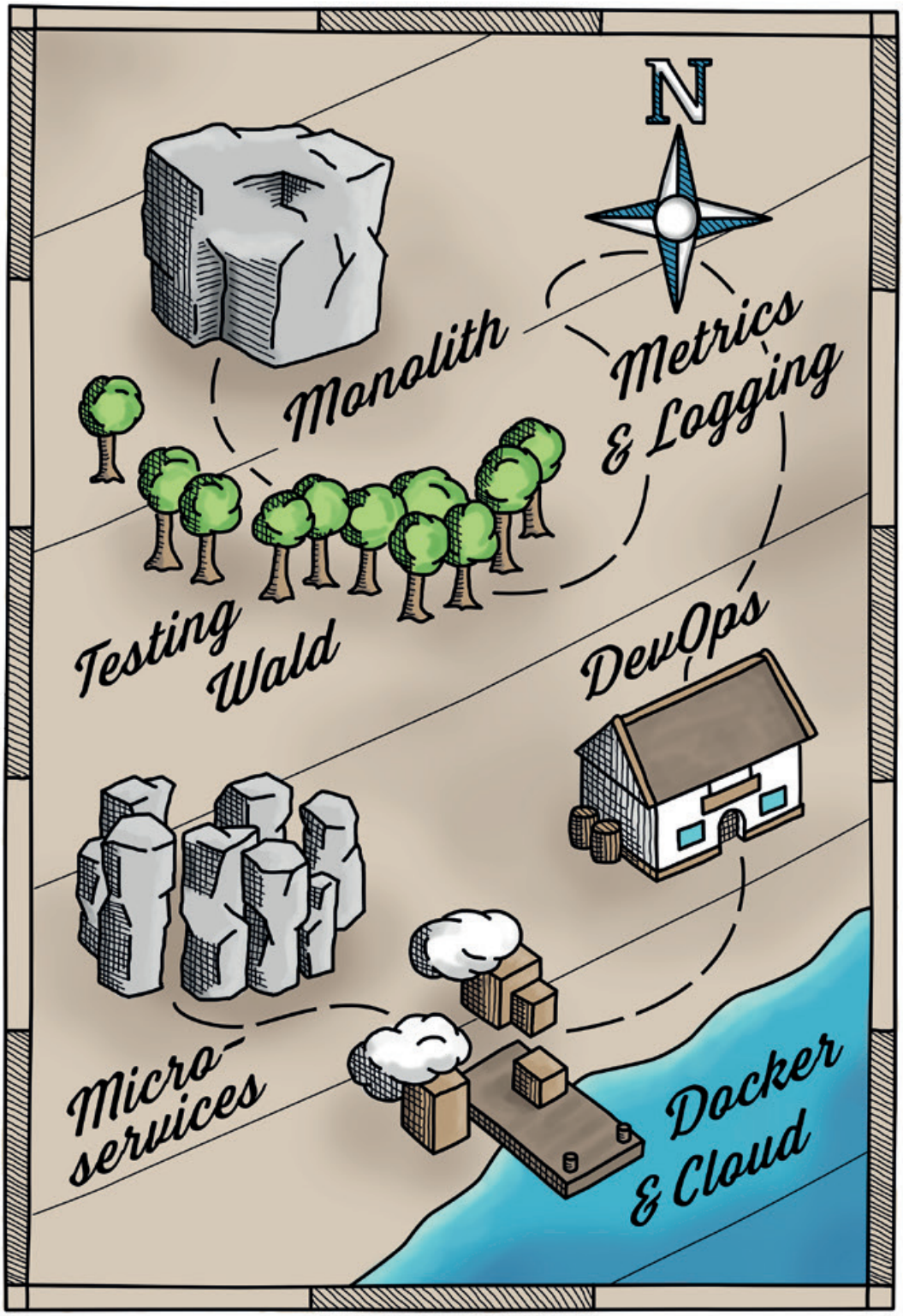
Vom Monolithen



zu Microservices

Vom Monolithen zu Microservices – ein Erfahrungsbericht

Andreas Weigel, synyx GmbH & Co. KG, und Jakob Fels, dm-drogerie markt GmbH + Co. KG

„Microservices" steht für ein Architektur-Muster, das den Fokus auf eine fachliche Modularisierung des gesamten Software-Stacks legt. Gestützt durch technischen Fortschritt und organisatorische Entwicklung, lassen sich damit besser nachvollziehbare Architekturen umsetzen, die viele Vorteile mit sich bringen.

