Martin Fowler
(fowler@acm.org)
ist Autor und renommierter Referent zum Thema Softwarearchitektur. Als Principal Consultant ist er bei ThoughtWorks auf objektorientierte Analyse und Design, UML, Entwurfsmuster und agile Softwareentwicklung spezialisiert und war einer der Erstunterzeichner des Agilen Manifests.

James Lewis
ist Principal Consultant bei ThoughtWorks, wo er als Leiter verschiedener Teams Softwaresysteme für Kunden in Europa entwickelt hat und jetzt als Berater für erfahrene Architektur- und Management-Gruppen tätig ist. Er hat Systeme auf der Grundlage von Microservices gebaut und arbeitet aktiv in der wachsenden Community, die sich mit diesem Architekturstil befasst.

Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr?

Der Begriff „Microservice Architecture" kam in den letzten Jahren auf, um Softwareanwendungen zu beschreiben, die aus voneinander unabhängig deploybaren Services bestehen. Auch wenn eine formale Definition bisher fehlt, gibt es gemeinsame Merkmale hinsichtlich der Organisation entlang von Business-Capabilities, der automatisierten Bereitstellung, der Logik in den Endpunkten und der dezentralen Steuerung von Programmiersprachen und Daten. Das Softwareunternehmen ThoughtWorks hat es sich zur Aufgabe gemacht, das Design, die Erstellung und die Verbreitung von Software zu verändern. Dabei ist in den letzten Monaten das Thema Microservices in unseren Projekten immer stärker in den Fokus gerückt. Wir analysieren in diesem Artikel die zunehmende Bedeutung von Microservices und versuchen eine Definition für die Microservice-Architektur zu finden.

Kurz gesagt, ist der Microservice-Stil ein Ansatz für die Entwicklung einer einzigen Anwendung in Form einer Reihe kleiner Services, die jeweils in einem eigenen Prozess laufen und die durch einfache Mechanismen kommunizieren – oft durch HTTP-Ressourcen-basierte APIs. Diese Dienste orientieren sich entlang *Business-Capabilities* und sind durch vollautomatisches Deployment unabhängig voneinander deploybar. Es gibt nur ein Minimum an zentraler Verwaltung dieser Dienste, die unterschiedliche Programmiersprachen wie auch Datenspeicher-Technologien verwenden können.

Um sich dem Begriff Microservice-Stil zu nähern, ist es hilfreich, ihn mit dem monolithischen Stil zu vergleichen – einer monolithischen Anwendung, die als eine Einheit gebaut ist (siehe Abbildung 1). Enterprise-Anwendungen werden oft in drei Hauptschichten gebaut: Für den Kunden sichtbar sind eine Benutzungsschnittstelle, eine Datenbank und eine Anwendung auf der Serverseite.

Diese serverseitige Anwendung ist ein Monolith, also eine logisch einzeln ausführbare Datei. Alle Änderungen am System würden den Aufbau und die Bereitstellung einer neuen Version der serverseitigen Anwendung bedeuten.

