[4] https://github.com/indyarni/modulithpatterns

[5] Evans, Eric: „Domain Driven Design", Tackling Complexity in the Heart of Software, Addison Wesley Longman, 2003.

[6] https://alistair.cockburn.us/hexagonal-architecture/

[7] http://www.codingthearchitecture.com/2016/04/25/layers_hexagons_features_and_components.html

[8] https://books.sonatype.com/mvnex-book/reference/multimodule.html

[9] https://guides.gradle.org/creating-multi-project-builds/

[10] https://www.youtube.com/watch?v=bVaiTPYlHFE

[11] https://www.osgi.org

[12] https://www.oracle.com/corporate/features/understanding-java-9-modules.html

[13] https://github.com/jboss-modules/jboss-modules

[14] https://youtu.be/VSiETATcoVw

Architekturpatterns in Modulithen – Teil 2

Wer mit wem reden darf

So manche Codebase macht nur auf den ersten Blick einen aufgeräumten Eindruck. Schön in Packages sortierter Code ohne Abhängigkeitsmanagement ist wie ein „aufgeräumtes" Kinderzimmer, bei dem einem die Lawine entgegenkommt, wenn man es wagt, die Schranktür aufzumachen. Um zu verhindern, dass Abhängigkeitszyklen und wuchernde Queraufrufe den Code zu einem „Big Ball of Mud" machen, gilt es, höllisch aufzupassen.

von Arnold Franke

Module in Softwaremonolithen sind toll, das haben wir im ersten Teil dieser Artikelserie gelernt. Das Schneiden von Code in sinnvolle fachliche Kontexte, aus denen man Module bildet, ist dabei schon die halbe Miete zur erfolgreichen Umsetzung dieser Architekturform – aber eben nur die halbe. Um überhaupt in den Genuss ihrer Vorzüge zu kommen, ist es ebenso wichtig, die Interaktionen und Abhängigkeiten der Module sorgfältig zu modellieren.

Schnittstellendesign im Modulithen

Das fängt schon beim Design der Modulschnittstellen an. Welche Operationen soll ein Modul nach außen bereitstellen und in welcher Form gibt es den anderen Modulen seine Daten preis? Ebenso essenziell: Welches Modul darf überhaupt von wem verwendet werden und wie vermeidet man Abhängigkeitszyklen, die die Eigenständigkeit der Module wieder kaputtmachen?

Verliert man diese Fragen aus den Augen, entsteht schnell Wildwuchs im Monolithen. Alles ist auf einmal von jedem abhängig und die Zuständigkeiten zwischen Modulen verschwimmen, womit plötzlich alle Vorteile der modulithischen Architekturform dahin sind. Wartbarkeit und Erweiterbarkeit gehen den Bach runter, unerwartete Nebeneffekte häufen sich. Und das, obwohl man doch so schön geschnittene Module hat. Damit das nicht passiert, gibt es einige Patterns zu befolgen, Antipatterns zu vermeiden und diverse Tools, die einen dabei unterstützen. In diesem Artikel geht es darum, wie man sein Abhängigkeitsmanagement im Modulithen nachhaltig umsetzt. Wir werden uns dabei wieder am Beispiel der Domain „Kino" entlanghangeln.

Welche Methoden gehören zur Schnittstelle?

Ein wichtiges Konzept in modularer Architektur ist, dass ein Modul die Hoheit über seinen fachlichen Subkontext hat und als einziges bestimmen darf, wie man zugehörige Daten manipuliert. Um das zu ermöglichen, muss das Modul seine interne Logik und Daten verstecken und darf nur Methoden veröffentlichen, die den Clients erlaubte Operationen bereitstellen und die interne Logik in sich kapseln. Wie im ersten Teil der Serie behandelt, ist die einfachste Möglichkeit, das in Java zu bewerkstelligen, das bewusste Setzen der Access Modifier von Methoden – *package private* für Interna und *public* für die Schnittstelle.

Hat man sein Modul intern in Subpackages unterteilt, leidet die Kapselung wieder, da die Subpackages modulinterne Methoden als *public* deklarieren müssen, um sich gegenseitig verwenden zu können. Man hat dann immer noch die Möglichkeit, öffentliche Schnittstellen durch Konventionen zu definieren. Ein geeigne-

Artikelserie

Teil 1: Ordnung ins Chaos bringen
Teil 2: Wer mit wem reden darf
Teil 3: Die Bude sauber halten

tes Mittel dafür ist das klassische Facade-Pattern [1]. Man erstellt in jedem Modul ein Java-Interface, das alle Schnittstellenmethoden definiert. Die Clientkonvention (die jedes Teammitglied kennen muss) ist dann, dass Zugriffe auf ein Modul nur über dessen Interface erlaubt sind.

Ein häufig anzutreffendes Antipattern nennt sich „Visibility for Testing", das Setzen von privaten Methoden auf *public*, nur um sie testen zu können. Das ist ein Schuss ins Knie, da man sich die schöne Kapselung gleich wieder kaputt macht. Das Bedürfnis, das zu tun, ist oft ein Hinweis auf schlecht gewählte Klassengröße bzw. Methodenabstraktion. Entweder eine private Methode ist Teil einer zu testenden Unit, die bereits eine *public* oder *package private* Methode hat, oder sie gehört zu einer tieferen Abstraktionsebene, die man als eigene, testbare Klasse extrahieren sollte.

Noch gefährlicher wird es, wenn man die Kapselung der Module mit dem Shared-Database-Antipattern umgeht. Es ist in Ordnung, dass alle Module eines Modulithen in dieselbe Datenbank schreiben, solange jedes Modul die Hoheit über seinen eigenen Bereich der Daten hat. Sobald aber zwei Module dieselbe Tabelle beschreiben, besteht wieder die Gefahr von auseinanderlaufender Logik und Konflikten, weshalb der Missbrauch der Datenbank als Schnittstelle dringend zu vermeiden ist.

Wie sehen die Methoden der Schnittstelle aus?

Beim Methodendesign der öffentlichen Schnittstelle eines Moduls ist es selten sinnvoll, einfach nur CRUD-Methoden [2] bereitzustellen. Damit reicht man im Grunde nur die Operationen der Persistenzschicht weiter und gibt den Clients wieder die volle Macht über die Daten. Besser ist es, Business Operation Methods nach den tatsächlichen Operationen zu designen, die auf dem fachlichen Subkontext des Moduls erlaubt sind (Listings 1 und 2).

Man sieht, dass im ersten Fall die Clients volle Macht über die Reservierungsdaten haben und jeder damit seine eigene Version der fachlichen Logik implementiert, was dann zu Inkonsistenzen führt. Im zweiten Fall ist die Logik da implementiert, wo sie hingehört – nämlich im Modul *Reservierung*. So wird die Separation of Concerns [3] eingehalten. Das Modul weiß, wie die Daten zu manipulieren sind und auch, dass noch weitere Dinge zu tun sind – wie hier ein Event verschicken, was die Clients gar nicht wissen können. Die Methode *abholen()* kann man gar nicht falsch benutzen und sie macht den Clientcode leicht verständlich, ohne dass man weitere Erklärung durch Kommentare o. ä. braucht.

Was gibt die Schnittstelle nach außen?

Das letzte Puzzleteil ist die Form, in der ein Modul seine Daten preisgibt. Das i-Tüpfelchen des Schnittstellendesigns ist es, ausschließlich Immutable Objects nach außen zu geben. Die Unveränderbarkeit der Objekte garantiert dem Entwickler an jeder Stelle im Code, dass es sich um originale Daten in validem Zustand vom zuständigen fachlichen Modul handelt, egal welchen Weg durch die Anwendung sie schon hinter sich haben. Insbesondere in der großen Codebase eines Modulithen ist das ein großer Vorteil, der das Fehlerpotenzial nochmal deutlich reduziert.

Üblicherweise gibt es in einem großen Projekt unterschiedliche Sichten auf die Daten. Ein Nutzer im Internet will Daten vermutlich in einer anderen Form präsentiert bekommen als das Backend sie in der Datenbank speichert. Ein fachliches Modul sollte daher eine externe fachliche Abstraktion auf seine Daten bereitstellen, mit denen Clients und andere Module arbeiten können. Metainformationen wie Erstellungszeitstempel oder sogar die Datenbank-ID sind dafür nicht unbedingt relevant,

Listing 1: Schnittstelle mit CRUD-Methoden

```
// Implementierung der Serviceschnittstelle im Reservierungsmodul
public Reservierung read(Long id) {
  return repository.findById(id);
}
public void update(Reservierung reservierung) {
  repository.save(reservierung);
}

public void delete(Long id) {
  repository.delete(id);
}
// Verwendung der Schnittstelle in Clientmodul 1:
// Reservierung abholen
Reservierung reservierung = reservierungService.read(reservierungId);
reservierung.setAbgeholt(true);
reservierungService.update(reservierung);

// Verwendung der Schnittstelle in Clientmodul 2:
// Reservierung abholen
reservierungService.delete(reservierungId);
```

Listing 2: Schnittstelle mit Business Operation Method

```
// Implementierung der Operation "abholen" im Reservierungsmodul
private void abholen (Long reservierungId) {
  Reservierung reservierung = repository.findById(reservierungId);
  reservierung.setAbgeholt(true);
  Reservierung reservierung = repository.save(reservierung);
  eventPublisher.publishEvent(new ReservierungAbgeholtEvent(reservier
ung));
}
// Verwendung der Schnittstelle in Clientmodul 1:
reservierungService.abholen(reservierungId);
// Verwendung der Schnittstelle in Clientmodul 2:
reservierungService.abholen(reservierungId);
```

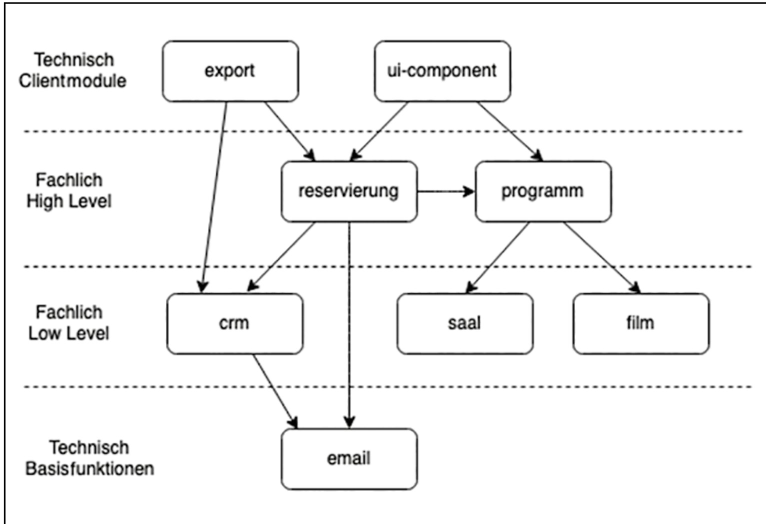


Abb. 1: Unidirektionaler Abhängigkeitsgraph

dafür kommen vielleicht weitere Informationen hinzu, die sich aus den Rohdaten ergeben.

Dabei muss man höllisch aufpassen, dass man nicht in das Antipattern „Too many layers of abstraction“ verfällt. Es gibt Projekte, die für jede technische Schicht und jedes externe System eine eigene Datenabstraktion einführen. Die schiere Menge an Konvertierungscode ist dann zu viel des Guten, und ein Feld in der gesamten Codebase hinzuzufügen, wird zur Mammutaufgabe. Hier schadet die Abstraktion mehr als sie nutzt. Verwende so wenig Abstraktion wie möglich, aber so viel wie nötig: „All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection“ [4].

Ein weiteres (Anti-)Pattern im Zusammenhang mit Datenrepräsentation ist „Active Record“ [5], wie es in manchen Web Application Frameworks existiert. Dabei hält ein in der Anwendung herumgereichtes Objekt nicht nur Daten, sondern auch zugehörige Logik und vor allem Methoden, die das Objekt direkt in die Datenbank persistieren können. Das bedeutet, dass man z. B. an jeder Stelle im GUI-Code einfach *.save()* auf einem x-beliebig veränderten Objekt aufrufen kann und es wird in der Datenbank aktualisiert. Was in einer kleinen Ein-Tabellen-Anwendung bequem und pragmatisch wirkt, wird in der großen Monolithen-Codebase zum Problem. Es wirft jede Kapselung von Logik über den Haufen, verletzt Single Responsibility und Separation of Concerns und kann zu einem Chaos führen, in dem sehr schwer nachzuvollziehen ist, wann und wo ein Objekt geändert wird.

Abhängigkeitsmanagement zwischen Modulen

Eine weitere Herausforderung, die sich in großen Softwaresystemen stellt, ist die Verdrahtung der Module untereinander. Welches Modul darf welches Modul verwenden? Darüber muss man sich sowohl bei einer Microservices-Landschaft als auch beim Modulithen Gedanken machen. Es ist dabei sinnvoll, sich die Hierarchie der Module bewusst zu machen. Welche fachlichen Module bieten eher grundlegende Funktionalität, welche bauen darauf auf und bedienen eher High Level Use Cases? Welche technischen Querschnittsmodule werden von der Fachlichkeit benutzt und welche müssen die Fachlichkeit kennen? Durch die Beantwortung dieser Fragen ergibt sich ein von oben nach unten gerichteter unidirektionaler Abhängigkeitsgraph, der eine leicht beherrschbare Struktur ins Projekt bringt.

Im Beispiel unserer Kinosoftware könnte man die eher stammdatennahen Module *saal* und *film* sowie das Kundenmodul als Low Level und die darauf aufbauenden *reservierung* und *programm* als High Level identifizieren. *email* steht am Ende des Graphen, weil es von der Fachlichkeit benutzt wird, und die *ui-component* sowie ein Datenexportmodul wären Beispiele für Module, die die Fachlichkeit kennen müssen. Auf einmal entsteht ein aufgeräumtes Bild unserer Anwendungsarchitektur, das man leicht in seinen Kopf bekommt, ohne viel Hintergrundwissen zu haben (**Abb. 1**).

Sehr wichtig ist dabei die Vermeidung von Abhängigkeiten, die in die falsche Richtung fließen. Insbesondere Cyclic Dependencies [6], also Zyklen im Abhängigkeitsgraphen, sind gefährlich. Wenn Module gegenseitig voneinander abhängen, dann macht das den modularen Ansatz kaputt, denn sie sind nicht mehr unabhängig voneinander verwend- und änderbar und bilden im Grunde ein neues Riesenmodul. Nebenwirkungen durch Änderungen in einem einzelnen Modul werden so viel schwerer abschätzbar, denn potenziell können dadurch alle Module im Zyklus und alle Module, die diese benutzen, beeinflusst werden oder kaputtgehen. In unserem

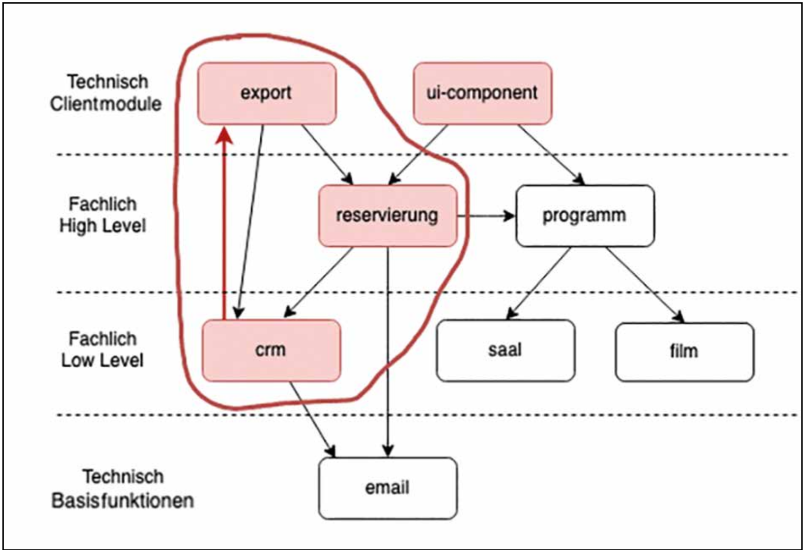
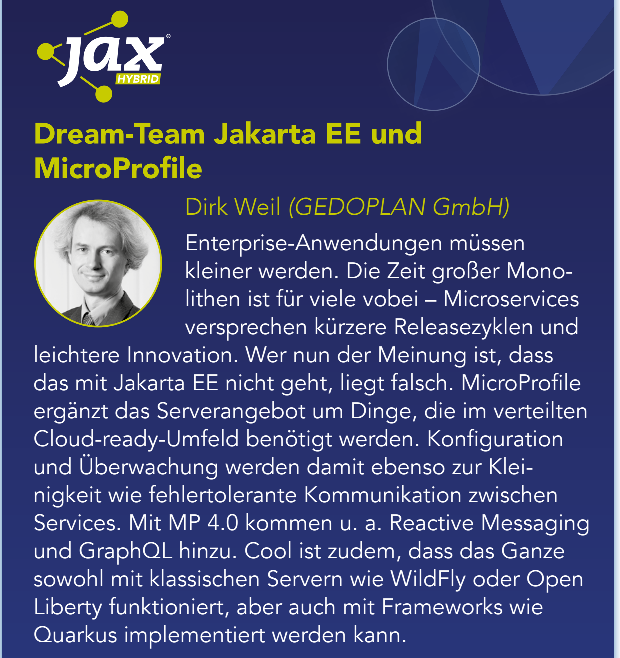


Abb. 2: Eine falsche Abhängigkeit führt zu einem großen Cycle

Beispiel in **Abbildung 2** ist das deutlich erkennbar, wenn man nur eine falsch gerichtete Abhängigkeit hinzufügt, wie die von CRM auf Export. Dadurch werden die Module *export*, *reservierung* und *crm* zu einem nicht mehr trennbaren Abhängigkeitszyklus. Wenn man jetzt im Modul *export* einen Fehler einbaut, kann dadurch das Modul *crm* kaputtgehen, was plötzlich auch *reservierung* und *ui-component* betrifft. Die halbe Anwendung ist auf einmal miteinander gekoppelt.

Steuern von Abhängigkeiten

Abhängigkeiten zwischen Modulen entstehen üblicherweise durch direkte Aufrufe und die Verwendung von Klassen eines anderen Moduls. Es gibt aber diverse Möglichkeiten, Abhängigkeiten darüber hinaus zu steuern.



Application Events

Indem man einen direkten Aufruf durch ein Application Event ersetzt, kann man die Abhängigkeit zwischen zwei Modulen umkehren. Ein Modul erzeugt ein Event eines bestimmten Typs, der zum Modul gehört. Ein anderes Modul registriert einen Event Listener, der auf diesen Event-Typ lauscht. Das empfangende Modul hat also eine Abhängigkeit zum sendenden Modul, indem es diesen Event-Typ kennt und nicht umgekehrt. Das ist insbesondere sinnvoll bei Aktionen, die nicht direkt zum fachlichen Use Case gehören, sondern eher Metaaufgaben übernehmen – wie zum Beispiel das Führen eines Auditlogs. In der Java-Welt ist so ein Pattern leicht selbst zu implementieren oder man nutzt z. B. das Application-Event-Feature [7] des Spring Frameworks wie in Listing 3.

Es ist in Modulithen so wie bei Microservices möglich, generell eine Event-getriebene Architektur umzusetzen, in der Interaktionen zwischen Modulen in erster Linie durch Events abgebildet werden. Dabei sind Aspekte wie Asynchronität, Transaktionen, Entkopplung der Module und gesteigerte Komplexität zu beachten und gegeneinander abzuwägen. Ein guter Kompromiss ist die Verwendung von Aufrufen bei direkten Kommandos an

Listing 3: Application Events

```
// Erzeugung des Events im Reservierungsmodul mit einem Spring Event
Publisher:
@Service
public class ReservierungService {
  private ApplicationEventPublisher eventPublisher;
  public void abholen(Long reservierungId) {
// Businesslogik der Methode
    [...]
    eventPublisher.publishEvent(new ReservierungAbgeholtEvent(
                                                        reservierung));
  }
}
// Reaktion auf das Event im Auditlogmodul mit einem Spring Event
Listener
@Component
public class ReservierungAbgeholtEventListener {

  @Eventlistener
  public void onApplicationEvent(ReservierungAbgeholtEvent event) {
  // Erzeugen des Auditlog-Eintrags aus dem Event
    [...]
  }
}
```

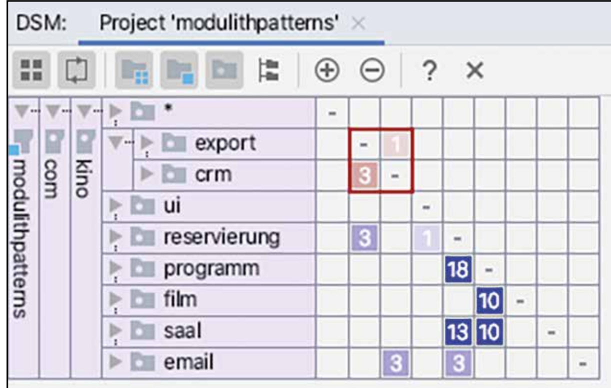


Abb. 3: IntelliJ Idea Dependency Matrix

ein Modul („Tu bitte das für mich") und die Verwendung von Events bei Status-Updates („Es ist etwas passiert, wer will, kann reagieren").

Plug-in Injection

Generell kann Dependency Injection dazu verwendet werden, Abhängigkeiten umzukehren, zum Beispiel im Spring Framework [8]. Modul A definiert ein Interface, das von Modul B als Bean implementiert wird. In Modul A kann man diese Bean jetzt mit dem Typ des Interface injizieren und ihre Methoden aufrufen, ohne das implementierende Modul B zu kennen. So können mehrere Module Plug-ins für die Funktionalität des aufrufenden Moduls schreiben, die dann dort als Liste injiziert und gleichförmig aufgerufen werden.

Das ermöglicht zum Beispiel die Umkehr der Abhängigkeiten des Exportmoduls zu den fachlichen Modulen aus **Abbildung 1**, sodass es zu einem Modul der Basisschicht wird, wie Listing 4 zeigt.

AOP

Aspektorientierte Programmierung [9] ermöglicht es, übergreifende Aspekte implizit auszuführen. Dafür definiert man beliebige Join Points, z. B. „alle Methodenaufrufe in Package xy" oder „alle Methoden, die Annotation @*XY* haben", auf die eine übergreifende Logik anzuwenden ist, und Aspects, die die auszuführende übergreifende Logik implementieren. Sogenannte Point Cuts definieren dann, an welchen Join Points welche Aspekte ausgeführt werden. Das Ganze ist so umsetzbar, dass sich die Module, in denen sich die Join Points befinden, und das Modul, in dem sich der Aspect befindet, nicht gegenseitig kennen. Dadurch kann man maximale Entkopplung erreichen, was allerdings auf Kosten der Übersichtlichkeit geht, denn durch AOP implizit ausgeführte Aktionen sind schwieriger nachvollziehbar als direkte Verbindungen.

Tooling

In einem großen Modulithen die Abhängigkeiten aller Module von Hand zu überwachen, ist ein Ding der Unmöglichkeit. Glücklicherweise gibt es jedoch eine ganze Reihe Tools, die während aller Phasen des Entwicklungsprozesses das Abhängigkeitsmanagement unterstützen.

Listing 4: Plug-in Injection

```
// Das Exportmodul definiert über ein Interface, in welcher Form es die
// Daten braucht:
package com.kino.export;
public interface ExportPlugin {
  String export(Long kundeId);
}

// Die fachlichen Module implementieren Plugins und entscheiden damit
// selbst, ob und welche Daten sie exportieren:
package com.kino.crm;
@Component
public class KundeExportPlugin implements ExportPlugin{
  @Autowired
  KundeService kundeService;
  @Override
  public String export(Long kundeId) {
    Kunde kunde = kundeService.findById(kundeId);
    return kunde.getName() + kunde.getTelefonnummer();
  }
}

package com.kino.reservierung;
@Component
public class ReservierungExportPlugin implements ExportPlugin {
  @Autowired
  ReservierungService reservierungService;
  @Override
  public String export(Long kundeId) {
    Reservierung reservierung = reservierungService.findByKundeId(kundeId);
    return reservierung.getNummer() + reservierung.getVorfuehrung();
  }
}

// Die Plugins werden ins Exportmodul injiziert und dort generisch
// aufgerufen. Die Rückgabewerte können verwendet werden, ohne die
// fachlichen Module zu kennen:
package com.kino.export;
@Service
public class ExportService {
  @Autowired
  List<ExportPlugin> exportPlugins;
  public String export(Long kundeId) {
    return exportPlugins.stream()
      .map(plugin -> plugin.export(kundeId))
      .collect(Collectors.joining(","));
  }
}
```

IntelliJ Idea und Eclipse

Idea bringt bereits einfache Werkzeuge zur Architekturanalyse mit. Man kann über seinen Code eine Dependency-Matrix generieren [10], die auf Package- und Klassenebene genau aufzeigt, wie viele Abhängigkeiten von wo nach wo bestehen. Verletzungen der Unidirektionalität erkennt man schnell an Abhängigkeiten, die über der Diagonale der Matrix auftauchen. Cycles kann man auflisten und identifizieren, wodurch sie verursacht werden. Für einen ersten Überblick über die Qualität der Codestruktur ist das meist schon ausreichend (**Abb. 3**; der Cycle aus **Abb. 2** ist sofort erkennbar).

In Eclipse kann man sich verschiedener Plug-ins bedienen, um Abhängigkeiten zu visualisieren und Cycles zu erkennen – wie eDepend [11] oder Java Dependency Viewer [12].

