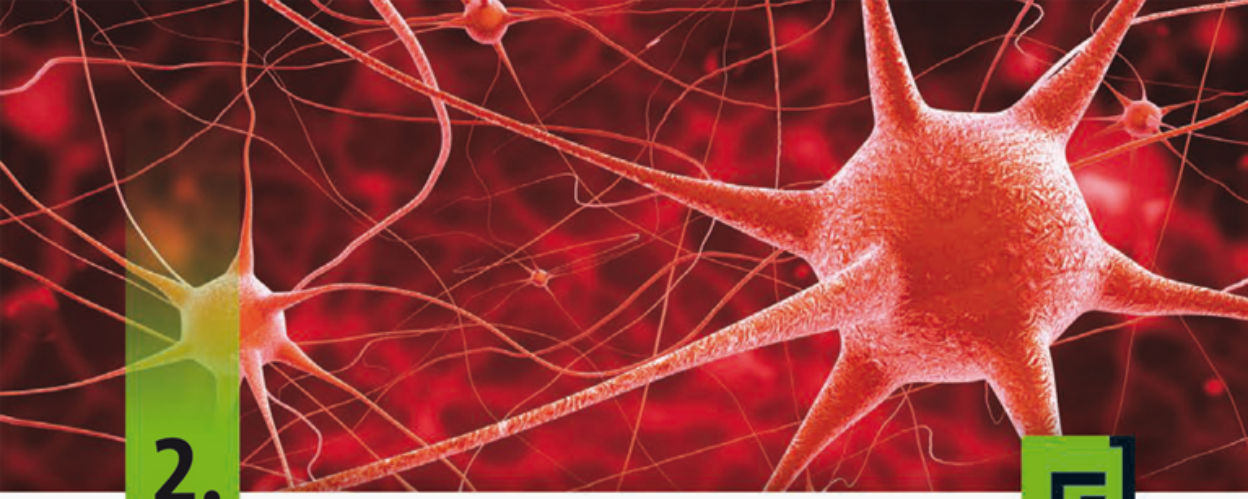

2. Auflage

Eberhard Wolff

Microservices

Grundlagen flexibler Softwarearchitekturen

dpunkt.verlag

Eberhard Wolff arbeitet seit mehr als 15 Jahren als Architekt und Berater – oft an der Schnittstelle zwischen Business und Technologie. Er ist Fellow bei der innoQ. Als Autor hat er über hundert Artikel und Bücher geschrieben – u.a. über Continuous Delivery – und als Sprecher auf internationalen Konferenzen vorgetragen. Sein technologischer Schwerpunkt liegt auf modernen Architekturansätzen – Cloud, Continuous Delivery, DevOps, Microservices oder NoSQL spielen oft eine Rolle.

Eberhard Wolff

Microservices

Grundlagen flexibler Softwarearchitekturen

2., aktualisierte Auflage

dpunkt.verlag

Eberhard Wolff
eberhard.wolff@gmail.com

Lektorat: René Schönfeldt
Copy-Editing: Sandra Gottmann (Münster-Nienberge)
Satz: Nadine Thiele
Herstellung: Susanne Bröckelmann
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN:
Print 978-3-86490-555-1
PDF 978-3-96088-413-2
ePub 978-3-96088-414-9
mobi 978-3-96088-415-6

2., aktualisierte Auflage 2018

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

Inhaltsverzeichnis

1 Vorwort

Microservices sind ein neuer Begriff – aber sie verfolgen mich schon lange. 2006 hielt Werner Vogels (CTO, Amazon) einen Vortrag auf der JAOO-Konferenz, wo er die Amazon Cloud und Amazons Partnermodell vorstellte [1]. Dabei erwähnte er das CAP-Theorem – heute Basis für NoSQL. Und dann sprach er von kleinen Teams, die Services mit eigener Datenbank entwickeln und auch betreiben. Diese Organisation nennen wir heute DevOps und die Architektur Microservices.

Später sollte ich für einen Kunden eine Strategie entwickeln, wie er moderne Technologien in seine Anwendung integrieren kann. Nach einigen Versuchen, neue Technologien direkt in den Legacy-Code zu integrieren, haben wir schließlich eine neue Anwendung neben der alten Anwendung mit einem völlig anderen modernen Technologie-Stack aufgebaut. Die neue und die alte Anwendung waren nur über HTML-Links gekoppelt – und über die gemeinsame Datenbank. Bis auf die gemeinsame Datenbank ist auch dieses Vorgehen im Kern ein Microservices-Ansatz. Das war 2008.

Ein anderer Kunde hatte schon 2009 seine komplette Infrastruktur in REST-Services aufgeteilt, die jeweils von einzelnen Teams weiterentwickelt wurden. Auch das nennen wir heute Microservices. Viele andere Unternehmen aus dem Internet-Bereich hatten damals schon ähnliche Architekturen.

In letzter Zeit wurde mir außerdem klar, dass Continuous Delivery [2] Auswirkungen auf die Software-Architektur hat. Auch in diesem Bereich haben Microservices viele Vorteile.

Und das ist der Grund für das Buch: Microservices sind ein Ansatz, den einige schon sehr lange verfolgen; darunter auch viele sehr erfahrene Architekten. Wie jeder Architekturansatz löst er sicher nicht alle Probleme – aber er kann eine interessante Alternative darstellen.

1.1 Überblick über Microservices

Microservice: vorläufige Definition

Im Mittelpunkt des Buchs stehen Microservices – ein Ansatz zur Modularisierung von Software. Modularisierung ist nichts Neues. Schon lange werden große Systeme in kleine Module unterteilt, um Software einfacher zu erstellen, zu verstehen und weiterzuentwickeln.

Das Neue: Microservices nutzen als Module einzelne Programme, die als eigene Prozesse laufen. Der Ansatz basiert auf der UNIX-Philosophie. Sie lässt sich auf drei Aspekte reduzieren:

- Ein Programm soll nur eine Aufgabe erledigen, und das soll es gut machen.
- Programme sollen zusammenarbeiten können.
- Nutze eine universelle Schnittstelle. In UNIX sind das Textströme.

Der Begriff Microservice ist nicht fest definiert. Kapitel 4 zeigt eine genauere Definition. Als erste Näherung dienen folgende Kriterien:

- Microservices sind ein Modularisierungskonzept. Sie dienen dazu, ein großes Software-System aufzuteilen – und beeinflussen die Organisation und die Software-Entwicklungsprozesse.
- Microservices können unabhängig voneinander deployt werden. Änderungen an einem Microservice können unabhängig von Änderungen an anderen Microservices in Produktion gebracht werden.
- Microservices können in unterschiedlichen Technologien implementiert sein. Es gibt keine Einschränkung auf eine bestimmte Programmiersprache oder Plattform.
- Microservices haben einen eigenen Datenhaushalt: eine eigene Datenbank – oder ein vollständig getrenntes Schema in einer gemeinsamen Datenbank.
- Microservices können eigene Unterstützungsdienste mitbringen, beispielsweise eine Suchmaschine oder eine spezielle Datenbank. Natürlich gibt es eine gemeinsame Basis für alle Microservices – beispielsweise die Ausführung virtueller Maschinen.
- Microservices sind eigenständige Prozesse – oder virtuelle Maschinen, um auch die Unterstützungsdienste mitzubringen.
- Dementsprechend müssen Microservices über das Netzwerk kommunizieren. Dazu nutzen Microservices Protokolle, die lose Kopplung unterstützen. Das kann beispielsweise REST sein – oder Messaging-Lösungen.

Dieser Ansatz betrachtet die Größe des Microservice nicht. Trotz des Namens »Microservice« ist die Größe für eine grobe Definition nicht so entscheidend.