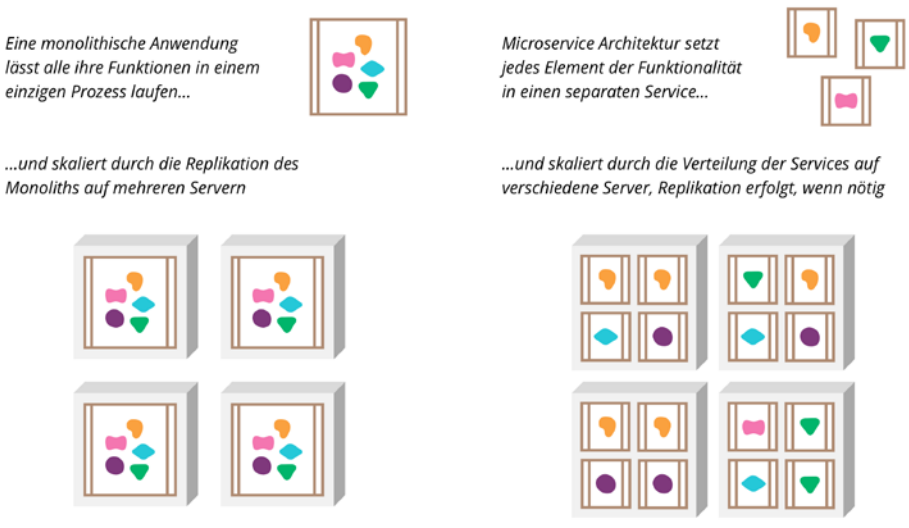

Abb. 1: Monolithen und Microservices

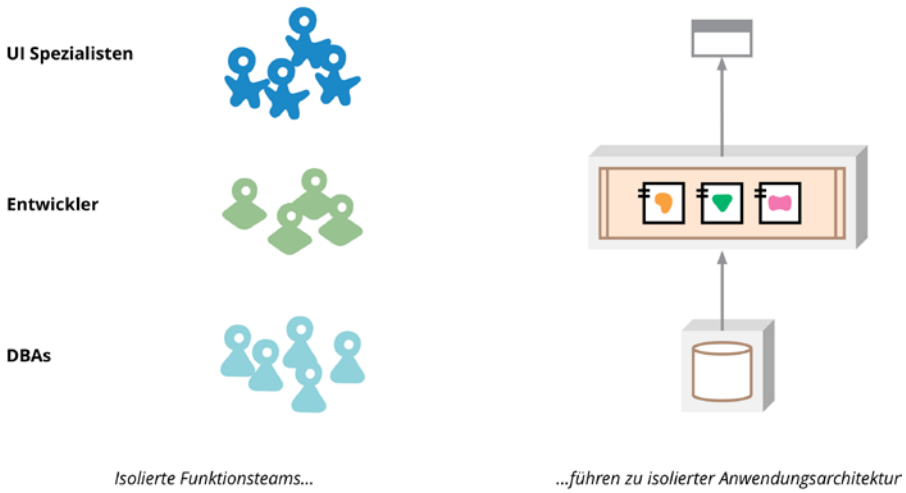

Abb. 2: Conways Law in Aktion

Komponententrennung durch Services

Solange wir uns erinnern können, besteht in der Softwareindustrie der Wunsch, Systeme zu bauen, indem man einzelne Komponenten zusammensteckt – ähnlich wie auch bei Systemen in der physischen Welt. In den letzten Jahrzehnten haben wir erhebliche Fortschritte mit großen Sammlungen von gemeinsamen Bibliotheken gesehen, die heute Teil der meisten Programmier-Plattformen sind.

Den Begriff der Komponente zu definieren ist aber gar nicht so einfach. Unsere Definition lautet: Eine *Komponente* ist eine Softwareeinheit, die unabhängig austauschbar und erweiterbar ist.

Microservice-Architekturen verwenden auch Bibliotheken, jedoch ist der bevorzugte Weg der Komponententrennung das Aufteilen in Services. In diesem Kontext bestehen *Bibliotheken* aus einem Zusammenschluss mehrerer zusammengehöriger Komponenten, die sich gegenseitig *in-memory* aufrufen. *Services* hingegen sind *out-of-process* - Komponenten, die mittels Web-Service-Anfragen oder *Remote Procedure Calls* kommunizieren.

Ein Hauptgrund für die Verwendung von Services als Komponenten – im Gegensatz zu Bibliotheken – ist die unabhängige Deploybarkeit von Services. Wenn man eine Anwendung hat, die mit mehreren Bibliotheken in einem einzigen Prozess läuft, muss nach einer Änderung an einer einzigen Komponente die gesamte Anwendung deployed werden. Ist jedoch die Anwendung in mehrere Services aufgeteilt, reicht es bei vielen Änderungen an einem Service, eben auch nur diesen Service neu zu deployen.

Das ist nicht immer der Fall – einige Änderungen werden Service-Schnittstellen verändern, was eine gewisse Koordination erforderlich macht. Doch das Ziel einer guten Microservice-Architektur ist es, die Abhängigkeiten zwischen Services dadurch zu minimieren, dass die Schnittstellen sinnvoll geschnitten werden und die Serviceverträge so gestaltet sind, dass sie in verschiedenen Versionen genutzt werden können.

Eine weitere Folge, wenn Services als Komponenten verwendet werden, sind die wesentlich expliziteren Schnittstellen zwischen Komponenten. Die meisten Programmiersprachen bieten keine guten Mechanismen, um explizite *Published Interfaces* (vgl. [Fow03]) zu definieren.

Häufig hindern nur Dokumentation und Disziplin die Aufrufer daran, die Kapselung einer Komponente aufzuweichen, was schnell zu einer übermäßig engen Kopplung zwischen den Komponenten führt. Services machen es einfacher, eine enge Kopplung mithilfe von expliziten Remote-Aufrufen zu vermeiden.

Es gibt jedoch auch Nachteile, Services so zu verwenden. Verteilte Aufrufe sind teurer als *in-process*-Aufrufe, deshalb müssen Remote-APIs grobkörniger sein – das macht sie meist umständlicher in der Benutzung. Außerdem ist das Verschieben von Verantwortlichkeiten zwischen einzelnen Komponenten mit Aufwand verbunden, wenn dabei Prozessgrenzen überschritten werden.

Als erste Annäherung können wir feststellen, dass sich Services gemäß einzelnen Laufzeitprozessen aufteilen lassen, aber das ist wirklich nur eine erste Annäherung. Ein Service kann auch aus mehreren Prozessen bestehen, die immer zusammen entwickelt und deployed werden, wie ein Anwendungsprozess und eine Datenbank, die nur von diesem Service verwendet werden.

Strukturierung nach Business-Capabilities

Wenn eine große Anwendung in Teile zerlegt wird, konzentriert sich das Management meist auf die technischen Schichten, was zur Entstehung von User-Interface-, Backend- und Datenbank-Teams führt. Sind Teams derart voneinander getrennt, können kleine Änderungen zu einem teamübergreifenden Projekt werden – und das kostet Zeit und Geld.

Ein gutes Team wird versuchen, das zu optimieren und sich für das kleinere von zwei Übeln entscheiden: die Logik in diejenige Anwendung zu zwingen, zu der das Team Zugang hat. Das ist ein Beispiel für Conways Law (siehe Abbildung 2): „Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure" (vgl. [Con68]).

Der Microservice-Ansatz zur Aufteilung von Services und Teams ist anders: Hier erfolgt die Organisation nach *Business Capabilities*. Solche Services implementieren Software für eine fachliche Geschäftseinheit durch den gesamten Technologie-Stack, einschließlich der Benutzungsoberfläche, Datenspeicherung und jeglicher externer Kommunikation mit anderen Systemen. Die Teams arbeiten funktionsübergreifend und haben in allen Bereichen der Softwareentwicklung Kompetenzen, wie zum Beispiel Datenbanken, Benutzungsschnittstellen und Projektmanagement (siehe Abbildung 3).

