Microservices

Mehr als nur ein Hype?

Alexander Schwartz

Architekturparadigma der Internetriesen

Die Internetriesen Amazon, Netflix und Google haben gezeigt, wie Produkte erfolgreich auf den Markt gebracht werden. Die Unternehmen sind gewachsen, doch die anfängliche monolithische Softwarearchitektur zeigte diesem Wachstum seine Grenzen auf. Um die Softwareentwicklung der großen Systeme wieder beherrschbar zu machen, setzten die Konzerne auf kleine, unabhängige Teams. Jeff Bezos von Amazon verwendete dafür 2002 den Begriff „Two Pizza Teams“: Ein Team sollte nur so groß sein, dass es von zwei (amerikanischen) Pizzen satt werden würde – dies ist eine Teamgröße von 5–10 Personen. Jedes Team ist „cross-functional“ aufgestellt und bringt damit alle notwendigen Kenntnisse und Fähigkeiten mit, angefangen bei Fachlichkeit, UI/UX, Programmierung bis hin zu Test und Betrieb. Die Kommunikation findet hauptsächlich zwischen den Teammitgliedern statt, der Abstimmungsbedarf zwischen den Teams wird reduziert [1, S. 9off.]. Gleichzeitig bekommt jedes Team die Verantwortung, sein eigenes System zu betreiben. Werner Vogels von Amazon fasste dies mit den Worten „You build it, you run it“ zusammen [2].

Melvin Conway beschrieb, dass Organisationen Systeme entwickeln, die ihre eigene Struktur widerspiegeln („Conway's law“) [3]. Entsprechend der oben beschriebenen Organisationsstruktur entwickelte sich ein Architekturstil, der heute als „Microservices“ bezeichnet wird. Dabei ist ein Team für einen oder mehrere Microservices verantwortlich, jedoch nie mehrere Teams für einen Microservice.

Definition Microservices

Microservices sind ein Architekturstil, bei dem die Anwendung in kleine Teile aufgeteilt wird, die unabhängig voneinander entwickelt und installiert werden. Sie laufen als eigenständige Prozesse und können unabhängig voneinander skaliert werden [8].

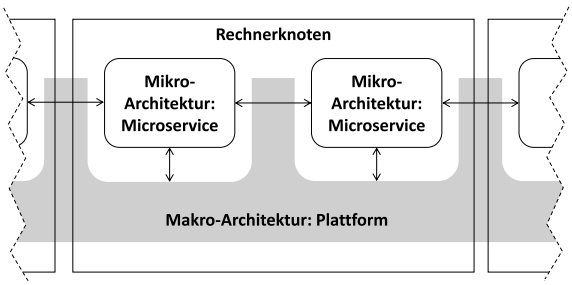

Abb. 1 Kommunikationsbeziehung Plattform/Microservice, Plattform überspannt mehrere Rechnerknoten

Die Architektur teilt sich auf in die interne Struktur der einzelnen Services (Mikroarchitektur), die Plattform für die Services (Makroarchitektur) und die Schnittstellen eines Service zur Plattform und zu anderen Services (Abb. 1).

Adam Wiggins, ein Mitgründer des Cloud-Service-Providers Heroku, beschreibt mit „The Twelve Factors“ unter anderem die folgenden Prinzipien: Jeder Service soll seine eigene Codebasis haben und damit unabhängig von anderen Services

DOI 10.1007/s00287-017-1078-6

Alexander Schwartz
msg systems ag,
Robert-Bürkle-Straße 1, 85737 Ismaning/München
E-Mail: alexander.schwartz@msg.group

*Vorschläge an Prof. Dr. Frank Puppe <puppe@informatik.uni-wuerzburg.de> oder an Dr. Brigitte Bartsch-Spörl <brigitte@bsr-consulting.de>

Alle „Aktuellen Schlagwörter“ seit 1988 finden Sie unter: http://www.is.informatik.uni-wuerzburg.de/as

weiterentwickelt werden können. Er verwaltet seine Abhängigkeiten selbst, anstatt sie in seiner Laufzeitumgebung vorauszusetzen. Abhängige Ressourcen wie Datenbanken und E-Mail-Services werden zur Laufzeit konfiguriert. Services exportieren ihre APIs über TCP- oder UDP-Ports. Sie bringen dafür zum Beispiel einen integrierten Webserver mit und sind idealerweise zustandslos in der Kommunikation [4].