Archunit

Archunit [13] ist eine Testing Library für Architekturregeln. Das API erlaubt es, Codebereiche wie Schichten, Slices etc. zu definieren. Dann kann man Regeln für Zugriffe zwischen den Bereichen festlegen und sie bei jedem Testlauf prüfen. Das eignet sich perfekt zur Definition von Modulen und Festlegung der Abhängigkeitshierarchie zwischen ihnen. Auch Dependency Cycles zwischen Modulen können automatisch erkannt werden, wie Listing 5 zeigt.

Auf Archunit aufbauend hat der Modulithen-Experte Oliver Drotbohm die Spring Boot Erweiterung Moduliths [14] kreiert, die erst vor kurzem in der Version 1.0.0 erschienen ist. Damit kann man sowohl durch einen einfachen Test die grundlegende modulare Struktur eines Spring-Boot-Modulithen sicherstellen (Listing 6) als auch komplexere Abhängigkeiten innerhalb einer Spring-Anwendung testen.

Listing 5: Archunit

```
@Test
void modulesAreFreeOfCycles() {
  JavaClasses classes = new ClassFileImporter().importPackages("com.
kino");
  ArchRule rule = slices()
    .matching("..kino.(*)..")
    .should().beFreeOfCycles();
  rule.check(classes);
}
// Mit unserem Dependency Cycle von Abb.2 schlagen die Tests mit diesem
Output fehl:
java.lang.AssertionError: Architecture Violation [Priority: MEDIUM] -
      Rule slices matching ..kino.(*)..' should be free of cycles' was violated
           (1 times):
Cycle detected: Slice crm -> Slice export -> Slice crm
[...]
```

jQAssistant

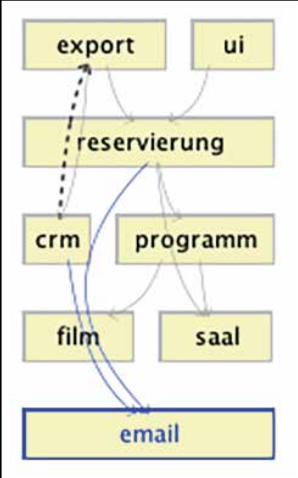


Abb. 4: Structure 101 generiert aus dem Code einen Abhängigkeitsgraph

jQAssistant [15] ist ein Maven-Plug-in, das eine Neo4j-Graphdatenbank mitbringt. Es kann die Codestruktur einer Anwendung in einen Graph einlesen und darauf dann Analysen fahren. Zu prüfende Regeln werden in Cypher definiert, der Query Language für Neo4j. So können neben einfachen Regeln wie Namenskonventionen von Klassen auch komplexe Strukturen auf Modulebene wie Cycles verifiziert werden (Listing 7). Die Einstiegshürde ist etwas höher, weil man sich mit Neo4j und Cypher auseinandersetzen muss, dafür sind die Möglichkeiten zur Definition von Regeln praktisch unbegrenzt.

Structure 101 und Co.

Structure 101 [16] ist eine kostenpflichtige Anwendungssuite für Softwarearchitekten. Das Herzstück ist eine Desktopanwendung, die den Code, den man ihr

Listing 6: Moduliths

```
@RunWith(SpringRunner.class)
@ModuleTest
public class ModulithsTest {
  @Test
  public void verifyModuleStructure() {
    Modules.of(KinoApplication.class).verify();
  }
}

// Output:
java.lang.AssertionError: Architecture Violation [Priority: MEDIUM] - Rule
     'slices matching 'com.kino.(*)..' should be free of cycles' was violated (
                                                                    1 times):
Cycle detected: Slice crm -> Slice export -> Slice crm
```

Listing 7: jQAssistent Cypher Query

```
----
 MATCH
  (c1:Component)-[:DEFINES_COMPONENT_DEPENDENCY]->(c2:Component),
  cycle=shortestPath((c2)-[:DEFINES_COMPONENT_DEPENDENCY*]->(c1))
 RETURN
  c1 as Component, nodes(cycle) as Cycle
----
```

vorwirft, automatisch analysiert und dann verschiedene Sichten und Auswertungen dafür bereitstellt. Für Modulithenbauer ist besonders interessant, dass man seine Module in der grafischen Oberfläche 1:1 nachmodellieren und daraus einen Soll-Abhängigkeitsgraph bauen kann (**Abb. 4**; unser Cycle ist durch den gestrichelten Pfeil erkennbar). Darin ist die Modulhierarchie definiert, die eingehalten werden muss, Verletzungen werden als Fehler grafisch hervorgehoben. Der Clou ist, dass man diese Soll-Architektur per DIE-Plug-in entwicklungsbegleitend verifizieren kann – Verletzungen werden erkannt, noch während man den Code schreibt. Für die Integration in Continous Integration Pipelines gibt es ein CLI, das den Abhängigkeitsgraph und weitere Regeln verifizieren und bei Verletzungen den Build fehlschlagen lassen kann. Auch beim Refactoring hilft Structure 101, indem man im Architekturdiagramm Klassen und Packages verschieben kann, um zu sehen, wie sich die Änderung auf die Abhängigkeiten auswirkt. In die gleiche Kerbe schlagen auch eine Reihe weiterer Architektursuiten wie Sonargraph/Sotograph [17], Lattix [18], Teamscale [19] und andere.

Fazit

Ebenso wichtig wie der Schnitt der Module sind ihre Interaktionen. Durch geschicktes Schnittstellendesign können wir die Nutzung eines Moduls für Entwickler intuitiv und frei von unerwünschten Nebenwirkungen gestalten. Die Struktur der Anwendung wird durch bewusstes Erzeugen und Vermeiden von Abhängigkeiten zwischen Modulen in einen Zustand versetzt, der es leicht macht, Änderungen an der richtigen Stelle durchzuführen und Auswirkungen zu kontrollieren. Diesen Zustand zu erreichen und zu erhalten, wird durch den Einsatz von bestimmten Patterns und unterstützenden Tools erleichtert, wobei der Nutzen den Mehraufwand schnell überwiegt. Bleibende Herausforderungen sind das Testen der monolithischen Codebase, das Durchführen von weitreichenden Refactorings und die Weiterentwicklung der Anwendungs- und Systemarchitektur eines Modulithen, die im letzten Teil der Serie adressiert werden.

Arnold Franke wirkt als Entwickler und Berater für die Kunden der synyx GmbH & Co KG in Karlsruhe. Dabei baut er als Komplexitätsvermeider und Software Craftsman nachhaltige, pragmatische Lösungen mit sauberem Code.

@indyarni

Links & Literatur

[1] Gang of Four – Design Patterns https://archive.org/details/designpatternsel00gamm/page/185

[2] CRUD: https://www.codecademy.com/articles/what-is-crud

[3] Separation of Concerns: https://en.wikipedia.org/wiki/Separation_of_concerns

[4] Wheeler, David: https://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)

[5] Active Record: Martin Fowler – Patterns of Enterprise Application Architecture

[6] Cyclic Dependencies: https://www.lattix.com/why-cyclic-dependencies-are-bad/

[7] Spring Application Events: https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#context-functionality-events

[8] Spring IOC: https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#beans

[9] Aspektorientierte Programmierung: https://de.wikipedia.org/wiki/Aspektorientierte_Programmierung

[10] Idea Dependency Matrix: https://www.jetbrains.com/help/idea/dsm-analysis.html

[11] eDepend: https://marketplace.eclipse.org/category/free-tagging/edepend-graphical-dependency-analysis-tool-340

[12] Java Dependency Viewer: https://marketplace.eclipse.org/content/java-dependency-viewer

[13] Archunit: https://www.archunit.org

[14] Moduliths: https://github.com/odrotbohm/moduliths

[15] jQassistant: https://jqassistant.org

[16] Structure 101: https://structure101.com

[17] Sonargraph/Sotograph: https://www.hello2morrow.com/products/sonargraph

[18] Lattix: https://www.lattix.com

[19] Teamscale: https://www.cqse.eu/en/teamscale/overview/

The easy Way to secure Microservices

Michael Hofmann *(Hofmann IT-Consulting)*

Jeder produktiv betriebene Microservice muß mit Security abgesichert sein. Um dies sicherzustellen, entsteht im Vergleich zu einem monolithischen System durch die hohe Anzahl an Services ein signifikanter Mehraufwand. Erfolgt der Betrieb dann noch in einer Public-Cloud, so darf weder die Kommunikation innerhalb der Infrastruktur des Cloud-Betreibers, noch die Verbindung über das Internet im Klartext erfolgen. Darüber hinaus müssen in jedem einzelnen Service entsprechende Autorisierungsprüfungen stattfinden. In dieser Session wird gezeigt, wie einfach und wie wenig aufwändig die Umsetzung der Securitymaßnahmen mit einem Service-Mesh-Tool, wie beispielsweise Istio, erfolgen kann. Mit ein paar kleinen Istio-Regeln wird die gesamte Kommunikation im Service Mesh mit mutual TLS (mTLS) abgesichert. Auch grundlegende Prüfungen der Service-to-Service-Kommunikation und der End-User-Autorisierung mittels JWT können von Istio übernommen werden. Die weiterführenden Berechtigungsprüfungen innerhalb eines Java Services werden durch die Verwendung der MicroProfile-Spezifikationen veranschaulicht.

Architekturpatterns in Modulithen – Teil 3

Die Bude sauber halten

Puh, endlich geschafft. Die Artikelserie geht dem Ende zu. Diese Menge Code zu einem anständigen Modulithen zu formen, war ganz schön anstrengend. Zum Glück ist er jetzt fertig, alle Arbeit ist getan! Wie? Weiterentwicklung? Wartung und Betrieb? Neue Anforderungen? Das Team skalieren? Technische Schulden? Aber das Ding ist doch ganz neu! Warum müssen wir da schon wieder ran? Tja, machste nix. Oder doch?

von Arnold Franke

Klar machen wir da was! Genau wie jede andere Software bleibt ein Modulith nur dann am Leben, wenn kontinuierlich weiter an ihm gearbeitet wird. Wenn man nicht ohnehin schon ständig neue Features und geänderte Anforderungen umsetzt, dann sind Sicherheitsupdates, abgekündigte Versionen, sich ändernde Abhängigkeiten, neue Tools und Erkenntnisse genug Gründe für nicht abreißende Weiterentwicklung. In einer großen, modulithischen Codebase gibt es dafür eine Menge Herausforderungen. Das können sein: großflächige Refactorings, häufiges Hinterfragen der System- und Anwendungsarchitektur und das Erreichen einer hohen Testabdeckung aller relevanten Aspekte, um dabei nichts kaputt zu machen. Wenn man das alles im Griff hat, dann spricht auch nichts dagegen, den Modulithen ganz Microservice-like mehrmals am Tag über die Continuous Delivery Pipeline auf Prod zu pushen.

Das Testuniversum

Je größer ein Modulith ist, desto mehr verschiedene Aspekte sind für die reibungslose Funktion entscheidend. Der Code in den Modulen muss tun was er soll, außerdem sollen die Module zusammengenommen das erwartete Verhalten an den Tag legen. Die Integration mit externen Abhängigkeiten und internen beweglichen Teilen wie Datenbanken, Caches und Events muss stabil laufen. Es gibt nichtfunktionale Anforderungen wie Performance, Resilience, Security. Fachliche Akzeptanzkriterien müssen eingehalten werden. Wie soll man all das testen und dabei nicht verrückt werden?

Es ist keine schlechte Idee, sich erst einmal an der klassischen Testpyramide [1] zu orientieren. Eine möglichst komplette Unit-Test-Abdeckung möchte man wahrscheinlich in den meisten Projekten haben. Aber braucht man Lasttests, wenn man nur 1000 Requests pro Tag beantwortet? Wie wichtig ist das automatisierte Sicherstellen von Performance, wenn kein echter User auf Antwort wartet? Welche Integrationstests braucht das Projekt unbedingt, welche sind vernachlässigbar? Muss man jeden fachlichen Edge Case abdecken oder reicht der Happy Path? Solche Fragen sollte man sich stellen und sich dann bewusst entscheiden, welchen Aspekt man in welcher Intensität automatisiert testen will. Dabei wird man den einen oder anderen Kompromiss zwischen kompletter Abdeckung aller Aspekte und dem dafür notwendigen Aufwand eingehen müssen.

Um ein einheitliches Bild in den Köpfen des Teams zu etablieren, eignet sich eine Visualisierung aller zu testenden Aspekte und der zugehörigen Testarten wie die Tabelle in **Abbildung 1** zeigt.

Dort sieht man auf den ersten Blick anhand der grünen Haken, welche Art von Test in welcher Technologie für das Testen welches Aspekts zuständig ist. Die gelben Symbole visualisieren unvermeidbare Überschneidungen. So kann es beispielsweise passieren, dass durch eine fehlerhafte Modulintegration auch ein fachlicher Akzeptanztest auf die Bretter geht, weil er diese Integration durchläuft, obwohl er sie gar nicht explizit testen will. Die API-Security-Tests dagegen werden dadurch nicht behelligt.

Artikelserie

Teil 1: Ordnung ins Chaos bringen
Teil 2: Wer mit wem reden darf
Teil 3: Die Bude sauber halten

Testart: / Aspekt:	Unittests (JUnit 5)	Integration Tests (mit AppContext)	BDD Tests (Cucumber)	Pen Tests (OWASP ZAP)
Code Unit	✅	⚠️	⚠️	
Modulintegration		✅	⚠️	
Akzeptanzkriterien			✅	
API Security				✅

✅ Test testet explizit diesen Aspekt
⚠️ Test kann implizit rot werden durch Fehler in diesem Aspekt

Abb. 1: Die Testmatrix visualisiert, welcher Test was testet

In einem großen Modulithen hat so eine Matrix für gewöhnlich noch deutlich mehr Zeilen und Spalten als in **Abbildung 1**. Man kann so eine Definition des Testuniversums in einem Teamworkshop als Ist-Zustand und Soll-Zustand erarbeiten. Am Ende hat jeder ein einheitliches Bild im Kopf und weiß, wo die Reise hingeht. Testentwicklung läuft koordinierter ab, während das Team Sicherheit gewinnt, denn fehlende Tests gehen einem jetzt nicht mehr so schnell durch die Lappen, und man weiß genau, wo man hin greifen muss, wenn ein Test rot wird.

Wie wichtig ist die Testabdeckung?

Ergänzt wird das Konzept durch Meaningful Test Coverage [2]. Das bedeutet zunächst, dass man nicht nur sinnlose Tests schreibt, die einfach jede Zeile Code durchlaufen, um eine hohe Coverage zu erzielen. Jeder Test sollte einen sinnvollen Aspekt testen und entsprechend spezifische Assertions durchführen. Außerdem bedeutet es, dass man sich bewusst entscheidet, manchen Code **nicht** zu testen, wie z. B. generierten Code, Testcode, Konfigurationscode, und ihn dann auch aus der Coverage-Analyse auszuschließen. Auf fachlicher Ebene kann das bedeuten, dass man vielleicht nicht jeden fachlichen Edge Case automatisiert testen will, sondern nur die essenziellen Use Cases. Bei solchen Entscheidungen sollten die entsprechenden Stakeholder ein Wörtchen mitzureden haben. Am Ende steht das hehre Ziel, möglichst 100 Prozent der Dinge, die man tatsächlich testen möchte, durch sinnvolle Tests abzudecken. Das gilt für die Zeilen Code bei Unit-Tests, für Integrationspunkte bei Integrationstests, fachliche Aspekte bei Akzeptanztests und entsprechend auch für alle anderen Arten von Tests.

Isolation von Tests

Mit einem Test nur exakt den Aspekt anzusprechen, den man damit auch testen will, ist oft gar nicht so einfach. Jedes getestete Stück Code verwendet andere Klassen oder Abhängigkeiten, die man eigentlich nicht mittesten will. Wenn man für den Integrationstest einer Schnittstelle die ganze Anwendung hochfährt, braucht man auch alle Abhängigkeiten wie Datenbank, Filesystem, Messagingsysteme, die mit der Schnittstelle gar nichts zu tun haben. In der komplexen Codebase eines Modulithen ist das so viel, dass Laufzeit, Ressourcenverbrauch und Aussagekraft der Tests stark darunter leiden. Daher ist die Isolation des zu testenden Aspekts beim Testaufbau eine Kunst für sich. Im Modulithen trifft man auf unterschiedlichste Isolationsszenarien. Daher ist es sinnvoll, dafür einen Werkzeugkasten mit den richtigen Isolationsmethoden parat zu haben.

Auf Unit-Test-Ebene gibt es für Java eine Menge an Mocking Libraries wie Mockito [3] und Powermock [4], die es leicht machen, eine Code Unit zu isolieren, indem man Mocks für alle Abhängigkeiten konfiguriert. Im Spring Framework gibt es viele Möglichkeiten, für integrative Tests nur genau die Teile der Anwendung hochzufahren, die man braucht, zum Beispiel die Spring Boot Test Slices [5] oder eigene Context-Konfigurationen. Auch sehr nützlich ist das realitätsnahe Nachstellen von Requests auf eigene Web-Controller mit Spring MockMVC [6].

Um externe Systeme in Tests außen vor zu lassen, kann man in einem eigenen Testprofil die Implementierungen der Interfaces zu den externen Systemen mit Mocks ersetzen und hat damit seine Anwendung nach außen isoliert. Eine (oft bessere) Alternative hierzu ist es, stattdessen die externe Abhängigkeit selbst durch einen Testmechanismus zu ersetzen und damit die Integration zur Abhängigkeit mitzutesten. Zum Beispiel kann man mit Wiremock [7] mit geringem Aufwand externe Webserver simulieren und genau konfigurieren, wie sie sich im Test verhalten sollen. Andere externe Abhängigkeiten wie Message-Systeme, Datenbanken, Verzeichnisdienste können mit den auf Docker aufbauenden Testcontainers [8] sehr schnell während des Tests hoch- und runtergefahren werden (Listing 1). Im Test wird ein Container mit einer echten MySQL gestartet. *@DataJpaTest* fährt den benötigten Slice des Spring Contexts hoch.

Listing 1

```
@DataJpaTest
class ReservierungRepositoryTest {
@Autowired
  private ReservierungRepository reservierungRepository;
@Test
  void testMySql() {
    try (MySQLContainer<?> mySqlContainer = new MySQLContainer<>()
        .withDatabaseName("kino")
        .withUsername("foo")
        .withPassword("bar")) {
    mySqlContainer.start();
      assertThat(reservierungRepository.findById(1L)).isNotNull();
    }
  }
}
```

Refactoring im großen Stil

Refactorings in einem Monolithen sind manchmal furchteinflößend. Wenn man einfach drauflos wurschtelt, verzettelt man sich schnell oder kommt in die Situation, Änderungen in hunderten Dateien auf einmal zu mergen. Oder man manövriert sich in aussichtslose Lagen, in denen man die ganze Arbeit wieder zurückrollen kann. Daher sollte man für große Refactorings einen Plan haben. Nach Schema F kann man dabei leider nur selten vorgehen, es gibt aber ein paar hilfreiche Methoden und Arbeitsweisen, die sich grob in „hilfreich für Fleißarbeit" und „hilfreich für Strukturänderung" unterteilen lassen.

Refactoring – Fleißarbeiten

Hierzu zählen Refactorings, die an sich nicht komplex sind, aber viele Stellen in der Codebase betreffen und daher nicht mal eben schnell runtergehackt sind. Ein Beispiel: Man will im gesamten Code alle Dependency Injections von Field Injection auf Constructor Injection umbauen, da man das für sinnvoll hält [9]. Oder man will einen Großteil der Integrationstests in Unit-Tests überführen. Das kann in einem großen Monolithen hunderte Klassen betreffen, die man von Hand ändern muss. Die Stellen zu identifizieren, ist in einem Monolithen meist einfacher als im verteilten System, denn man hat ja den gesamten Code in einem Repo. Eine Volltextsuche oder die IDE listen schnell alle Baustellen auf.

Je nach Dringlichkeit und Größe des Refactorings kann man natürlich einen Unglücksraben bestimmen, der u. U. über Wochen alles auf einmal durchrockt und sich danach nochmal genauso lange Urlaub nehmen muss. Oft ist es aber klüger, den Zustand über die Zeit hinweg mit kleinen Änderungen zu verbessern. Dafür ist es erst einmal notwendig, dass jedes Teammitglied den Zielzustand genau kennt und den Grund für das Refactoring versteht. Das erreicht man am einfachsten, wenn man die Änderung an einem echten Beispiel gemeinsam durchspielt. Danach ist die wichtigste Regel die Boyscout Rule [10]. Man verlässt jedes Stück Code, das man anfasst, sauberer, als man es angetroffen hat – und dazu gehört auch, das geplante Refactoring an dieser Stelle durchzuführen. So führt sich das Refactoring über die Zeit „von allein" durch, bis man dann irgendwann den letzten Rest in einem Rutsch wegarbeitet.

Für das Boyscouting und die Beobachtung des Fortschritts ist es manchmal hilfreich, alle zu ändernden Stellen im Code zu markieren, wofür sich in Java zum Beispiel Marker-Annotationen eignen. Man erstellt eine Annotation wie *@ShouldBeAUnitTest* und klatscht sie erst einmal an jede Testklasse, die man umbauen will. Wer über die Annotation stolpert, der weiß dann gleich, was zu tun ist, und man kann von der IDE leicht die verbleibenden Stellen zählen lassen. Codekommentare sind dafür ein unzureichender Ersatz, da sie nicht so einfach referenzierbar und anfällig für Tippfehler sind.

Es kann sich auch lohnen, eine Refactoring-Kennzahl einzuführen, die den kontinuierlichen Fortschritt greifbar macht und immer präsent ist. Zum Beispiel ist es möglich, ein Plug-in für SonarQube zu schreiben, das in jeder statischen Codeanalyse die Anzahl der verbleibenden Field Injections ausweist [11]. Das Gleiche lässt sich per ArchUnit [12] auch zum Testzeitpunkt prüfen und in eine Datei oder einen beliebigen anderen Store ausgeben. So kann man die Kennzahl nach und nach verbessern, ohne dass deswegen gleich Tests fehlschlagen (Listing 2).

Strukturelle Refactorings

In den ersten beiden Teilen der Serie haben wir viel Zeit damit verbracht, die interne Architektur des Modulithen mit Modulschnitten und bewussten Abhängigkeiten zu modellieren. Um diese hohe Qualität in der Struktur zu halten, muss man sie während der Weiterentwicklung immer wieder hinterfragen. Sollte man hier eine Abstraktion einführen? Verdient dieser Subcontext ein eigenes Modul? Sollte die Abhängigkeit dieser beiden Module nicht umgedreht werden? Dadurch identifizierte strukturelle Refactorings sind häufig größerer Natur.

Listing 2

```
@Test
void checkRemainingFieldInjections() {
  JavaClasses classes = new ClassFileImporter().importPackages("com.kino");
ArchRule rule = FreezingArchRule.freeze(
              fields()
                .should()
                .notBeAnnotatedWith(Autowired.class));
rule.check(classes);
}
// Führt zum diesem Fileoutput:
Field <com.kino.export.ExportService.kundeService> is annotated with
                                        @Autowired in (ExportService.java:0)
Field <com.kino.export.ExportService.reservierungService> is annotated
                                   with @Autowired in (ExportService.java:0)
```

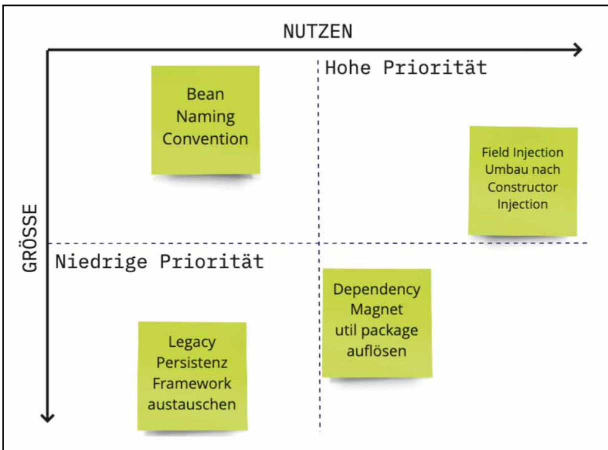


Abb. 2: Das Schulden-Board visualisiert und priorisiert technische Schulden

Man geht sie am besten an, indem man sie möglichst in Teilschritte zerlegt, die jeweils einen funktionierenden und deploybaren Zwischenstand haben. Durch häufigeres Mergen kleinerer Änderungen und einen kürzeren Feedbackzyklus kann man das Risiko eines großen Refactorings reduzieren und auftretende Probleme gezielter beheben. Wenn man ein großes Refactoring auf einmal ohne Zwischenschritte durchführt, dann ist auch der Schmerz beim Merge größer, und bei danach entdeckten Bugs ist es schwieriger, die Ursache davon in dem Riesen-Merge zu identifizieren.

Es gibt Tools, mit denen man so ein großes Refactoring im Voraus durchspielen kann. In einem Structure101-Architekturdiagramm [13] kann man Klassen oder Packages per Drag and Drop verschieben und sieht sofort, welche Auswirkungen die Änderung hat. So stellt man oft schon vorher fest, ob ein Refactoring den gewünschten Effekt hat, kann potenzielle Probleme identifizieren und Teilschritte ableiten. Einen gewissen Schutz gegen unbeabsichtigte Verletzungen der Architektur bieten automatisierte Strukturtests, wie z. B. ArchUnit, die Architekturregeln definieren, gegen die der Code bei jedem Testlauf geprüft wird.