Produkte, nicht Projekte

Die meisten Softwarevorhaben, die wir sehen, werden in Form von Projekten organisiert: Ziel ist es, ein Stück Software zu entwickeln, das nach dem Projekt als abgeschlossen betrachtet wird. Die Software wird einem Wartungsteam übergeben – das Entwicklungsteam ist nicht länger verantwortlich.

Befürworter von Microservices sind gegen dieses Modell und empfehlen, dass Teams ihre Produkte über den gesamten Lebenszyklus hinweg betreuen sollen. Inspirierend hierfür ist Amazons Leitgedanke „You build it, you run it" (vgl. [Gra06]).

Entwickler erfahren, wie sich ihre Software im Betrieb verhält, und haben direkten Kontakt zu den Nutzern, da sie bei diesem Modell auch einen Teil des Supports übernehmen.

Die Produktmentalität ist mit der Aufteilung auf *Business Capabilities* verknüpft. Anstatt Software als eine Liste von Funktionen zu betrachten, die zu einem bestimmten Zeitpunkt fertig sind, geht es hier um eine kontinuierliche Beziehung. Die fortwährende Frage ist, wie die Software die Nutzer besser unterstützen und die Leistung der *Business Capabilities* optimieren kann. Es gibt natürlich keinen Grund, warum der gleiche Ansatz nicht auch für monolithische Anwendungen verwendet werden kann – aber durch die feinere Granularität von Services ist es einfacher, dass eine persönliche Beziehung zwischen Service-Entwicklern und ihren Nutzern entsteht.

„Smart Endpoints and dumb Pipes“

Bei der Implementierung von Kommunikationsstrukturen zwischen Prozessen haben wir viele Produkte und Ansätze gesehen, bei denen die Kommunikationsmechanismen mit sehr viel Intelligenz ausgestattet sind. Ein gutes Beispiel hierfür ist der *Enterprise Service Bus (ESB)*. ESB-Produkte enthalten häufig anspruchsvolle Features für Message-Routing, Choreographie, Transformation und die Anwendung von Geschäftsregeln.

Die Vertreter von Microservices bevorzugen einen alternativen Ansatz: Intelligente Endpunkte und dumme Verbindungen (*Smart Endpoints and dumb Pipes*). Ziel von Anwendungen im Microservice-Stil ist es, so entkoppelt und in sich kohäsiv wie möglich zu sein. Sie enthalten ihre eigene Domänenlogik und agieren im klassischen Unix-Sinne mehr als Filter – sie empfangen eine Anfrage, verarbeiten diese und erstellen eine Antwort.

Diese Services sind eher dazu gemacht, RESTful-Protokolle zu verwenden als komplizierte Protokolle wie WS-Choreography, BPEL oder gar die Orchestrierung durch ein zentrales Tool. Die am häufigsten verwendeten Protokolle sind *HTTP request-response* mit Resource APIs und leichtgewichtigem Messaging. Ein passendes Leitmotiv hierfür ist: „Be of the web, not behind the web” (vgl. [Rob06]).

Microservice-Teams verwenden die Prinzipien und Protokolle, auf denen das World Wide Web (und zu einem großen Teil Unix) aufgebaut ist. Oft verwendete

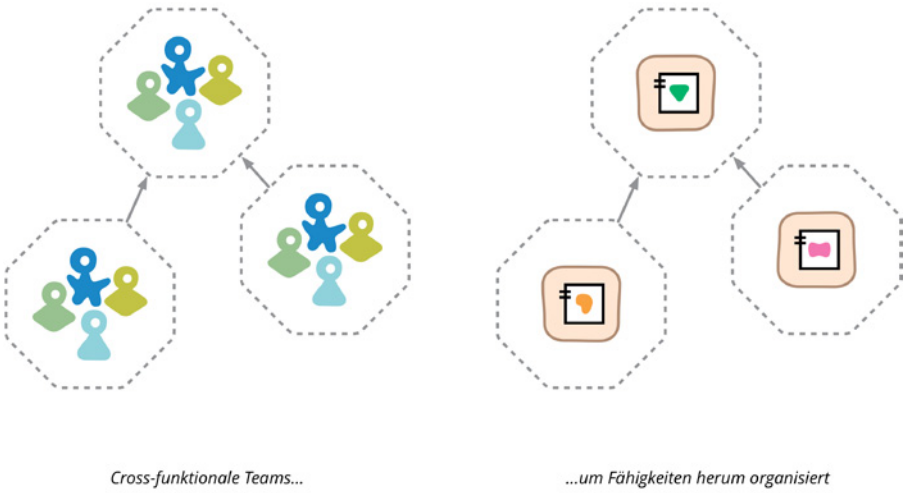

Abb. 3: Servicegrenzen durch Teamgrenzen verstärkt

Ressourcen können dabei mit sehr wenig Aufwand seitens Entwicklung oder Betrieb gecached werden.

Der zweite Ansatz, der häufig verwendet wird, ist Messaging über einen leichtgewichtigen Message-Bus. Die gewählte Infrastruktur ist in der Regel dumm („dumm“ meint hier, dass es nur um den Austausch von Nachrichten geht). Einfache Implementierungen wie RabbitMQ oder ZeroMQ machen nicht mehr, als eine zuverlässige asynchrone Message-Infrastruktur zu liefern. Die Intelligenz liegt weiterhin in den *Endpoints*, den Services – sie senden und empfangen einfach Nachrichten.