In seiner Mikroarchitektur kann sich jedes Team für einen Service unabhängig von den anderen Services und Teams für Programmiersprache, Build-Tools und Datenbank entscheiden, um die angebotenen Dienste optimal zu liefern. Diese Entscheidungen können auch geändert werden, ohne dass andere Services davon betroffen sind. Wichtig ist, dass alle Schritte der Build- und Run-Phasen automatisiert werden, da manuelle Prozesse aufgrund der Vielzahl von Microservices und deren Instanzen zu teuer wären.

Die Plattform kann Instanzen eines Microservice auf einem oder mehreren Rechnerknoten installieren und starten. Die Anzahl der Instanzen kann sich zur Laufzeit ändern. Mechanismen zur Verteilung von Aufrufen (Loadbalancer) sind heutzutage meist Teil der Plattform. Über Health-Checks kann die Plattform überwachen, ob ein Service korrekt funktioniert.

Die Aufgabenverteilung zwischen Plattform und Services ist eine wichtige Architekturentscheidung. Im Open-Source-Software-Stack von Netflix übernehmen die Services viele Aufgaben und registrieren sich selbstständig bei einer Service-Registry (Eureka[1]), nutzen clientseitige Bibliotheken für eine Lastverteilung der Aufrufe an nachgelagerte Systeme (Ribbon[2]) und kümmern sich um die Fehlerbehandlung, wenn eine Schnittstelle nicht oder zu langsam antwortet (Hystrix[3]). Im Gegensatz dazu übernimmt die Plattform Kubernetes[4] das Service-Discovery und ein Service-Mesh wie envoy[5] oder istio[6] als Teil der Plattform die Lastverteilung, Fehlerbehandlung und das Monitoring für Schnittstellen zwischen Services.

Je mehr Funktionen die Makroarchitektur anbietet, desto einfacher erscheint die Implementierung der Microservices. Allerdings sind alle Funktionen der Makroarchitektur zu Beginn eines Projekts zwischen allen Nutzern abzustimmen, damit die Microservices so entwickelt werden, dass sie nach außen gleichartig erscheinen und effizient betrieben werden können. Zu Beginn eines Projekts liegen jedoch nur wenige Erfahrungen und belastbare Anforderungen vor und der Entscheidungsprozess selbst benötigt Zeit. Je mehr Funktionalität in der Makroarchitektur vorgesehen ist, desto häufiger sind Aktualisierungen notwendig, die dann auch zeitgleiche Änderungen an den Microservices bedingen.

Je weniger Funktionen in der Makroarchitektur vorgesehen sind, desto stabiler wird sie langfristig sein. Die Entscheidungen werden dann auf Microserviceebene getroffen: Jedes Team bestimmt seinen eigenen spätestmöglichen Zeitpunkt („last responsible moment“), um die jeweilige Entscheidung zu treffen. Damit gewinnen die Teams Zeit, um mehr Informationen zu sammeln und können bessere Entscheidungen treffen [7].

Schneiden von Services

Systeme können entsprechend der nichtfunktionalen Eigenschaften ihrer Komponenten in Microservices geschnitten werden. Kriterien können beispielweise Transaktionsvolumen, Verfügbarkeit, Datenschutz oder erwartete Änderungsrate und damit Release-Zyklus der Komponente sein. Damit kann jeder Microservice die benötigten nichtfunktionalen Eigenschaften optimal abbilden, anstatt dass ein Monolith die Summe aller nichtfunktionalen Eigenschaften abbilden muss.

Die Schnitte anhand der nichtfunktionalen Eigenschaften reichen in der Regel aber nicht aus, um Services zu erhalten, die von einem einzelnen Team verantwortet werden können. Hier bietet sich die fachliche Dekomposition entsprechend des Domain Driven Designs an [5]. Dabei werden fachliche Objekte der Anwendungsdomäne und ihre Abhängigkeiten und Interaktionen beschrieben. Bildet die technische Architektur diese Dekomposition ab und nutzt sie die fachlichen Begriffe, so kann sich eine gemeinsame Sprache zwischen Fachseite und Entwicklung etablieren, die eine langfristige Weiterentwicklung der Software unterstützt. Mit dem Konzept der Bounded Contexts wird akzeptiert, dass nicht jede Komponente die gleiche Sicht auf die fachlichen Objekte erfordert [6].

[1] Vgl. https://github.com/Netflix/eureka.
[2] Vgl. https://github.com/Netflix/ribbon.
[3] Vgl. https://github.com/Netflix/Hystrix.
[4] Vgl. https://kubernetes.io/.
[5] Vgl. https://github.com/lyft/envoy.
[6] Vgl. https://github.com/istio/istio.