Technische Schulden

Der Beweggrund für Refactorings sind häufig technische Schulden – also technische Mängel, die man in der Vergangenheit bewusst in Kauf genommen hat oder die sich über die Zeit angesammelt haben. Das ist per se nichts Schlimmes, solange man nicht die Kontrolle darüber verliert. Gerade in großen Projekten besteht die Gefahr, dass unbemerkt ein Berg entsteht, der nie wieder abzutragen ist und die Weiterentwicklung auf ewig bremst. Daher ist es wichtig, Auswirkung und Umfang der technischen Schulden immer im Blick zu haben und zu kontrollieren, indem man den Berg durch gezielte Refactorings kontinuierlich verringert.

Für Fans von Post-Its ist das technische Schulden-Board ein guter Weg, den Überblick über den Berg zu behalten (**Abb. 2**). Es handelt sich um ein Priorisierungs-Board mit zwei Achsen für den Mehrwert und die Größe einer technischen Schuld. Jedes Mal, wenn das Team eine technische Schuld identifiziert, wird sie auf dem Board in den beiden Achsen eingeordnet. Diese Visualisierung hilft bei der Einschätzung der Gesamtschuld und bei der Priorisierung der nächsten Refactorings – die kleinsten Schulden mit dem höchsten Mehrwert werden zuerst bearbeitet.

Statische Codeanalyse kann sowohl bei der Erkennung als auch bei der Verfolgung von technischen Schulden helfen. In Tools wie SonarQube [14] erkennt man Hotspots zum Beispiel schnell an Kennzahlen wie der Cyclomatic Complexity [15] und der Duplikationsrate. Auch generelle Gesundheitsmetriken wie die Größe von Klassen und Funktionen können Hinweise auf problematische Stellen geben. Wie oben beschrieben, kann man technische Schuld auch durch eigene Kennzahlen visualisieren, die man in einem SonarQube-Plug-in anzeigt oder von automatisierten Tests ausgeben lässt.

Die Reißleine

Im schlimmsten Fall kann es passieren, dass ein großes Softwareprojekt durch eigene und äußere Restriktionen in einen so maroden Zustand gerät, dass die Weiterentwicklung zu langsam und aufwendig wird, um sich zu lohnen, oder das Risiko zu groß wird, dabei etwas kaputtzumachen. Man spricht dann von einem Big Ball of Mud. Bevor man aufgibt und alles wegschmeißt, kann man immer noch die Reißleine ziehen und eine technische Rettungsphase starten. Man identifiziert die größten Probleme, die eine Weiterentwicklung verhindern, und fokussiert sich darauf, diese mit gezielten Refactorings zu beseitigen, während man die fachliche Weiterentwicklung stoppt oder zumindest auf das absolut Nötigste reduziert. Ein solcher Kraftakt ist für alle Stakeholder unangenehm, aber oft die bessere Alternative zur Kapitulation. Damit einhergehen muss aber auch ein Kulturwandel, der neue Arbeitsweisen und Mindsets mit sich bringt – sonst wird sich an der Qualität des Ergebnisses auch nichts ändern und am Ende des Rettungsversuchs steht gleich der nächste Big Ball of Mud.

Weiterentwicklung des Modulithen

Bei Software in der Größenordnung eines Modulithen ist der Funktionsumfang für gewöhnlich nie „komplett". In einer großen fachlichen Domain gibt es immer Raum für neue Ideen und Funktionen, für einfachere Bedienung und mehr Automatisierung. Es gibt viele Voraussetzungen zu erfüllen, um den Entwicklungs-Flow nachhaltig zu sichern.

Continuous Delivery

Die Größe des Modulithen sollte keinen Einfluss darauf haben, wie oft der aktuelle Codestand an den Benutzer ausgeliefert wird. Wenn alle wichtigen Aspekte durch automatische Tests abgedeckt sind, das Entwicklungsteam seinen Code kontinuierlich integriert und sowohl Build-Prozess als auch Deployment automatisiert sind, spricht nichts dagegen, Continuous Delivery zu praktizieren

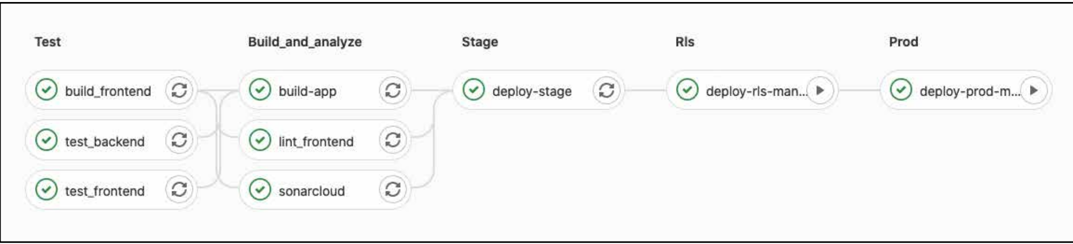


Abb. 3: Eine CI/CD Pipeline dauert beim Modulithen etwas länger, ist aber unverzichtbar

und jedes fertige Feature sofort auf Produktion auszurollen (**Abb. 3**). Natürlich führen lange Compile- und Build-Zeiten, die Menge an Tests und die Größe des Artefakts dazu, dass dieser Prozess länger dauert als beim typischen Microservice. Aber es ist immer noch besser, ihn kontinuierlich anzustoßen als nur alle x Wochen nach festem Releasezyklus. Dadurch verringern sich die Time to Market, das Risiko und die Kosten pro Release. Die Qualität der Software ist immer in auslieferbarem Zustand, und langfristige Planungen werden nicht durch den Releasezyklus eingeschränkt.

Das Festhalten an einem Releasezyklus kann je nach Projektcharakter und Rahmenbedingungen gute Gründe haben, ist aber häufig nur ein Hinweis, dass das Vertrauen in Qualität und Tests der Software noch nicht ausreicht. Zum Glück sind das meist Dinge, die man selbst in der Hand hat und die man aus eigener Kraft in einen Continuous-Delivery-fähigen Zustand bringen kann.

Cognitive Load und Komplexität

Ein Modulith wird optimalerweise von einem Team moderater Größe weiterentwickelt und betreut. Dabei muss das Team die fachliche Domain gut kennen, wissen, wie diese im Code umgesetzt ist, alle eingesetzten Technologien beherrschen, alle Abhängigkeiten kennen, die Architektur im Blick haben, die Betriebsinfrastruktur verstanden haben und so weiter und so fort. Alle Aspekte, die zur Entwicklung des Modulithen gehören, müssen in den Köpfen der Teammitglieder gleichzeitig Platz haben – man nennt das Cognitive Load [16]. Wenn diese zu groß wird und die Köpfe des Entwicklungsteams „voll“ sind, leidet darunter sowohl die bestehende Software als auch jede Weiterentwicklung. Teile des Systems werden vernachlässigt, weil man sich nicht mehr um alles kümmern kann. Die Qualität und die Umsetzungsgeschwindigkeit sinken, weil man nicht mehr alle Zusammenhänge im Kopf haben kann. Es bleibt weniger Zeit für Weiterentwicklung, weil der Erhalt des Status quo so viel Anstrengung bedeutet.

Das ist einer der vielen Gründe, warum es so unglaublich wichtig ist, das KISS-Prinzip (Keep it simple, stupid) [17] auf allen Ebenen einzuhalten und stets den einfachsten möglichen Weg zu wählen, um ein Problem zu lösen. Es ist eine der wichtigsten Voraussetzungen für nachhaltige Software, dass man es schafft, die Komplexität niedrig zu halten, während man weiterentwickelt und verbessert. Dementsprechend ist die Vermeidung von Komplexität auch einer der wichtigsten Skills eines Entwicklers.

Komplexität skalieren

Nun ist es leider nahezu unmöglich, steigende Komplexität zu vermeiden, wenn man den Funktionsumfang einer Software ständig erweitert. Klar kann man das Team vergrößern, um die Cognitive Load weiter zu verteilen, aber Teams skalieren nur begrenzt. Ab acht Entwicklern wird es meist kritisch mit dem Kommunikations- und Koordinations-Overhead [18]. Irgendwann hat man einen Punkt erreicht, an dem man die Komplexität der Software und die Größe des Teams nicht mehr erhöhen kann.

Micro Frontends – The Past, The Presence And The Future

David Leitner
(SQUER Solutions GmbH)

Unser erstes MicroFrontend-Projekt begann fast auf den Tag vor genau fünf Jahren. Seither haben wir unzählige Projekte mit diesem architektonischen Stil begleitet. In diesem Zeitraum haben sich die technischen Möglichkeiten stetig und erheblich verbessert. So konnten wir die anfänglich laienhaften Ansätze, welche client-seitig auf iFrames basierten, durch Konzepte wie Webpack Module-Federation oder EcmaScript Import-Maps ersetzen. Wir sind von Edge-Side-Includes zu ausgereiften Frameworks übergegangen, die es uns erlauben MicroFrontends auch Server-seitig komfortabel und sicher zu implementieren. Und selbst während der Umsetzung können wir durch fortgeschrittene Methoden zur Verwaltung von MonoRepos und verteilten, inkrementellen Builds völlig neue Methoden ausschöpfen. Schlussendlich haben wir sogar Wege gefunden, um mit ausgefeilt Konzepten wie „federated GraphQL“ komplexe Integrationsszenarien abzubilden. Doch es ist nicht alles Gold was glänzt. Mittels Beispielen aus der Praxis werden wir detailliert auf die Entwicklung dieser verschiedenen Ansätze eingehen. Dies wird verdeutlichen, dass all diese neuen Konzepte und Tools zu hoher Komplexität führen (können) und daher bedacht und mit ausreichend architekturellen Fingerspitzengefühl angewandt werden müssen. Letztendlich werden Sie in diesem Vortrag die verschiedenen Möglichkeiten zur Implementierung von MicroFrontends in der Vergangenheit, Gegenwart und Zukunft erkennen, und danach ein Gefühl haben, welcher der verfügbaren Ansätze sich für Ihre spezifische Anforderung am besten eignet.

Will man dennoch mehr skalieren und die Software erweitern, dann bleibt nur der Divide-and-Conquer-Ansatz [19]. Man löst einen Teil des Modulithen heraus und macht daraus einen eigenen Service mit eigener Deployment-Einheit. Aus dem großen Team werden zwei Teams, die sich die Cognitive Load untereinander aufteilen und dann wieder unabhängig voneinander skalieren können.

Eine Alternative dazu ist es, den Teamfokus nach und nach von Weiterentwicklung auf Wartung und Betrieb zu verschieben. Das ermöglicht es, bei gleicher Teamgröße und immer langsamer wachsender Komplexität die Software weiter zu warten und aktuell zu halten. Dieser Schritt geht unweigerlich einher mit einer neuen Zusammensetzung der Rollen im Team. Die Pioniere und Erfinder machen Platz für Städtebauer und Umsetzer [20], [21]. Das ist keine negative Entwicklung, wenn man sie bewusst in Kauf nimmt und nicht von ihr überrascht wird.

Einen neuen Service extrahieren

Das Aufteilen eines Modulithen in zwei oder mehr Services ist aus vielen Gründen nicht trivial. Es gibt neue Herausforderungen in der Infrastruktur, im Betrieb, in der Organisationsstruktur, bei Themen wie Performance, Resilience und Monitoring. Auf all das möchte ich hier nicht eingehen und lieber auf Kollegen verweisen, die das schon sehr gut abgedeckt haben [22].

Wenn man die reine Codesicht und Anwendungsarchitektur betrachtet, sollte es aber recht einfach sein, einen Service aus einem gut strukturierten Modulithen herauszuschneiden. Man entscheidet sich für ein Modul, das einen eigenen, gut gekapselten Subcontext der Domain enthält, und schiebt es komplett in ein neues Repo. Abhängigkeiten zu anderen Modulen werden durch Netzwerk-Calls ersetzt. Im ersten Wurf werden aus Methodenaufrufen meist HTTP Calls und aus Application Events werden Events auf einem externen Event Bus. Dabei ist auch bei den neu entstandenen Services darauf zu achten, dass es keine zyklischen Abhängigkeiten zwischen den Services gibt, sonst baut man wieder einen Deployment-Monolithen.

Leider gibt es bei so einer Trennung auch großes Potenzial für Duplikationen, wenn technische Querschnittsthemen oder Datenstrukturen in beiden Services gebraucht werden. Der oben genannte Overhead für „Dinge außenrum“ kommt noch dazu. Letztendlich ist es immer ein Trade-off, den man damit eingeht, und für den man sich sehr bewusst unter Abwägung aller Vor- und Nachteile entscheiden muss.

That's all Folks

Jetzt stehen wir hier mit vollem Kopf, die Synapsen überwältigt mit Patterns, Methoden, Tools, Best Practices. Wenn man das jetzt alles so macht wie es die Artikelserie sagt, und genauso anwendet wie beschrieben, dann kommt doch unglaublich tolle Software dabei raus, oder? Ein Modulith der Glückseligkeit und viel besser als mit Microservices? Vielleicht. Vielleicht auch nicht. Gute Software entsteht nicht ohne gesunden Menschenverstand, ohne ständiges Hinterfragen und ohne über den Tellerrand zu gucken. Die Methoden aus den Artikeln haben sich in Problemen aus der Praxis bewährt, aber man soll ja nicht Lösungen für irgendein fremdes Problem finden, sondern für sein eigenes. Darum hoffe ich, dass ich die Werkzeugkiste der Leser ein bisschen erweitern und den Sinn für saubere, modulare Architektur schärfen konnte, ohne den Horizont dabei zu sehr einzuschränken. Ich wünsche viel Spaß beim Refactoring!

Arnold Franke wirkt als Entwickler und Berater für die Kunden der synyx GmbH & Co KG in Karlsruhe. Dabei baut er als Komplexitätsvermeider und Software Craftsman nachhaltige, pragmatische Lösungen mit sauberem Code.

@indyarni https://blog.synyx.de

Links & Literatur

[1] https://martinfowler.com/bliki/TestPyramid.html

[2] https://synyx.de/blog/code-coverage-with-significance/

[3] https://site.mockito.org

[4] https://github.com/powermock/powermock

[5] https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-test-auto-configuration.html

[6] https://spring.io/guides/gs/testing-web/

[7] http://wiremock.org

[8] https://www.testcontainers.org

[9] http://olivergierke.de/2013/11/why-field-injection-is-evil/

[10] https://www.oreilly.com/library/view/97-things-every/9780596809515/ch08.html

[11] https://docs.sonarqube.org/latest/extend/developing-plugin/

[12] https://www.archunit.org

[13] https://structure101.com

[14] https://www.sonarqube.org

[15] https://blog.ndepend.com/understanding-cyclomatic-complexity/

[16] https://techbeacon.com/app-dev-testing/forget-monoliths-vs-microservices-cognitive-load-what-matters

[17] https://t2informatik.de/wissen-kompakt/kiss-prinzip/

[18] https://www.toptal.com/product-managers/agile/scrum-team-size

[19] https://effectivesoftwaredesign.com/2011/06/06/divide-and-conquer-coping-with-complexity/

[20] http://agilebusinessmanifesto.com/agilebusiness/a-structure-for-continuous-innovation-pioneers-settlers-town-planners/

[21] https://de.wikipedia.org/wiki/Teamrolle#Teamrollen_nach_Belbin

[22] https://synyx.de/events/jug-hessen-vom-monolithen-zu-microservices-ein-erfahrungsbericht/

Microfrontends mit Angular im Praxiseinsatz

Architektur für agile Teams

Wir haben mehrere namhafte Unternehmen gefragt, warum und wie sie Microfrontends nutzen. Ein Mittelständler gibt Auskunft, warum er sich bewusst dagegen entschieden hat.

von Manfred Steyer

In den letzten Jahren habe ich einige Kunden bei der Umsetzung von Microfrontends mit Angular unterstützt. Dabei unterscheiden sich die Realisierungen teilweise beträchtlich voneinander. Der Grund dafür sind die vielen verschiedenen Implementierungsvarianten von Microfrontends, mit denen unterschiedliche Vor- und Nachteile einhergehen. Diese müssen im konkreten Fall gegen die vorherrschenden Architekturziele abgewogen werden.

In diesem Artikel zeige ich Implementierungsvarianten sowie deren Konsequenzen anhand von praktischen Beispielen auf. Dazu habe ich einige Kunden befragt – fünf dieser Gespräche präsentiere ich hier nach einem kurzen Überblick. Um zu unterstreichen, dass Microfrontends nicht immer passend sind, gibt es am Ende noch einen Fall, in dem sich ein Unternehmen gegen diesen Architekturstil entschieden hat.

Was sind Microfrontends?

Ähnlich wie bei Microservices im Backend ist die Idee von Microfrontends, mehrere kleine anstatt einer einzigen großen Anwendung zu nutzen. Die Vorteile dieser Art der Modularisierung sind in erster Linie nicht technischer, sondern organisatorischer Natur, da sie weitgehend autarke Teams ermöglichen. Jedes Team kümmert sich um sein eigenes Microfrontend und soll sich nur wenig mit anderen Teams abstimmen. Das bedeutet auch, dass Technologiestacks sowie Architekturstile, die am besten zur jeweiligen Domäne passen, zum Einsatz kommen können. Einen firmeninternen Technologiemix führt man natürlich nicht aus Jux und Tollerei ein. Bei Softwaresystemen und Produktsuiten, die über eine Dekade oder länger hinweg entwickelt werden, ist es jedoch durchaus sinnvoll, zwischendurch auf aktuelle Stacks wechseln zu können. Ein weiterer Vorteil von Microfrontends ist die Möglichkeit, jedes Frontend separat bereitstellen zu können. Teams blockieren sich also nicht mehr gegenseitig und das bringt Agilität.

Wo sind die Herausforderungen?

So verlockend diese Vorteile auch klingen mögen – problemlos umsetzbar sind sie nicht. Das liegt unter anderem daran, dass aktuelle SPA Frameworks sowie Bundling-Lösungen (noch) nicht mit dieser Idee im Hinterkopf geschaffen wurden. Das bedeutet, dass die Teams ggf. selbst Infrastrukturcode zur Orchestrierung der einzelnen Microfrontends entwickeln müssen [1]. Außerdem gilt es, verschiedene Fragen zu klären, z. B., wie die einzelnen Frontends dem Benutzer als großes Ganzes präsentiert werden sollen, aber auch, wie die Kommunikation zwischen den Frontends erfolgen kann. Zudem benötigt man eine Art Single Sign-on für alle Einzellösungen. Daneben müssen die Teams entscheiden, ob und wie Komponenten zwischen den Microfrontends zu teilen sind. Gerade das ist ein sehr delikates Thema, denn während man gerade komplexe Komponenten wiederverwenden möchte, widerstrebt solch ein Austausch der grundlegenden Idee hinter Microservices, nämlich Abhängigkeiten zwischen Teams zu verhindern.

Eine ideale Lösung gibt es hier wohl nicht. Vielmehr müssen anhand der verschiedenen Architekturanforderungen passable Ansätze gefunden werden, die mehr Vor- als Nachteile mit sich bringen. Dieses Dilemma sollte Softwarearchitekten jedoch ohnehin bekannt sein. Deswegen zielen auch einige der nachfolgend gestellten Fragen sowohl auf die gewählten Maßnahmen wie auch auf die beobachteten Konsequenzen – sowohl in positiver als auch in negativer Hinsicht – ab.

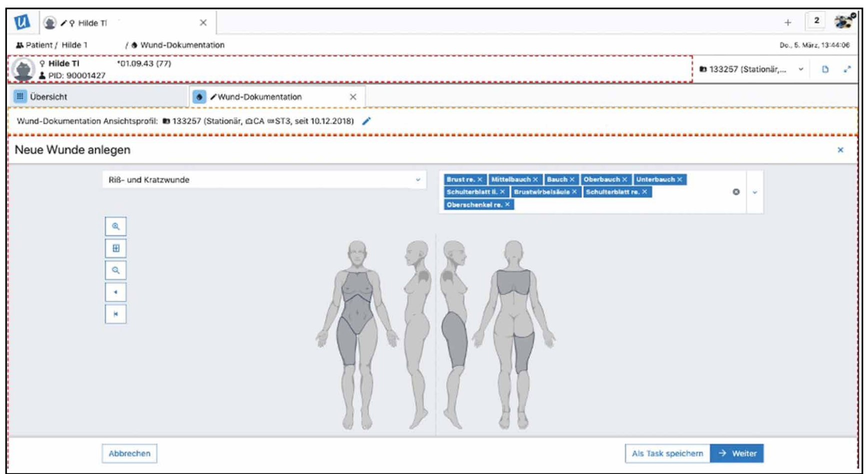


Abb. 1: AGFA Healthcare – die gestrichelten Linien deuten verschiedene Microfrontends an

AGFA Healthcare

Welche Art von Anwendung setzen Sie um?

Wir implementieren ein Krankenhausinformationssystem bzw. ein klinisches Informationssystem, das alle relevanten Workflows im klinischen Ablauf von Ärzten, Pflegekräften und anderen Care-Providern abbildet (**Abb. 1**).

Warum haben Sie sich für eine Microfrontend-Architektur entschieden?

Es handelt sich bei unserer Lösung um eine Produktsuite, bestehend aus mehreren individuellen Teilprodukten, die dem Benutzer als großes Ganzes (Composite Application) präsentiert werden sollen und verschiedene Workflows abbilden. Außerdem wollten wir ein Baukastensystem zur Herstellung von Composite-Lösungen für unterschiedliche Märkte. Wir benötigten zudem einen Ansatz, der auch bei vielen Teams mit einer verteilten Entwicklung funktioniert.

Die Architektur erleichtert ein unabhängiges Deployment von Teilen der Applikation. Der Umfang eines Updates kann dadurch reduziert werden, und es muss nicht immer das ganze Frontend ausgetauscht werden. Der Ansatz sollte unterschiedliche Entwicklungsgeschwindigkeiten in verschiedenen Teams und eine klare Abgrenzung von Medizinprodukten zu Nichtmedizinprodukten ermöglichen. Zukunftssicherheit war uns hierbei ebenso wichtig, da sich unterschiedliche Frontend-Technologien in unterschiedlichen Generationen zu einem Gesamtprodukt zusammenfassen lassen.

Wie haben Sie diese Architektur umgesetzt?

Zum einen kamen iFrames zum Einsatz, u. a., um bestehende Funktionalitäten einbinden zu können. iFrames bieten eine sehr gute Isolation der einzelnen Microfrontends. Zum anderen haben wir für die neueren Produktentwicklungen bereits auf Web Components gesetzt.

Inwieweit sind die Vorteile, die Sie sich von dieser Architektur erhofft haben, eingetreten?

Die einzelnen Produkte können möglichst unabhängig voneinander von verschiedenen Teams entwickelt werden. Außerdem führt der Ansatz zu klaren Schnittstellen zwischen den einzelnen Funktionalitäten. Auch die Wiederverwendbarkeit ist sehr gut. Durch die klaren Schnittstellen konnte auch ein hoher Grad an organisatorischer Skalierbarkeit erreicht werden (viele unabhängige Teams), da die Aufgaben besser verteilbar sind.

Welche Herausforderungen sind aufgetreten?

Wir mussten Code schreiben, um die einzelnen iFrames zu koordinieren und damit einhergehende Usability-Probleme zu kompensieren. Außerdem mussten wir ein Framework für die übergreifende Kommunikation der einzelnen iFrames erstellen, welches z.B. Kontextänderungen an alle iFrames bzw. Web Components propagiert bzw. Kontextänderungen entgegennimmt und propagiert.

Wir haben eine umfängliche UX- und UI-Guideline erstellt, damit die Applikationen trotz Modularität eine einheitliche UX bieten. Wir haben wiederverwendbare Komponenten (CSS, Angular) und APIs erstellt, damit nicht in jedem Microfrontend das Rad neu erfunden werden muss. Durch die Standardisierung der Authentifizierung und Autorisierung musste nicht jedes Team das Tokenmanagement (JWT) selbst realisieren. Außerdem mussten Memory Footprint und Performance der Anwendung bereits frühzeitig im Auge behalten werden, da gemeinsame Bibliotheken und Frameworks pro iFrame redundant geladen werden.

E-Banking und Beraterapplikation bei einer Bank

Welche Anwendung haben Sie mit Microfrontends umgesetzt?

Wir haben u. a. eine eBanking-Applikation für Kunden sowie eine Beraterapplikation für Bankmitarbeiter umgesetzt.

Warum haben Sie sich für eine Microfrontend-Architektur entschieden?

Aufgrund unserer Microservices-Architektur im Backend war es der nächste logische Schritt, auch das Frontend in Richtung Microfrontend-Architektur zu bringen. Des Weiteren spielte auch die Anzahl der Teams eine Rolle. Wir haben ca. sieben Teams, die fachlich nach Domänen aufgestellt und für Backend bis Frontend verantwortlich sind. Ein weiterer Grund war das unabhängige Ausliefern der einzelnen Microfrontends und des Applikationsrahmens. Daneben half uns die Architektur beim Umsetzen eines Tranchenkonzepts. Dabei wird jeder Kunde einer Tranche zugeteilt und jede Tranche bekommt unterschiedliche Versionen ausgeliefert.