In einer monolithischen Applikation laufen Komponenten *in-process* und die Kommunikation zwischen ihnen geschieht über Methoden- oder Funktionsaufrufe. Das größte Problem bei der Umwandlung eines Monolithen in eine Microservice-Architektur ist die Veränderung der Kommunikationsmuster. Die einfache Überführung einer *in-memory*-Methode in einen RPC-Aufruf endet leicht in einer sehr Overhead-lastigen Kommunikation und ist auf Dauer nicht performant. Stattdessen sollte man die feingranulare und kontinuierliche Kommunikation in eine Kommunikation mit weniger, aber dafür umfangreicheren Nachrichten umwandeln.

Dezentrale Governance

Eine der Folgen von zentraler Governance und Standardisierung ist die Tendenz, sich auf eine einzige technische Plattform festzulegen. Die Erfahrung zeigt, dass dieses Vorgehen sehr einschränkt, denn nicht jedes Problem lässt sich mit der gleichen Lösung erschlagen. Auch wenn monolithische Applikationen zu einem gewissen Grad von der Benutzung mehrerer Programmiersprachen profitieren können, ist der Einsatz mehrerer Programmiersprachen eher unüblich.

Werden die Komponenten einer großen Anwendung in Services aufgesplittet, haben wir beim Bau jedes einzelnen Service die Wahl zwischen unterschiedlichen Optionen:

- Sie wollen Node.js nutzen, um eine einfache Statusseite zu bauen? Tun Sie das.
- Sie verwenden C++ für eine besonders kritische Echtzeit-Komponente? Prima.
- Sie meinen, ein Austausch der Datenbank würde zu besserem Leseverhalten einer Komponente führen?

Für jede Aufgabe gibt es das passende Werkzeug. Doch nur weil man etwas *tun kann*, bedeutet das natürlich noch nicht, dass man es unbedingt *tun sollte* – aber ein System auf diese Weise zu teilen bedeutet, dass man Optionen hat, wenn es darauf ankommt.

Teams, die Microservices bauen, bevorzugen auch bei Standards einen anderen Ansatz. Anstatt eine Reihe definierter Standards irgendwo auf Papier festzuhalten, wollen sie nützliche Werkzeuge schaffen, die anderen Entwicklern helfen können, ähnliche Probleme zu lösen. Diese Werkzeuge werden in der Regel während der Implementierung erstellt und dann mit einer größeren Gruppe von Entwicklern geteilt. Da Git und GitHub heute de facto die Versionskontrollsysteme der Wahl sind, werden Open-Source-Praktiken auch

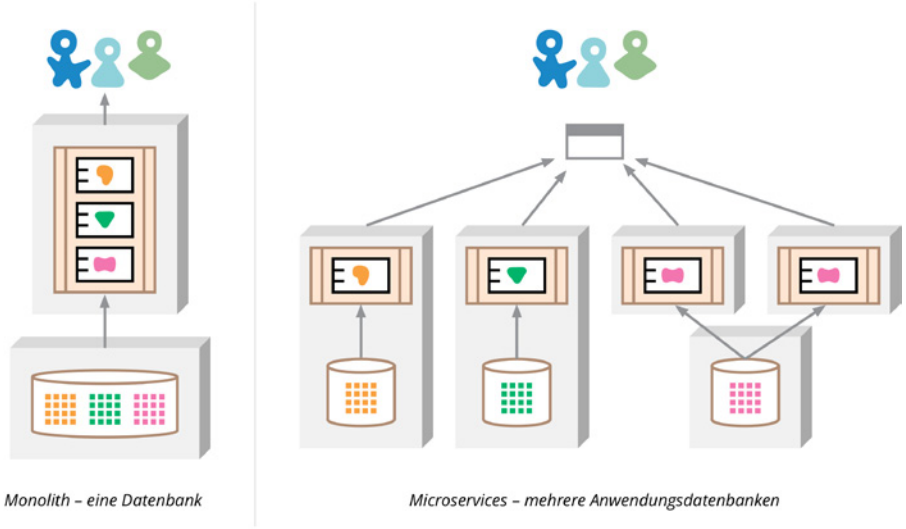

Abb. 4: Gegenüberstellung: Monolith versus Microservices

intern immer häufiger eingesetzt. Netflix ist ein sehr gutes Beispiel hierfür.

Die Microservice-Community mag keinen Overhead. Nicht, dass die Vertreter keinen Wert auf *Service Contracts* legt. Ganz im Gegenteil – es gibt in der Regel sogar mehr davon. Aber es gibt andere Möglichkeiten, diese *Service Contracts* zu verwalten.

Patterns wie *Tolerant Reader* (vgl. [Fow11-a]) und *Consumer-Driven-Contracts* (vgl. [Rob06]) werden oft für Microservices verwendet. Diese helfen dabei, die *Contracts* unabhängig voneinander weiterzuentwickeln. Die Ausführung von *Consumer-Driven-Contracts* als Teil des Builds erhöht das Vertrauen und bietet eine schnelle Rückmeldung darüber, ob ein Service noch funktioniert.

Wir kennen beispielsweise ein Team in Australien, das den Aufbau neuer Services mit *Consumer-Driven-Contracts* vorantreibt. Das Team benutzt einfache Werkzeuge, um *Consumer-Driven-Contracts* zu definieren. Die Contracts werden dann Teil des automatisierten Builds, noch bevor der Code für den neuen Service überhaupt geschrieben wurde. Der Service wird dann nur bis zu dem Punkt gebaut, an dem er den *Service-Contract* erfüllt – eine elegante Methode, beim Bau neuer Software das *YAGNI*-Dilemma (*You Ain't Gonna Need It*) zu vermeiden. Diese Techniken und Werkzeuge, die rundherum entstehen, verringern die zeitliche Kopplung zwischen den Services und begrenzen dadurch den Bedarf für eine zentrale Verwaltung der Contracts.