Monolithen

Microservices grenzen sich von Deployment-Monolithen ab. Ein Deployment-Monolith ist ein großes Software-System, das nur als Ganzes auf einmal deployt werden kann. Es muss als Ganzes durch alle Phasen der Continuous-Delivery-Pipeline wie Deployment, Test, Abnahme und Release laufen. Durch die Größe des Deployment-Monolithen dauert dieser Prozess länger als bei kleineren Systemen. Das reduziert die Flexibilität und erhöht die Kosten der Prozesse. Der Deployment-Monolith kann intern modular aufgebaut sein – nur müssen alle diese Module gemeinsam in Produktion gebracht werden.

1.2 Warum Microservices?

Microservices dienen dazu, Software in Module aufzuteilen und dadurch die Änderbarkeit der Software zu verbessern.

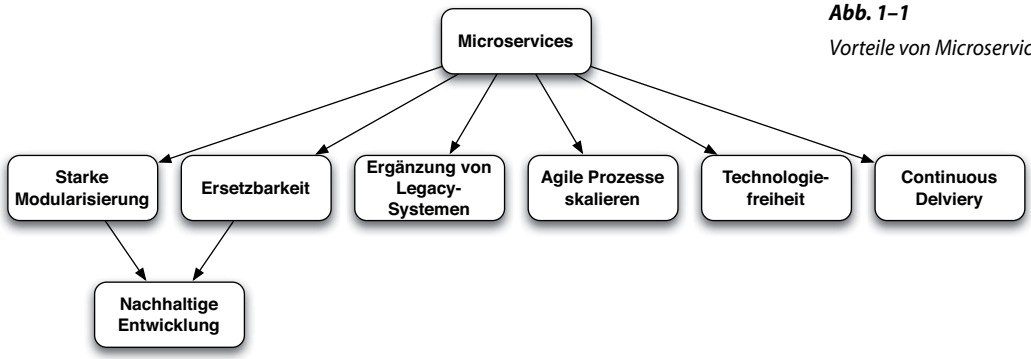

Abb. 1–1
Vorteile von Microservices

Microservices haben einige wesentliche Vorteile:

Starke Modularisierung

- Microservices sind ein *starkes Modularisierungskonzept*. Wird ein System aus Software-Komponenten wie Ruby GEMs, Java JARs, .NET Assemblies oder Node.js NPMs zusammengestellt, schleichen sich leicht ungewünschte Abhängigkeiten ein. Irgendwo referenziert jemand eine Klasse oder Funktion, wo sie eigentlich nicht genutzt werden soll. Und nur wenig später sind im System so viele Abhängigkeiten, dass eine Wartung oder Weiterentwicklung praktisch unmöglich ist. Microservices hingegen kommunizieren über explizite Schnittstellen, die mit Mechanismen wie Messages oder REST umgesetzt sind. Dadurch sind die technischen Hürden für die Nutzung eines Microservice höher. So schleichen sich ungewünschte Abhängigkeiten kaum ein. Es sollte zwar möglich sein, auch in Deployment-Monolithen eine gute Modularisierung zu erreichen. Die Praxis zeigt aber, dass die Architektur von Deployment-Monolithen meistens zunehmend schlechter wird.

Leichte Ersetzbarkeit

- Microservices können leichter ersetzt werden. Andere Komponenten nutzen einen Microservice über eine explizite Schnittstelle. Wenn ein Service dieselbe Schnittstelle anbietet, kann er den Microservice ersetzen. Der neue Microservice muss weder die Code-Basis noch die Technologien des alten Microservice übernehmen. An solchen Zwängen scheitert oft die Modernisierung von Legacy-Systemen. Kleine Microservices erleichtern die Ablösung weiter. Gerade die Ablösung wird bei der Entwicklung von Systemen oft vernachlässigt. Wer denkt schon gerne darüber nach, wie das gerade erst geschaffene wieder ersetzt werden kann? Die einfache Ersetzbarkeit von Microservices reduziert außerdem die Kosten von Fehlentscheidungen. Wenn die Entscheidung für eine Technologie oder einen Ansatz auf einen Microservice begrenzt ist, kann im Extremfall einfach der Microservice komplett ersetzt werden.

Nachhaltige Software-Entwicklung

- Die starke Modularisierung und die leichte Ersetzbarkeit erlauben eine nachhaltige Software-Entwicklung. Meistens ist die Arbeit an einem neuen Projekt recht einfach. Bei längerer Projektlaufzeit lässt die Produktivität nach. Ein Grund dafür ist die Erosion der Architektur. Das vermeiden Microservices durch die starke Modularisierung. Ein weiteres Problem sind die Bindung an alte Technologien und die Schwierigkeiten, alte Module aus dem System zu entfernen. Hier helfen Microservices durch die Technologiefreiheit und die Möglichkeit, Microservices einzeln zu ersetzen.

Legacy-Anwendung erweitern

- Der Einstieg in eine Microservices-Architektur ist einfach und bringt bei alten Systemen sogar sofort Vorteile: Statt die unübersichtliche alte Code-Basis zu ergänzen, kann das System mit einem Microservice ergänzt werden. Der kann bestimmte Anfragen bearbeiten und alle anderen dem Legacy-System überlassen. Er kann Anfragen vor der Bearbeitung durch das Legacy-System modifizieren. So muss nicht die gesamte Funktionalität des Legacy-Systems abgelöst werden. Der Microservice ist auch nicht an den Technologie-Stack des Legacy-Systems gebunden und kann mit modernen Ansätzen entwickelt werden.

Time-to-Market

- Microservices erlauben ein besseres Time-to-Market. Wie schon erwähnt, können Microservices einzeln in Produktion gebracht werden. Wenn in einem großen System jedes Team für einen oder mehrere Microservices zuständig ist und Features nur Änderungen an diesen Microservices benötigen, kann das Team ohne weitere Koordinierung mit anderen Teams entwickeln und Features in Produktion bringen. So können Teams ohne große Koordination an vielen Features parallel arbeiten, sodass mehr Features in derselben Zeit in Produktion gebracht werden können als bei einem Deploy-

ment-Monolithen. Microservices helfen dabei, agile Prozesse auf große Teams zu skalieren, indem das große Team in kleine Teams mit eigenen Microservices aufgeteilt wird.

Unabhängige Skalierung

- Jeder Microservice kann unabhängig von den anderen Services skaliert werden kann. Dadurch ist es nicht notwendig, das gesamte System zu skalieren, wenn nur wenige Funktionalitäten intensiv genutzt werden. Das kann oft eine entscheidende Vereinfachung sein.

Technologiefreiheit

- Bei der Umsetzung von Microservices herrscht Technologiefreiheit. Dadurch kann eine neue Technologie in einem Microservice erprobt werden, ohne dass andere Services betroffen sind. Das senkt das Risiko für die Einführung neuer Technologien und neuer Versionen vorhandener Technologien, da sie in einem kleinen Rahmen eingeführt und getestet werden können, in dem die Kosten kalkulierbar sind. Ebenso ist es möglich, spezielle Technologien für bestimmte Funktionalitäten zu nutzen – zum Beispiel eine spezielle Datenbank. Das Risiko ist gering, weil der Microservice jederzeit ersetzt oder entfernt werden kann. Die neue Technologie ist auf einen oder wenige Microservices beschränkt. Das reduziert das Risiko und ermöglicht vor allem unabhängige Technologie-Entscheidungen für unterschiedliche Microservices. Außerdem erleichtert es die Entscheidung für den Einsatz und die Evaluierung von neuen, hoch innovativen Technologien. Das kommt der Produktivität der Entwickler zugute und verhindert das Veralten der Technologie-Plattform. Aktuelle Technologien ziehen außerdem qualifiziertere Mitarbeiter an.

Continuous Delivery

- Für Continuous Delivery [1] sind Microservices vorteilhaft. Die Microservices sind klein und können unabhängig voneinander deployt werden. Die Umsetzung einer Continuous-Delivery-Pipeline ist wegen der Größe des Microservice einfach. Das Deployment eines einzelnen Microservice ist risikoärmer als das Deployment eines großen Monolithen. Es ist also einfacher, das Deployment eines Microservice abzusichern – beispielsweise durch den parallelen Betrieb verschiedener Versionen. Für viele Microservice-Nutzer ist Continuous Delivery der wesentliche Grund für die Einführung von Microservices.