Der kleinste mögliche Schnitt eines Microservice enthält alle zu einer Transaktion benötigten fachlichen Objekte. Der größte sinnvolle Schnitt beinhaltet einen kompletten Bounded Context, da innerhalb die fachlichen Objekte eine hohe Kohäsion und die Bounded Contexts untereinander eine geringe Kopplung haben.

Datenhaltung

Eine von mehreren Services gemeinsam genutzte Datenbank steht im Gegensatz zur unabhängigen Weiterentwicklung der einzelnen Services, da sie eine enge Kopplung der Services bedeutet. Außerdem können die beteiligten Services nicht mehr unabhängig voneinander skaliert werden, da sie sich mit ihrer Transaktionslast gegenseitig beeinflussen. Daher erhält jeder Microservice, der eine Datenhaltung benötigt, seine eigene Datenbank – oder zumindest sein eigenes Datenbankschema, um später in eine eigene Datenbank umziehen zu können. Benötigen andere Services diese Daten, so greifen sie über APIs darauf zu oder lassen sich über Veränderungen über asynchrone Events benachrichtigen.

Sollen Daten serviceübergreifend analysiert werden, so kann ein gemeinsamer Datenbestand aus den verschiedenen Services zusammengeführt oder aus den historischen Events der einzelnen Services aufgebaut werden. Die Analysen können jedoch auch direkt auf den historischen Events aufsetzen und diese als Stream verarbeiten.

Kommunikation zwischen Services

Werden die Services in der beschriebenen Form geschnitten, so entsteht ein verteiltes System, das auf mehreren Rechnerknoten läuft. Damit ergeben sich Herausforderungen rund um Bandbreite, Latenz und Zuverlässigkeit in der Kommunikation zwischen den Services [12]. Benötigt ein Service die Daten eines anderen Services, so kann er sich diese bei Bedarf in dem Moment holen, indem er sie benötigt. Er muss Vorkehrungen treffen für den Fall, dass der Aufruf sehr lange dauert oder zu einem Fehler führt. Darüber hinaus darf er das aufgerufene System nicht überlasten.

Ein Service kann sich von anderen Services entkoppeln, indem er die benötigten Daten in seiner eigenen Datenbank als Kopie hält und sie entsprechend seiner Anforderungen strukturiert und indexiert. So kann er Servicelevels für Anfragen garantieren, die auf Daten eines anderen Services basieren. Für jedes Datum gibt es jedoch immer nur einen Service, der die Datenhoheit hat und Veränderungen an den Daten durchführen darf. Alle anderen Services nutzen eine Replik.

Es gibt verschiedene Möglichkeiten, die Repliken aktuell zu halten: Bei Polling-Ansätzen fragen die Nutzer der Daten periodisch beim anderen Service an. Dies ist robust und erlaubt einen einfachen, zustandslosen Server ohne zusätzliche Middleware. Bei Publish-Subscribe-Ansätzen wird ein Message-Bus benötigt, der als Teil der Makroarchitektur implementiert wird. Der Service mit der Datenhoheit sendet Informationen zu neuen, geänderten und gelöschten Daten als Nachrichten an ein Topic im Message-Bus, das die Clients abonnieren. Clients erhalten die Information schneller als im Polling-Verfahren, dies wird jedoch mit einer komplexeren Architektur erkauft.

Eine Alternative für die Kommunikation zwischen Services bieten asynchrone fachliche Events. Ein Service kann sie nutzen, um seine Datenkopie zu aktualisieren oder direkt eigene Aktionen auszulösen. Events unterscheiden sich von der Datenreplikation dadurch, dass das Event die fachliche Änderung an den Daten beschreibt (z. B. bei einem Kunden „E-Mail-Adresse bestätigt“ oder „Adresse geändert auf XY“). Events können über Polling oder Publish-Subscribe weitergegeben werden.

Serviceschnitt am Beispiel eines Online-Shops

Die Ideen aus den vorherigen Abschnitten soll folgendes Beispiel eines vereinfachten Online-Shops erläutern: Gemäß der nichtfunktionalen Eigenschaften erfolgt ein erster Serviceschnitt in Bereiche, z. B. in Startseite (hohe Aufrufzahl, hohe Volatilität), Produktsuche (Suchindex über alle Daten benötigt), Detailseite (starke Verzweigung zu weiteren Funktionen, mögliche Differenzierung nach Produkt), Warenkorb (auf vielen Seiten eingebunden), Check-Out (Erfassen von sensiblen Zahlungsinformationen) und Produktpflege (Master für Produktdaten, nur intern erreichbar).