Auf den Höhepunkt treibt vor allem Amazon die dezentrale Governance mit seinem Ethos „You build it, You run it." Teams sind für alle Aspekte der Software, die sie bauen – einschließlich ihres Betriebs – rund um die Uhr verantwortlich. Eine Dezentralisierung von Verantwortung in diesem Ausmaß ist nicht die Norm, aber wir sehen immer mehr Unternehmen, die die komplette Verantwortung ihren Entwicklungsteams übergeben. Netflix ist ein weiteres Unternehmen, das diesen Ethos lebt. Nachts um drei Uhr nicht mehr vom Pager geweckt zu werden, ist sicherlich ein starker Anreiz, qualitativ hochwertigen Code zu schreiben. Diese Ansätze stehen im kompletten Gegensatz zum traditionellen und zentralen Governance-Modell.

Dezentrales Datenmanagement

Dezentralisierung von Daten bedeutet auf der höchsten abstrakten Ebene zunächst, dass das konzeptionelle Modell der Welt sich je nach System unterscheidet. In einem großen Unternehmen ist die Integration dieser verschiedenen Modelle häufig ein Problem. Zum Beispiel hat der Vertrieb eine andere Sicht auf denselben Kunden als der Support. Was im Vertrieb mit dem Begriff „Kunde" gemeint ist, kann vom Support anders bezeichnet werden. Verwenden zwei Abteilungen doch einmal denselben Begriff, vergeben sie mitunter andere Attribute, oder noch schlimmer, gemeinsame Attribute mit unterschiedlicher Semantik.

Dieses Problem der Integration tritt meistens zwischen Applikationen auf. Es kann aber auch *innerhalb* einer Anwendung auftreten, insbesondere wenn die Applikation in separate Komponenten unterteilt ist. Ein sinnvoller Ansatz, dies zu betrachten, ist der *Bounded Context* (vgl. [Fow14]), ein Begriff aus dem *Domain-Driven Design (DDD)*.

DDD teilt eine komplexe Domäne in mehrere voneinander abgegrenzte Kontexte und definiert die Beziehungen zwischen ihnen. Dieses Vorgehen ist sowohl für monolithische Architekturen als auch für Microservices sinnvoll. Aber es besteht ein natürlicher Zusammenhang zwischen einem Service und den Kontextgrenzen, der hilft, die Trennung zu verdeutlichen – wie auch schon im Abschnitt über die Ausrichtung auf Business-Capabilities beschrieben.

Neben der Dezentralisierung über konzeptionelle Modelle entscheiden Microservices auch über die Dezentralisierung der Datenspeicherung. Während für monolithische Anwendungen eine jeweils eigene logische Datenbank für die Persistenz bevorzugt wird, wollen Unternehmen häufig eine einzige Datenbank für eine ganze Reihe von Anwendungen nutzen (siehe **Abbildung 4**). Viele dieser Entscheidungen sind durch Anbieter kommerzieller Modelle rund um Lizenzierung beeinflusst.

Microservices lassen lieber jeden Service seine eigene Datenbank verwalten. Dies können entweder verschiedene Datenbanken der gleichen Technologie oder ganz unterschiedliche Datenbank-Systeme sein – ein Ansatz, der P*olyglot Persistence* genannt wird (vgl. [Fow11-b]). Dieser Ansatz kann auch innerhalb von Monolithen verwendet werden, taucht aber meistens im Zusammenhang mit Microservices auf.

Die Dezentralisierung der Verantwortung für Daten durch Microservices hat Folgen für die Verwaltung von Updates. Der übliche Ansatz für den Umgang mit Updates sind Transaktionen, um im Falle einer Aktualisierung mehrerer Ressourcen ihre Konsistenz zu garantieren. Dieses Vorgehen wird oft in monolithischen Anwendungen angewendet.

Das Verwenden von Transaktionen erhöht die Konsistenz, birgt allerdings eine hohe zeitliche Kopplung, was problematisch bei der Verwendung mehrerer Services ist. Verteilte Transaktionen sind notorisch schwer zu implementieren, weshalb Microservice-Architekturen den Schwerpunkt auf transaktionslose Koordination zwischen Services legen (vgl. [Hoh04]). Dabei wird aber explizit anerkannt, dass die Konsistenz nur *Eventual Consistency* ist und dass Probleme durch kompensierende Operationen behandelt werden müssen.

Inkonsistenzen in dieser Weise zu verwalten, ist eine neue Herausforderung für viele Entwicklungsteams, doch es ist eine Herausforderung, die oftmals zum Business passt. Unternehmen müssen mit einer gewissen Inkonsistenz umgehen, um schneller auf Nachfragen und Fehler reagieren zu können. Es lohnt sich, Abstriche in der Konsistenz zu machen, solange die Kosten für die Fehlerbeseitigung bei Inkonsistenzen geringer sind als die Kosten für Geschäftsverluste mit größerer Konsistenz.

Die Automatisierung von Infrastruktur

Infrastruktur-Automatisierung hat sich in den letzten Jahren enorm entwickelt. Die Entwicklungen im Cloud-Computing – insbesondere Angebote wie Amazon-Web-Services – haben die operative Komplexität von Entwicklung, Deployment und Betrieb von Microservices reduziert. Viele der Produkte oder Systeme, die mit Microservices gebaut werden, werden von Teams mit langjähriger Erfahrung in Continuous Delivery (vgl. [Fow13]) und Continuous Integration entwickelt. Dabei machen diese Teams ausgiebigen Gebrauch von der Infrastruktur-Automatisierung (siehe Abbildung 5).