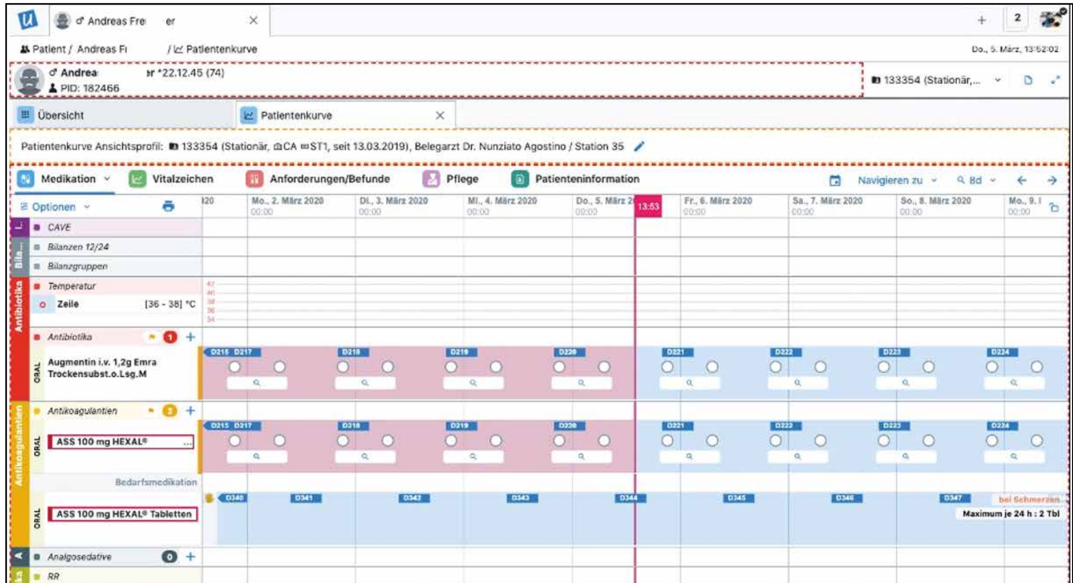


Abb. 2: AGFA Healthcare – die Ansicht setzt sich aus Bestandteilen mehrerer Microfrontends zusammen

Wie haben Sie diese Architektur umgesetzt?

Microfrontends werden zur Laufzeit lazy geladen (*ocLazyLoad*) und in die Rahmenanwendung integriert (auf Basis von AngularJS). Teilweise werden aber auch weitere SPAs über Hyperlinks und SSO eingebunden. Die Kommunikation zwischen den Frontends findet über ein eigenes Event-Handling-Konzept (auf Basis von Observables) statt. Wir haben keine direkte Kommunikation zwischen den einzelnen Microfrontends, sondern nur zwischen Microfrontends und der Rahmenanwendung. Die Authentifizierung erfolgt über OAuth. Über HttpInterceptors wird das Token an die REST Services übergeben. Für diese Zwecke gibt es ein unternehmensinternes Framework, das immer wiederkehrende Tasks übernimmt. All diese zentralen Libs werden einmal in den Rahmen geladen und die Microfrontends laden nur applikationsspezifischen Code.

Derzeit denken wir auch über den Einsatz von bzw. die Migration auf Angular nach.

Inwieweit sind die Vorteile, die Sie sich durch diese Architektur erhofft haben, eingetreten?

Die Vorteile liegen überwiegend in unabhängigen Releases, d. h., Teams können unabhängig von anderen Teams arbeiten.

Mit welchen Herausforderungen hatten Sie zu kämpfen und wie sind Sie damit umgegangen?

Eine Herausforderung war die Vorgabe der Framework-Versionen durch die Rahmenanwendung (AngularJS, Lodash etc.), die durch die Microfrontends abgestimmt eingesetzt werden mussten. Der initiale Aufwand zum Einrichten der Entwicklungsumgebung ist auch eine Herausforderung, denn zur Entwicklung muss die Rahmenanwendung am Arbeitsplatz aufgesetzt werden. Es ist nicht möglich, die Microfrontends ohne Rahmen zu entwickeln oder zu testen.

Onlinelohnabrechnung bei DATEV

Welche Anwendung haben Sie als Microfrontend umgesetzt?

Gegenstand der Onlineanwendung ist ein umfangreiches Lohnabrechnungsprogramm, das häufigen gesetzlichen Änderungen unterliegt und durch mehrere unabhängige Teams betreut wird.

Warum haben Sie sich für eine Microfrontend-Architektur entschieden?

Aufgrund des Umfangs des Projekts und der häufigen gesetzlichen Änderungen nutzen wir die Vorteile separater Releases und losgelöster Testbarkeit einer Microfrontend-Architektur.

Wie haben Sie diese Architektur umgesetzt?

Die Microfrontends werden derzeit in einem zentralen Monorepo gepflegt und als einzelne Anwendungen betrieben. Da wir Nx einsetzen, können wir mit Linting-Regeln sicherstellen, dass die einzelnen Domänen nicht miteinander kommunizieren. Somit lässt sich das Monorepo später, wenn unser Team auf mehrere aufgesplittet wird, auch auf verschiedene Repositories aufteilen. Ob wir das wirklich wollen, entscheiden wir aber erst, wenn es so weit ist.

Die Navigation zwischen den Anwendungen erfolgt per Hyperlink. Der Aufrufkontext wird per Queryparameter zwischen den Anwendungen kommuniziert. Weitere benötigte Informationen werden anschließend am Server angefragt. Die Authentifizierung erfolgt pro App mit einem serverseitig bereitgestellten Token.

Derzeit verhindern wir nicht, dass mehrere Microfrontends dieselben Bibliotheken beinhalten, da diese möglichst autark voneinander sein sollen. Innerhalb eines Microfrontends gibt es jedoch Überlegungen, Bestandteile mittels Browsercaching und @angular/pwa vorzuhalten.

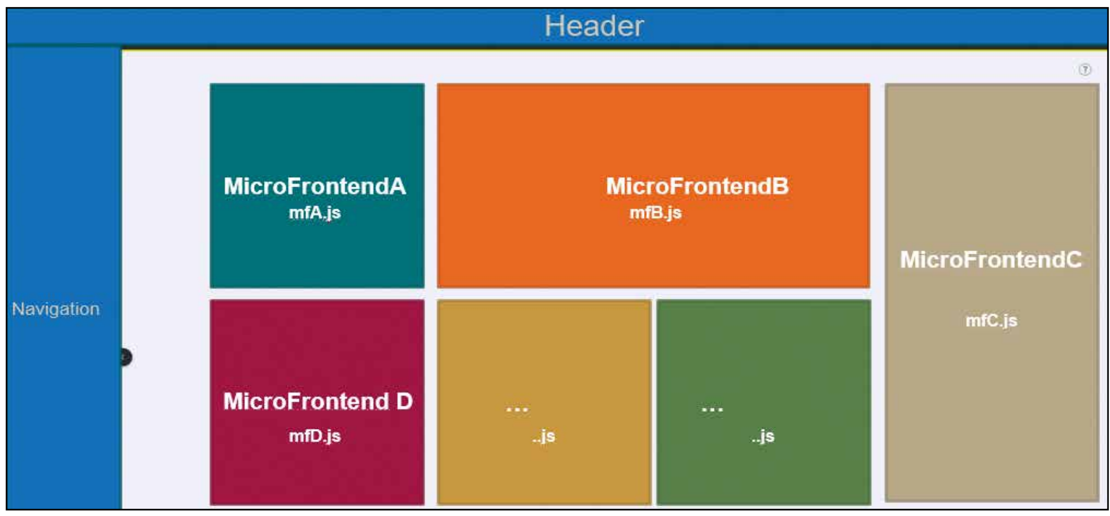


Abb. 3: Schematische Darstellung der Bankanwendung – die Integration erfolgt über Lazy Loading ganzer Anwendungen

Inwieweit sind die Vorteile, die Sie sich durch diese Architektur erhofft haben, eingetreten?
Während der Aufteilung des Ausgangsprojekts sind Abhängigkeiten zwischen Komponenten schnell aufgefallen, wodurch eine saubere Entkoppelung erreicht werden konnte. Die entstandenen Anwendungen können unabhängig voneinander kompiliert und freigegeben werden. Das Tooling (Nx) ermöglicht eine klare Darstellung der Abhängigkeiten innerhalb des Projekts und zeigt Auswirkungen einzelner Änderungen auf. Die klare Strukturierung und Aufteilung erleichtern die Verortung neuer Bestandteile.

Mit welchen Herausforderungen hatten Sie zu kämpfen, und wie sind Sie damit umgegangen?
Der Umgang mit bestehenden und neuen Konfigurationen erwies sich anfangs als schwierig. Zentrales und übergreifendes Styling ist in Verortung und Verwendung weiterhin eine Herausforderung. Die große Abweichung der Projektstruktur von einem gewöhnlichen Angular-Projekt führt zu einer gewissen Einarbeitungszeit für Neuzugänge – selbst für geübte Angular-Entwickler.

Interne Anwendung zur Unterstützung von Prozessen in einer Bank

Welche Anwendung haben Sie als Microfrontend umgesetzt?
Wir wollten diverse interne Finanzdienstleistungsanwendungen in einer Shell vereinigen.

Warum haben Sie sich für eine Microfrontend-Architektur entschieden?
Wir wollten eine starke Isolation zwischen Front- und Backofficeanwendungen, um unabhängige Produktteams zu begünstigen.

Wie haben Sie diese Architektur umgesetzt?
Um eine starke Isolation der einzelnen Anwendungen, die teilweise von Teams in unterschiedlichen Ländern entwickelt werden, zu gewährleisten, setzen wir auf iFrames. Die bekannten Nachteile von iFrames haben wir mit einer Bibliothek [1], die wir auf npm bereitgestellt haben, weitgehend kompensiert. Diese Bibliothek nutzt für die Kommunikation zwischen den Microfrontends die Möglichkeiten von postMessage [2]. Da Web Components mittlerweile eine gewisse Reife erreicht haben, denken wir darüber nach, künftig auch diese Option zu nutzen. Die Authentifizierung und Authorisierung erfolgt primär über Tokens. Dazu kommen OAuth2 und OpenId Connect zum Einsatz. Die Tokens werden über den Session Storage geteilt.

Wir verhindern nicht, dass dieselben Framework-Bestandteile mehrfach geladen werden. Das hat in unserem Szenario keine Priorität.

jax HYBRID

Die Microfrontend-Revolution: webpack Module Federation und Angular

Manfred Steyer
(SOFTWAREarchitekt.at)

Die Umsetzung von Microfrontends war bis jetzt alles andere als einfach. Da gängige Frameworks und Build-Werkzeuge diese Idee nicht einmal ansatzweise kannten, musste man ordentlich in die Trickkiste greifen. Die von webpack gebotene Module Federation leitet hier einen entscheidenden Richtungswechsel ein. Sie erlaubt es, separat kompilierte Anwendungsteile zur Laufzeit zu laden und Bibliotheken zwischen ihnen zu teilen. In dieser Session erfahren Sie von Microfrontend-Pionier Manfred Steyer am Beispiel einer Angular-Anwendung, wie Sie diesen Mechanismus zur Schaffung von Microfrontends nutzen können. Neben den Schönwetterszenarien diskutieren wir auch weiterführende Bereiche wie dynamische Module Federation oder der Umgang mit Versionskonflikten. Am Ende der Session haben Sie einen Überblick über das Thema und wissen, wie Sie Module Federation in Ihren Projekten nutzen können.

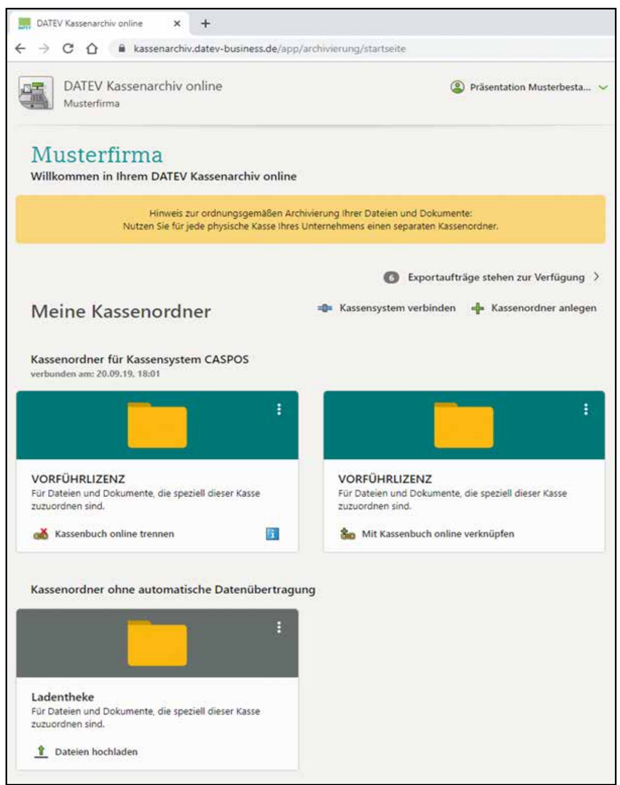


Abb. 4: DATEV Kassenarchiv online – die Integration erfolgt über Hyperlinks

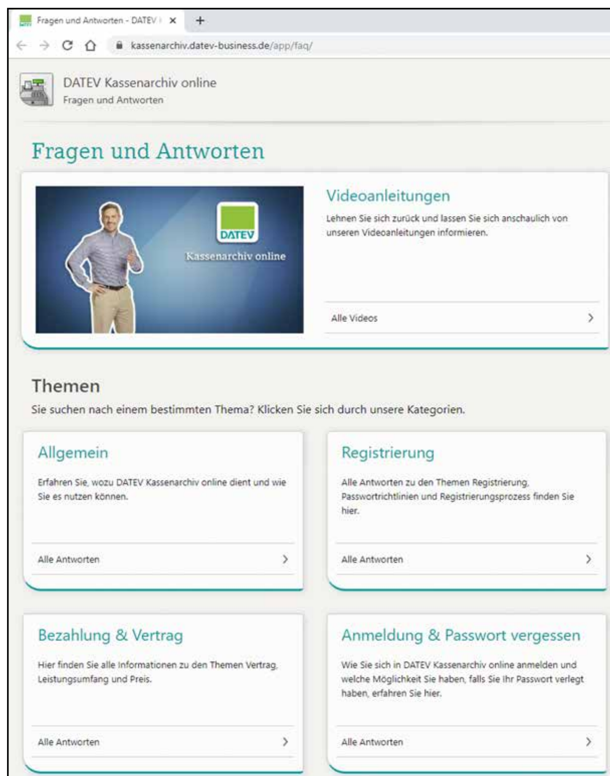


Abb. 5: DATEV Kassenarchiv online – contentgetriebenes Microfrontend für FAQ

Inwieweit sind die Vorteile, die Sie sich durch diese Architektur erhofft haben, eingetreten?

Die oben beschriebenen Effekte sind eingetreten. Die geografisch verteilten Teams können autark arbeiten.

Mit welchen Herausforderungen hatten Sie zu kämpfen und wie sind Sie damit umgegangen?
Die Behandlung des Back- und Forward-Buttons verursachte im Zusammenhang mit iFrames die meisten Probleme.

DATEV Kassenarchiv online

Beschreiben Sie bitte kurz, welche Anwendung Sie als Microfrontend umgesetzt haben.
Mit DATEV Kassenarchiv online werden täglich Kassendaten aus elektronischen Registrierkassen in der DATEV-Cloud revisionssicher archiviert. Als direkte Kundenschnittstelle ermöglicht das Frontend neben der Benutzerregistrierung die Bestellung und Verwaltung von Kassenordnern zur Darstellung der Registrierkassen an der Oberfläche. Dort bieten die Kassenordner den Import und Export der Kassendaten sowie die Verknüpfung zum DATEV-Kassenbuch online zur digitalen und ordnungsgemäßen Kassenbuchführung.

Warum haben Sie sich für eine Microfrontend-Architektur entschieden?
Das Online-Frontend von DATEV Kassenarchiv online besteht aus den drei Microfrontends Registrierungsanwendung, Kassenarchiv (**Abb. 4**) und FAQ (**Abb. 5**). Diese unterscheiden sich sowohl im Design als auch in den genutzten Technologien. Durch die strikte Trennung liefert jedes Microfrontend nur den Code aus, den es selbst benötigt. Somit werden Probleme, die durch inkompatible Bibliotheken auftreten können von vornherein vermieden. Zudem sinken dadurch die Build-Größen und folglich die Ladezeiten. In den seltensten Fällen müssen alle drei Builds geladen werden, da es zwischen den Anwendungen nur wenige fachliche Berührungspunkte gibt. Das erübrigt auch einen komplizierten Mechanismus zur Kommunikation zwischen den Microfrontends. Des Weiteren wird die FAQ-Anwendung durch andere Personen redaktionell gepflegt. Durch die Trennung können die Anwendungen unabhängig voneinander weiterentwickelt und releast werden.

Wie haben Sie diese Architektur umgesetzt?
Es handelt sich um unabhängige Anwendungen in eigenen Code-Repositories. Die Hauptanwendung verwaltet gemeinsam genutzten Code als Shared Library in einem Monorepo. Diese Library kann separat als npm-Paket gebaut und somit auch von anderen Anwendungen genutzt werden.

Wie werden die einzelnen Frontends zu einem Gesamtsystem integriert?
Die Frontends laufen unter derselben Domain mit einem gemeinsamen Basispfad und individuellen Anwendungspfaden. Innerhalb der Anwendungen wird über *href*-Verlinkungen auf die anderen Anwendungen verwiesen. Der in der Registrierungsanwendung registrierte Benut-

zer wird zum Log-in in der Hauptanwendung verwendet, wobei derselbe Backend Service zum Einsatz kommt.

Wie findet die Kommunikation zwischen den Frontends statt?
Mangels anwendungsübergreifender Use Cases gibt keinen Mechanismus, um Daten direkt zwischen den Microfrontends auszutauschen.

Wie findet die (gemeinsame) Authentifizierung statt?
Nur in der Hauptanwendung ist die Authentifizierung mittels Benutzername und Passwort erforderlich. Zur Verknüpfung eines Kassenordners mit DATEV Kassenbuch online ist die zusätzliche Authentifizierung mittels OAuth2 und OpenID Connect an zentralen DATEV Services vonnöten. Dadurch besteht ein Single Sign-on für DATEV Unternehmen online und DATEV Kassenbuch online.

Verhindern Sie, dass dieselben Framework-Bestandteile mehrfach geladen werden?
Aktuell setzen zwar die Registrierungsanwendung und die Hauptanwendung mit Angular auf dasselbe Framework, jedoch wird die Registrierung in der Regel nur einmalig aufgerufen. Sofern direkt im Anschluss das Log-in in die Hauptanwendung erfolgt, wird Angular mit seinen Abhängigkeiten erneut geladen. Dies wird in Kauf genommen, da es sich beim Aufruf beider Anwendungen um einen seltenen Use Case handelt. Bei der FAQ-Anwendung kommen andere Technologien zum Einsatz, sodass kaum gemeinsamer Code existiert.

Inwieweit sind die Vorteile, die Sie sich durch diese Architektur erhofft haben, eingetreten?
Durch die Trennung können bedenkenlos unterschiedliche Technologien eingesetzt werden. Beispielsweise hat die Einführung eines neuen npm-Pakets in einem Microfrontend keinerlei Auswirkungen auf die anderen Frontends. In diesen entsteht weder zusätzlicher Testaufwand noch wachsen deren Build-Größen mit. Ferner kann die FAQ-Anwendung jederzeit vom Redaktionsteam gepflegt und veröffentlicht werden, ohne mit der Registrierungs- oder der Hauptanwendung in Konflikt zu geraten.

Die erhofften Vorteile der Unterteilung in Microfrontends haben sich für uns eingestellt, sodass wir diesen Ansatz erneut wählen würden. Um dem erhöhten Pflegeaufwand beizukommen, gibt es Überlegungen, die einzelnen Frontends in ein gemeinsames Monorepo zu überführen.

Mit welchen Herausforderungen hatten Sie zu kämpfen, und wie sind Sie damit umgegangen?
Gemeinsamer Code wurde zunächst in beiden Angular-Anwendungen dupliziert. Um den Aufwand der doppelten Pflege sowie das Risiko des damit verbundenen Vergessens zu vermeiden, wurde die Shared Library eingeführt. Diese erfüllt zwar ihren angedachten Zweck, jedoch gestaltet sich die Bereitstellung und Einbindung neuer Versionen der Library als aufwendig. Das führt dazu, dass die Registrierungsanwendung häufig auf einem älteren Stand ist als die Hauptanwendung. Ebenso verhält es sich mit Updates der npm-Pakete.

jax HYBRID

Angular Architektur Workshop: Strategic Design mit Nx und Micro Frontends

Manfred Steyer
(SOFTWAREarchitekt.at)

In diesem interaktiven Workshop lernen Sie von Manfred Steyer – Angular GDE und Trusted Collaborator im Angular-Team – wie sich große und skalierbare Geschäftsanwendungen mit Angular entwickeln lassen. Dazu betrachten wir zunächst die Nutzung von Strategic Domain Design im Frontend sowie die Umsetzung mit Nx Monorepos. Wir betrachten Ansätze zum Erzwingen von Architekturvorgaben und Inkrementelle Builds sowie den Build-Cache zum Beschleunigen von Builds und Testläufen. Diese Ideen werden nach und nach ausgebaut und münden in die Realisierung von Micro Frontends mit dem brandneuen webpack Module Federation und Web Components auf der Basis von Angular Elements. Nach den einzelnen Übungen haben Sie eine Fallstudie, die Sie als Vorlage für eigene Vorhaben nutzen können. Am Ende wissen Sie nicht nur, wie sich Micro Frontends mit dem Stand der Technik umsetzen lassen, sondern auch, ob dieser Architekturstil zu Ihnen passt und welche Alternativen Sie haben. Außerdem sind Sie in der Lage, die einzelnen Optionen vor dem Hintergrund Ihrer Vorhaben zu bewerten.

SaaS-Lösung im Medizinbereich – bewusst keine Microfrontend-Architektur

Welche Art von Anwendung setzen Sie um?
Wir entwickeln im Kundenauftrag für ein mittelständisches Unternehmen aus Potsdam eine Prozessplattform als SaaS-Lösung im Medizinbereich.

Warum haben Sie sich gegen eine Microfrontend-Architektur entschieden?
Wir haben nur ein agiles Team. Derzeit gibt es keinen Anwendungsfall für ein separates Deployment der Bestandteile. Microfrontends hätten die Komplexität der Anwendung und des Deployments unnötig erhöht. Dank Nx Monorepos und Zugriffseinschränkungen zwischen Bibliotheken haben wir dennoch die Freiheit, später auf Microfrontends umzustellen, weil wir mit diesen Zugriffseinschränkungen in sich geschlossene Subdomänen erzwingen.

Wie haben Sie diese Architektur umgesetzt?
Wir nutzen ein Nx Monorepo und schneiden die Anwendung nach Domänen. Bibliotheken einer Domäne

dürfen nicht mit Bibliotheken einer anderen Domäne kommunizieren. Das stellen wir mit den von Nx gebotenen Zugriffsbeschränkungen sicher. Für das State Management nutzen wir NgRx.

Inwieweit sind die Vorteile, die Sie sich durch diese Architektur erhofft haben, eingetreten?
Die Komplexität ist überschaubar. Durch den Einsatz von Nx Monorepos und den Schnitt nach Domänen haben wir eine einheitliche Projektstruktur.

Fazit

Die vorgestellten Lösungen unterstreichen, dass Microfrontends nützlich sind, wenn es mehrere Teams zu koordinieren gilt. Dadurch, dass jedes Team möglichst autonom vorgehen kann, erhöht sich die Agilität. Dabei fällt auch auf, dass in allen Fällen, in denen Microfrontends eingesetzt wurden, mehrere voneinander weitgehend entkoppelbare Domänen vorliegen. Bei den Umsetzungsstrategien kristallisieren sich zwei Hauptströmungen heraus: In Fällen, in denen alle Frontends als große und integrierte Anwendung zu präsentieren sind, kommt eine Shell – in einem Interview auch als Rahmenanwendung bezeichnet – zum Einsatz. Dabei handelt es sich um eine SPA, die in der Lage ist, weitere separat kompilierte SPAs zu laden.

In Fällen, in denen eine hohe Isolation benötigt wird oder Legacy-Anwendungen einzubinden sind, kommen iFrames zum Einsatz. Die beiden Unternehmen, die unter anderem hierauf setzen, sind sich der Nachteile von iFrames sehr wohl bewusst und haben eigene Bibliotheken geschrieben, um diese zu kompensieren. Eine davon wurde auf npm veröffentlich [2]. Aus eigener Erfahrung kann ich sagen, dass man als Benutzer hier nicht merkt, dass es sich um iFrames handelt.

Interessant ist aber auch, dass die beiden Firmen, die iFrames verwenden, nach neueren Entwicklungen in Richtung Web Components streben bzw. diesen noch sehr jungen Ansatz bereits verwenden. Dabei handelt sich zweifelsfrei um den moderneren Ansatz, der einige mit iFrames verbundene Herausforderungen erst gar nicht aufkommen lässt, dafür jedoch Legacy-Lösungen nicht unterstützt.

Ein weiterer Ansatz zum Laden der Microfrontends wurde als Lazy Loading bezeichnet. Hier wird eine AngularJS-Erweiterung genutzt, um separat bereitgestellte AngularJS-Anwendungen zu laden. Das funktioniert leider bei neueren Frameworks nicht mehr so einfach, da hier die Build-Werkzeuge wie webpack und somit das Angular CLI verlangen, dass der gesamte Quellcode beim Kompilieren vorliegt. Erst danach wird er in einzelne Teile gesplittet. Eine Lösung stellen auch hier Web Components oder einfach das manuelle Nachladen ganzer Anwendungs-Bundles dar. Im letzteren Fall müsste die Shell per JavaScript *script*-Tags erzeugen.