Jeder Bereich wird dann nach Bedarf in kleinere Microservices unterteilt: Am Beispiel der Detailseite könnten dies z. B. Bildervorschau, Größen- und Farbauswahl und Produktvorschläge zu ähnlichen Produkten sein.

Auch wenn die Suche keine eigenen Daten speichert, wird sie die benötigten Daten aus der

Produktpflege replizieren, um sie in einem inversen Index abzulegen. Über ein Event „Produkt aktualisiert" der Produktpflege kann sie ihren Datenbestand aktualisieren. Wenn der Service Kundenregistrierung das Event „Neuer Kunde registriert" versendet, so kann der Service Newsletter als Reaktion darauf eine Willkommens-E-Mail verschicken.

Ob ein Artikel einem Kunden im Online-Shop als verfügbar angezeigt wird, ist abhängig von verschiedenen Faktoren. Bei einem Versandhaus z. B. für Modeartikel können hierfür die Bestände aus dem Lager abzüglich eines Puffers für Inventurdifferenzen, die erwarteten Lieferungen der Lieferanten und die Anzahl der von Kunden kürzlich in den Warenkorb gelegten Artikel berücksichtigt werden. Für jede dieser Informationen hat ein anderer Service die Datenhoheit. Das Versandhaus wird diese Informationen mit Heuristiken bewerten: Auf der einen Seite soll der Bestand möglichst weit abverkauft werden, auf der anderen Seite soll der Kunde möglichst selten die Meldung bekommen, dass der Artikel, den er eben noch als verfügbar sah, nun doch vergriffen ist. In den ersten Schritten des Einkaufprozesses nutzen die Services replizierte Daten, um die Skalierbarkeit zu gewährleisten. In der Suche wird die Verfügbarkeit als weiteres Attribut in den Suchindex integriert. Auf der Detailseite führt ein eigenständiger Verfügbarkeitsservice die replizierten Daten zusammen und zeigt sie an. Der Check-out kann je nach Warengruppe, Preis oder Restbestand über einen synchronen Aufruf den Artikel zeitlich befristet reservieren oder ebenfalls replizierte Daten nutzen. Der Versandhändler Amazon verschickt nach dem Check-out eine Bestellbestätigung mit dem Hinweis, dass der Kaufvertrag erst dadurch zustande kommen soll, dass Amazon eine Versandbestätigung des Artikels verschickt. Ob dies nun rechtlich haltbar ist oder nicht – es deutet darauf hin, dass auch im Check-out replizierte Datenbestände und Heuristiken möglich sind.

Weiterführende Beispiele, wie ein fiktiver Check-out-Prozess eventbasiert oder mit Prozessorchestrierung aussehen könnte, finden sich im Artikel von Tobias Flore [13] und der sich daran anschließenden Diskussion.

Microservices weitergedacht

Wer die Microservices als zu unstrukturiert erachtet, sich aber dem Gedanken von autonomen Systemen anschließen kann, wird eine zusätzliche vertikale Strukturierung begrüßen. Diese wurde vom Online-Shop Otto.de als Vertikale [9] vorgestellt und später als Self-Contained System beschrieben [10]. Jedes Self-Contained System besteht aus einem (manchmal auch mehreren) Microservices. Es bringt sein eigenes Web-Frontend mit, kümmert sich um die eigene Datenhaltung und bringt die gesamte benötigte fachliche Logik selbst mit. Mit umliegenden Services ist es idealerweise nur per Weblinks lose gekoppelt, alternativ kommuniziert es asynchron im Backend. Daten aus anderen Self-Contained Systems werden bei Bedarf repliziert.

Wer bei den Services den Fokus auf Fachlichkeit legen möchte und der Infrastruktur mehr Aufgaben zukommen lassen will, der findet mit Function as a Service (FaaS) neue Möglichkeiten: Mit Amazon Lambda, Google Cloud Functions oder Apache OpenWisk[7] können einzelne Funktionen unabhängig voneinander bereitgestellt und skaliert werden.

Wann sind Microservices der richtige Ansatz?

Dem Ansatz von kleinen Teams, die unabhängig voneinander entwickeln, sind auch im Kontext agiler Softwareentwicklung viele große Unternehmen gefolgt: So haben in Deutschland Otto [9] und Kaufhof-Galleria die Software-Entwicklung für ihre Online-Shops auf diese Prinzipien umgestellt. Klassische Unternehmen wie der Finanzkonzern ING in den Niederlanden haben auf agile Softwareentwicklung und Microservices umgestellt.