Auch wenn dies kein Artikel über Continuous Delivery ist, wollen wir doch auf einige wesentliche Aspekte eingehen. Weil man so viel Vertrauen wie möglich in die Funktionalität der Software legen möchte, führt man viele *automatisierte Tests* durch. Der Durchlauf der lauffähigen und getesteten Software durch die einzelnen Stationen einer Pipeline bedeutet außerdem *automatisiertes Deployment* in alle Umgebungen.

Es ist relativ einfach, eine monolithische Anwendung in verschiedene Umgebungen zu deployen. Wenn man einmal in die Automatisierung des *Path to Production* für einen Monolithen investiert hat, scheint es auf einmal gar nicht mehr so schwierig, gleich mehrere Anwendungen auf diese Art und Weise auszuliefern. Man darf nicht vergessen, dass es eines der Ziele von Continuous Delivery ist, das Deployment so „langweilig“ wie möglich zu machen. Ob also eine oder drei Anwendungen durch eine Pipeline laufen, sollte von daher kein Unterschied sein, solange es „langweilig“ bleibt.

Ein weiterer Bereich, in dem wir umfangreiche Infrastruktur-Automatisierungen sehen, ist der Betrieb von Microservices in Produktion. Obwohl es – wie gesagt – bei einem „langweiligen” Deployment kaum Unterschiede zwischen Monolithen und Microservices gibt, kann die Betriebslandschaft für beide doch sehr unterschiedlich aussehen (siehe Abbildung 6).

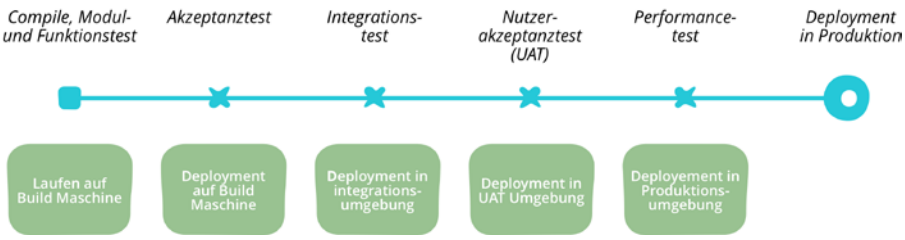

Abb. 5: Basis einer Build-Pipeline

„Design for failure“

Werden Services als Komponenten verwendet, müssen Anwendungen so gestaltet werden, dass sie den Ausfall von Services tolerieren können. Jeder Serviceaufruf kann aufgrund der Nichtverfügbarkeit des Anbieters fehlschlagen. Der Aufrufer muss darauf so tolerant wie möglich reagieren. Damit ergibt sich im Vergleich zu einem monolithischen Aufbau eine zusätzliche Komplexität: Microservice-Teams müssen ständig prüfen, wie Serviceausfälle die User-Experience beeinflussen. Die „Simian Army” von Netflix (vgl. [Net]) zum Beispiel provoziert gewollt Ausfälle von Services und sogar ganzer Rechenzentren im laufenden Betrieb, um sowohl die Widerstandsfähigkeit als auch das Monitoring der Anwendung zu testen.

Da Services theoretisch jederzeit ausfallen können, müssen die Fehler schnell erkannt und Services zeitnah und automatisch wiederhergestellt werden. Microservice-Anwendungen legen viel Wert auf ein Echtzeit-Monitoring der Anwendung, wobei sowohl technische Elemente (z. B. wie viele Anfragen pro Sekunde die Datenbank erhält) als auch geschäftsrelevante Kennzahlen (z. B. wie viele Bestellungen pro Minute eingehen) überwacht werden.

Wenn etwas schief geht, kann *Semantic Monitoring* als Frühwarnsystem funktionieren, um das Entwicklungsteam zu alarmieren. Monitoring ist in Microservice-Architekturen besonders wichtig, da *Choreography* und *Event Collaboration* (vgl. [Fow06]) zu schwer vorhersehbarem Verhalten führen können. Monitoring hilft dabei, Fehlverhalten schnell zu entdecken.

Monolithen können genauso transparent gebaut werden wie Microservices – das sollten sie sogar. Der Unterschied ist, dass es bei Microservices essenziell ist, zu wissen, wenn Services in den unterschiedlichen Prozessen nicht mehr miteinander kommunizieren können. Bei Bibliotheken, die innerhalb eines Prozesses laufen, ist diese Art der Transparenz weniger nützlich. Microservice-Teams benötigen ein durchdachtes Monitoring und Logging für jeden einzelnen Service. Dies können Dashboards sein, die den Up/Down-Status sowie eine Vielzahl von operativen und geschäftsrelevanten Kennzahlen zeigen. Details zum Status der *Circuit Breaker*, dem aktuellen Durchsatz oder der Latenz sind weitere Beispiele, die wir oft in der „freien Wildbahn“ beobachten.