Alle diese Gründe sprechen für die Einführung von Microservices. Welche Gründe am wichtigsten sind, hängt von dem Szenario ab. Die Skalierung agiler Prozesse und Continuous Delivery sind oft aus einer Geschäftssicht wichtig. Kapitel 5 widmet sich den Vorteilen von Microservices im Detail und geht auch auf die Priorisierung ein. Wo so viel Licht ist, ist auch Schatten. Daher wird Kapitel 6 noch detailliert darlegen, welche Herausforderungen bei der Umsetzung von Micro-

services existieren und wie man mit ihnen umgehen kann. Im Wesentlichen sind das die folgenden:

Beziehungen sind versteckt.

- Die Architektur des Systems besteht aus den Beziehungen der Services. Aber ohne Weiteres ist nicht klar, welcher Microservice welchen anderen aufruft. Dadurch wird Architekturarbeit zu einer Herausforderung.

Refactoring ist schwierig.

- Die starke Modularisierung hat auch Nachteile: Refactorings, bei denen Funktionalitäten zwischen Microservices verschoben werden, sind schwer umsetzbar. Die Aufteilung des Systems in Microservices ist nachträglich nur schwer zu ändern. Diese Probleme kann man durch geschicktes Vorgehen abmildern.

Fachliche Architektur ist wichtig.

- Die Aufteilung des Systems in fachliche Microservices ist wichtig, weil dadurch auch die Aufteilung in Teams festgelegt wird. Fehler bei der Aufteilung auf dieser Ebene beeinflussen auch die Organisation. Nur eine gute fachliche Aufteilung kann die unabhängige Entwicklung der Microservices gewährleisten. Da Änderungen an der Aufteilung schwierig sind, können Fehler gegebenenfalls nur schwer korrigiert werden.

Betrieb ist komplex.

- Ein System, das aus Microservices besteht, hat viele Bestandteile, die deployt, überwacht und betrieben werden müssen. Das erhöht die Komplexität im Betrieb und die Anforderungen an die Betriebsinfrastruktur. Microservices erzwingen eine Automatisierung der Betriebsprozesse, da sonst ein Betrieb der Plattform zu aufwendig ist.

Verteilte Systeme sind komplex.

- Die Komplexität für die Entwickler wächst: Ein Microservice-System ist ein verteiltes System. Aufrufe zwischen Microservices können wegen Netzwerkproblemen fehlschlagen. Aufrufe über das Netzwerk sind langsamer und haben eine geringere Bandbreite, als dies bei Aufrufen innerhalb eines Prozesses der Fall wäre.

[1] *http://jandiandme.blogspot.com/2006/10/jaoo-2006-werner-vogels-cto-amazon.html*

[2] Eberhard Wolff: Continuous Delivery: Der pragmatische Einstieg, dpunkt.verlag, 2014, ISBN 978-3864902086

Teil I

Motivation und Grundlagen

Dieser Teil des Buchs zeigt, was Microservices sind, warum Microservices so interessant sind und wo sie gewinnbringend genutzt werden können. So wird an praktischen Beispielen klar, was Microservices in welchen Szenarien bewirken.

Kapitel 2 erläutert die Struktur des Buchs. Um die Bedeutung von Microservices zu illustrieren, enthält Kapitel 3 konkrete Szenarien für die Nutzung von Microservices.

2 Einleitung

In diesem Kapitel steht das Buch selber im Mittelpunkt: Abschnitt 2.1 beschreibt kurz das Konzept des Buchs, Abschnitt 2.2 beschreibt die Zielgruppe und Abschnitt 2.3 gibt einen Überblick über die Kapitel und Struktur des Buchs. Abschnitt 2.4 erläutert die Bedeutung der Essays im Buch. Abschnitt 2.5 beschreibt Pfade durch das Buch für die verschiedenen Zielgruppen und Abschnitt 2.6 enthält schließlich die Danksagung.

Errata, Links zu den Beispielen und weitere Informationen finden sich unter *http://microservices-buch.de/*.

2.1 Überblick über das Buch

Das Buch gibt eine ausführliche Einleitung in Microservices. Dabei stehen die Architektur und Organisation im Mittelpunkt, ohne dass technische Umsetzungsmöglichkeiten vernachlässigt werden. Ein vollständig implementiertes Beispiel für ein Microservice-System zeigt eine konkrete technische Umsetzung. Technologien für Nanoservices zeigen, dass es sogar noch kleiner als Microservices geht. Das Buch vermittelt alles, um mit dem Umsetzen von Microservices loszulegen.

2.2 Für wen ist das Buch?

Das Buch wendet sich an Manager, Architekten und Techniker, die Microservices als Architekturansatz einführen wollen.

- Microservices setzen auf die wechselseitige Unterstützung von Architektur und Organisation. *Manager* lernen in der Einführung die grundlegenden Ideen von Microservices kennen und können dann vor allem auf die organisatorischen Auswirkungen fokussieren. *Manager*
- *Entwickler* erhalten eine umfassende Einleitung in die technischen Aspekte und können damit die notwendigen Fähigkeiten aufbauen, *Entwickler*

um Microservices umzusetzen. Ein konkretes Beispiel für eine technische Umsetzung von Microservices und zahlreiche weitere Technologien z. B. für Nanoservices helfen dabei mit dem Verständnis.

Architekten

- *Architekten* lernen Microservices aus einer Architekturperspektive kennen und können sich gleichzeitig in technische oder organisatorische Fragen vertiefen.

Im Buch gibt es Hinweise für eigene Experimente und Möglichkeiten zur Vertiefung. So kann der Interessierte das Gelesene praktisch ausprobieren und sein Wissen selbstständig erweitern.

2.3 Übersicht über die Kapitel

Teil I

Der erste Teil des Buchs zeigt die Motivation für Microservices und die Grundlagen der Microservices-Architektur. Das Kapitel 1 hat schon die grundlegenden Eigenschaften, Vor- und Nachteile von Microservices erläutert. Kapitel 3 zeigt zwei Szenarien für den Einsatz von Microservices: eine E-Commerce-Anwendung und ein System zur Verarbeitung von Signalen. Dieser Teil vermittelt einen ersten Einblick von Microservices und zeigt auch schon Anwendungskontexte.

Teil II

Teil II erläutert nicht nur Microservices genauer, sondern beschreibt auch die Vor- und Nachteile:

- Kapitel 4 beleuchtet die *Definition* des Begriffs »Microservices« aus drei Perspektiven: der Größe eines Microservice, dem Gesetz von Conway, nach dem Organisationen nur bestimmte Software-Architekturen hervorbringen können, und schließlich aus einer fachlichen Sicht anhand von Domain-Driven Design und BOUNDED CONTEXT.
- Die *Gründe* für Microservices zeigt Kapitel 5. Microservices haben nicht nur technische, sondern auch organisatorische Vorteile und auch aus Geschäftssicht gibt es gute Gründe für Microservices.
- Microservices haben aber auch ganz eigene Herausforderungen, die Kapitel 6 zeigt. Dazu gehören technische Herausforderungen, aber auch solche bei der Architektur, Infrastruktur und dem Betrieb.
- In Kapitel 7 steht eine Abgrenzung zwischen Microservices und SOA (Service-Oriented Architecture) im Vordergrund. Auf den ersten Blick scheinen diese beiden Konzepte eng verwandt. Bei genauerer Betrachtung gibt es aber erhebliche Unterschiede.

Teil III

Im Teil III geht es um die Umsetzung von Microservices. Der Teil zeigt, wie die Vorteile aus Teil II erreicht werden und wie die Herausforderungen gelöst werden können.