Ein anderer Ansatz ist der Einsatz von Hyperlinks. Das erinnert an Produktsuiten wie Office 365 oder Googles G Suite. Dieser Ansatz ist wohl der einfachste, aber dafür bemerkt der Benutzer auch den Anwendungswechsel. Letzteres fällt nicht ins Gewicht, wenn sich ein Anwender eine Zeit lang in einem Frontend aufhält, und lässt sich mit Caching und Service Workers kompensieren.

Ein weiterer, hier nicht betrachteter Ansatz ist das Zusammensetzen der Frontends am Server. Das widerstrebt jedoch der Funktionsweise von clientseitig gerenderten SPAs und bietet sich somit eher für klassische Websites an. Beispiele dafür sind u. a. Amazon oder Zalando.

Eine interessante Beobachtung ist auch, dass ein Unternehmen, das ein Monorepo verwendet, sich den Einsatz mehrerer separater Repositories überlegt. Während dieses Vorgehen Microfrontends im engeren Sinne entspricht, überlegt sich das Team hinter einer anderen Fallstudie die Vereinigung mehrerer Repos in einem Monorepo. Das zeigt, dass beide Ansätze funktionieren, aber auch, dass beide Ansätze mit ihren eigenen Herausforderungen kommen.

In den Fallstudien, die heute schon auf Monorepos setzen, wurde mit Linting-Regeln eine klare Trennung zwischen Domänen sichergestellt. Das bringt kurzfristig die gewünschte Entkopplung und langfristig die Möglichkeit, die Lösung auf mehrere Repos zu splitten. Zwischen den Zeilen lässt sich auch herauslesen, dass ein paar der interviewten Unternehmen bereits die Microfrontends-Idee genutzt haben, bevor es dieses Wort gab – unter anderem in den Zeiten von AngularJS oder, als Web Components noch kein Thema waren.

Das letzte Interview liegt mir besonders am Herzen, weil es zeigt, dass – trotz all der aktuell vorherrschenden Euphorie für Microfrontends – dieser Ansatz nicht überall geeignet ist. Vor allem, wenn nur ein Team existiert, kann ein gut strukturierter Monolith die zweckmäßigere Lösung sein. Die erwähnten Linting Rules helfen beim Erzwingen der gewünschten Entkopplung und daneben muss sich das Team nicht mit den teils komplizierten Konsequenzen von Microfrontends belasten. Außerdem entspricht das der Art, in der SPAs gedacht sind, und dank Lazy Loading lassen sich einzelne – wenn auch nicht separat kompilierte Programmteile – bei Bedarf laden.

Manfred Steyer ist Trainer und Berater mit Fokus auf Angular, Google Developer Expert und Trusted Collaborator im Angular-Team. Er schreibt für O'Reilly, das deutsche Java Magazin und Heise Developer. Unter www.ANGULARarchitects.io bietet er und sein Team Angular-Schulungen und Beratung in Deutschland, Österreich und der Schweiz an.

www.softwarearchitekt.at

Links & Literatur

[1] https://www.softwarearchitekt.at/aktuelles/6-steps-to-your-angular-based-microfrontend-shell/

[2] https://www.npmjs.com/package/@microfrontend/controller

[3] https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

Von reaktiven UIs und expliziten Datenflüssen

Reaktive UI-Programmierung

Moderne Webapplikationen sollen reaktiv sein. Zumindest propagieren das viele etablierte Web-Frameworks. Aber was bedeutet Reaktivität in diesem Kontext eigentlich genau? Warum sollte man sie einsetzen und inwiefern erleichtert sie dem Frontend-Entwickler das Leben?

Der Begriff „reactive" oder Reaktivität hat sich mittlerweile fest im Umfeld der Softwareentwicklung etabliert. Doch in meinem Alltag habe ich nicht das Gefühl, dass jeder damit das Gleiche meint. In Summe fallen mir besonders drei unterschiedliche Auslegungen von Reaktivität auf. Zur ersten Gruppe, zu der ich mich auch zähle, gehören überwiegend Frontend-Entwickler. Sie verwenden den Begriff der Reaktivität vor allem im Zusammenhang mit funktional geprägten Programmierstilen und Frameworks. Auch die zweite Gruppe besteht aus Entwicklern, diese sind allerdings eher Backend-fokussiert. Sie verbinden mit Reaktivität das Verhalten ganzer Systeme im Sinne des Reaktiven Manifests [1] und seiner Prinzipien: antwortbereit, widerstandsfähig, elastisch und nachrichtenorientiert.

Die dritte und mit Abstand größte Gruppe assoziiert mit „reactive" einfach nur ergonomische, intuitive Bedienbarkeit des UI. Insbesondere gehören hierzu direktes Feedback auf Eingaben und eine in sich konsistente Darstellung der Anwendung. Diese Eigenschaft wird nachfolgend als Responsiveness bzw. responsive bezeichnet.

Bezogen auf Frontend-Entwicklung sind sowohl die erste als auch die dritte Interpretation von Reaktivität relevant. Viele verbreitete Frontend Frameworks schreiben sich reaktive Programmierung auf die Fahne, und eine gute Responsiveness ist einer der Hauptgründe für den Einsatz solcher Frameworks.

Reactive, responsive oder beides?

Hat also Responsiveness etwas damit zu tun, ob die eingesetzte Technologie reaktiv ist? Eigentlich erstmal nicht. Es lassen sich auch mit jQuery oder purem JavaScript superergonomische Webseiten bauen, die allen Ansprüchen an User Experience (UX) gerecht werden. Aber der Einsatz von Frameworks wie React oder Angular vereinfacht es in der Regel, dieses Ziel zu erreichen. Vor allem wenn die Anwendung größer wird, helfen diese Frameworks bei der Strukturierung. Man kann also sagen: Reaktivität und Responsiveness bedingen sich zwar nicht, widersprechen sich aber auch nicht. Sie stehen orthogonal zueinander.

Reaktive UI-Programmierung

Die etablierten Frontend Frameworks brüsten sich damit, reaktiv zu sein. Hier ein paar Beispiele:

- Svelte [2] ist „truly reactive"
- React [3] trägt Reaktivität im Namen
- Angular [4] hat eine „change detection"
- Vue [5] hat ein „unobtrusive reactivity system"

Und was bedeutet das jetzt? Eigentlich nichts anderes, als dass das UI immer den aktuellen State der Applikation widerspiegelt. Denn das UI reagiert immer sofort auf Datenänderungen und bleibt dabei jederzeit in sich konsistent, zeigt also an allen relevanten Stellen die geänderten Daten an.

Aber das sollte ja eigentlich keine Besonderheit sein. Der Benutzer erwartet stets, dass das UI den Zustand der Applikation repräsentiert. Das als Entwickler zu realisieren, ist jedoch nicht so einfach. Wenn ein UI rein imperativ umgesetzt wird, muss sich der Entwickler quasi per Hand darum kümmern, dass auf alle Wertänderungen, die anzeigerelevant sind, richtig reagiert wird. Gegebenenfalls müssen auch weitere Teile in der Applikation aktualisiert werden, die sich auch auf das UI auswirken. So ist es schwierig, den Überblick zu behalten, welche UI-Komponente direkt oder indirekt von welchen Daten abhängt. Generell ist manuelle Synchronisation von State sehr fehleranfällig. Es gibt überall lokal zwischengespeicherte Werte in den UI-Komponenten, und die Abhängigkeiten, woher diese Daten ursprünglich stammen, sind nicht mehr nachvollziehbar. So kann es leicht passieren, dass das, was im UI angezeigt wird, nicht mehr den tatsächlichen Zustand der Applikation widerspiegelt.

Eine mögliche Alternative zur imperativen Programmierung ist Reactive Programming. Wie der Name schon vermuten lässt, kommen wir dem reaktiven UI hier ein bisschen näher. Die Definition von Reactive Programming

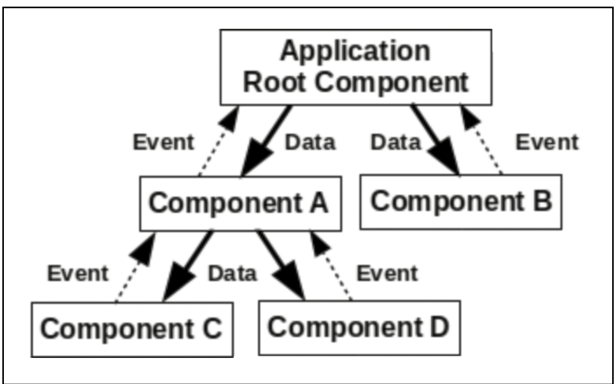


Abb. 1: Komponentenbaum – Data down, Events up

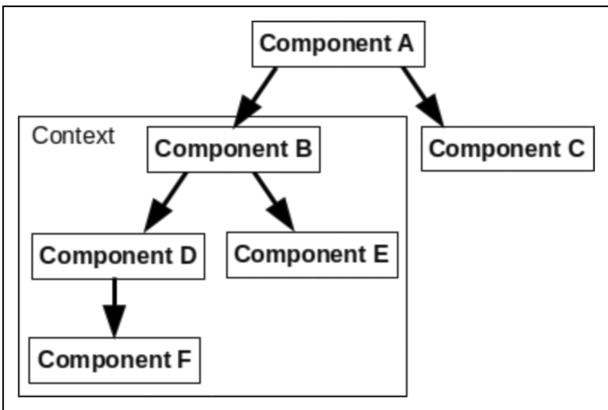


Abb. 2: Dependency Injection im Komponentenbaum

auf Wikipedia: „In computing, reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change.“ [6] Der Teil von Reactive Programming, auf den sich eigentlich alle reaktiven UI Frameworks direkt beziehen, ist „the propagation of change“. Hinter diesem Ausdruck steckt nichts weiter, als dass Datenabhängigkeiten explizit gemacht werden. Wenn sich ein Wert innerhalb eines Programms ändert, werden auch alle anderen von ihm abgeleiteten Werte aktualisiert. Das beliebteste Beispiel hierfür ist Excel: Wenn man den Wert einer Zelle ändert, aktualisiert sich auch die abgeleitete Summe sofort. Bezogen auf das UI hat das zur Konsequenz, dass alle angezeigten Daten immer konsistent mit dem Zustand der Applikation sind.

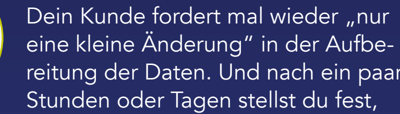
In Reactive Programming wird der Datenfluss durch die Anwendung explizit modelliert. Hierbei wird das Konzept von Streams verwendet. In den meisten UI Frameworks gibt es jedoch keine Streams, sondern der Datenfluss wird durch die Hierarchie der UI-Komponenten beschrieben. Es gibt aber auch UI Frameworks, die direkt auf Streams als Kernkonzept bauen. Der vermutlich bekannteste Vertreter dieses Ansatzes ist Cycle.js [7].

Datenfluss durch den Komponentenbaum

Eigentlich sind alle reaktiven Frontend Frameworks komponentenbasiert. Doch wie spiegelt sich jetzt der Datenfluss durch die Komponenten wider? Das gesamte UI der Anwendung ist als Baum von Komponenten strukturiert. In **Abbildung 1** ist schematisch ein solcher Komponentenbaum dargestellt. Die Pfeile zeigen dabei den Datenfluss. Von oben werden Daten nach unten an die Kindknoten weitergereicht. Idealerweise sollten die Kindknoten ihre Eltern nicht kennen und daher nicht direkt mit ihnen kommunizieren. Stattdessen übermitteln in einer sauberen Architektur die Kindknoten Änderungen an Eltern über Events. Durch das Einhalten dieser Prinzipien innerhalb des UI wird der Datenfluss explizit, was ein wichtiges Kriterium für reaktive Programmierung ist.

In den meisten Frameworks gibt es zusätzlich zum direkten Weiterreichen der Daten an die Kindknoten einen zusätzlichen Mechanismus, um Daten weiterzugeben, nämlich per Dependency Injection. Daten werden dann einem gesamten Teilbaum zur Verfügung gestellt. Das wird in React und Svelte als Context API bezeichnet. **Abbildung 2** verdeutlicht Dependency Injection im Komponentenbaum. Component B spannt einen Kontext auf, und nicht nur alle direkten, sondern auch die indirekten Kindknoten – also insgesamt Component D, E und F – können auf die Daten des Kontexts zugreifen. Wichtig ist hierbei, dass auch die per Kontext bereitgestellten Daten bei Änderungen für ein Update sorgen und so ebenfalls ein Datenfluss explizit modelliert wird.

Wenn man sich eine einzelne Komponente genauer ansieht, wie schematisch in **Abbildung 3** dargestellt, wird noch ein weiteres Detail deutlich: Komponenten können auch einen lokalen State haben. Es gibt zwei Möglichkeiten, wie Daten in eine Komponente gelangen können: entweder von oben durch den Elternknoten bzw. Kontext oder von unten per Event. Man kann zwei Arten von Events unterscheiden: die von den eigenen Kind-



jax HYBRID

Hibernate Tips 'n' Tricks: Schnelle Lösungen für typische Probleme und Anwendungsfälle

Thorben Janssen *(Freiberufler)*

Dein Kunde fordert mal wieder „nur eine kleine Änderung" in der Aufbereitung der Daten. Und nach ein paar Stunden oder Tagen stellst du fest, dass sich das nicht so leicht im Code umsetzen lässt. Hättest du doch besser den Datenbankzugriff angepasst? Und überhaupt, das hat doch bestimmt schon mal jemand gemacht! Die gute Nachricht ist, in vielen Fällen gibt es wirklich schon ein Hibernate Feature, das dir den Großteil der Arbeit abnimmt. Eine Annotation oder wenige Zeilen Code reichen bereits aus, um Multi-Tenancy zu implementieren, datenbankspezifische Datentypen zu unterstützen, SQL-Schnipsel auf Entitäten abzubilden, die Elemente einer Assoziation in einer vorgegebenen Reihenfolge zu lesen, UUIDs zu generieren und als Primärschlüssel zu verwenden, Änderungen in einem Auditlog zu dokumentieren und vieles mehr.

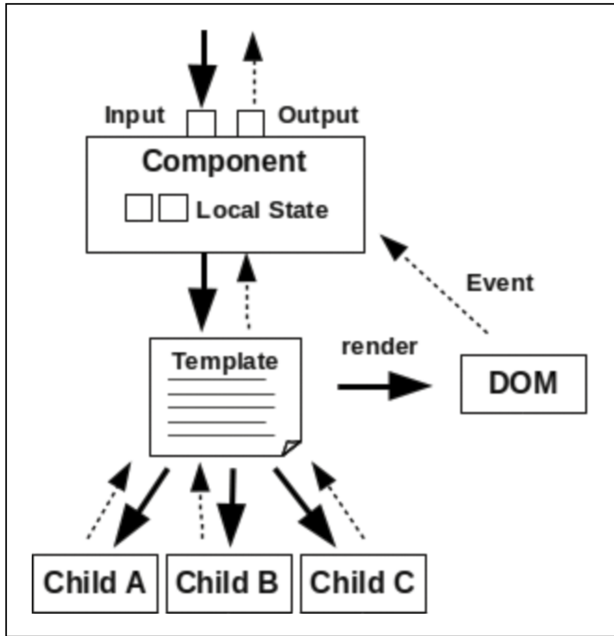


Abb. 3: Komponente im Detail

komponenten geworfenen Events und die Events, die direkt aus dem DOM stammen. Letztere sind in der Regel Benutzerinteraktionen. beispielsweise die Eingabe in ein Input-Feld oder ein Klick mit der Maus. Wenn Informationen aus einem Event weiterhin für die Komponente relevant sind, aber nicht für ihre Elternkomponente, muss sie sich diese Daten selbst merken. Genau für solche Anwendungsfälle benötigt man einen lokalen State.

Der Komponentenbaum und die Kommunikation unter den Komponenten ist der Schlüssel zur Reaktivität im UI. Um Propagation of Change als Kernkonzept konsequent umzusetzen, müssen die Datenflüsse explizit werden. Genau das passiert im Komponentenbaum. Es gibt nur zwei Möglichkeiten, den Zustand einer UI-Komponente zu beeinflussen: entweder die hinuntergereichten Daten ändern sich oder es wird ein Event geworfen. Die nach unten gereichten Daten können entweder aus der direkten Elternkomponente stammen oder ihren Ursprung höher im Baum haben, beispielsweise per Context API. Sowohl das Hinunterreichen als auch das Konsumieren der Daten ist explizit. Gleiches gilt für die Kommunikation nach oben über Events. Sowohl Werfen als auch Fangen der Events machen den Datenfluss explizit sichtbar. Ergänzend sei noch gesagt, dass ein Komponentenbaum natürlich nicht der einzige Weg ist, um reaktive UIs zu bauen, momentan ist es jedoch der etablierteste.

jax HYBRID

Cassandra – Geschwindigkeit hat seinen Preis: Ein Projektbericht

Thomas Strauss
(AUSY Technologies Germany AG)

Cassandra wirbt mit unlimitierter Skalierbarkeit, und in der Tat machen wir die Erfahrung, dass Cassandra dabei keine Obergrenzen kennt. Allerdings muss man als Entwickler einen gewissen Preis dafür bezahlen, um in den Genuss der versprochenen Leistung zu gelangen. In diesem Praxisbericht wird der verteilte Datastore Cassandra vorgestellt und mit den Erfahrungen aus dem harten Projektalltag angereichert. Dabei wird die Funktionsweise von Cassandra erläutert und es werden Hinweise gegeben, wie man mit Cassandra erfolgreich sein, aber auch wie man damit scheitern kann.

Fazit

Was die meisten Leute unter einem reaktiven UI verstehen, sind Responsiveness und eine generell gute UX. Frontend-Entwickler verstehen jedoch zumeist etwas ganz anderes darunter, nämlich eine Spielart von reaktiver Programmierung, um das UI zu implementieren. Man könnte einen Zusammenhang zwischen Responsiveness und reaktiven Frameworks sehen, aber eigentlich haben sie erst einmal nicht unmittelbar etwas miteinander zu tun. Im Kern reaktiver Programmierung geht es um Propagation of Change, also darum, dass der Datenfluss durch die Anwendung explizit modelliert ist. Änderungen am State können so niemals zu einem inkonsistenten Zustand mit dem UI führen. Das hört sich im ersten Augenblick selbstverständlich an, ist aber mit imperativer Programmierung gar nicht so leicht umzusetzen. Der reaktive Werkzeugkasten macht es einem da deutlich leichter.

Klassischerweise werden Datenflüsse direkt als Streams modelliert, doch die meisten reaktiven Frameworks weichen davon ab. Sie bauen einen Komponentenbaum auf und machen den Fluss der Daten auf diese Weise explizit. So sorgt der Komponentenbaum dafür, dass alle Teile des UI konsistente Daten haben. Im Prinzip sind sich React, Svelte, Vue oder Angular, was diesen Grundgedanken angeht, sehr ähnlich, auch wenn sie sich sonst in vielerlei Hinsicht unterscheiden. Hauptsache, die View ist in sich konsistent und spiegelt immer den aktuellen Zustand der Anwendung wider. Und das ist eben mit dem reaktiven Ansatz viel einfacher zu erreichen als ohne.

Hendrik Müller ist Enterprise Developer bei der OPEN KNOWLEDGE GmbH in Oldenburg. Mit dem Schwerpunkt auf Webtechnologien begleiten ihn momentan Angular und Web Components durch den Tag. Abseits davon beschäftigt er sich leidenschaftlich mit dem Design von Programmiersprachen.

Links & Literatur

[1] https://www.reactivemanifesto.org
[2] https://svelte.dev
[3] https://reactjs.org
[4] https://angular.io
[5] https://vuejs.org
[6] https://en.wikipedia.org/wiki/Reactive_programming
[7] https://cycle.js.org

Ein Überblick über Umsetzungsstrategien

Verteilte Transaktionen in verteilten Systemen

Verteilte Transaktionen sind out, das haben die Softwarearchitekten im Laufe der letzten Jahre erkannt. Neuere Persistenzsysteme bieten die Funktionalität für verteilte Transaktionen gar nicht an oder empfehlen, dieses Transaktionsverhalten, falls doch vorhanden, nur in Ausnahmefällen zu verwenden. Doch was kann man als Softwarearchitekt empfehlen, wenn die fachlichen Anforderungen so gestaltet sind, dass Daten, die in mehreren Datentöpfen persistiert werden, zueinander konsistent sein müssen? Der nachfolgende Überblick soll die Auswahl der passenden Umsetzungsstrategie abgestimmt auf den jeweiligen Use Case erleichtern.

von Michael Hofmann

Verteilte Transaktionen (XA Transactions) basieren auf dem Two-Phase-Commit-(2PC-)Protokoll. Dieses Protokoll war in der Vergangenheit essenzieller Bestandteil eines jeden Datenbanksystems und wurde somit das zentrale Konzept, um das Konsistenzproblem bei verteilten Datenbanken zu lösen. Doch wo liegen nun die Probleme beim Einsatz dieses Protokolls? Durch die synchronen Eigenschaften des Protokolls werden verteilte System miteinander sehr eng gekoppelt. Der Ausfall eines der beiden Systeme wird auch das andere System stark beeinträchtigt. Neben dem Absturz eines der beiden Systeme, was eher selten der Fall ist, kann es aber auch zu Problemen in der Kommunikation untereinander kommen [1]. Beide Fehlerwahrscheinlichkeiten zusammen genommen erhöhen das gesamte Ausfallrisiko. Nachdem man sich endlich eingestanden hat, dass Ausfälle passieren können, wurde begonnen, nach Alternativen zu suchen. Doch dazu später mehr.

Das 2PC-Protokoll ist, wie der Name schon sagt, in zwei Phasen unterteilt. Phase 1 ist die sogenannte Prepare-Phase, die im zweiten Schritt mit der Commit-Phase abgeschlossen wird. Jede XA Transaction muss diese beiden Phasen durchlaufen, wodurch die Ausführung der Transaktion sehr langsam wird. Ein Mehr an Kommunikation erhöht wiederum die Wahrscheinlichkeit eines Fehlers. Für den Fall eines Absturzes muss das Datenbanksystem eine Recovery-Funktionalität besitzen, die das System wieder in einen fehlerfreien Zustand versetzen kann. All diese Anforderungen an das 2PC-Protokoll erhöhen die Komplexität eines Datenbanksystems enorm. Bei Hochlastsystemen hat man sich daher als Softwarearchitekt immer schon zweimal überlegt, ob der Use Case eine verteilte Transaktion erforderlich macht oder ob man darauf verzichten kann.

Doch es gibt auch ganz triviale Gründe für die Vermeidung von verteilten Transaktionen: Sie sind schlicht und ergreifend gar nicht möglich. Aktuelle Systeme kommunizieren über REST APIs auf Basis von HTTP miteinander und HTTP bietet eben keine verteilten Transaktionen an. Oder es werden zur Speicherung Datenbanksysteme verwendet, die keine verteilten Transaktion unterstützen.

ACID vs. BASE

Mit dem Wegfall von verteilten Transaktionen und der Persistierung in verteilen Systemen hat sich ein neues Konsistenzmodell etabliert. Früher hat allein ACID als gängiges Konsistenzmodell dominiert, auch die verteilten Transaktionen garantier(t)en diese Eigenschaften. ACID steht für Atomic, Consistent, Isolated und Durable und stellte sehr strenge Bedingungen an das Datenbanksystem. Mit dem erfolgreichen Festschreiben der Transaktion (Commit) kann man sich sicher sein, dass die Änderungen an den Daten definitiv sofort und

dauerhaft ausgeführt wurden. Alle nachfolgenden Leseoperationen liefern immer den aktuellen Datenzustand.

Im Zuge der NoSQL-Bewegung wurde vermehrt ein neues Konsistenzmodell etabliert: BASE. BASE steht für Basic Availability, Soft-State, Eventual Consistency. Die Implementierung von BASE ermöglicht den NoSQL-Systemen, bezüglich einer sofortigen Konsistenz und Datenaktualität ein paar kleine Abstriche in Kauf zu nehmen. Was aber nicht bedeuten soll, dass die Änderungen an den Daten verloren gehen, sondern manchmal erst etwas zeitversetzt ausgeführt werden. Den Vorteil, den man sich mit diesem Trade-off erarbeitet hat, liegt in der sehr guten Skalierbarkeit und der besseren Resilienz solcher Systeme. Ein möglicher Ausfall wird also als wahrscheinlich akzeptiert und kann somit auch besser kompensiert werden.

Schlussendliche Konsistenz (Eventual Consistency)

Der Preis, den man bei BASE zu zahlen hat, wird als Eventual Consistency bezeichnet. Leider ist die deutsche (Falsch-)Übersetzung („eventuelle Konsistenz") ein wenig unglücklich, da die Konsistenz nicht eventuell ist, sondern auch bei BASE definitiv garantiert wird. Nur eben ein wenig später als bei ACID. Dieser Umstand wird mit der deutschen Entsprechung „schlussendliche Konsistenz" besser ausgedrückt.

Die Erfahrung der letzten Jahre hat immer wieder gezeigt, dass Softwarearchitekten und Anwender sich mit diesem Umstand erst einmal anfreunden müssen. Es wurden immer wieder Fragen gestellt, wie beispielsweise: „Wann sind die Daten in dem anderen System gespeichert und wie soll der Anwender damit umgehen, dass er unter Umständen ‚alte' Daten angezeigt bekommt?"