Evolutionäres Design

Anhänger von Microservices haben in der Regel Erfahrungen mit *Evolutionary De-*

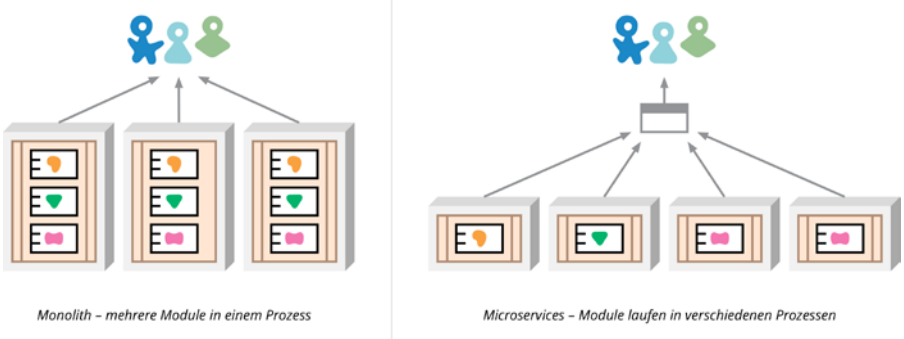

Abb. 6: Modulares Deployment kann unterschiedlich aussehen

sign und sehen die Zerlegung von Software in Services als weiteres Werkzeug, um Änderungen zu kontrollieren, ohne dadurch den Entwicklungsprozess zu verlangsamen. Änderungskontrolle bedeutet nicht unbedingt weniger Änderungen – mit der richtigen Einstellung und den richtigen Werkzeugen kann man häufig, schnell und gut kontrollierte Änderungen an Software vornehmen.

Wann immer man versucht, Software in Komponenten zu zerlegen, steht man vor der einen Frage: Nach welchen Prinzipien entscheidet man, in welche Teile die Anwendung geschnitten werden soll? Die Kerneigenschaft einer Komponente ist die Idee der unabhängigen Austauschbarkeit und Erweiterungsfähigkeit. Das setzt voraus, dass wir uns vorstellen können, eine Komponente neu zu bauen, ohne dabei ihre Kollaborateure zu beeinträchtigen. Tatsächlich gehen viele Microservice-Teams noch weiter: Sie gehen explizit davon aus, dass sie viele Services schnell wieder „einmotten" werden, statt sie langfristig zu entwickeln.

Die Website der Zeitschrift The Guardian ist ein gutes Beispiel für eine Anwendung, die ursprünglich als Monolith gebaut, aber dann in Richtung Microservices weiterentwickelt wurde. Der Monolith ist immer noch der Kern der Website, neue Funktionalitäten werden aber inzwischen in Form von Microservices hinzugefügt. Diese neuen Services nutzen die API des Monolithen.

Dieser Ansatz ist besonders praktisch für Funktionalität, die von Natur aus temporär ist, wie zum Beispiel spezielle Seiten zu Sportveranstaltungen. Eine solche Komponente kann mithilfe einer *Rapid Development Language* erstellt und wieder entfernt werden, sobald das Ereignis vorbei ist. Wir haben ähnliche Ansätze in einem Finanzinstitut gesehen, wo neue Dienstleistungen für den aktuellen Markt aufgenommen und nach ein paar Monaten oder sogar schon nach Wochen wieder verworfen werden.

Diese Gewichtung der Austauschbarkeit ist ein Spezialfall des Prinzips, Modularität durch *Patterns of Change* zu gestalten. Nach diesem Prinzip möchte man Dinge, die sich zur gleichen Zeit ändern, im gleichen Modul pflegen. Teile eines Systems, die sich nur selten ändern, sollten in anderen Services entwickelt werden, als solche die sich besonders häufig ändern. Werden zwei Services immer wieder gleichzeitig verändert, so ist das ein Zeichen dafür, dass beide zusammengeführt werden sollten.

Komponenten in Services zu implementieren, bietet auch die Chance für eine granularere Release-Planung. Beim Monolithen erfordert jede Änderung den vollständigen Build und das Deployment der gesamten Anwendung. Mit Microservices muss man nur diejenigen Services neu deployen, die sich geändert haben. Das kann den Release-Prozess vereinfachen und beschleunigen. Nachteil ist, dass man sich Gedanken machen muss, ob die Änderungen an einem Service auch Auswirkungen auf ihre Aufrufer haben.

Der traditionelle Integrationsansatz versucht, dieses Problem durch Versionierung zu lösen. Microservice-Vertreter präferieren jedoch, Versionierung als letzten Ausweg zu nutzen (vgl. [Bya13]). Ein gewisses Maß an Versionierung kann vermieden werden, wenn Services so gestaltet werden, dass sie so tolerant wie möglich mit Änderungen anderer Service-Anbieter umgehen können.

Sind Microservices die Zukunft?

Unser Hauptziel beim Schreiben dieses Artikels war es, die wesentlichen Ideen und Prinzipien von Microservices zu erklären. Wir sind davon überzeugt, dass Microservices ein wichtiges Konzept ist, das für Enterprise-Anwendungen ernsthaft in Betracht gezogen werden sollte. Wir haben bereits mehrere Systeme in diesem Stil gebaut und uns auch mit anderen Teams ausgetauscht, die diesen Stil verwenden und bevorzugen.

Als erste Anwender der Microservice-Architektur sind uns unter anderem Amazon, Netflix, The Guardian, der Government Digital Service der britischen Regierung, www.realestate.com.au, Forward und www.comparethemarket.com bekannt. Darüber hinaus gibt es viele Organisationen, die schon lange auf eine Art und Weise arbeiten, die wir als Microservices bezeichnen würden, ohne dass diese es jemals so genannt haben. Trotz dieser positiven Erfahrungen können wir nicht sagen, dass wir uns hundertprozentig sicher sind, ob Microservices die zukünftige Richtung von Softwarearchitektur bestimmen. Auch wenn unsere bisherigen Erfahrungen im Vergleich zu monolithischen Anwendungen positiv sind, sind wir uns aber auch der Tatsache bewusst, dass bisher nicht genügend Zeit vergangen ist, damit wir uns ein endgültiges Urteil erlauben könnten.