- Das Kapitel 8 beschreibt die Architektur von Microservice-Systemen. Neben der fachlichen Architektur geht es auch um übergreifende technische Herausforderungen.
- Kapitel 9 zeigt die verschiedenen Möglichkeiten zur Integration und Kommunikation zwischen Microservices. Dazu zählt nicht nur eine Kommunikation über REST oder Messaging, sondern auch eine Integration der UIs und die Replikation von Daten.
- Kapitel 10 zeigt Möglichkeiten zur Architektur eines Microservice. In diesem Bereich gibt es verschiedene Möglichkeiten, um die Microservices aufzubauen wie CQRS, Event Sourcing oder hexagonale Architektur. Schließlich geht es auch um geeignete Technologien für typische Herausforderungen.
- Das Testen steht im Mittelpunkt von Kapitel 11. Tests müssen weitgehend unabhängig sein, um das unabhängige Deployment der einzelnen Microservices zu ermöglichen. Dennoch müssen die Tests nicht nur die einzelnen Microservices, sondern auch das Gesamtsystem testen.
- Der Betrieb und Continuous Delivery stehen im Mittelpunkt von Kapitel 12. Microservices erzeugen viel mehr deploybare Artefakte und erhöhen damit die Ansprüche an die Infrastruktur. Das ist eine wesentliche Herausforderung bei der Einführung von Microservices.
- Im nächsten Schritt zeigt Kapitel 13, wie Microservices auch die Organisation beeinflussen. Schließlich sind Microservices eine Architektur, die auch die Organisation beeinflussen und verbessern soll.

Teil IV

Der letzte Teil des Buchs zeigt, wie Microservices ganz konkret technisch umgesetzt werden können. Dort geht es dann hinunter bis auf die Code-Ebene:

- Ein vollständiges Beispiel einer Microservices-Architektur zeigt Kapitel 14. Sie basiert auf Java, Spring Boot, Docker und Spring Cloud. Ziel ist, eine einfach zu nutzende Anwendung bereitzustellen, um die Konzepte aus dem Buch ganz praktisch zu verdeutlichen und eine Basis für eigene Implementierungen und Experimente zu bieten.
- Noch kleiner als Microservices sind die Nanoservices aus Kapitel 15. Sie erzwingen aber auch spezielle Technologien und einige Kompromisse. Das Kapitel zeigt verschiedene Technologien mit den jeweiligen Vor- und Nachteilen.
- Kapitel 16 zeigt zum Abschluss, wie Microservices konkret adaptiert werden können.

2.4 Essays

Das Buch enthält Essays, die Microservices-Experten geschrieben haben. Die Aufgabe war, auf ungefähr zwei Seiten wichtige Erkenntnisse zu Microservices festzuhalten. Manchmal ergänzen die Essays das Buch, manchmal beleuchten sie andere Themen und manchmal widersprechen sie auch dem Rest des Buchs. Es gibt eben bei Software-Architekturen oft keine klaren Antworten, sondern verschiedene Meinungen und Möglichkeiten. Die Essays bieten die Chance, verschiedene Standpunkte kennenzulernen, um sich dann eine eigene Meinung zu bilden.

2.5 Pfade durch das Buch

Das Buch bietet für jede Zielgruppe passende Inhalte (siehe Tab. 2–1). Natürlich kann und sollte jeder auch Kapitel lesen, die vielleicht nicht zur eigenen Rolle gehören. Aber der Fokus der Kapitel liegt auf der jeweiligen Rolle.

Tab. 2–1
Pfade durch das Buch

Kapitel	Entwickler	Architekten	Manager
3 – Microservice-Szenarien	X	X	X
4 – Was sind Microservices?	X	X	X
5 – Gründe für Microservices	X	X	X
6 – Herausforderungen bei Microservices	X	X	X
7 – Microservices und SOA		X	X
8 – Architektur von Microservice-Systemen		X	
9 – Integration und Kommunikation	X	X	
10 – Architektur eines Microservice	X	X	
11 – Testen von Microservices und Microservice-Systemen	X	X	
12 – Betrieb und Continuous Delivery von Microservices	X	X	
13 – Organisatorische Auswirkungen der Architektur			X
14 – Beispiel-Implementierung von Microservices	X		
15 – Technologien für Nanoservices	X	X	
16 – Wie mit Microservices loslegen?	X	X	X

Wer nur an dem groben Inhalt eines Kapitels interessiert ist, kann das Fazit des Kapitels lesen. Wer direkt ganz praktisch einsteigen will, sollte mit den Kapiteln 14 und 15 anfangen, bei denen konkrete Technologien und Code im Mittelpunkt stehen.

Die Anleitungen zu eigenen Experimenten in den Abschnitten »Selber ausprobieren und experimentieren« können die Basis zu einer selbstständigen Vertiefung des Gelernten sein. Wenn ein Kapitel besonders wichtig erscheint, kann man die Aufgaben dazu durcharbeiten, um die Themen des Kapitels genauer kennenzulernen.

2.6 Danksagung

Alle, mit denen ich das diskutiert habe, die Fragen gestellt oder mit mir zusammengearbeitet haben – viel zu viele, um sie alle zu nennen. Der Dialog hilft sehr und macht Spaß!

Besonders erwähnen möchte ich Jochen Binder, Matthias Bohlen, Merten Driemeyer, Martin Eigenbrodt, Oliver B. Fischer, Lars Gentsch, Oliver Gierke, Boris Gloger, Alexander Heusingfeld, Christine Koppelt, Andreas Krüger, Tammo van Lessen, Sascha Möllering, André Neubauer, Till Schulte-Coerne, Stefan Tilkov, Kai Tödter, Oliver Wolf und Stefan Zörner

Eine wichtige Rolle hat auch mein Arbeitgeber, die innoQ, gespielt. Viele Diskussionen und Anregungen meiner Kollegen finden sich in diesem Buch.

Schließlich habe ich meinen Freunden, Eltern und Verwandten zu danken, die ich für das Buch oft vernachlässigt habe – insbesondere meiner Frau.

Und natürlich gilt mein Dank all jenen, die an den in diesem Buch erwähnten Technologien gearbeitet und so die Grundlagen für Microservices gelegt haben.

Last but not least möchte ich dem dpunkt.verlag und René Schönfeldt danken, der mich sehr professionell bei der Erstellung des Buchs unterstützt hat.

2.7 Änderungen in der 2. Auflage

In der zweiten Auflage ist vor allem der Abschnitt 4.3 zu DDD komplett überarbeitet. DDD ist eine wichtige Basis für Microservices, sodass eine bessere Erläuterung sinnvoll ist. Das Beispiel in Kapitel 14 ist komplett überarbeitet, da sich die Technologien weiterentwickeln und mittlerweile neben dem Netflix-Stack auch Alternativen verfügbar sind, die nun in Kapitel 14, aber auch im Rest des Buchs Erwähnung finden.

2.8 Links & Literatur

[1] Eberhard Wolff: Continuous Delivery: Der pragmatische Einstieg, dpunkt.verlag, 2. Auflage, 2016, ISBN 978-3-86490-371-7

3 Microservice-Szenarien

Dieses Kapitel zeigt einige Szenarien, in denen die Nutzung von Microservices sinnvoll ist. Abschnitt 3.1 betrachtet die Modernisierung einer Legacy-Webanwendung. Dieses Szenario ist der häufigste Einsatzkontext von Microservices. Ein völlig anderes Szenario stellt Abschnitt 3.2 vor. Dort geht es um die Entwicklung eines Signalsystems, das als verteiltes System mit Microservice umgesetzt wird. Abschnitt 3.3 zieht ein Fazit aus den Szenarien und lädt zu einer eigenen Bewertung ein.

3.1 Eine E-Commerce-Legacy-Anwendung modernisieren

Szenario

Die Raffzahn Online Commerce GmbH betreibt einen E-Commerce-Shop, von dem der Umsatz des Unternehmens wesentlich abhängt. Es ist eine Webanwendung, die sehr viele unterschiedliche Funktionalitäten anbietet. Dazu zählen die Benutzerregistrierung und -verwaltung, Produktsuche, Überblick über die Bestellungen und der Bestellprozess – das zentrale Feature einer E-Commerce-Anwendung.