In solchen Fällen war immer das Beispiel aus dem Onlinebanking hilfreich. Jahrelang war man es als Bankkunde gewohnt, nach einer Überweisung den reduzierten Kontosaldo sofort zu sehen, obwohl die Überweisungsliste erst sehr viel später die Überweisung angezeigt hat. Wieso sollte dieses Verhalten des Onlinebanking nicht auch in anderen Systemen gängige Praxis sein? Nach dem Abwägen der Vorteile bezüglich Resilienz und Skalierbarkeit (durch BASE) wird bei den meisten Use Cases die schlussendliche Konsistenz durch den Anwender dann doch akzeptiert.

Die „neuen" Umsetzungsstrategien

Neue Softwaresysteme, die mittels Service-zu-Service-Calls arbeiten und auch verteilte Datentöpfe verwenden, sollten nach den Erfahrungen der letzten Jahre nicht mehr mit verteilten Transaktionen arbeiten. Doch welche Möglichkeiten gibt es, zumindest die schlussendliche Konsistenz zu erreichen, ohne dabei Datenverluste zu erleiden? Viele der nachfolgenden Strategien existieren teilweise schon seit Jahrzehnten, sind jedoch heutzutage immer noch das Mittel der Wahl.

Best Efforts 1PC

Der erste Gedanke, den man bezüglich Transaktionssteuerung hat, ist die Umsetzung mit Best Efforts 1PC. Ein Artikel von Dr. David Syer [2] aus dem Jahr 2009 hat diesen Ansatz sehr gut beschrieben. Das Grundprinzip basiert darauf, zwei getrennte Transaktionen so zu verschachteln, dass erst ganz spät am Ende der Ablauflogik der Commit gegen beide beteiligte Systeme ausgeführt wird. Alle möglichen Fehlerfälle der Businesslogik wurden zuvor gelöst oder sind erst gar nicht aufgetreten. Der Commit der beiden Transaktionen erfolgt unmittelbar aufeinander, es befindet sich also keine Zeile Code mehr dazwischen. Die Annahme, die dahintersteht, ist die Hoffnung, dass zwischen den beiden Commits nichts passieren wird. Was aber stattfinden kann, sind die klassischen Probleme bei der Kommunikation mit verteilten Systemen. Es kann also durchaus sein, dass der erste Commit erfolgreich bestätigt wird, die Ausführung des zweiten Commits wegen einem Netzwerkfehler abgebrochen wird und die Daten per Rollback zurückgerollt werden. Das Endergebnis ist ein inkonsistenter Zustand der Daten. Um das zu vermeiden, könnte eine erneute Ausführung des zweiten Commits versucht werden, jedoch kann auch diese Fehlerkompensation den konsistenten Gesamtzustand nicht garantieren, wenn das zweite Datenbanksystem für längere Zeit nicht erreichbar ist.

Die mögliche Inkonsistenz, wenn auch sehr unwahrscheinlich, führt in erster Reaktion zur Ablehnung dieses Ansatzes. Auf den zweiten Blick passt diese Strategie sehr gut zu Use Cases, bei denen der Verlust der Datenänderung nicht ins Gewicht fällt oder der mögliche Datenverlust wegen der fehlenden Transaktionseigenschaften in den Folgesystemen akzeptiert werden muss. IoT-Systeme schicken so viele Messdaten, dass es in der Regel auf den Verlust von ein paar wenigen Daten nicht ankommt. Oder das Schreiben erfolgt gegen ein System, das gar keine Transaktionen kennt, wie beispielsweise ein LDAP-Server.

Transactional Outbox oder Outgoing Transactions

Ist der Verlust der Daten nicht akzeptabel, so kann als Alternative die Transactional Outbox [3] eingesetzt werden. Das Prinzip hierbei ist das transaktionale Schreiben in die fachlichen Tabellen der Anwendung und in eine sogenannte Outgoing Table innerhalb derselben Datenbank. Die Outgoing Table enthält die Nachrichten, die an das andere System übertragen werden müssen. Ein weiterer Prozess selektiert diese Datensätze und kümmert sich um die gesicherte Übertragung an das andere System. Dieser Schritt erfolgt wiederum innerhalb einer Transaktion.

Damit ergeben sich ein paar wichtige Vorteile. Die Daten in der Outgoing Table gehen nicht verloren. Bei einem Fehler in der Übertragung der ausgehenden Datensätze kann sie so oft wiederholt werden, bis sie erfolgreich durchgeführt wurde. Das geschieht ohne Beeinträchtigung der ursprünglichen Transaktion, da diese bereits festgeschrieben wurde. Falls die Daten in der Outgoing Table schon im gewünschten Format vorliegen (beispielsweise im JSON-Format), muss sich der Übertragungsprozess nicht mehr um die Konvertierung kümmern. Der Übertragungsprozess kann Bestandteil der Anwendung sein oder auch als eigenständiger Prozess betrieben werden. Darüber hinaus kann der Prozess so generisch im-

plementiert werden, dass er für viele Outgoing Tables in unterschiedlichen Systemen verwendet werden kann. Im Kubernetes-Umfeld wäre ein sogenannter Sidecar-Container mit dieser Funktionalität sehr gut einsetzbar.

Damit der Transfer der Nachrichten nicht zum Erliegen kommt, ist die Überwachung des Übertragungsprozesses ein kritischer Faktor. Da sich ein eigenständiger Prozess in der Regel leichter überwachen lässt als ein Thread innerhalb der Anwendung, sollte der Übertragungsprozess außerhalb der eigentlichen Anwendung laufen. Hier leistet Kubernetes sehr gute Dienste, indem der getrennt laufende Container mit Health Checks versehen werden kann.

Change Data Capture (CDC)

Da in den meisten Fällen die Persistierung der Daten in einem relationalen Datenbanksystem erfolgt, bietet sich mit CDC [4] eine Möglichkeit, die nach demselben Funktionsmuster wie die Outgoing Table funktioniert. Der einzige Unterschied liegt darin, dass die Änderungen in der Datenbank aus dem Transaktionslog gelesen werden. Somit kann für jede Tabelle der Datenbank eine CDC-Überwachung definiert werden. Erst wenn die Änderung an den Tabellen im Transaktionslog festgeschrieben wurde, startet der CDC-Prozess mit seiner Arbeit. Viele der großen kommerziellen Datenbankhersteller bieten diese Funktionalität schon seit sehr vielen Jahren an. Aber keine Sorge, wer sich im Open-Source-Umfeld aufhält, findet bei Debezium [5] die notwendige Unterstützung für seine Datenbank.

Ein Nachteil von CDC liegt darin, dass die betroffenen Tabellen, so wie sie in der Datenbank definiert werden, auch durch das Transaktionslog überwacht werden. Eine fachliche Änderung innerhalb der Datenbank kann sich aber über mehrere Tabellen erstrecken. Das hat zur Folge, dass solche Änderungen wieder zu einer Gesamtheit zusammengeführt werden müssen. Hierfür bietet es sich an, die Outgoing Table von oben einzusetzen, die dann vom CDC-Prozess überwacht wird. Wer will, kann zur denormalisierten Befüllung der Outgoing Table altbekannte Datenbankobjekte wie Trigger und Stored Procedure einsetzen. Somit bietet die Kombination aus Outgoing Transactions und CDC ein mächtiges Werkzeug, um Daten zuverlässig zu übertragen.

Single Source of Truth (SSOT)

Die Strategie SSOT [6] ist auch nicht neu. Das Prinzip dahinter besagt, dass die Wahrheit der Daten an einer einzigen Stelle liegen soll. Von dieser Stelle aus können sich andere Systeme die Daten holen und sind zum Zeitpunkt des Abrufens sicher, dass es sich um den aktuellen Stand handelt. Übertragen auf unsere Problemstellung der Kompensation nicht vorhandener verteilter Transaktionen ergibt sich also die folgende Möglichkeit: Datenänderung in einem Service werden nicht sofort in der zugehörigen Datenbank selbst gespeichert, sondern stattdessen transaktional an ein Messagingsystem übertragen. Die Services besitzen einen Message Consumer, der auf diese Nachrichten lauscht. Sobald die Nachricht von den Consumern empfangen werden, führen sie die Änderung in der jeweiligen Datenbank aus. Die Rückübertragung der Nachricht vom Messagingsystem in die Datenbank erfolgt wieder innerhalb einer Transaktion. Eine ausführliche Beschreibung unter Verwendung von Kafka kann auf dem Blog von Confluent [7] nachgelesen werden.

Der Einsatz dieser indirekten Datenänderung macht vor allem dann Sinn, wenn auch noch andere Systeme an den Änderungen interessiert sind. Sobald diese Nachrichten als Events modelliert und als grundsätzliches Kommunikationsmuster eingesetzt werden, spricht man von Event Sourcing [8]. Ein Nachteil beim SSOT liegt in der Tatsache begründet, dass auch der Service, der die Änderungen direkt vom Clientaufruf empfangen hat, diese nicht sofort in seiner Datenbank speichert. Der Client bekommt die Änderung als erfolgreich bestätigt, sobald die Nachricht ins Messagingsystem übertragen worden ist. Verzögert sich nun die Verarbeitung durch den eigenen Message Consumer, bekommt der Client, falls er direkt danach die vermeintlich geänderten Daten abfragt, immer noch die alten Werte angezeigt. Dieses Verhalten ist einem Anwender nur sehr schwer zu erklären und im Normalfall geht der Anwender davon aus, dass seine Änderungen nicht ordnungsgemäß ausgeführt wurden. Damit verschwindet das Vertrauen in die Korrektheit, wodurch auch die schlussendliche Konsistenz in Frage gestellt wird.

Sagas

Die letzte Möglichkeit in dieser Aufzählung sind Sagas [9]. Sagas sind Abfolgen von lokalen Transaktionen, die wie-

derum weitere lokale Transaktionen in anderen Services über Events oder Trigger auslösen. Sobald ein Fehler auftritt, wird eine Serie von sogenannter Kompensationstransaktionen ausgelöst, die die zuvor gemachten Änderungen wieder rückgängig machen. Die Weiterleitung der Events bzw. Trigger kann dabei über ein Messagingsystem oder durch direkte synchrone Aufrufe stattfinden. Die Steuerung dieser Abfolge der lokalen Transaktionen kann dezentral, also mittels Choreografie, oder zentral über eine sogenannte Orchestrierung erfolgen.

Die Umsetzung mit einer zentralen Steuerungslogik ähnelt sehr stark dem Vorgehen bei einer verteilten Transaktion (2PC). Auch hier ist der zentrale Transaktionsmanager für die Abfolge der Transaktionen in den beteiligten Transaktionspartnern zuständig. Im Fehlerfall muss er alle Transaktionspartner über das Rollback informieren, wodurch diese dann für das Zurückführen der Änderung zuständig sind. Im Fall der Datenbanken wird einfach das sogenannte After Image verworfen und das Before Image wieder als aktueller Datenstand gespeichert. Dieses Rollback kann bei Sagas nur sehr umständlich umgesetzt werden. Jeder Service muss die Businesslogik für das Zurückrollen der Änderungen selbst implementieren, was einen nicht unerheblichen Programmier- und Testaufwand darstellt.

Es gibt jedoch noch einen weiteren speziellen Nachteil, den man bei Sagas kaum verhindern kann: Sobald ein Service die Änderungen der sogenannten Saga-Transaktion lokal bei sich ausgeführt hat, können diese Daten von anderen Services über einen API-Aufruf gelesen werden. Werden diese Änderungen jetzt mit einer Kompensationstransaktion im Rahmen der Saga wieder zurückgerollt, bekommt der API-Aufrufer davon nichts mit. In der Datenbankwelt spricht man von Phantom oder Dirty Reads. Der API-Aufrufer hat also Daten gelesen, die er eigentlich nie hätte sehen sollen, da die zugrundeliegende Transaktion später zurückgenommen wurde.

Beim dezentralen Ansatz zur Koordination, also mit Choreografie, kommt es immer wieder zu dem Fall, dass eine Saga-Transaktion empfangen wird und eine weitere Saga-Transaktion ausgelöst werden muss. Es findet also eine lokale Transaktion gegen die Datenbank und die Erstellung einer weiteren Saga-Transaktion statt. Im Fall einer asynchronen Weiterleitung der Saga-Transaktion ergibt sich somit wieder die Notwendigkeit einer verteilten Transaktion. Der Übergang in Richtung Event Sourcing kann fließend damit einhergehen.

Fazit

Man sollte je nach Use Case entscheiden, welche der aufgeführten Strategien am besten passt. Je einfacher das gewählte Verfahren, desto einfacher ist die Umsetzung und desto geringer ist die Wahrscheinlichkeit, Daten in einem inkonsistenten Zustand zu hinterlassen. So kann es durchaus sein, dass ein Best Efforts 1PC für den speziellen Fall genau die richtige Wahl darstellt.

Outgoing Transactions in Kombination mit CDC bieten ein relativ einfach verständliches Verfahren, das ohne Datenverluste große Sicherheit bietet. Auch ohne den Einsatz von CDC kann man mit vernünftigem Aufwand eine sichere Lösung selbst implementieren, die mit der passenden Generik mehrfach zum Einsatz kommen kann. Dieser Mehrfacheinsatz führt automatisch zu einer breiten Testbasis und außerdem amortisiert sich der anfängliche Implementierungsaufwand sehr schnell.

Der SSOT-Ansatz verschwimmt in den Projekten sehr oft mit Event Sourcing. Wenn die Umsetzung von SSOT schon nach den Prinzipien von Event Sourcing erfolgt (Events als Nachrichtenformat), ist die Möglichkeit des Einsatzes von Event Sourcing zumindest nicht blockiert. Ob Event Sourcing für ein Projekt der richtige Ansatz ist, sollte jedoch unter Einbeziehung weiterführender Kriterien diskutiert und aktiv entschieden oder abgelehnt werden.

Sagas können zum Teil sehr kompliziert werden und bieten somit in der Umsetzung einiges an Fehlerpotential. Die notwendigen Kompensationstransaktionen erhöhen ebenso den Aufwand für die Verwendung. Darüber hinaus müssen diverse Entscheidungen zur Transaktionskoordination und zur Weiterleitung der Transaktionen getroffen werden. Ohne passende Frameworks sollten Sagas nicht die erste Wahl zur Vermeidung von verteilten Transaktionen sein. Leider ist die Auswahl an Saga Frameworks noch sehr gering. Aktuell stehen im Java-Bereich nur sehr wenige Saga Frameworks zur Auswahl (u. a. Eventuate Tram Sagas [10]) und MicroProfile bemüht sich seit Langem, mit MicroProfile Long Running Actions (LRA) [11] einen Standard zu etablieren. Bleibt also abzuwarten, ob sich in diesem Gebiet in Zukunft noch etwas bewegt.

Michael Hofmann ist freiberuflich als Berater, Coach, Referent und Autor tätig. Seine langjährigen Projekterfahrungen in den Bereichen Softwarearchitektur, Java Enterprise und DevOps hat er im deutschen und internationalen Umfeld gesammelt.

info@hofmann-itconsulting.de

Links & Literatur

[1] https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

[2] https://www.javaworld.com/article/2077963/distributed-transactions-in-spring--with-and-without-xa.html?page=2

[3] https://microservices.io/patterns/data/transactional-outbox.html

[4] https://en.wikipedia.org/wiki/Change_data_capture

[5] https://debezium.io/blog/2019/02/19/reliable-microservices-data-exchange-with-the-outbox-pattern/

[6] https://en.wikipedia.org/wiki/Single_source_of_truth

[7] https://www.confluent.io/blog/messaging-single-source-truth/

[8] https://martinfowler.com/eaaDev/EventSourcing.html

[9] https://microservices.io/patterns/data/saga.html

[10] https://github.com/eventuate-tram/eventuate-tram-sagas

[11] https://github.com/eclipse/microprofile-lra

CQRS und DDD

Gemeinsam mehr erreichen

Die Verbreitung der Akronyme CQRS und DDD nimmt seit einigen Jahren stetig zu: Vor allem im Kontext von Microservices begegnet man den beiden Konzepten immer häufiger. Was steckt dahinter?

von Golo Roden

CQRS steht für Command Query Responsibility Segregation und bezeichnet die strikte Trennung der Schreib- und der Lesevorgänge einer Anwendung beziehungsweise eines API. In der klassischen Anwendungsarchitektur wird die mittlere Ebene, die typischerweise über ein API verfügt, als eine in sich geschlossene Einheit gesehen, auf die vom UI aus zugegriffen wird. CQRS schlägt stattdessen vor, die Aktionen des Benutzers ganz klar zu kategorisieren:

- Auf der einen Seite stehen Commands, die eine Intention des Benutzers ausdrücken und zumeist eine Zustandsänderung innerhalb der Anwendung nach sich ziehen. Da sie die Intention ausdrücken, sollten sie nicht mit Create, Update und Delete benannt werden. Diese Verben bezeichnen nämlich technische Implikationen, nicht fachliche Absichten. In der Fachsprache tauchen üblicherweise andere, domänenspezifische Begriffe auf, die sich in den Commands entsprechend wiederfinden sollten.
- Auf der anderen Seite stehen Queries, die den aktuellen Zustand der Anwendung abfragen, ihn aber nicht ändern. Queries entsprechen also reinen Lesevorgängen, wohingegen Commands das Schreiben in einer Anwendung repräsentieren.

Viel mehr als das ist CQRS zunächst einmal nicht. Es sagt insbesondere nichts über die konkrete Implementierung aus, auch nicht über die Struktur der Datenhaltung. CQRS gibt noch nicht einmal vor, wie das API strukturiert und seine Routen benannt sein sollten: Wichtig ist einzig und allein, dass das Schreiben vom Lesen getrennt wird – auf welche Art auch immer.

CQRS in einem API implementieren

Eine einfache Möglichkeit, CQRS in einem API zu implementieren, besteht darin, sich gedanklich von REST zu verabschieden und stattdessen nur noch *POST* und *GET* einzusetzen. *POST* steht dabei aus naheliegenden Gründen für schreibende Vorgänge (also für Commands), *GET* wird hingegen zum Lesen (also für Queries) verwendet. Da die Semantik wie Anlegen, Ändern oder Löschen von Dingen nun nicht mehr über das HTTP-Verb ausgedrückt wird, muss die eigentliche Aktion Bestandteil des URL werden.

Queries beziehen sich immer auf den aktuellen Stand und entsprechen vom Verb her letztlich immer dem Lesen. Bei ihnen genügt deshalb eine substantivische Beschreibung, welche Daten des Zustands gelesen werden sollen.

Bevor man beginnt, ein API zu implementieren, lohnt es sich, eine Liste der gewünschten Commands und Queries zu erstellen und zumindest für einen Moment über ihre Benennung nachzudenken. Das wiederum funktioniert natürlich nur in einem gegebenen fachlichen Kontext. Deshalb wird im weiteren Verlauf des Artikels eine einfache To-do-Listen-Anwendung als Beispiel verwendet, wie man sie beispielsweise aus dem Projekt TodoMVC [1] kennt.

Doch selbst das genügt noch nicht, um festzulegen, welche Commands und Queries man benötigt. Dafür muss nämlich zuerst noch mehr Klarheit bezüglich der tatsächlichen Funktionalität geschaffen werden. **Abbildung 1** zeigt das dazu passende UI:

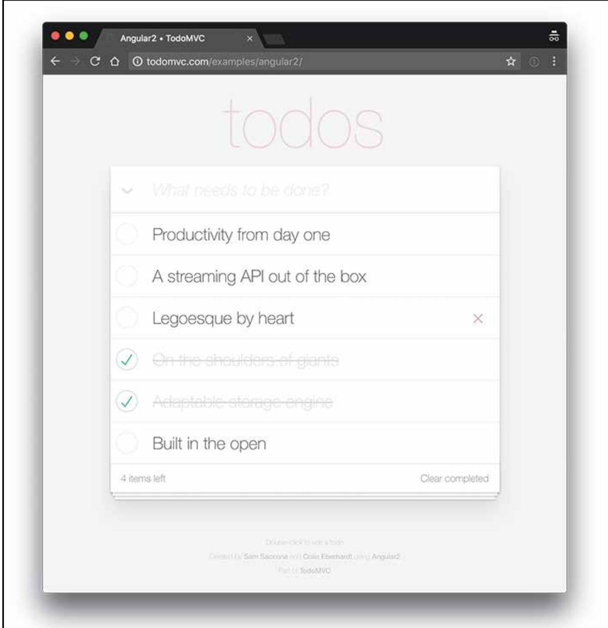


Abb. 1: UI der To-do-Anwendung

- Der Anwender soll sich neue Aufgaben notieren können, wobei eine Aufgabe lediglich aus einer Beschreibung besteht.
- Die Beschreibung einer Aufgabe soll sich auch im Nachhinein noch ändern lassen.
- Aufgaben können, wenn sie erledigt wurden, vom Anwender abgehakt werden.
- Das Abhaken einer Aufgabe soll sich zurücknehmen lassen. Das ist beispielsweise dann relevant, wenn eine Aufgabe fälschlicherweise oder aus Versehen abgehakt wurde.
- Gelegentlich erübrigen sich Aufgaben im Lauf der Zeit auch von selbst. Sie sind zwar hinfällig, aber genau genommen nicht erledigt. Insofern wäre es falsch, sie abzuhaken. Vielmehr sollen solche Aufgaben verworfen werden können.
- Zu guter Letzt soll es für den Anwender möglich sein, erledigte beziehungsweise hinfällige Aufgaben zu archivieren. Solche Aufgaben sollen dann zwar nicht gelöscht werden, über das UI aber nicht mehr sichtbar sein.

Wie bereits erwähnt empfiehlt es sich an der Stelle, nicht aus Gewohnheit auf Create, Update und Delete zu setzen, sondern Begriffe zu suchen, die auch in der Fachlichkeit enthalten sind. Das führt dazu, dass das API und die Fachlichkeit näher aneinanderrücken, was das Verständnis der Verwendern für das API wiederum verbessert: Man muss nicht eine fachliche und eine technische Sprache und das Mapping zwischen beiden lernen – es genügt, wenn man sich in der Fachsprache auskennt.

Commands und Queries definieren

Das Command zum Notieren einer neuen Aufgabe sollte also nicht *Create Task* heißen, denn Create ist von seiner Bedeutung her zu weit weg von „notieren" (abgesehen davon handelt es sich bei der Anwendung um eine To-do- und nicht um eine Taskliste, weshalb auch das Wort Task unpassend ist). Deutlich besser wäre daher z. B. *Note Todo*.

Auch das Bearbeiten einer Aufgabe ist nicht einfach nur ein *Update Todo*, da Update ein Wort ist, das einen technischen Vorgang beschreibt, aber keinen fachlichen. Besser, weil inhaltlich treffender, wäre hier etwa *Edit Todo*. Spätestens beim Abhaken, dem Verwerfen und dem Archivieren merkt man sehr deutlich, wie unpassend *Update* und *Delete* sind: Während das eine viel zu unspezifisch ist, trifft das andere genau genommen in keinem einzigen Fall zu – schließlich sollen die verschiedenen Aktionen rückgängig gemacht werden können, und beim Archivieren ist sogar explizit gefordert, dass die Aufgabe zwar nicht mehr angezeigt, aber eben nicht gelöscht wird. Würde man sich hier mit dem klassischen Vokabular begnügen, hätte man lauter *Update Todo*-Commands, die man nicht voneinander unterscheiden könnte und die deshalb ziemlich beliebig wären.

Passender wären zum Beispiel *Tick Off Todo*, *Resume Todo* und *Discard Todo* zum Abhaken, Fortsetzen und Verwerfen von Aufgaben. Zum Archivieren schließlich könnte man das naheliegende *Archive Todo* verwenden. Benennt man nun die Routen des API entsprechend, hat man ein sprechendes API, bei dem man sich nicht über die (am Ende doch nur mangelhafte) Zuordnung von HTTP-Verben zu fachlichen Aktionen Gedanken machen muss, und das sich – wenn man die Fachlichkeit kennt – intuitiv verwenden lässt:

```
POST /note-todo
POST /edit-todo
POST /tick-off-todo
POST /resume-todo
POST /discard-todo
POST /archive-todo
```

Alle diese Formulierungen stehen bereits im Imperativ, also in der Befehlsform, was hervorragend die Tatsache widerspiegelt, dass es sich um Commands handelt: Der Anwender beauftragt den Computer bzw. die Anwendung damit, etwas für ihn zu erledigen, was den Zustand der Anwendung verändert. Genau das trifft die zuvor beschriebene Erklärung eines Commands in CQRS. Es repräsentiert die Intention des Anwenders und verändert den Zustand der Anwendung.

Die Queries lassen sich nach diesen Vorüberlegungen deutlich einfacher formulieren. Da das Verb bei ihnen festgeschrieben ist, genügt wie bereits erwähnt eine substantivische Beschreibung dessen, was zu lesen ist. Im einfachsten Fall ist das eine Liste aller noch nicht archivierten Aufgaben, unabhängig von ihrem sonstigen Zustand. Selbstverständlich könnte man noch weitere Sichten auf die Daten definieren, aber da eine To-do-Liste selten Tausende von Einträgen enthält, kann man es sich an der Stelle auch einfach machen und eine einzige Liste für alles verwenden und sie gegebenenfalls auf dem Client filtern: *GET /todos*

Und die Datenhaltung?

Damit ist nun das API gemäß CQRS definiert. Die Implementierung eines Clients dürfte relativ einfach vonstattengehen und auch der Server dürfte nicht allzu schwierig umzusetzen sein. Interessant ist an der Stelle höchstens noch die Frage nach der Datenhaltung. Sollte man, wenn man statt REST auf ein CQRS-basiertes API setzt, eine klassische relationale Datenbank zum Speichern von Daten verwenden?