Die Komplexität löst sich damit nicht auf, wird jedoch leichter handhabbar. Zudem verschiebt sie sich teilweise in andere Bereiche, zum Beispiel in der Infrastruktur bei der Bereitstellung einer Orchestrierung von Containern oder in der Organisation bei dem richtigen Schnitt der Domäne und den zuständigen Entwicklungsteams. Vorteile können dabei unter anderem Verständnis der Domäne, Geschwindigkeit im Release-Zyklus oder Skalierung der Software beziehungsweise der Entwicklungsteams sein.

Dieser Artikel zeigt, wie im Rahmen der Software-Entwicklung der Coupon-Verwaltung bei dm-drogerie markt eine monolithische Anwendung in eine Microservice-Architektur transformiert wurde, wobei zunächst gezielt mit einem strukturierten Monolithen angefangen wurde.

Neue Möglichkeiten durch Umdenken und Innovation

Herrscht mangelnde Kommunikation zwischen dem Entwicklungsteam und dem Fachbereich, beschränkt sich die Kommunikation weitestgehend auf den Austausch von Last- und Pflichtenheft. Somit ist es unumgänglich, dass der Funktionsumfang vom Fachbereich im Vorfeld vollständig definiert wird, das Entwicklungsteam darauf aufbauend die Hardware-Dimensionierung einplant und die Software-Entwicklung durchführt. Je nach Dauer der Entwicklung und mangels Feedback des Endanwenders ist schließlich unklar, ob das Ergebnis alle Bedürfnisse erfüllt.

Das Resultat ist ein Software-Entwicklungsprozess nach dem Wasserfallmodell. Parallele Entwicklung in der gleichen Codebasis führt zu Abstimmungsaufwänden und aufgeschobenen und damit seltenen Releases. Die Folge ist meist eine monolithische Anwendung mit einem komplexen und aufwendigen Software-Stack für den Betrieb, der nur schwierig zu skalieren ist.

Eine Reaktion darauf in der Software-Entwicklung ist die Realisierung von horizontalen Schnitten (Präsentations-, Logik-, Datenhaltungs-Schicht) zur Definition von Abhängigkeitsbeziehungen und Abstraktion von Komponenten. Um in diesem Rahmen Synergie-Effekte zu erzielen, entstand das Architekturmuster „Service-orientierte Architektur" (SOA), das sich auf die Orchestrierung von Services innerhalb der Logikschicht konzentriert. Das Ergebnis ist eine stetig wachsende Anwendung, die nur im Ganzen deployt werden kann, mit zunehmendem Funktionsumfang schwer überschaubar ist und hohes Potenzial für Seiteneffekte zwischen unabhängigen Domänen aufweist. Wachsende funktionale Anforderungen können nur langsam bedient werden, da viel Zeit in die Wartung und den Betrieb der Anwendung fließt.

Um die Time-to-Market zu minimieren, sind drei wesentliche Punkte zu adressieren. Zum Ersten sind ein Umdenken in der Art und Weise der Kommunikation und das gegenseitige Verständnis zwischen allen Projektbeteiligten zwingend notwendig, unabhängig von ihrer Rolle. Zum Zweiten muss die Domäne verstanden und der gesamte Software-Stack entsprechend geschnitten sein. Damit die aufgeteilten Services performant ausgerollt werden können, braucht es zudem eine automatisierte Provisionierung des Infrastruktur-Stacks.