Diese Anwendung ist ein Deployment-Monolith: Sie kann nur als Ganzes deployt werden. Bei einer Änderung eines Features muss die gesamte Anwendung neu ausgeliefert werden. Der E-Commerce-Shop arbeitet mit anderen Systemen zusammen – beispielsweise der Buchhaltung und der Lagerhaltung.

Gründe für Microservices

Der Deployment-Monolith war zwar als wohlstrukturierte Anwendung gestartet, aber über die Jahre haben sich mehr und mehr Abhängigkeiten zwischen den Modulen eingeschlichen. Aus diesem Grund ist die Anwendung mittlerweile kaum noch wart- und änderbar. Außerdem ist die ursprüngliche Architektur schon lange nicht mehr ange-

messen für die aktuellen Anforderungen. So wurde beispielsweise die Produktsuche sehr stark verändert, weil die Raffzahn Online Commerce GmbH sich vor allem in diesem Bereich von den Konkurrenten abheben will. Ebenso sind mehr und mehr Möglichkeiten geschaffen worden, wie Kunden Probleme ohne Kundenservice selber lösen können. Dadurch konnte die Firma ihre Kosten erheblich reduzieren. Dementsprechend sind diese beiden Module mittlerweile sehr groß, intern sehr komplex aufgebaut und haben auch viele Abhängigkeiten zu anderen Modulen, die ursprünglich nicht eingeplant waren.

Langsame Continuous Delivery Pipeline

Raffzahn setzt auf Continuous Delivery und hat eine Continuous-Delivery-Pipeline etabliert. Die Pipeline ist kompliziert und hat eine lange Durchlaufzeit, weil der komplette Deployment-Monolith getestet und in Produktion gebracht werden muss. Einige der Tests dauern Stunden. Schnellere Durchlaufzeiten durch die Pipeline wären sicher wünschenswert.

Parallele Arbeit ist kompliziert.

Es gibt Teams, die an verschiedenen neuen Features arbeiten. Aber die parallele Arbeit ist kompliziert: Die Struktur der Software ist dafür zu schlecht. Die einzelnen Module sind zu schlecht separiert und haben zu viele Abhängigkeiten untereinander. Da alles nur gemeinsam deployt werden kann, muss der gesamte Deployment-Monolith vorher auch getestet werden. Das Deployment und die Testphase sind ein Flaschenhals. Wenn ein Team gerade ein Release in der Deployment-Pipeline hat, aber bei dem Release in diesen Phasen ein Problem auftaucht, müssen alle anderen Teams warten, bis die Änderung erfolgreich deployt worden ist. Und der Durchlauf durch die Deployment-Pipeline muss koordiniert werden. Nur ein Team kann zu einem Zeitpunkt im Test und Deployment sein. So wird geregelt, welches Team welche Änderung wann in Produktion bringen darf.

Flaschenhals bei den Tests

Neben dem Deployment müssen auch die Tests koordiniert werden. Wenn der Deployment-Monolith durch einen Integrationstest läuft, dürfen in dem Test nur die Änderungen eines Teams enthalten sein. Es gab Versuche, mehrere Änderungen auf einmal zu testen. Dann war bei einem Fehler nicht klar, woher das Problem kam, und es gab lange und komplexe Fehleranalysen.

Ein Integrationstest dauert ca. eine Stunde. Pro Arbeitstag sind realistisch sechs Integrationstests möglich, weil Fehler behoben und die Umgebungen wieder hergerichtet werden müssen. Bei zehn Teams kann ein Team im Schnitt ungefähr alle zwei Tage eine Änderung in Produktion bringen. Oft muss ein Team aber Fehleranalyse betreiben – dann verlängert sich die Integration. Daher nutzen einzelne Teams Feature Branches, um sich von der Integration zu entkoppeln: Sie nehmen ihre Änderungen an einem separierten Zweig in der Versionskon-

trolle vor. Bei der Integration dieser Änderungen in den Hauptzweig kommt es immer wieder zu Problemen: Änderungen werden aus Versehen beim Mergen wieder entfernt oder die Software hat plötzlich Fehler, die durch die getrennte Entwicklung aufgetreten sind. Die Fehler können erst nach einer Integration in langwierigen Prozessen ausgemerzt werden.

Also bremsen sich die Teams durch die Tests gegenseitig aus. Letztendlich arbeiten alle Teams zwar an den eigenen Modulen, aber an derselben Code-Basis, sodass sie sich ausbremsen. Durch die gemeinsame Continuous-Delivery-Pipeline und die dadurch notwendige Koordination sind letztendlich die Teams nicht dazu in der Lage, unabhängig und parallel zu arbeiten.

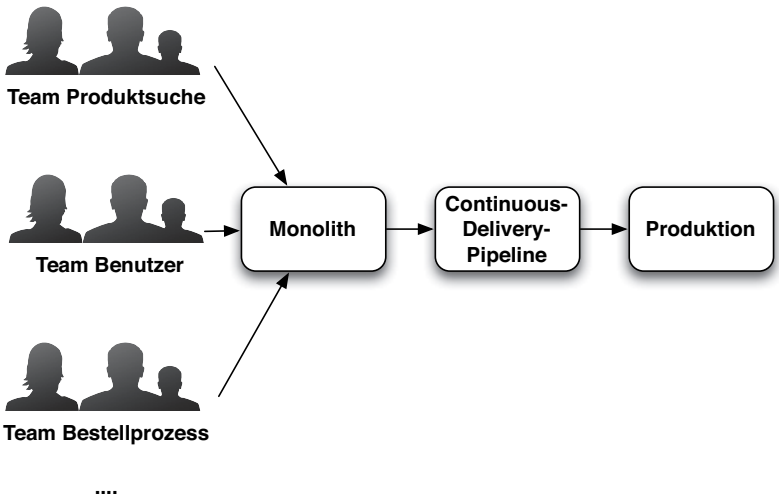

Abb. 3–1
Teams bremsen sich durch den Monolithen gegenseitig aus.

Vorgehen

Die Raffzahn Online Commerce GmbH hat sich wegen der vielen Probleme dazu entschieden, kleine Microservices von dem Deployment-Monolithen abzuspalten. Die Microservices implementieren jeweils ein Feature wie beispielsweise die Produktsuche und werden von einem Team verantwortet. Das Team ist von der Anforderungsaufnahme bis zum Betrieb der Anwendung vollständig verantwortlich. Diese Microservices kommunizieren mit dem Monolithen und anderen Microservices über REST. Auch die Benutzeroberfläche ist anhand der fachlichen Anwendungsfälle auf die einzelnen Microservices aufgeteilt. Jeder Microservice liefert die HTML-Seiten für seine Anwendungsfälle aus. Zwischen den HTML-Seiten der Microservices darf es Links geben. Aber es ist nicht erlaubt, auf die Datenbanktabellen der ande-

ren Microservices oder des Deployment-Monolithen zuzugreifen. Ein Datenaustausch zwischen den Services darf ausschließlich über REST oder durch die Verlinkung der HTML-Seiten erfolgen.

Die Microservices können unabhängig voneinander deployt werden. Dadurch ist es möglich, Änderungen in den Microservices ohne Koordinierung mit anderen Microservices oder Teams auszuliefern. Das vereinfacht die parallele Arbeit an Features erheblich und reduziert gleichzeitig den Koordinierungsaufwand.

Der Deployment-Monolith ist durch die Ergänzung um die Microservices wesentlich weniger Änderungen unterworfen. Für viele Features sind gar keine Änderungen am Monolithen mehr notwendig. Daher wird der Deployment-Monolith nun seltener deployt und geändert. Eigentlich war der Plan, den Deployment-Monolithen irgendwann vollständig abzulösen. Aber mittlerweile erscheint es wahrscheinlich, dass der Deployment-Monolith einfach zunehmend seltener deployt wird, weil die meisten Änderungen in den Microservices stattfinden. Dann stört der Deployment-Monolith aber nicht mehr. Eine vollständige Ablösung ist eigentlich überflüssig und erscheint wirtschaftlich nicht mehr sinnvoll.