Auf den ersten Blick spricht nichts dagegen. Ein erster Entwurf für ein Datenmodell könnte vorsehen, dass es eine Tabelle namens *Todos* gibt. Die Commands würden auf den einzelnen Datensätzen dieser Tabelle agieren und die Query würde alle Datensätze zurückgeben, die nicht als archiviert markiert wurden. Die einzige Herausforderung an der Stelle ist das Mapping von fachlichen auf technische Konstrukte: So würde in der Datenbank das *Note Todo* beispielsweise einem *INSERT* entsprechen, alle anderen Commands hingegen einem *UPDATE*.

Den aktuellen Anforderungen würde dieser Ansatz genügen, doch wäre er nicht besonders ausbaufähig. Insbesondere für das Reporting ist dieses Vorgehen nicht besonders gut geeignet, da keinerlei historische Informationen aufbewahrt werden. Gerade bei einer To-do-Liste könnte es aber interessant sein, gewisse Statistiken aus den Daten zu ermitteln. Beispielsweise könnte man folgende Fragen beantworten wollen:

- Wie viel Zeit vergeht zwischen dem Notieren und dem Abhaken einer Aufgabe?
- Wie häufig wird eine Aufgabe editiert, bevor sie abgehakt wird?
- Wie viele Aufgaben werden verworfen, obwohl sie zuvor mindestens dreimal editiert wurden?
- Wie häufig wird innerhalb von 30 Sekunden das Abhaken rückgängig gemacht?

Diese Liste ließe sich endlos fortsetzen. Das Erschreckende daran ist, dass sich keine einzige dieser Fragen ohne Weiteres beantworten ließe, weil die Datenbasis diese Informationen nicht enthält. Theoretisch hätte man diese Informationen aber speichern können – schließlich sind alle Aktionen, nach denen gefragt wird, irgendwann passiert. Warum liegen sie dann nicht vor?

Das Problem liegt darin begründet, dass das beschriebene Vorgehen die Historie verliert. Editiert ein Anwender beispielsweise eine Aufgabe mehrfach, lassen sich die einzelnen Bearbeitungsschritte nicht mehr nachvollziehen. Schlimmer noch: Es lässt sich noch nicht einmal mehr nachvollziehen, wann oder ob die Aufgabe überhaupt jemals editiert wurde. Selbstverständlich kann man das in den Griff bekommen, indem man ein Feld für einen Zeitstempel der letzten Aktualisierung einfügt. Doch auch das hilft noch nicht in allen Fällen, da sich die einzelnen Bearbeitungsstände nicht separat abrufen lassen. Hierzu müsste man entweder eine ganze Reihe von Feldern oder gleich eine neue Tabelle anlegen.

Da man nicht weiß, welche Datenabfragen zukünftig gestellt werden, kann man die Tabellen nicht auf die entsprechenden Antworten optimieren. Man sammelt daher entweder zu viele oder zu wenige Daten, ganz sicher aber stets die falschen. Werden dann Fragen wie die oben genannten gestellt, lassen sie sich – wenn überhaupt – nur mit Glück beantworten. In der Regel wird die Antwort auf jede dieser Fragen lauten, dass man zunächst Code schreiben müsse, der die dafür relevanten Daten erfasst und speichert. Anschließend müsse man lediglich noch ein paar Wochen oder Monate warten, und schon könne man die gewünschte Auswertung liefern. Dass das alles andere als befriedigend ist, liegt auf der Hand.

Vom Status quo zu Domain Events

Was man also braucht, ist eine andere Art der Datenspeicherung: Anstatt den aktuellen Zustand zu speichern, sollte man die einzelnen Veränderungen speichern, die zum Status quo geführt haben. Das klingt zunächst danach, als ob man die eingegangenen Commands sammeln sollte. Commands haben jedoch die Eigenschaft, dass eine Anwendung sie ausführen oder verweigern kann. Aus der reinen Tatsache, dass ein Command an ein API geschickt wurde, lässt sich noch nicht folgern, dass es auch erfolgreich ausgeführt wurde. Viel inter-

Listing 1

id	eventType	order	timestamp	payload
1	todoNoted	1	29.12.2019 19:05	{ description: 'go shoping' }
1	todoEdited	2	29.12.2019 19:07	{ description: 'go shopping' }
1	todoTickedOff	3	02.01.2019 10:13	{}
1	todoArchived	4	02.01.2019 13:49	{}

essanter als das reine Command ist also die Reaktion darauf: Was wurde von der Anwendung in fachlicher Hinsicht entschieden, sozusagen als Reaktion auf das Command? Diese Reaktionen sind schnell identifiziert, da sie sich – zumindest im vorliegenden Beispiel – originär auf die Commands abbilden lassen. Das muss jedoch nicht bei jeder Anwendung so sein und insbesondere bei komplexeren Anwendungen ist das Vorgehen nicht mehr ganz so einfach:

- Eine Aufgabe wurde notiert.
- Eine Aufgabe wurde editiert.
- Eine Aufgabe wurde abgehakt.
- Eine Aufgabe wurde fortgesetzt.
- Eine Aufgabe wurde verworfen.
- Eine Aufgabe wurde archiviert.

Es fällt auf, dass alle diese Formulierung in der Vergangenheitsform stehen. Das hat den einfachen Grund, dass es sich hierbei um Fakten handelt, die als Reaktion auf ein Command entstanden sind. Diese Dinge sind passiert und es gibt nichts auf der Welt, das man tun könnte, um diese Ereignisse ungeschehen zu machen. Natürlich lässt sich eine Gegentransaktion ausführen, doch die reine Tatsache, dass ein Ereignis einmal stattgefunden hat, ist unumkehrbar. Der Effekt lässt sich ausgleichen – das Ereignis an sich nicht.

Diese Ereignisse werden als Domain Events bezeichnet und sie lassen sich in einer immer weiterwachsenden Liste von Ereignissen speichern. Dafür eine Datenbankstruktur aufzusetzen, ist sehr simpel: Außer *INSERT* und *SELECT* wird nichts benötigt. Wie das API bei CQRS reduziert man an der Stelle nun also die Datenbank auf einfaches Schreiben und Lesen. Als Ergebnis erhält man eine Liste, die der im Listing 1 ähnelt. Zusätzlich zu den bereits beschriebenen Informationen enthält sie außerdem noch eine *id*-Spalte, damit man mehrere Aufgaben unterscheiden kann, und eine *order*, mit der sich die Reihenfolge der Events in Bezug auf eine Aufgabe ermitteln lässt.

Speichert man Domain Events auf diese Weise, kann man nicht nur den aktuellen Zustand einer Aufgabe wieder ermitteln, sondern auch die zuvor angesprochenen Fragen beantworten:

- Wie viel Zeit vergeht zwischen dem Notieren und dem Abhaken einer Aufgabe? Anders formuliert: Wie viel Zeit liegt zwischen dem *todoNoted* und dem *todoTickedOff* Event? Die Antwort lässt sich anhand des Timestamp rasch ermitteln: Zwischen den beiden Domain Events sind 3 Tage, 15 Stunden und 8 Minuten vergangen. Berechnet man diese Differenz für mehrere Aufgaben, kann man außerdem angeben, wie viel Zeit üblicherweise im Durchschnitt pro Aufgabe vergeht.
- Wie häufig wird eine Aufgabe editiert, bevor sie abgehakt wird? Anders formuliert: Wie viele *todoEdited* Events gibt es nach dem *todoNoted* Event, aber vor dem zugehörigen *todoTickedOff* Event? Die Antwort für die Beispielaufgabe lautet 1.
- Wie viele Aufgaben werden verworfen, obwohl sie zuvor mindestens dreimal editiert wurden? Anders formuliert: Wie viele Aufgaben weisen mindestens drei *todoEdited* und außerdem auch ein *todoDiscarded* Event auf? Im vorliegenden Beispiel trifft das nicht zu, weshalb die Antwort 0 lautet.
- Wie häufig wird innerhalb von 30 Sekunden das Abhaken rückgängig gemacht? Anders formuliert: Bei wie vielen Aufgaben folgt innerhalb von 30 Sekunden ein *todoResumed* Event auf ein *todoTickedOff* Event? Da in dem Beispiel kein *todoResumed* Event enthalten ist, liegt die Häufigkeit bei 0.

Diese Art der Datenspeicherung nennt man Event Sourcing, weil Domain Events als Quelle genutzt werden, um Informationen zu ermitteln. Eine Datenbank, die wie die zuvor beschriebene Liste funktioniert, nennt man Event Store. Es liegt auf der Hand, dass man CQRS und Event Sourcing unabhängig voneinander nutzen kann – immerhin kam Event Sourcing in der Beschreibung von CQRS zunächst nicht vor und selbstverständlich lassen sich auch Ereignisse in einer Liste speichern, ohne dass das damit verbundene API nach CQRS strukturiert sein muss. Aber man kann die beiden Konzepte eben auch wunderbar miteinander verbinden.

Wenn man so vorgehen möchte, resultieren Commands also letztlich in Domain Events, die in einem

DDD Workshop: Collaborative Modeling und Strategisches Design

Henning Schwentner
(WPS – Workplace Solutions)

Eine grundsätzliche Frage bei der Softwareentwicklung ist, wie man die Domäne richtig schneiden kann. In Zeiten von Microservices ist das besonders bewusst geworden, aber auch unabhängig davon muss man diese Frage beantworten. Ziel ist es, zu einer tragfähigen Architektur und einem guten Domänenmodell zu kommen. Dabei ist erstens wichtig, mit Collaborative-Modeling-Techniken wie Event Storming und Domain Storytelling die Domäne zu verstehen. Im zweiten Schritt werden die Bounded Contexts und Subdomänen herausgearbeitet. Der Workshop wird aus Vorträgen und Übungen bestehen. Wir werden uns Beispiele aus der Praxis anschauen, und in den Übungen erarbeiten sich die Teilnehmer selbst an einer Beispieldomäne eine Aufteilung in Bounded Contexts. Der Workshop wird folgendermaßen strukturiert sein: Einführung und Überblick DDD, Collaborative Modeling, Strategisches Design mit Bounded Context, Strategisches Design mit Subdomains Der Workshop eignet sich für Entwickler und Architekten, aber auch Kollegen aus dem Fachbereich. Keine Laptops nötig; wir konzentrieren uns auf die Domäne.

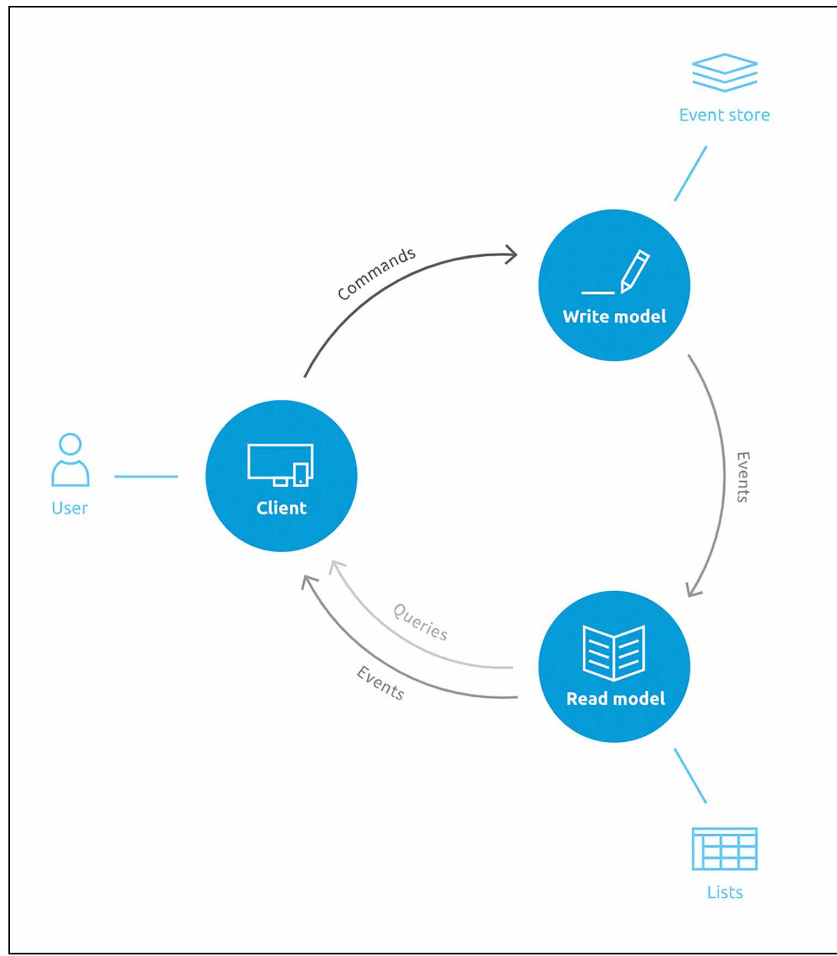


Abb. 2: Zusammenhang zwischen DDD, Event Sourcing und CQRS

Event Store gespeichert werden. Zum Auslesen müssen die Domain Events wieder abgespult werden. Dieser Vorgang wird als Replay bezeichnet. Allerdings gibt es dabei auch einen gravierenden Haken: Je mehr Events gesammelt und gespeichert werden, desto länger dauert es, ein Replay durchzuführen, da die Anzahl der Events zunehmend größer wird.

Allerdings gibt es einen einfachen Ausweg. Da Domain Events per Definition niemals geändert oder gelöscht werden, ergibt ein Replay bis zu einem bestimmten Zeitpunkt immer dasselbe Ergebnis. Um beim obigen Beispiel zu bleiben: Es spielt für den Replay der ersten drei Events keine Rolle, ob danach noch weitere Events kommen oder nicht – und wenn ja, wie viele. Das wiederum bedeutet, dass sich das Ergebnis eines Replays problemlos cachen lässt. Ein solcher zwischengespeicherter Wert wird beim Event Sourcing als Snapshot bezeichnet. Statt einen Replay stets von Anfang an durchführen zu müssen, kann man daher auf einen Snapshot zurückgreifen und muss nur noch die seither aufgetretenen Events abspielen, um auf den aktuellen Stand zu kommen. Werden solche Snapshots regelmäßig geschrieben, lässt sich damit eine nahezu lineare Performance erzielen. Außerdem lässt sich das Konzept dazu nutzen, Views vorzuberechnen: Schreibt man stets nach jedem Event einen neuen Snapshot, hat man jederzeit den aktuellen Zustand der Anwendung griffbereit und kann ihn präsentieren.

Semantik und Struktur von Domain Events

Da CQRS und Event Sourcing zwei technisch geprägte Konzepte sind, lösen sie auch ausschließlich technische Probleme. Die Semantik und Struktur von Domain Events ergibt sich jedoch weder aus dem einen noch aus dem anderen – ob Events also sinnvoll modelliert sind und fachlich relevante Daten enthalten, wird durch CQRS und Event Sourcing nicht beeinflusst. Die beiden Mechanismen beschreiben lediglich, wie sich Daten auf Basis von Events speichern und wie sich fachlich sprechende APIs entwickeln lassen. Woher kommt also die Semantik? Beziehungsweise wie geht man beim Entwurf einer Anwendung vor, um sicherzugehen, dass die verwendeten Commands und Events auch tatsächlich das abbilden, was der Fachlichkeit nützt?

Das ist aus naheliegenden Gründen keine technische Frage, weshalb sie sich auch nicht technisch beantworten lässt. Stattdessen benötigt man ein Konzept, um fachliche Konzepte bereits beim Entwurf und während der Entwicklung zu verstehen, zu greifen und in Code zu implementieren. Genau hier kommt der Ansatz Domain-driven Design (DDD) ins Spiel: DDD ist eine Methodik, um in einem interdisziplinären Team ein gemeinsames Verständnis und eine gemeinsame Sprache für die Fachlichkeit zu entwickeln. DDD rückt also, anders als beispielsweise Test-driven Design (TDD) kein technisches Artefakt in den Fokus, sondern die Fachlichkeit und deren Verständnis.

Dadurch kommunizieren unter anderem Fachexperten, Entwickler und Designer frühzeitig miteinander und bekommen auf diesem Weg ein gemeinsames Verständnis der zu entwickelnden Thematik. Die gemeinsame Sprache, die in DDD als Ubiquitous Language bezeichnet wird, vereinfacht die Kommunikation in den späteren Phasen der Entwicklung spürbar. Obwohl die Begriffe dieser Sprache von der modellierten Fachdomäne abhängen, folgen sie doch gewissen Regeln.

So gibt es in DDD beispielsweise wie in CQRS Commands, die die Wünsche der Anwender darstellen. Domain Events hingegen sind die von der Anwendung als Reaktionen auf die Commands geschaffenen Fakten, die nicht mehr veränderlich sind und deshalb in der Vergangenheitsform stehen. Das klingt inzwischen bereits

vertraut, obwohl es unabhängig von CQRS und Event Sourcing entstanden ist. Die gemeinsame Logik der Commands und Events ist in sogenannten Aggregates gekapselt, die zudem auch den Zustand der Anwendung in logische Einheiten kapseln. Eine der Besonderheiten von DDD ist, dass es nicht die Daten der Anwendung in den Mittelpunkt rückt, sondern die Prozesse. Warum das wichtig ist, hat Steve Yegge sehr anschaulich in seinem lesenswerten Blogeintrag „Execution in the Kingdom of Nouns“ [2] beschrieben.

Gemeinsam mehr erreichen

Obwohl DDD unabhängig von CQRS und Event Sourcing entstanden ist und sich daher unabhängig von diesen beiden Ansätzen einsetzen lässt, ergänzen sich die drei Konzepte hervorragend. Dabei wird den Events eine tragende Rolle zuteil: In DDD fungieren sie als fachliche und semantische Grundlage für die Modellierung. In Event Sourcing legt bereits der Name nahe, dass es ebendiese Events sind, die es als Veränderungen zu speichern gilt. Und in CQRS dienen Events schließlich der Synchronisation von Schreib- und Leseseite. **Abbildung 2** stellt den Zusammenhang zwischen DDD, Event-Sourcing und CQRS an Hand des Datenflusses einer Anwendung grafisch dar. Im Write Model, also der Schreibseite der Anwendung, verwendet man DDD, um eingehende Commands zu verarbeiten und sie in Domain Events zu transformieren. Diese Events legt man anschließend mit Hilfe von Event Sourcing in einen Event Store. Danach werden die Events für die Synchronisation der Schreib- und der Leseseiten, die in einer CQRS-Anwendung existieren, an das Read Model übertragen. Dort interpretiert man die Events und aktualisiert die zu lesenden Views gemäß den individuellen Anforderungen.

Auf diese Art fügen sich die drei Konzepte wie Puzzleteile zu einem großen Ganzen zusammen. Als Ergebnis erhält man einen Ansatz zum Entwickeln von Anwendungen, der sich nah an der Fachlichkeit bewegt, hervorragende Analysemöglichkeiten bietet, und der sich außerdem gut skalieren lässt. Dank des Einsatzes von DDD ist von Vornherein ein interdisziplinäres Team involviert, weshalb auf diesem Weg in kürzerer Zeit bessere Software gelingt.

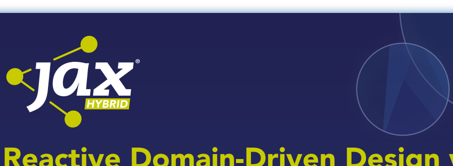
Fazit

Selbstverständlich gibt es zu jedem der drei einzelnen Konzepte, CQRS, Event Sourcing und DDD, noch weitaus mehr zu schreiben. Insbesondere gibt es, bedingt durch die andere Art der Architektur, eine Reihe von Sonderfällen, die man zwar nicht alle im finalen Code berücksichtigen muss, denen man sich aber zumindest bewusst sein sollte. Folgt man während der Entwicklung von Software diesen Konzepten, erhält man als Ergebnis eine Reihe von Vorteilen gegenüber klassischer Architektur und einem herkömmlichen Vorgehensmodell – man hat aber eben auch mit einer anderen Art von Problemen zu kämpfen.

Ob die Vor- oder die Nachteile überwiegen, muss für jede Anwendung einzeln bewertet werden. Tendenziell lässt sich aber sagen, dass es kein Vorgehen gibt, das keinerlei Probleme mit sich brächte, man durch den Einsatz von CQRS und Co. aber zumindest eine ganze Reihe üblicher Probleme lösen kann – was insbesondere im Bereich der Datenauswertung praktisch ist. Auch die Tatsache, dass DDD zu einem besseren Verständnis der Fachlichkeit bei allen Beteiligten führt, ist ein enormer Gewinn: Wer schreibt schließlich bessere Software als der, der nicht nur technisch, sondern auch fachlich weiß, was er tut?

Golo Roden ist Gründer und CTO der the native web GmbH, ein auf native Webtechnologien spezialisiertes Unternehmen. Für die Entwicklung moderner Webanwendungen bevorzugt er JavaScript und Node.js und hat mit „Node.js & Co.“ das erste deutschsprachige Buch zu diesem Thema geschrieben. Darüber hinaus ist er journalistisch für Fachmagazine und als Referent und Contentmanager für Konferenzen im In- und Ausland tätig. Für sein qualitativ hochwertiges Engagement in der Community wurde Golo dreifach als Microsoft MVP ausgezeichnet.

www.thenativeweb.io



Reactive Domain-Driven Design with Message Streaming

Vaughn Vernon *(Kalele/VLINGO)*

Everyone is talking about Event-Driven Architectures and Streaming Data. How many are actually using these successfully? If you haven't started yet, how can you leverage the power of this approach? Even if you have already achieved a measure of success, what can you do to ensure business viability and long-term flexibility in design that responds well to the inevitable on-going change? Also, why stop at Event-Driven when your entire system can be Reactive through Message-Driven architectures, both inside single services and across many collaborating services? Learn about DDD Context Mapping with Open Host Service and Published Language, and how to integrate using Reactive implementations that transform streaming to well-designed solutions. Employing the right tools will help a lot.

Links & Literatur

[1] http://todomvc.com

[2] http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html

Strategisches Mapping mit Wardley Maps

Lagebewusstsein schaffen!

Nur wenn wir die Landschaft um uns herum kennen, können wir unsere Route durch unwegsames Gelände sinnvoll planen. Landkarten (Maps) helfen uns dabei, Hindernisse gezielt zu umgehen, auf dem Weg befindliche Chancen zu erkennen und günstige Ausgangspositionen für kommende Etappen zu finden. Zeichnen wir also eine Karte!

von Tom Asel

Strategische Entscheidungen sollten nicht aus dem Bauch heraus getroffen werden, so viel ist klar. Wir benötigen verlässliche Informationen, anhand derer wir unsere Strategie planen können. Aber wie können wir diese Informationen sinnvoll so zusammenstellen, dass daraus ein realistisches Bild entsteht? Wardley Maps sind eine Methode, um bestehende Annahmen visuell so anzuordnen, dass sich ein Bild ergibt, das Landkarten nicht unähnlich ist. Bereits die Erstellung einer Map, also das Anordnen von zueinander in Bezug stehenden Komponenten, kann zur Gewinnung neuer Erkenntnisse beitragen. Die Methode visualisiert bestehende Annahmen und macht sie damit greifbar für Diskussionen. Durch die Überprüfung (impliziter) Annahmen kann ein gemeinsames Verständnis entwickelt werden. Maps eignen sich als Kommunikationsmittel zur Darstellung komplexer Sachverhalte sehr gut und bilden damit eine Basis zur eigentlichen Strategieentwicklung.

Was ist eine Wardley Map?

Wir betrachten ein beispielhaftes Szenario, das sich an einen realen Anwendungsfall anlehnt. In diesem Szenario bietet ein Unternehmen seinen Kunden eine Übersicht über Verträge und Leistungen über ein webbasiertes Kundenportal an. Das Kundenportal greift auf ein cloudbasiertes CRM-System und auf ein internes System zur Vertragsführung zu. Letzteres ist an eine Standardlösung von SAP angebunden, um Ein- und Auszahlungsvorgänge abzuwickeln. Kundenportal, Vertragsführung und SAP werden inhouse im eigenen Rechenzentrum betrieben, das CRM wird als Software-as-a-Service-Angebot genutzt. Zusätzlich entwickelt das Unternehmen eine Kunden-App, die über ein Backend an das Kundenportal angebunden ist. Die App soll dessen Funktionen auch zur mobilen Nutzung anbieten. Die Entwicklung von App und Backend hat gerade erst begonnen.

Abbildung 1 zeigt eine Wardley Map unseres Szenarios, bestehend aus einem Bezugspunkt (Anchor) und miteinander verbundenen bedeutsamen Dingen (Components). Der Bezugspunkt ist der Nordpfeil unserer Landkarte, Positionen werden relativ zu diesem Bezugspunkt in der Karte vermerkt. Auf unserer Karte ist der Kunde der Bezugspunkt, alle Components werden also bezogen auf den Kunden ausgerichtet. Direkt verknüpft mit dem Bezugspunkt sind Components, die Bedarfe (Needs) repräsentieren. In unserem Fall drückt die Map aus, dass der Kunde mit den Components „App“ und „Kundenportal“ interagiert.

Components repräsentieren alle Dinge von Bedeutung im Kontext unserer strategischen Betrachtung. Es handelt sich um Abstraktionen, maximale Vereinfachungen komplexer Zusammenhänge. Softwaresysteme können ebenso in einer Map als Component vorkommen wie Dienstleistungen, Wissen oder physikalische Kräfte. Daumenregel: Ist es strategisch bedeutsam, wird es als Component Teil der Map.