Ein agiler Software-Entwicklungsprozess fördert einen engeren Austausch zwischen allen Produktbeteiligten. Mit dem Fokus auf die Entwicklung eines Anwendungsfalls schaffen Fachverantwortliche, Produktverantwortliche und Entwickler ein gemeinsames Verständnis für die nächste zu entwickelnde Funktionalität mit dem höchsten Mehrwert für den Kunden. Dieser iterative Entwicklungszyklus bietet den Raum zur stetigen Ausrichtung der Produktvision und stellt sicher, dass die aktuell relevanten Bedürfnisse des Kunden befriedigt werden.

Einen Ansatz für den richtigen Schnitt bietet Eric Evans mit Domain Driven Design *[1]*. Darauf aufbauend existieren verschiedene Methodiken wie das Event-Storming, mit dem sich Domänengrenzen (Bounded Contexts) und zusammenhängende Domänen (Aggregates) eines Anwendungsfalls identifizieren lassen. Die Konzentration auf abgegrenzte Anwendungsfälle erleichtert das Verständnis sowie die Kommunikation bei der Lösungsfindung. Schließlich muss die Aufteilung in Services umgesetzt werden, um von unabhängigen Deployments und flexibler Skalierung zu profitieren. Der technologische Rahmen ist damit neuen Anforderungen ausgesetzt und muss neu gedacht werden.

Die Entwicklung von kleinen Services bedeutet, dass mehr Infrastruktur notwendig ist und diese schnell provisioniert werden muss. Auf der einen Seite kann man für diesen Zweck diverse Cloud-Dienste verwenden, auf der anderen Seite existieren die notwendigen Werkzeuge Open-Source-basiert und als On-Premise-Lösungen. Zum Beispiel kann mit Spring Boot *[2]* eine vollständige und lauf-

fähige Anwendung erstellt werden, die alle notwendigen Abhängigkeiten mit sich bringt.

Mit systemD *[3]* vereinfacht sich die Steuerung einer solchen Anwendung. Container wiederum ermöglichen eine schlanke Isolierung der Betriebssystem-Virtualisierung (etwa mit Docker *[4]*). Schließlich gibt es auch für die Orchestrierung der Container verschiedene Alternativen (wie Kubernetes *[5]* oder Docker Swarm *[6]*). Diese Werkzeuge folgen dem „Everything as Code"- Ansatz und bieten eine Beschreibungssprache zur Konfiguration der Laufzeitumgebung. Die Infrastruktur ist damit versionierbar und reproduzierbar. Es muss zusätzlich die Abwägung getroffen werden, ob die Anwendungskonfiguration mit dem Deployment ausgeliefert wird und damit jederzeit nachvollziehbar, aber starr ist, oder ob ein Konfigurationsserver als weitere Abhängigkeit eine zur Laufzeit flexible Konfiguration erlaubt.

Die Komplexität liegt schließlich im Aufbau der Orchestrierung, jedoch bietet eine solche Infrastruktur die Möglichkeit einer Continuous Deployment Pipeline; also einen automatisierten Prozess, um die Anwendung in Produktion zu nehmen und damit die Time-to-Market zu senken.

Vom Monolithen zu Microservices bei dm-drogerie markt

Um sowohl für die wachsenden fachlichen als auch die technischen Anforderungen vorbereitet zu sein, ist dm-drogerie markt mit der Unterstützung von synyx den Pfad dieser Transformation von einem monolithischen hin zu einem auf Microservices basierenden System gegangen.