Herausforderungen

Durch die Umsetzung der Microservices entsteht zunächst zusätzliche Komplexität: Die vielen Microservices benötigen eigene Infrastrukturen. Parallel muss der Monolith weiter unterstützt werden.

Die Microservices umfassen wesentlich mehr Server und stellen daher ganz andere Anforderungen. Das Monitoring und die Verarbeitung der Logdateien müssen damit umgehen, dass die Daten auf verschiedenen Servern anfallen. Also müssen die Informationen zentral konsolidiert werden. Außerdem muss eine wesentlich größere Anzahl Server angeboten werden – und zwar nicht nur in Produktion, sondern auch in den verschiedenen Test-Stages und auch Umgebungen für die einzelnen Teams sind sie notwendig. Das stellt wesentlich höhere Anforderungen an die Infrastruktur-Automatisierung. Es müssen nicht nur zwei unterschiedliche Arten von Infrastrukturen für den Monolithen und die Microservices unterstützt werden, sondern unter dem Strich auch wesentlich mehr Server.

Vollständige Migration ist langwierig.

Die zusätzliche Komplexität durch die beiden unterschiedlichen Software-Arten wird sehr lange vorhanden sein, denn die vollständige Migration weg vom Monolithen ist ein langwieriger Prozess. Wenn der Monolith niemals vollständig abgelöst wird, werden auch die zusätzlichen Infrastrukturkosten bestehen bleiben.

Testen bleibt eine Herausforderung.

Eine weitere Herausforderung ist das Testen: Bisher wurde der gesamte Deployment-Monolith in der Deployment-Pipeline getestet. Diese Tests sind aufwendig und dauern lange, weil sie alle Funktionalitäten im Deployment-Monolithen testen müssen. Wenn jede Änderung an jedem Microservice durch diese Tests geschickt wird, dauert es sehr lange, bis die Änderungen in Produktion sind. Außerdem müssen die Änderungen koordiniert werden, denn jede Änderung sollte einzeln getestet werden, damit gegebenenfalls klar ist, welche Änderung einen Fehler ausgelöst hat. Damit ist dann gegenüber dem Deployment-Monolithen nicht viel gewonnen: Das Deployment wäre zwar unabhängig voneinander möglich, aber die Test-Stages vor dem Deployment müssen immer noch koordiniert und von jeder Änderung einzeln durchlaufen werden.

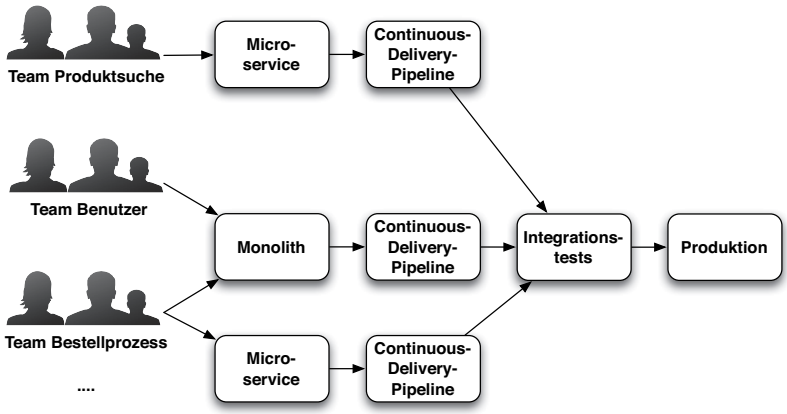

Abb. 3–2
Entkoppelte Arbeit durch Microservices

Aktueller Stand der Migration

Abbildung 3–2 zeigt den aktuellen Stand: Die Produktsuche arbeitet auf ihrem eigenen Microservice und vollständig unabhängig vom Deployment-Monolithen. Eine Koordinierung mit den anderen Teams ist kaum noch notwendig. Nur im letzten Schritt des Deployments müssen der Deployment-Monolith und die Microservices gemeinsam getestet werden. Diesen Schritt muss jede Änderung des Monolithen und aller Microservices durchlaufen. Dadurch ist ein Flaschenhals entstanden. Das Team »Benutzer« arbeitet gemeinsam unter anderem mit dem Team »Bestellprozess« am Deployment-Monolithen. Diese Teams müssen sich trotz Microservices immer noch eng abstimmen. Daher hat das Team »Bestellprozess« einen eigenen Microservice umgesetzt, der einen Teil des Bestellprozesses umfasst. Änderungen in diesem Teil des Systems sind im Vergleich zum Deployment-Monolithen nicht nur wegen der jüngeren Code-Basis schneller umgesetzt, sondern auch weil die Koordination mit den anderen Teams entfällt.

Aufstellung der Teams

Voraussetzung für die unabhängige Arbeit an Features ist die Aufstellung der Teams nach Fachlichkeiten wie Produktsuche, Benutzer oder Bestellprozess. Wenn stattdessen die Teams nach technischen Merkmalen wie UI, Middle Tier oder Datenbank aufgestellt sind, muss für jedes Feature jedes Team beteiligt werden. Schließlich wird ein Feature meistens Änderungen in UI, Middle Tier und Datenbank umfassen. Um Koordination zwischen den Teams zu minimieren, ist eine Aufstellung der Teams nach Fachlichkeiten auf jeden Fall sinnvoll. Microservices unterstützen die Unabhängigkeit durch eine technische Unabhängigkeit der einzelnen Services. Deswegen müssen Teams sich auch viel weniger bezüglich Basistechnologien und grundlegenden technischen Entwürfen koordinieren.

Die Tests müssen ebenfalls modularisiert werden. Jeder Test sollte einem einzigen Microservice zugeschlagen werden. Dann reicht es, wenn der Test bei Änderungen an diesem Microservice ausgeführt wird. Außerdem kann es sein, dass der Test dann als Unit-Test umgesetzt werden kann statt als Integrationstest. So wird die Testphase, in der alle Microservices und der Monolith gemeinsam getestet werden, zunehmend kürzer. Das verringert das Problem der Koordination für die letzte Testphase.

Die Migration hin zu einer Microservices-Lösung hat einige Performance-Probleme erzeugt und auch Probleme bei Netzwerkausfällen. Diese Schwierigkeiten konnten allerdings mit der Zeit gelöst werden.

Nutzen

Dank der neuen Architektur können Änderungen wesentlich schneller deployt werden. Eine Änderung kann von einem Team innerhalb von 30 Minuten in Produktion gebracht werden. Der Deployment-Monolith hingegen wird wegen der teilweise noch nicht automatisierten Tests nur wöchentlich deployt.

Neben der höheren Geschwindigkeit sind die Deployments der Microservices auch sonst wesentlich angenehmer: Es ist viel weniger Koordinierung notwendig. Fehler können schneller gefunden und behoben werden, weil die Entwickler noch sehr genau wissen, woran sie gearbeitet haben – schließlich ist das nur 30 Minuten her.

Letztendlich wurde das Ziel erreicht: Die Entwickler können viel mehr Änderungen an dem E-Commerce-Shop vornehmen. Das ist möglich, weil die Teams ihre Arbeit wesentlich weniger koordinieren müssen und weil die Deployments der Services unabhängig voneinander erfolgen können.

Die Möglichkeit, unterschiedliche Technologien zu nutzen, haben die Teams sparsam genutzt: Der bisher verwendete Technologie-Stack war ausreichend. Zusätzliche Komplexität durch den Einsatz von unterschiedlichen Technologien wollten die Teams vermeiden. Allerdings wurde für die Produktsuche die lange überfällige Suchmaschine eingeführt. Diese Änderung konnte von dem Team, das für die Produktsuche verantwortlich ist, alleine durchgeführt werden. Zuvor war die Einführung dieser neuen Technologie lange Zeit unterbunden worden, weil das Risiko als zu groß eingeschätzt wurde. Und einige Teams haben mittlerweile neue Versionen der Bibliotheken aus dem Technologie-Stack in Produktion, weil sie auf die Bug Fixes angewiesen waren. Dazu war keine Koordination über die Teams notwendig.