Die Ausrichtung der Components erfolgt entlang der vertikalen Achse (Value), abhängig von der Sichtbarkeit des Werts für den Kunden (unser Bezugspunkt, Anchor). Components, mit denen der Kunde direkt interagiert, sind für ihn auch direkt wahrnehmbar und werden daher weiter oben angeordnet. Weiter unten stehende Components sind, relativ zum Bezugspunkt Kunde, weniger sichtbar – aber nicht weniger wichtig. Der Wert einer Component und seine vom Bezugspunkt aus wahrnehmbare Sichtbarkeit sind zwei voneinander zu unterscheidende Merkmale. Durch die Verbindungen zwischen den Components ergibt sich eine Wertkette (Value Chain), bei der Components aufeinander aufbauen.

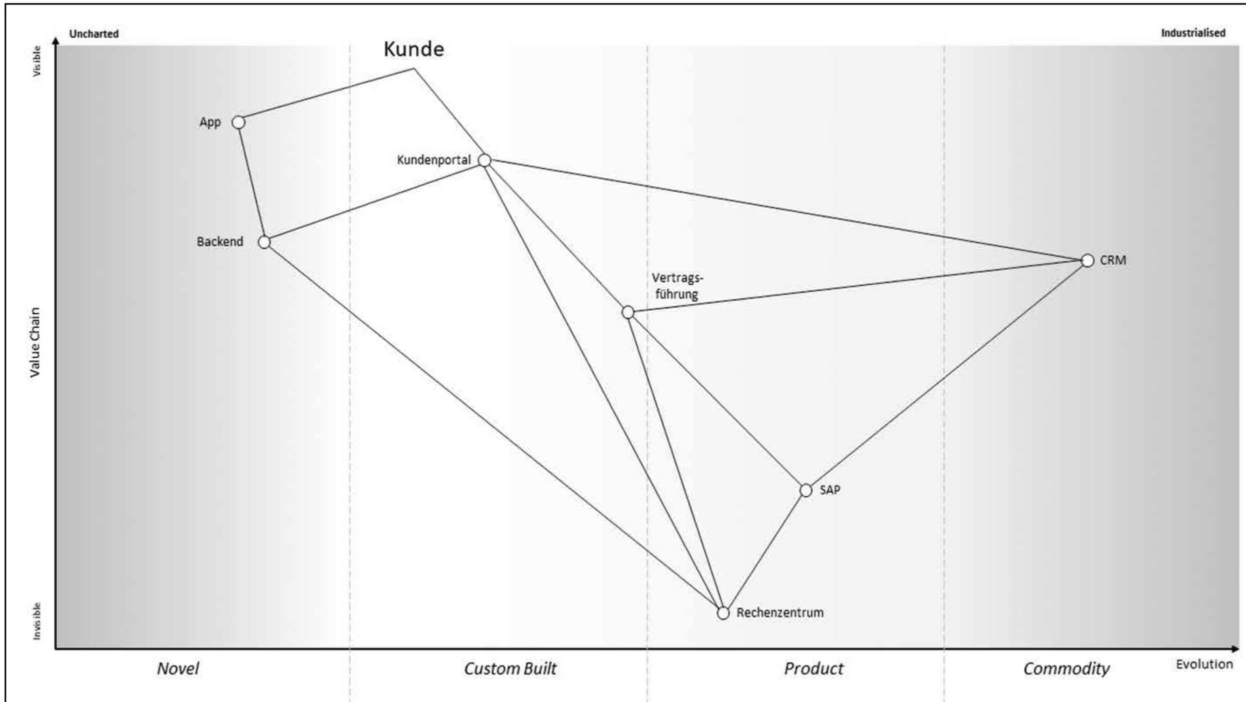


Abb. 1: Eine Landkarte des Beispielszenarios

Je weiter oben sich eine Component befindet, umso erkennbarer ist ihr Wert für den Kunden, je weiter rechts sie sich befindet, desto höher und reifer ist sie entwickelt. Die horizontale Achse beschreibt den Grad der Entwicklung (Evolution). Dieser sagt etwas über den Charakter der jeweiligen Component aus: Stellen wir ein Softwaresystem als Component am linken Rand der Map dar, sagen wir damit aus, dass das System sich am Anfang seiner evolutionären Entwicklung befindet. Gleiches gilt für Components, die Wissen oder Methoden repräsentieren. Steht eine Component weit links, kann das beispielsweise bedeuten, dass Technologien oder Konzepte eingesetzt wurden, zu denen bisher im Unternehmen wenig Wissen verfügbar ist. Dadurch erhält die Component im betrachteten Kontext experimentellen Charakter. Das birgt Chancen für neuartige Ideen und somit künftiges Potenzial sowohl für Wettbewerbsvorteile als auch für Risiken. Am linken Rand der Evolutionsachse sind die Unwägbarkeiten hoch. Je weiter eine Component sich entlang der Achse bewegt, umso reifer wird sie, d. h. umso geringer die Unwägbarkeit, und umso besser lassen sich Chancen und Risiken erkennen und bewerten. Die Einteilung der Achse in vier Segmente verdeutlicht das: Im ersten Segment (Genesis) findet die „Werdung“ statt. Der Fokus liegt auf der Erforschung und Erkundung und der ursprünglichen Entstehung von etwas Neuem. Im zweiten Segment handelt es sich um spezifische, maßgeschneiderte (Custom-Built-)Lösungen. Im dritten Segment erhält eine Component hinsichtlich ihrer Entwicklung und Reife die Eigenschaften eines Produkts. Components im vierten Segment sind so weit entwickelt, dass sie allgegenwärtig und einfach verfügbar sind (Commodity + Utility), vergleichbar mit Elektrizität, Radioempfang oder, dank Cloud-Computing, Rechenleistung. Nicht alle Components erreichen auch die höchste Entwicklungsstufe, aber sie entwickeln sich entlang der Achse in diese Richtung – freilich in unterschiedlicher Geschwindigkeit.

Nun haben wir alle Bausteine kennengelernt, um eine Karte einer bestimmten Landschaft zu zeichnen: Wir haben einen Bezugspunkt (Anchor) und können strategisch relevante Elemente (Components) relativ dazu positionieren (Value). Entfernungen zwischen den Elementen sind für uns unterscheidbar und bedeutsam (Evolution). Diese Karte können wir anderen zeigen, um über die Landschaft zu diskutieren. Der beste Weg, eine Map zu erstellen, ist, dies direkt in einer Gruppe zu tun, gemeinsam mit den jeweiligen Experten. Dabei ergeben sich wertvolle Diskussionen: „Warum ist das so

Über die Methode

Namensgeber Simon Wardley war in seinen Funktionen als CEO von Fotango und später als Vice President Cloud bei Canonical tätig. In seinem Buch beschreibt er, wie die Unzufriedenheit mit gängigen Methoden (SWOT-Analysen, 2x2 Charts, …), unzuverlässige Ratschläge von Business Consultants und aus dem Bauch heraus getroffene Managemententscheidungen ihn zur Suche nach neuen Ansätzen zur strategischen Planung inspirierten. Die von ihm als „Topographical Intelligence in Business Strategy“ bezeichnete Methode hat sich in vielen verschiedenen Anwendungsbereichen, auch unterhalb der CEO-Ebene, unter dem Namen Wardley Mapping etabliert.

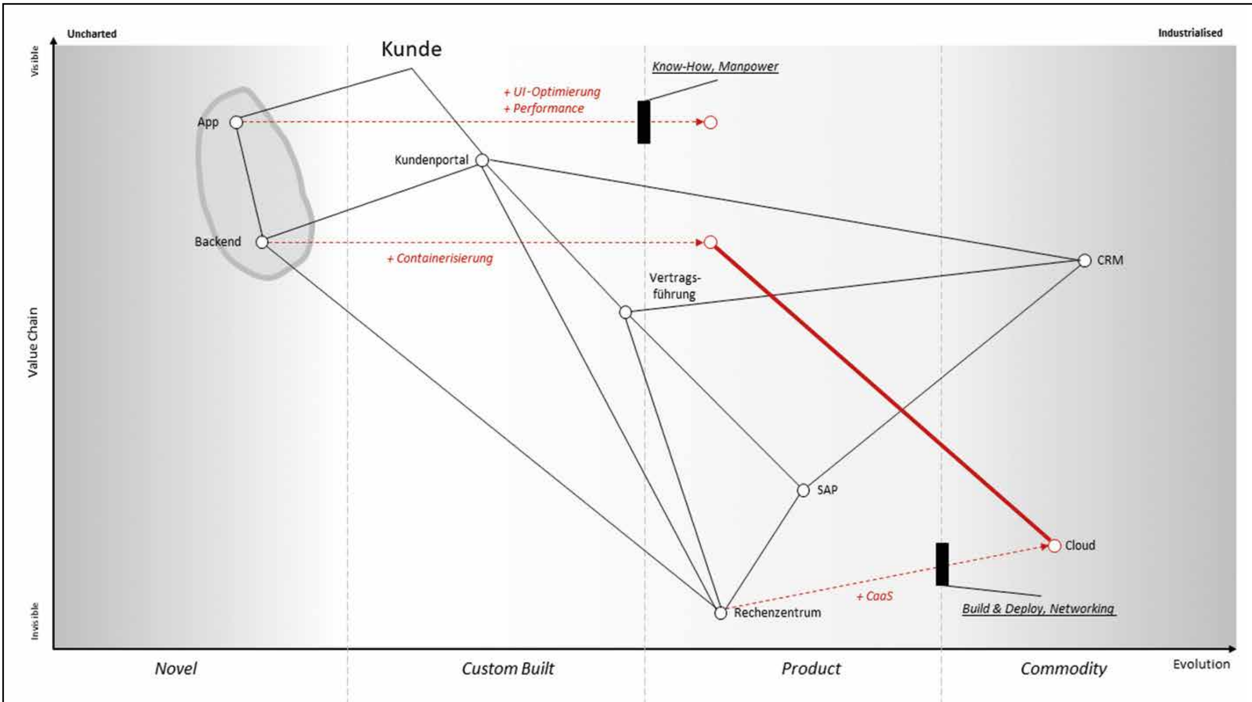


Abb. 2: Wardley Maps mit klimatischen Einflüssen

weit unten?", „Warum wird X weiter als Y entwickelt dargestellt?", „Ist A wirklich Teil der Landschaft? Warum ist B nicht abgebildet? Wo ist C?". Durch kritische Auseinandersetzung werden Annahmen hinterfragt und Erkenntnisse geteilt – ein gemeinsames Verständnis wird aufgebaut und weiter gefördert. Wer dabei an Domain-driven Design (DDD) und die Ubiquitous Language denkt, liegt genau richtig. Sobald wir ein gemeinsames Verständnis der Landschaft haben, können wir uns den Umgebungsbedingungen (Climate) widmen und diese ebenfalls in unsere Map übertragen.

Klimatische Einflüsse

Bodenwetterkarten enthalten neben der geografischen Abbildung der Landschaft weitere Informationen, etwa über Hoch- und Tiefdruckgebiete, Windverhältnisse, Niederschläge und Temperaturen. Die klimatischen Verhältnisse wirken auf die Landschaft ein und beeinflussen die geltenden Bedingungen. Je mehr relevante Details unsere Karte enthält, umso besser erkennen wir, dass der beste Weg von A nach B nicht zwangsläufig eine Gerade ist. Ein steiler Aufstieg an einer auf der Karte verzeichneten Felswand (Landscape) ist, je nach klimatischen Verhältnissen, anstrengend aber machbar, oder eben ein lebensgefährliches Wagnis. Dieser Informationsvorsprung ist entscheidend und stellt laut Simon Wardley den Unterscheid zwischen Maps und klassischen Methoden der Strategieplanung (Kasten: „Über die Methode") dar.

Um eine erfolgreiche Strategie zu entwickeln, müssen wir uns explizit und ganz bewusst mit den Dingen beschäftigen, auf die wir keinen Einfluss haben. Während wir uns die Gegebenheiten der Landschaft zunutze machen und sie auch ein Stück weit verändern können, entziehen sich klimatische Einflüsse unserem Willen: Wir können sie nicht steuern, aber wir können sie entdecken und auch diese Erkenntnisse für uns vorteilhaft nutzen. Bewegungen an Märkten, strategische Entscheidungen von Wettbewerbern, technologischer oder gesellschaftlicher Wandel sind klimatische Einflüsse, die wir auf unseren Karten abbilden können und wollen.

In **Abbildung 2** werden klimatische Gegebenheiten genutzt, um eine Strategie zu entwerfen. App und zugehöriges Backend werden parallel vom gleichen Team entwickelt. Der gemeinsame Kontext wird durch die Umrandung sichtbar gemacht, da er für die strategische Entwicklung eine Rolle spielt. Beide Systeme sind noch in einem frühen experimentellen Status und nicht produktionsreif. Es müssen das UI optimiert und Performanceprobleme ausgeräumt werden, um eine positive Kundenerfahrung und somit Akzeptanz am Markt zu ermöglichen. Der rote Pfeil drückt die angestrebte Bewegung (Movement) entlang der Evolutionsachse aus. Die Software soll ihren experimentellen Charakter mit der Zeit verlieren und die Eigenschaften eines reifen Produkts erhalten. Die strategische Entscheidung wird zusammen mit der dafür zu nehmenden Hürde abgebildet: In unserem Fall mangelt es an Wissen und Kapazität, was als schwarzer Balken dargestellt wird. Der Balken beschreibt Inertia, die zu überwindende „Massenträgheit". Nur wenn diese Hürde überwunden werden kann, ist der Weg frei für die Evolution von einer Eigenentwicklung (Custom Built) zu einem reifen und konkurrenzfähigen System mit Produktcharakter.

Cloud-Computing ermöglicht nicht nur die Nutzung von SaaS, sondern auch alternative Betriebskonzepte für Eigenentwicklungen, was Einfluss auf die Bereitstellung

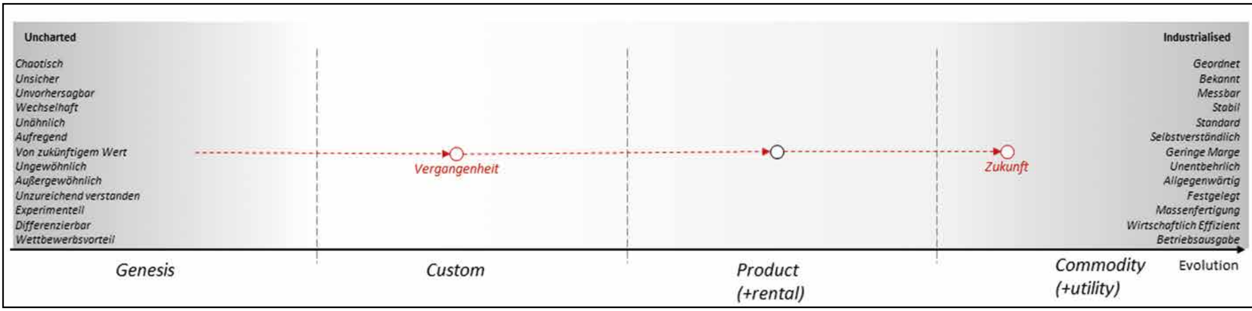


Abb. 3: Evolution von der Uncharted zur Industrialised Domain

hat. Der rote Pfeil beschreibt den Schritt weg vom rechenzentrumsbasierten Betrieb standardisierter VMs (Product) hin zu Container as a Service in einer Public Cloud (Commodity). Dieser Schritt ist jedoch nicht ohne Weiteres gangbar, denn auch hier existiert Inertia, also Widerstand durch eigene Trägheit. Das Know-how muss aufgebaut werden, Deployment und Betrieb erfordern neue infrastrukturelle Maßnahmen, die physische Anbindung zwischen Cloud und dem eigenen Haus muss ggf. neu dimensioniert werden. Sind diese Herausforderungen bewältigt, ergeben sich neue Möglichkeiten für andere Components auf der Map, sie können sich weiterentwickeln. Das Backend profitiert von der Umstellung auf containerbasierte Bereitstellung und kann cloudbasiert betrieben werden, sobald dieses Betriebsmodell zu einem späteren Zeitpunkt bereit ist. Die rote Linie beschreibt diese Verbindung der Components in der Zukunft. Dazu lohnt es sich, das Konzept der Evolution noch einmal genauer zu betrachten.

Evolution

Die Evolution beginnt mit dem Auftreten einer Component am linken Rand, der sogenannten Uncharted Domain (**Abb. 3**). Ihre Charakteristiken sind geprägt von Begriffen wie „experimentell“, „chaotisch“, „unzureichend verstanden“. Wir können schwer oder gar nicht vorhersehen, wie sich Dinge in der Uncharted Domain entwickeln werden. Daraus ergibt sich zum einen ihr hohes Potenzial in der Zukunft, aber auch ihr hohes Risiko durch eben diese Unsicherheit.

Je weiter sich eine Component entwickelt, umso mehr weicht die Unsicherheit der Gewissheit und es wird möglich, im industriellen Maßstab zu denken. Aus den ersten Laborversuchen mit transistorbasierten CPUs werden spezialisierte Kleinserien gefertigt, die später Produktreife erlangen und schließlich kostengünstig in Massenfertigungsprozessen in ausreichender Stückzahl produziert und für jeden erhältlich angeboten werden können. Je weiter wir uns auf der Map an den rechten Rand bewegen, umso mehr gelangen wir in die Industrialised Domain. Die Differenzierbarkeit zwischen verschiedenen Produkten nimmt ab, dafür sind diese wohl verstanden und lassen sich effizient her- und bereitstellen. Die Verfügbarkeit steigt rapide an, die Kosten pro Stück fallen. Je weiter entwickelt eine Component ist, umso eher kann sie die Ausgangsbasis für neue Entwicklungen sein, die dann ihrerseits wieder in der Uncharted Domain einsetzen. Aus kostengünstig in hoher Stückzahl gefertigten CPUs wurden Serverfarmen und wurde schließlich Cloud-Computing, bei dem die Rechenleistung (nicht die Hardware) allgegenwärtig verfügbar ist. In unserem Beispiel ist die Evolution des Betriebsmodells vom eigenen Inhouserechenzentrum hin zu cloudbasierten Containerdiensten die Basis für die technologische Weiterentwicklung des Backends. Das App-Experiment entwickelt sich zu einem reifen Produkt, je weiter es in die Industrialised Domain vorstößt.

Strategische Entscheidungen treffen

Bislang wurden die rein visuellen Eigenschaften von Wardley Maps besprochen. Mit Kenntnis der Landschaft und der Kräfte, die auf diese einwirken, können wir uns nun den strategischen Entscheidungsprozessen widmen. Kennen wir die Landschaft und die klimatischen Einflüsse, können wir individuelle Handlungen ableiten und universelle Prinzipien, die in jedem Kontext zutreffen, anwenden. Muster helfen uns dabei, diese zu erkennen und anzuwenden. Simon Wardley beschreibt dies mit den fortgeschrittenen Konzepten Doctrine und Leadership und gibt dem interessierten Leser Patternkataloge und Strategeme an die Hand, um eigene Strategi-

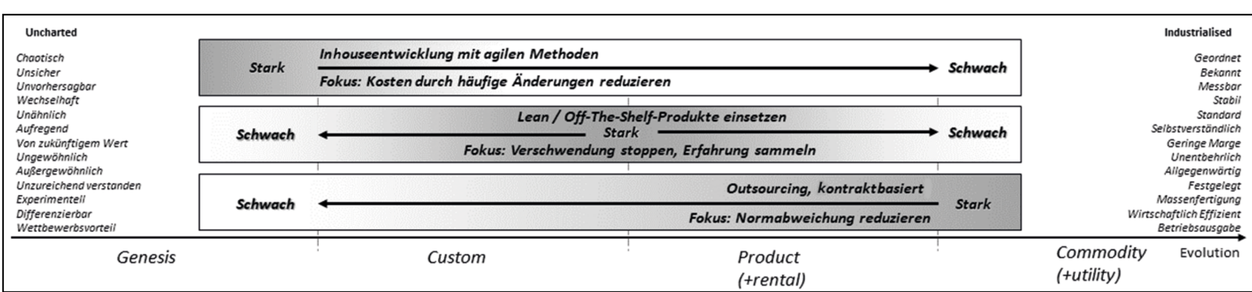


Abb. 4: Verschiedene Methoden eignen sich je nach Evolutionsgrad besser

en basierend auf Maps effektiver entwickeln zu können. Diese Konzepte sprengen den Rahmen dieses Artikels, sollen aber nicht gänzlich unerwähnt bleiben. Zusammen mit dem Konzept des Strategy Cycles bilden sie den wahren Kern des strategischen Mappings.

Um das zu verdeutlichen, wenden wir das gesammelte Wissen an und greifen das Pattern „No Single Method fits all“ heraus. Es besagt, dass es keine Methode gibt, die sich sinnvoll universell auf alle Components einer Map anwenden lässt. Beziehen wir das auf die unterschiedlichen Ansätze zur Softwareentwicklung und übertragen sie in das Konzept von Uncharted und Industrialised Domain, ergibt sich das Bild aus **Abbildung 4**.

Sowohl App als auch Backend befinden sich in der Uncharted Domain. Das bedeutet, dass mit häufigen Änderungen zu rechnen ist, sozusagen „by Design“. Dies ist eine wesentliche Eigenschaft von Components in der Genesis-Phase der Evolution und somit der Uncharted Domain. In diesem Fall bietet es sich an, mit agilen Methoden auf den steten Wandel zu reagieren, um die dadurch anfallenden Kosten unter Kontrolle zu behalten. Elemente am rechten Rand der Skala unterliegen diesem Wandel weniger stark. Ihre Charakteristiken sind wohl bekannt und somit kalkulierbar. Für Eigenentwicklungen, die es bis zu diesem Stadium schaffen, könnte (nicht muss!) Outsourcing eine Option sein. Das CRM ist beispielsweise von Anfang an als SaaS im Einsatz und keine individuelle Eigenentwicklung.

Fazit, Anwendungsmöglichkeiten und Ausblick

Wir haben die auffälligsten Merkmale von Wardley Maps angesprochen und anhand einfacher Beispiele verdeutlicht. Für fortgeschrittene Themen wie Doctrine, Gameplay und die Patternkataloge wird auf das frei verfügbare, unter CC-BY-SA lizensierte Werk von Simon Wardley verwiesen, da dieser Artikel dafür nicht ausreichend Raum bieten kann [1], [2].

Die Entwicklung einer Map im Team kann im Rahmen eines Workshops stattfinden. Das Ergebnis kann leicht eine ganze Wand füllen. Entscheidend ist hier der nächste Schritt: Die Reduktion auf das Wesentliche, z. B. durch Aufspaltung einer Map in mehrere. Auf diesem Weg werden immer wieder neue Erkenntnisse über bestehende Annahmen gesammelt (Kasten: „Anmerkungen zur Arbeit mit Wardley Maps“).

Im Kontext von DDD lassen sich Wardley Maps nutzen, um strategische Entscheidungen über Bounded Contexts zu treffen. Welche Veränderungen zeichnen sich ab? Welches Team und welche Methode eignen sich für welche Region der Karte? In welchen Bounded Context (nicht in welches System!) lohnen sich Investitionen, um neue Chancen nutzbar zu machen? Wardley Maps sind ein weiteres Werkzeug in unserer Designtoolbox, auf das wir im Bedarfsfall zugreifen können.

Auch wenn Wardley Maps ursprünglich für strategische Entscheidungen auf CEO-Level entwickelt wurden, können wir sie überall dort anwenden, wo wir uns ein Bild von komplexen Situationen verschaffen wollen, um fundierte Entscheidungen zu treffen. Das gilt für das strategische Architekturmanagement des Enterprise-Architekten ebenso wie für Projektleiter und System Owner, die eine vorausschauende (eben strategische) Planung betreiben wollen.

Abschließend lässt sich noch die Frage aufgreifen, ob es möglich ist, mit Wardley Maps komplexe Sachverhalte der realen Welt wirklich treffend darzustellen. Auch dazu gibt es ein mehr als treffendes Zitat von Simon Wardley: „All models are wrong but some are useful.“

Anmerkungen zur Arbeit mit Wardley Maps

- Bedürfnisse der Nutzer (Needs) stehen über allem. Nutzer sind Kunden, Stakeholder, …
- Akzeptiere Unsicherheit. Nutze die Map, um sie kenntlich zu machen.
- Annahmen müssen klar kommuniziert werden.
- Sämtliche Konzeption muss auf ständige Evolution ausgelegt sein.
- Gehe iterativ vor, wende Erlerntes an.

Tom Asel ist freiberuflicher Softwarearchitekt und Trainer. Als Enterprise-Architekt und Software Engineer beschäftigt er sich mit technischen Konzepten vom frühen Entwurf bis tief in die Implementierung hinein.

Links & Literatur

[1] https://medium.com/wardleymaps

[2] https://learnwardleymapping.com

Wardley Maps in der Softwareentwicklung: Risiken entdecken und Strategien entwickeln

Tom Asel *(Freiberufler)*

Langlebige Entscheidungen werden in der Softwareentwicklung auf allen Ebenen getroffen – nicht nur vom Management: Make or buy? Spring oder Micronaut? Java oder Kotlin? Solide Lösung oder Workaround? Wäre es da nicht hilfreich, ein leichtgewichtiges Werkzeug zu haben, das uns bei der Entscheidungsfindung unterstützt? Wardley Maps sind genau das: ein niederschwelliges, visuelles Werkzeug, um strategische Planung und Kommunikation von Annahmen, Zielen und Risiken zu vereinfachen. In der Session wird gezeigt, wozu strategische Karten in der Softwareentwicklung angewendet werden können, um Risiken zu identifizieren und Strategien zu kommunizieren.