Die steigende Zahl der Marketing-Kampagnen bei dm-drogerie markt führt zu einem stetig wachsenden Funktionsumfang im Backend des Coupon-Verwaltungssystems. Das alte System wurde direkt in der bestehenden Integrations-Plattform, basierend auf starren proprietären Komponenten, entwickelt und unterlag den klassischen, oben beschriebenen Strukturen. Das Resultat nach nur wenigen Jahren war ein Zustand, der Erweiterungen kaum zuließ und hohen Aufwand für die Wartung erforderte. Zu diesem Zeitpunkt wurde mit der Entwicklung des Online-Shops begonnen, wobei erkannt wurde, dass eine Anbindung der beiden Systeme nicht ohne Weiteres möglich war. Eine funktionierende Interaktion der Systeme setzte die Ablösung der bestehenden Coupon-Verwaltung durch eine Neuentwicklung voraus.

Der Weg zum Monolithen

Wenn ein Unternehmen einen Größenschwellwert überschreitet, wird versucht, Synergie-Effekte zu erzielen. Analog zum Vorgehen in der Software-Entwicklung werden horizontale Schichten eingeführt und beispielsweise technische Teams wie ein Betriebsteam bereitgestellt, das generische und umfassende Lösungen für die Entwicklerteams anbietet. Aufgrund der hohen Priorität des Online-Shops war das Angebot des Betriebsteams auf dessen technische Bedürfnisse ausgerichtet. Diese Rahmenbedingungen bieten viele Funktionen zum Projektstart, stellen aber gleichzeitig ein Korsett dar, das nur mit großem Kommunikations- und Arbeitsaufwand anpassbar ist.

Es wurde ein Software-Stack angeboten, der viele Funktionen umfasste, wie Active- und Standby-Software-Load-Balancer, Proxy inklusive Varnish und HTTP-Server sowie Web-Container *(siehe Abbildung 1)*. Die Komponenten liegen aufgrund von Ausfallsicherheit jeweils in zweifacher Dimensionierung vor. Außerdem wurden ein Deployment sowie ein Anwendungsmonitoring angeboten. Da zusätzlich zu dieser Ausgangssituation die funktionalen sowie nicht-funktionalen Anforderungen noch nicht feststanden, wurde ein iterativer Entwicklungsprozess gewählt und mit einem fachlich-strukturierten Monolithen angefangen. Darüber hinaus einigte sich das Team – zum einen aus Know-how-Gründen, zum anderen aufgrund der Funktionalität – auf das Spring-Framework und insbesondere Spring Boot.

Als zentrales Backend-System hat die Coupon-Verwaltung zur Aufgabe, viele Schnittstellen anzubieten, die von unterschiedlichen Systemen konsumiert werden. Die Konsumenten spiegelten unterschiedliche Anwendungsfälle wider, benötigten individuelle Logik und Perspektiven auf die Daten. Aus diesem Grund war es intuitiv, die Software anhand der Konsumenten aufzuteilen („Consumer Driven Slices").

Als „Maven Multi Module"-Projekt wurde jedes Modul als eigenständige und lauffähige Anwendung entwickelt. Erst beim Deployment entsteht aus den Modulen ein Monolith. Mit Disziplin und Tool-Unterstützung (wie SonarQube *[7]*) wurde sichergestellt, dass keine zyklischen Abhängigkeiten entstehen. Aufgrund der gerin-