Bewertung

Das Ablösen eines Monolithen durch das Einführen von Microservices ist schon fast ein Klassiker für die Einführung von Microservices. Monolithen weiterzuentwickeln und mit neuen Features zu versehen, ist aufwendig. Die Komplexität und damit die Probleme des Monolithen nehmen über die Zeit zu. Eine vollständige Ablösung durch eine andere Software ist schwierig, weil dann die Software komplett ersetzt werden muss – und das ist risikoreich.

Schnelle und unabhängige Entwicklung neuer Features

Gerade bei Unternehmen wie der Raffzahn Online Commerce GmbH sind die schnelle Entwicklung neuer Features und die parallele Arbeit an mehreren Features überlebenswichtig. Nur so können Kunden gewonnen und davon abgehalten werden, zu anderen Anbietern zu wechseln. Das Versprechen, mehr Features schneller zu entwickeln, machen Microservices für viele Einsatzkontexte sehr attraktiv.

Einfluss auf die Organisation

Dieses Beispiel verdeutlicht auch den Einfluss von Microservices auf die Organisation. Die Teams arbeiten jeweils auf eigenen Microservices. Weil die Microservices unabhängig voneinander entwickelt und deployt werden können, ist die Arbeit der Teams voneinander entkoppelt. Dazu darf aber ein Microservice nicht von mehreren Teams parallel weiterentwickelt werden. Zu der Microservices-Architektur gehört eine Organisation der Teams entsprechend den Microservices: Jedes Team ist für einen oder mehrere Microservices zuständig, die eine isolierte Funktionalität umsetzen. Diese Beziehung zwischen Organisation und Architektur ist gerade bei Microservices sehr wichtig. Die Teams kümmern sich um alle Belange des Microservice von der Anforderungsaufnahme bis hin zur Betriebsüberwachung. Selbstverständlich können gerade für den Betrieb gemeinsame Infrastrukturdienste für Logging oder Monitoring genutzt werden.

Und schließlich: Wenn das Ziel ein einfaches und schnelles Deployment in Produktion ist, reicht die Umstellung der Architektur auf Microservices nicht aus. Die gesamte Continuous-Delivery-Pipeline muss auf Hindernisse untersucht und diese müssen ausgeräumt werden. Das zeigen im Beispiel die Tests: Ein gemeinsames Testen sollte auf das notwendige Minimum beschränkt sein. Jede Änderung muss einzeln einen Integrationstest mit den anderen Microservices durchlaufen, aber der darf nicht besonders lange dauern.

Amazon macht es schon lange

Das hier beschriebene Szenario hat einige Parallelen zu dem, was Amazon schon sehr lange tut – und aus demselben Grund: Um möglichst schnell und einfach neue Features auf der Website umsetzen zu können. 2006 hat Amazon nicht nur seine Cloud-Plattform vorgestellt, sondern auch darüber gesprochen, wie sie Software entwickeln. Wesentliche Merkmale:

- Die Anwendung ist in verschiedene Services aufgeteilt.
- Die Services liefern jeweils ein Teil der Webseite. Beispielsweise gibt es einen Service für die Suche, einen weiteren Service für die Empfehlungen usw. Die einzelnen Services werden dann zusammen in der UI dargestellt.
- Es gibt jeweils ein Team, das für einen solchen Service zuständig ist. Und zwar sowohl für die Entwicklung neuer Features wie auch für den Betrieb des Service. Das Motto lautet: »You build it – you run it!« (Etwa: Wenn du es schreibst, musst du es auch betreiben!)
- Gemeinsame Basis dieser Services ist die Cloud-Plattform – letztendlich virtuelle Maschinen. Sonst gibt es keine Software-Vorgaben. Die Teams haben also weitgehende Technologiefreiheit.

Damit hatte Amazon wesentliche Merkmale von Microservices bereits 2006 realisiert. Und Amazon hatte DevOps durch Teams mit Betriebsexperten und Entwicklern umgesetzt. Dieser Ansatz erzwingt eine weitgehende Automatisierung des Deployments, denn der manuelle Aufbau von Servern ist in Cloud-Umgebungen nicht sinnvoll umsetzbar – und damit ist zumindest ein Aspekt von Continuous Delivery verwirklicht.

Fazit: Microservices wird bei einigen Firmen bereits seit Langem praktiziert – vor allem bei Firmen mit Internet-basiertem Geschäftsmodell. So hat der Ansatz schon lange seine praktischen Vorteile bewiesen. Außerdem haben Microservices Synergieeffekte mit anderen modernen Ansätzen wie Continuous Delivery, Cloud oder DevOps.

3.2 Ein neues Signalsystem entwickeln

Szenario

Die Suche nach vermissten Flugzeugen oder Schiffen ist ein komplexes Unterfangen. Schnelles Reagieren kann Leben retten. Dazu sind verschiedene Systeme notwendig. Einige liefern Signale – seien es beispielsweise Funksignale oder Radarsignale. Die Signale müssen aufgezeichnet und verarbeitet werden. So kann aus den Funksignalen beispielsweise eine Peilung entstehen, die dann mit den Radarbildern abgeglichen werden muss. Und schließlich gibt es weitere Verarbeitung durch Menschen. Sowohl die Auswertungen als auch die Rohdaten müssen den verschiedenen Rettungskräften zur Verfügung gestellt werden. Die Signal GmbH baut Systeme genau für diesen Einsatzkontext. Die Systeme werden individuell zusammengestellt, konfiguriert und für den jeweiligen Kundenkontext angepasst.

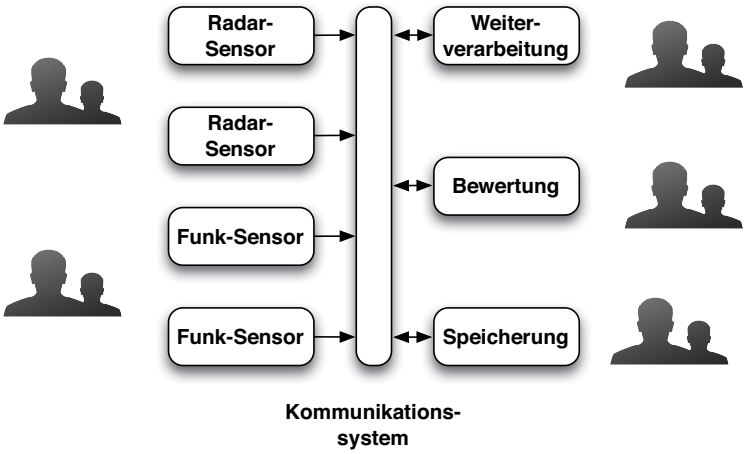

Abb. 3–3
Überblick über das Signalsystem

Gründe für Microservices

Das System besteht aus verschiedenen Bestandteilen, die jeweils auf unterschiedlichen Rechnern laufen. Die Sensoren sind im zu überwachenden Gebiet verteilt und mit eigener Intelligenz ausgestattet. Diese Rechner sollen jedoch die weitere Verarbeitung auf keinen Fall übernehmen und die Daten auch nicht speichern. Dazu ist die Hardware nicht leistungsfähig genug und ein solches Vorgehen ist auch z. B. wegen der Datensicherheit nicht wünschenswert.

Verteiltes System

Aus diesen Gründen ist das System ein verteiltes System. Die verschiedenen Funktionalitäten sind im Netzwerk verteilt. Das System ist

unzuverlässig, da einzelne Bestandteile und die Kommunikation zwischen den Bestandteilen ausfallen können.

Es wäre denkbar, einen großen Teil des Systems in einem Deployment-Monolithen unterzubringen. Aber bei näherer Betrachtung müssen die einzelnen Teile sehr unterschiedlichen Anforderungen genügen. Die Verarbeitung der Daten benötigt eher viel CPU und einen Ansatz, bei dem viele Algorithmen die Daten bearbeiten können. Dazu gibt es Lösungen, die aus einem Daten- oder Event-Strom Ereignisse auslesen und bearbeiten. Die Speicherung verlangt einen ganz anderen Fokus: Im Wesentlichen müssen die Daten in einer für unterschiedliche Auswertungen geeigneten Datenstruktur vorgehalten werden. Hierfür eignen sich moderne NoSQL-Datenbanken. Aktuelle Daten sind wichtiger als alte – sie müssen schneller zugreifbar sein, während alte Daten irgendwann sogar gelöscht werden. Für Analysen durch Menschen müssen die Daten ausgelesen und aufbereitet werden.