Oft genug sind die tatsächlichen Folgen einer Architekturentscheidung erst mehrere Jahre später erkennbar. Wir haben Projekte gesehen, in denen ein gutes Team mit einem starken Wunsch nach Modularität eine monolithische Architektur gebaut hat, die im Laufe der Zeit zerfallen ist. Viele Leute glauben, dass ein solcher Verfall mit Microservices weniger wahrscheinlich ist, da die Service-Grenzen explizit und schwer zu umgehen sind. Bis wir aber genügend Systeme über eine gewisse Zeitspanne beobachtet haben, können wir nicht wirklich beurteilen, wie Microservice-Architekturen reifen.

Es gibt sicherlich gute Gründe zu erwarten, dass Microservices sich negativ entwickeln. Bei jedem Versuch, Software in Komponenten aufzuteilen, hängt der Erfolg davon ab, wie gut die Software in Komponenten geschnitten werden kann. Es ist schwer zu definieren, wo genau die Grenzen von Komponenten liegen sollen. Evolutionäres Design erkennt diese Schwierigkeiten an und stellt deshalb die Bedeutung des leichten Refactorings von Komponenten in den Vordergrund.

Aber wenn Komponenten Services mit verteilter Kommunikation sind, ist Refactoring viel schwieriger als mit *in-process* Bibliotheken. Code über Servicegrenzen hinweg zu bewegen, ist schwierig – alle Änderungen in der Benutzungsoberfläche müssen zwischen den Teilnehmern abgestimmt werden. Außerdem muss die Rückwärtskompatibilität beachtet werden und auch die Tests werden komplizierter.

Ein weiteres Problem entsteht, wenn Komponenten nicht sauber zueinander passen. Dann wird die Komplexität aus dem Inneren einer Komponente oft in die Verbindungen zwischen Komponenten verschoben. Nicht nur, dass so Komplexität verschoben wird, sie landet auch dort, wo sie nicht unbedingt hingehört und schwer zu kontrollieren ist. Man denkt schnell, dass alles in Ordnung ist, wenn man sich die Innenseite einer Komponente ansieht, übersieht dabei aber die chaotischen Verbindungen zwischen den Services.

Schließlich spielt auch die Kompetenz des Teams eine Rolle. Neue Techniken werden in der Regel schneller von erfahrenen Teams angenommen. Aber eine Technik, die ein erfahrenes Team effektiver macht, ist nicht unbedingt für ein weniger schnell lernendes Team geeignet. Wir haben viele dieser Teams beobachtet, die

chaotische monolithische Architekturen gebaut haben. Ein schwaches Team wird immer ein schwaches System bauen. Doch es bleibt abzuwarten, ob Microservices diese Art von Chaos reduzieren oder verschlimmern. Wir haben auch Ansichten gehört wie: „Man sollte mit Microservices nicht auf der grünen Wiese anfangen. Stattdessen sollte man mit einem Monolithen starten." Dies ist ein durchaus plausibles Argument.

Wir haben diesen Artikel also mit vorsichtigem Optimismus geschrieben. Bisher haben wir ausreichend viele Microservice-Architekturen gesehen, um zu denken, dass es sich um einen lohnenden Weg handelt. Wir können nicht sicher sagen, wo wir am Ende landen. Doch das ist schließlich eine der großen Herausforderungen in der Softwareentwicklung – wenn wir Entscheidungen für die Zukunft treffen, verfügen wir meist nur über unvollständige Informationen. ■

Der Beitrag wurde ebenfalls in der Printausgabe von OBJEKTspektrum 01/2015 veröffentlicht.

Links

[Bya13] B. Byars, Enterprise Integration Using REST, 2013, siehe: http://martinfowler.com/articles/enterpriseREST.html#versioning

[Con68] M.E. Conway, How Do Commitees Invent?, 1968, siehe: http://www.melconway.com/Home/Committees_Paper.html

[Fow03] M. Fowler, PublishedInterface, 2003, siehe: http://martinfowler.com/bliki/PublishedInterface.html

[Fow06] M. Fowler, EventCollaboration, 2006, siehe: http://martinfowler.com/eaaDev/EventCollaboration.html

[Fow11-a] M. Fowler, TolerantReader, 2011, siehe: http://martinfowler.com/bliki/TolerantReader.html

[Fow11-b] M. Fowler, PolyglotPersistence, 2011, siehe: http://martinfowler.com/bliki/PolyglotPersistence.html

[Fow13]M. Fowler, ContinuousDelivery, 2013, siehe: http://martinfowler.com/bliki/ContinuousDelivery.html

[Fow14] M. Fowler, BoundedContext, 2014, siehe: http://martinfowler.com/bliki/BoundedContext.html

[Gra06] J. Gray, A Conversation with Werner Vogels: Learning form the Amazon technology platform, 2006, siehe: https://queue.acm.org/detail.cfm?id=1142065

[Hoh04] G. Hohpe, PubishedInterface, 2004, siehe: http://www.eaipatterns.com/ramblings/18_starbucks.html

[Net] Netflix, Simian Army siehe: https://github.com/Netflix/SimianArmy

[Rob06] I. Robinson, Consumer-Driven Contracts: A Service Evolution Pattern, 2006, siehe: http://martinfowler.com/articles/consumerDrivenContracts.html