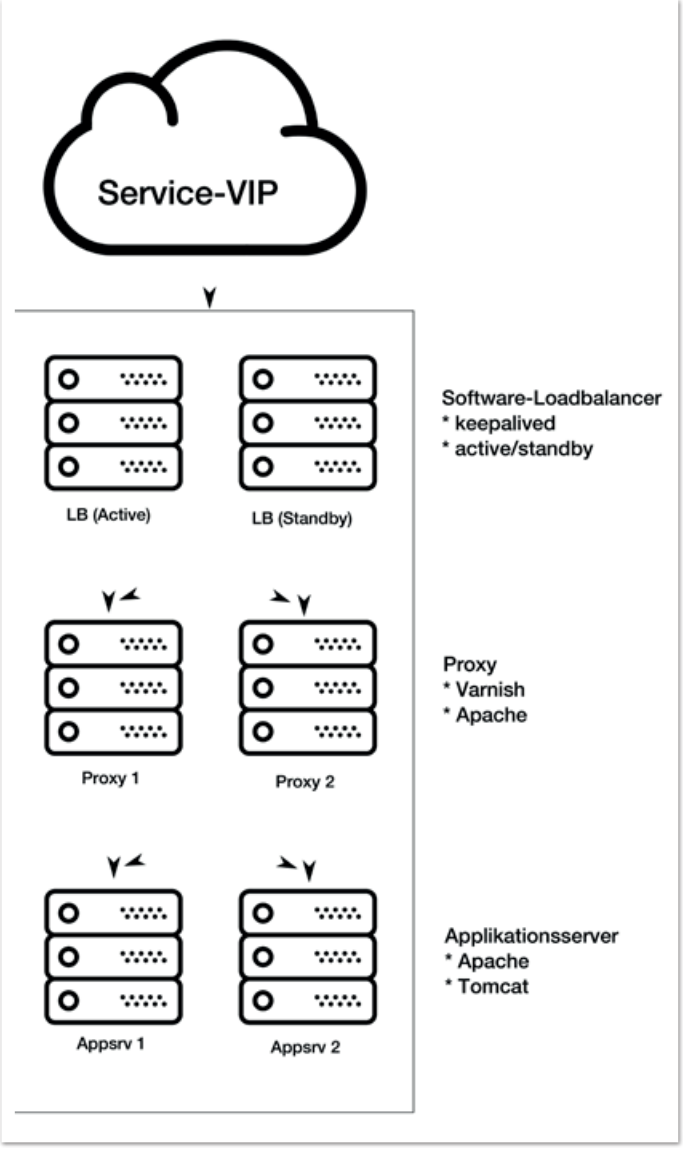


Abbildung 1: Ausgereifter und umfangreicher Software-Stack

gen Anforderungen an das Datenmodell sowie der zeitlich getrennten Schreibzugriffe wurde auf Datenbankebene integriert.

Microservice-Infrastruktur bei dm-drogerie markt

Zur gleichen Zeit wurde mit höchster Priorität der dm-Online-Shop entwickelt. Da das Bedürfnis der zuständigen Entwickler nach Rahmenbedingungen für Microservices zunahm, folgte ein Umdenken an mehreren Stellen. Das Betriebsteam ging mehr auf die Entwicklungsteams ein und reduzierte den Software-Stack für den Betrieb einer Anwendung. Er bestand lediglich aus einem HTTP-Server und einem Web-Container *(siehe Abbildung 2)*.

Für die ersten Microservices des Online-Shops wurde zunächst eine Basis-Infrastruktur bereitgestellt *(siehe Abbildung 3)*. Es kamen verschiedene Komponenten aus dem Netflix-OSS-Stack *[8]* zum Einsatz, wie zum Beispiel eine Service-Discovery (Eureka *[9]*), die eine einfache Skalierung ermöglicht. Zusätzlich wurden dedizierte Load Balancer durch Client Side Load Balancing (Ribbon *[10]*) ersetzt.

Alle Microservices stehen in einer DMZ und Zugriffe werden durch einen Proxy authentifiziert. Die Microservices selber übernehmen letztlich die Autorisierung. Der Proxy dient auch als API-Gateway (Zuul *[11]*) und vereinheitlicht als Reverse-Proxy die Schnittstellen zu den Microservices.

Mit der „dm glückskind"-App, die Coupons abfragen und aktivieren kann, entstand die Anforderung einer Funktion, die einen zentralen Sicherheitsmechanismus benötigte, um Kunden zu identifizieren. Da dieser Mechanismus in der Microservice-Infrastruktur schon bereitstand, war das der richtige Zeitpunkt, mit dem ersten Microservice auf die neue Infrastruktur zu migrieren. Das relevante Modul wurde aus dem Monolithen gelöst und als Microservice deployt.