Eine Gemeinsamkeit ist, dass sie sich einen Wettbewerbsvorteil durch häufige Releases, Experimente und differenzierte Technologie erhoffen. Sie alle haben in Automatisierung der Releases, Self-Service-Funktionalität für die Entwicklungsteams und Monitoring investiert. Sie ermöglichen ihnen eine kontinuierliche Weiterentwicklung ihrer Systeme in Bezug auf Funktion und Architektur. Diesen Unternehmen scheint gemeinsam zu sein, dass hier fachliche Individualsoftware entwickelt wird und jedes dieser Unternehmen technische Komponenten als Open Source zur Verfügung stellt.

Es lockt die Möglichkeit, die Architektur kontinuierlich durch den Austausch einzelner Services erneuern zu können. Gleichzeitig entwickelt sich das Microservice-Ökosystem mit einer hohen Geschwindigkeit weiter, was eine hohe

[7] Vgl. http://openwhisk.org/.

Veränderungsbereitschaft für die Mikro- und Makroarchitektur fordert.

Microservices als Architektur für das Enterprise?

Auch wenn immer mehr Erfolgsgeschichten von Microservicearchitekturen veröffentlicht werden, so sind Microservices nicht für jedes Unternehmen die richtige Wahl. Natürlich werden Erfolge groß gefeiert und es ergibt sich ein Survivorship Bias. Ein blindes Anwenden der Technologie führt jedoch zu großen Aufwänden, denen keine Verbesserungen gegenüberstehen. Für Unternehmen bedeutete es große Investitionen in Talente, Kultur und Technologie. Ohne Exzellenz in Automatisierung und Monitoring ist das Ziel nicht zu erreichen. Ein prominentes Gegenbeispiel zu Cloud und Microservices ist der Online-Service StackOverflow. In regelmäßigen Abständen veröffentlicht dieses Unternehmen seine Architektur und die Traffic-Eckpunkte seiner Website [11]. Das Unternehmen legt dar, warum es sich immer wieder gegen Cloud und Microservices entschieden hat und für physikalische Hardware und eine monolithische Architektur – bei gleichzeitig hoher Weiterentwicklungsgeschwindigkeit.

Daher sollte der Architekturansatz Microservices – wie jeder andere auch – in der jeweiligen Situation geprüft und bewertet werden. Dabei helfen die vorhandenen Erfahrungsberichte und Best Practices, die zeigen, dass es sich um mehr als nur einen Hype handelt. Diejenigen, die sich für Microservices entscheiden, können bei der Umsetzung auf eine große Zahl an Open-Source-Projekten zurückgreifen.

Literatur

1. Kim G, Humble J, Debois P, Willis J (2016) DevOps Handbook. How to Create World-Class Agility, Reliability, & Security in Technology Organizations. IT Revolution Press, Portland
2. Gray J (2006) A Conversation with Werner Vogels. ACM QUEUE (May 2006), pp 14–22
3. Conway ME (1968) How do committees invent. Datamation 14(4):28–31
4. Wiggins A (2017) The Twelve-Factor App. Letzte Änderung 30.1.2012, https://12factor.net/, letzter Zugriff: 15.6.2017
5. Evans E (2003) Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley
6. Vernon V (2013) Implementing Domain-Driven Design. Addison Wesley
7. Poppendieck M, Poppendieck T (2003) Lean Software Development: An Agile Toolkit. Addison Wesley
8. Lewis J, Fowler M (2017) Microservices, https://martinfowler.com/articles/microservices.html, letzter Zugriff: 17.5.2017
9. Kaus S, Steinacker G, Wegner O (2013) Teile und herrsche: Kleine Systeme für große Architekturen. OBJEKTspektrum 5:8–13
10. Self-Contained Systems, http://scs-architecture.org/, letzter Zugriff: 17.5.2017
11. Stack Overflow: The Architecture – 2016 Edition, https://nickcraver.com/blog/2016/02/17/stack-overflow-the-architecture-2016-edition/, letzter Zugriff: 4.6.2017
12. Deutsch P: Fallacies of Distributed Computing, https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing, letzter Zugriff: 8.6.2017
13. Flore T: Wer Microservices richtig macht, braucht keine Workflow Engine und kein BPMN, https://blog.codecentric.de/2015/09/wer-microservices-richtig-macht-braucht-keine-workflow-engine-und-kein-bpmn/, letzter Zugriff: 8.6.2017