BDD, Logging, Metrics und Tracing – Voraussetzung für Microservices

Das Ersetzen des Backend für die Coupon-Verwaltung birgt nicht nur technische Herausforderungen. Um den korrekten Betrieb zu gewährleisten, existierten in der Vergangenheit viele manuelle Tests. Es gibt zwei Gründe, aus denen der Fachbereich zu zeitintensiven, manuellen Tests greift, obwohl sie für das Ausrollen einer Funktion hinderlich sind. Entweder es ist technisch nicht möglich, einen automatisierten Test zu implementieren, weil Teilprozesse keine Schnittstelle haben, oder es fehlt an Vertrauen in das Entwicklungsteam. Auf beide Umstände muss das Entwicklungsteam von Projektbeginn an eingehen.

Wo es keine Schnittstellen gab, wurde Zeit in Kommunikation und Entwicklung investiert, um diese herzustellen. Außerdem wurde mit Development, Stage, Performance, Release und Production ein Fünf-Stages-Konzept etabliert, um dem Ziel, Continuous Deployment, näherzukommen.

Das Vertrauen wurde durch eine hohe Priorität auf vollständige Akzeptanz-Tests von Beginn an aufgebaut. Als Teil der Definition-of-Ready und damit noch vor dem Refinement werden in einem sogenannten „Drei-Amigo-Meeting" (Fachverantwortlicher, Produktverantwortlicher und Entwickler) alle Tests für die in der nächsten Story zu implementierende Funktion definiert. Diese Akzeptanz-Tests werden in der Beschreibungssprache gherkin *[12]* (Given - When - Then) für alle Projektbeteiligten verständlich erstellt und mit dem Werkzeug Cucumber *[13]* implementiert. Aus einem dedizierten Projekt heraus werden die unterschiedlichen Stages als Black-Box getestet. Somit konnten die Tests auch nach der Extraktion des Moduls aus dem Monolithen sicherstellen, dass die Funktionalität als Microservice weiterhin gewährleistet ist.

Um einer neuen Infrastruktur vertrauen zu können, muss man sehen und verstehen, wie sich die Anwendung im Live-Betrieb verhält. Dafür gibt es drei Mechanismen:

Mit Logging kann man ereignisbasierte Informationen sammeln. Diese Informationen werden mit den Werkzeugen ElasticSearch *[14]*, Logstash *[15]* und Kibana *[16]* zentral für alle Teams durchsuchbar bereitgestellt.

Metrics sind Laufzeit-Aggregate zur Wahrnehmung der Performance. Die grafische Oberfläche Grafana *[17]* stellt die Performance-Informationen überwiegend aus InfluxDB-Datenbanken *[18]* dar. Mittels Telegraf senden die Microservices dafür täglich etwa zehn Gigabyte an Daten.

Tracing sind auf Anfragen basierende Daten zur Korrelation von Laufzeit-Informationen. Eine Tracing-Infrastruktur mit OpenZipkin *[19]* ist gerade im Aufbau. Mit Spring-Cloud-Sleuth *[20]* kann anhand von Tracing-Ids und Span-Ids schon jetzt in den Logs-Anfragen korreliert werden.

Viele Hürden passiert, aber noch nicht am Ziel

Je nach Kultur, Code-Basis und Rahmenbedingungen kann eine solche Transformation Zeit brauchen. Es sind Experimente notwendig und ein guter Umgang mit Fehlentscheidungen. Als gute Entscheidung hat sich herausgestellt, den Abstand zwischen Betriebs- und Entwicklerteam zu minimieren und damit die DevOps-Kultur zu leben. Dediziert verstärken betriebsnahe Mitarbeiter die Entwicklungsteams für mehrere Tage die Woche und begleiten den Entwicklungsprozess.

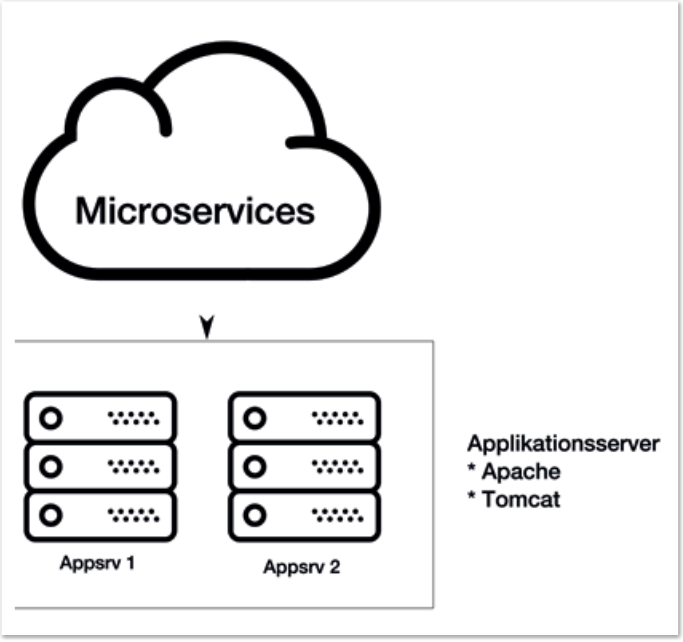


Abbildung 2: Schlanker Software-Stack, bestehend aus HTTP-Server und Web-Container

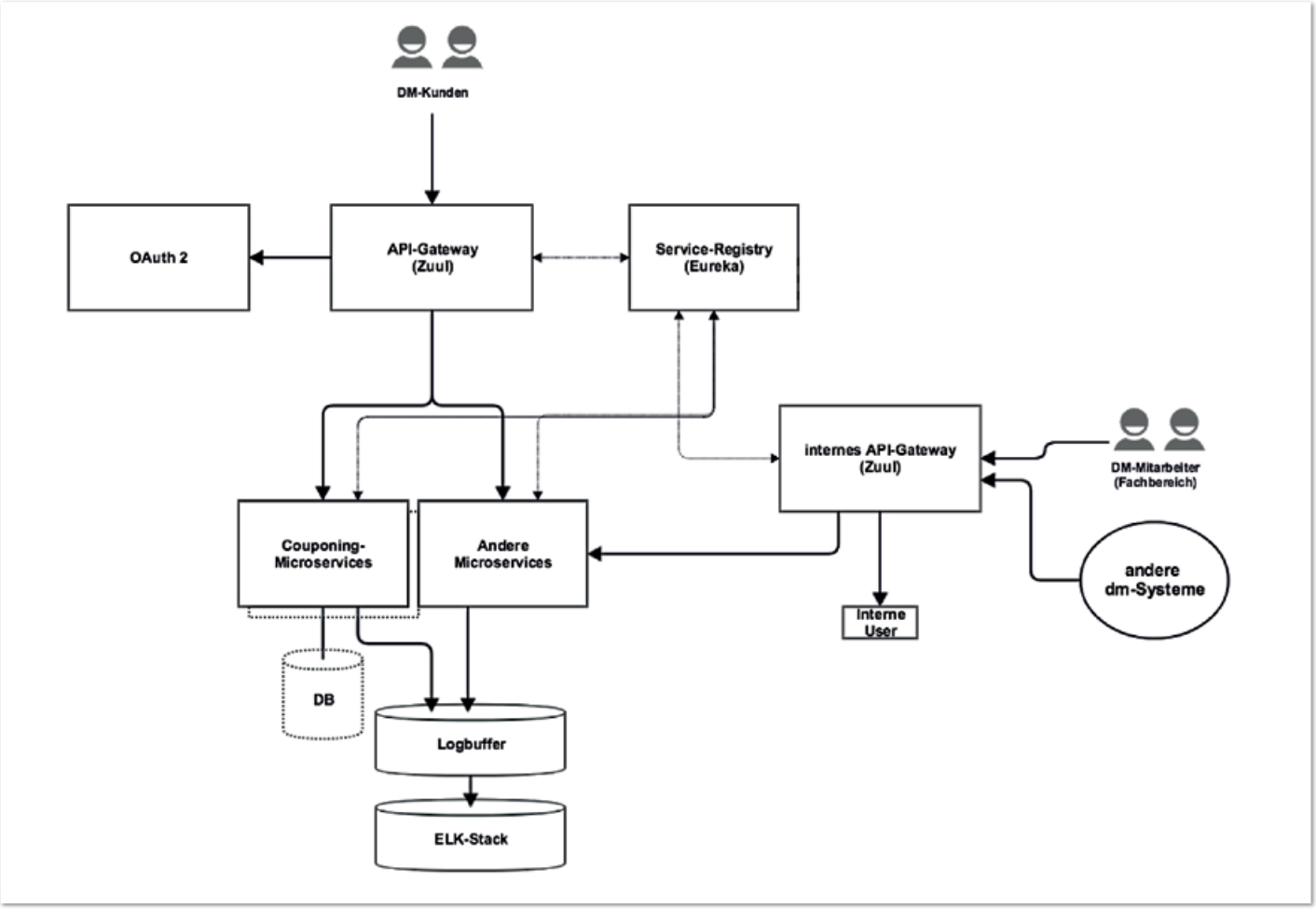


Abbildung 3: Schematischer Überblick über die Microservice-Infrastruktur

Es hat sich als keine so gute Entscheidung erwiesen, zentrale Komponenten zwischen Teams zu teilen, die bei Änderungen enormen Kommunikationsaufwand bedeuten. Die laufende Transformation zeigt allerdings jetzt schon deutliche Vorteile:

- Services können unabhängig voneinander deployt und skaliert werden
- Bessere Fokussierung auf Domäne und einfache Kommunikation
- Keine Seiteneffekte auf fremde Domänen durch Modularisierung
- Schnelle Releases durch Continuous Deployment Pipeline

Auch für dm-drogerie markt ist der Weg noch nicht zu Ende. Die automatisierte Provisionierung der Microservices – und damit eine Optimierung des Deployments mit Containern – sowie die Orchestrierung sind die nächsten spannenden Themen auf dem Pfad zu einer hoch flexiblen und automatisierten Infrastruktur.

Weiterführende Informationen

[1] Eric Evans, Domain-Driven-Design, Tackling Complexity in the Heart of Software, Addison-Wesley, 2003, ISBN 978-0-321-12521-7
[2] *https://projects.spring.io/spring-boot*
[3] *https://www.freedesktop.org/wiki/Software/systemd*
[4] *https://www.docker.com*
[5] *https://kubernetes.io*
[6] *https://github.com/docker/swarm*
[7] *https://www.sonarqube.org*
[8] *https://netflix.github.io*
[9] *https://github.com/Netflix/eureka*
[10] *https://github.com/Netflix/ribbon*
[11] *https://github.com/Netflix/zuul*
[12] *https://github.com/cucumber/cucumber/wiki/Gherkin*
[13] *https://cucumber.io*
[14] *https://www.elastic.co/de/products/elasticsearch*
[15] *https://www.elastic.co/de/products/logstash*
[16] *https://www.elastic.co/de/products/kibana*
[17] *https://grafana.com*
[18] *https://www.influxdata.com*
[19] *http://zipkin.io*
[20] *https://cloud.spring.io/spring-cloud-sleuth*

Andreas Weigel
weigel@synyx.de

Andreas Weigel ist Software-Entwickler und Berater bei synyx GmbH & Co. KG und unterstützt seit mehreren Jahren verschiedene Kunden bei der Entwicklung von Software-Lösungen im agilen Umfeld. Dabei beschäftigt er sich primär mit Architektur-Entscheidungen, der vollumfänglichen Automatisierung des Entwicklungsprozesses sowie der Umsetzung von individuellen Lösungen.