Technologie-Stack pro Team

Jede dieser Aufgaben stellt ganz andere Anforderungen. Daher benötigt jede von ihnen neben einem eigenen Technologie-Stack auch ein eigenes Team. Das Team besteht aus den technischen Experten für die jeweilige Aufgabe. Dazu kommen Personen, die entscheiden, welche Features die Signal GmbH am Markt platziert, und daraus neue Anforderungen ableiten. Systeme für die Verarbeitung und Sensoren sind jeweils eigene Produkte, die eigenständig am Markt positioniert werden.

Integration anderer Systeme

Ein weiterer Grund für die Nutzung von Microservices ist die Integration anderer Systeme. Sensoren und Logik gibt es auch von anderen Herstellern. Die Integration solcher Lösungen ist in Kundenprojekten immer wieder eine Anforderung. Mit Microservices können andere Systeme leichter integriert werden, weil die Integration unterschiedlicher verteilter Bestandteile der Normallfall ist.

Aus diesen Gründen haben die Architekten der Signal GmbH entschieden, das System tatsächlich als verteiltes System umzusetzen. Dabei soll ein Team seine jeweilige Fachlichkeit in mehreren kleineren Microservices umsetzen. Dadurch soll die Austauschbarkeit der Microservices weiter verbessert und auch die Integration anderer Systeme soll vereinfacht werden.

Fest steht nur eine gemeinsam genutzte Kommunikationsinfrastruktur, mit der die Microservices untereinander kommunizieren können. Die Kommunikationstechnologie steht in vielen verschiedenen Programmiersprachen und Plattformen zur Verfügung, sodass es keine Einschränkungen bezüglich der konkreten Technologie gibt. Zur reibungslosen Kommunikation müssen die Schnittstellen der Microservices untereinander klar definiert werden.

Herausforderungen

Der Ausfall der Kommunikation zwischen den Microservices ist eine wesentliche Herausforderung. Das System muss benutzbar bleiben, auch wenn es zu Netzwerkausfällen kommt. Dazu müssen Technologien genutzt werden, die mit solchen Ausfällen zurechtkommen. Das Problem ist aber durch Technologien alleine nicht in den Griff zu bekommen. Es muss fachlich entschieden werden, was beim Ausfall eines Systems geschehen soll. Wenn beispielsweise alte Daten ausreichend sind, können Caches helfen. Oder es kann möglich sein, einen einfacheren Algorithmus zu nutzen, der ohne die Abfrage der anderen Systeme auskommt.

Hohe technologische Komplexität

Die technologische Komplexität der Gesamtlösung ist sehr hoch. Es werden unterschiedlichste Technologien eingesetzt, um den Anforderungen der verschiedenen Bestandteile gerecht zu werden. Dabei können die Teams, die an den einzelnen Systemen arbeiten, weitgehend unabhängige Technologie-Entscheidungen treffen. So können sie die jeweils passende Lösung umsetzen.

Leider bedeutet das aber auch, dass Mitarbeiter nicht mehr so einfach zwischen Teams wechseln können. Als beispielsweise gerade bei der Speicherung der Daten viel zu tun war, konnten die Mitarbeiter aus den anderen Teams kaum helfen, weil sie noch nicht einmal die Programmiersprache beherrschten, die dieses Team benutzt – ganz zu schweigen von den spezifischen Technologien wie beispielsweise der verwendeten Datenbank.

Ein System mit einer solchen Vielzahl an Technologien zu betreiben, ist eine Herausforderung. Aus diesem Grund gibt es in diesem Bereich eine Standardisierung: Alle Microservices müssen weitgehend identisch zu betreiben sein. Es sind virtuelle Maschinen, sodass die Installation recht einfach ist. Dazu kommt ein standardisiertes Monitoring, das Datenformate und Technologien festlegt. So können die Anwendungen einfach zentral überwacht werden. Neben dem typischen betrieblichen Monitoring kommen dazu noch die Überwachung fachlicher Werte und schließlich auch eine Auswertung der Log-Dateien.

Nutzen

Der wesentliche Nutzen von Microservices in diesem Zusammenhang ist die gute Unterstützung für die verteilte Natur des Systems. Die Sensoren sind an verschiedenen Standorten, sodass ein zentrales System sowieso kaum sinnvoll ist. Diesen Umstand hat sich die Architektur dann zunutze gemacht, indem das System noch weiter in kleine Microservices aufgeteilt worden ist, die im Netzwerk verstreut sind. Dadurch

wurde die Austauschbarkeit der Microservices weiter erhöht. Außerdem unterstützt der Microservices-Ansatz die Technologievielfalt, die dieses System auszeichnet.

Time-to-Market wie bei dem anderen Beispiel ist in diesem Szenario bei Weitem nicht so wichtig. Es wäre auch gar nicht so gut umsetzbar, weil die Systeme bei verschiedenen Kunden installiert sind, und daher können sie gar nicht ohne Weiteres neu installiert werden. Allerdings werden einige Ideen aus dem Continuous-Delivery-Bereich genutzt. Konkret: die weitgehend einheitliche Installation und das zentrale Monitoring.

Bewertung

Microservices passen als Architekturentwurf sehr gut zu dem Szenario. Das System kann davon profitieren, dass bei der Umsetzung typische Probleme durch die bekannten Vorgehensweisen aus dem Microservices-Bereich gelöst werden können – beispielsweise die Technologiekomplexität und der Betrieb der Plattform.

Dennoch ist das Szenario keines, das sofort mit dem Begriff »Microservice« bezeichnet werden würde. Daraus lassen sich verschiedene Dinge ableiten:

- Microservices sind breiter einsetzbar, als es auf den ersten Blick scheint. Auch außerhalb der webbasierten Geschäftsmodelle können Microservices viele Probleme lösen – wenn es auch ganz andere als bei Webunternehmen sind.
- Tatsächlich nutzen viele der Projekte in verschiedenen Bereichen schon länger Microservice-Ansätze – wenn sie das vielleicht auch selber nicht so nennen und auch nicht vollständig umsetzen.
- Durch Microservices können diese Projekte Technologien nutzen, die im Microservice-Umfeld gerade realisiert werden. Und sie können von den Erfahrungen zum Beispiel in Bezug auf Architektur aus diesem Umfeld lernen.

3.3 Fazit

Dieses Kapitel hat zwei unterschiedliche Szenarien in völlig verschiedenen Bereichen gezeigt: ein Websystem, bei dem schnelles Time-to-Market wichtig ist, und ein System zur Signalverarbeitung, das von der Natur her schon verteilt ist. Die Architekturprinzipien sind ähnlich – wenn auch aus unterschiedlichen Gründen.

Ebenso gibt es einige gemeinsame Herangehensweisen. Dazu zählen die Aufteilung der Teams nach Microservices, die Anforderungen an die Automatisierung der Infrastruktur sowie andere organisatorische Themen. In anderen Bereichen ergeben sich aber Unterschiede. Für das Signalsystem ist die Möglichkeit, unterschiedliche Technologien zu nutzen, essenziell, weil sie sowieso sehr viele unterschiedliche Technologien nutzen müssen. Für das Websystem ist es nicht so wichtig. In dem Szenario spielen die unabhängige Entwicklung, das schnelle und einfache Deployment und letztendlich die bessere Time-to-Market die entscheidende Rolle.

Wesentliche Punkte

- Microservices bieten sehr viele Vorteile.
- Eine wesentliche Motivation bei webbasierten Anwendungen können Continuous Delivery und schnelles Time-to-Market sein.
- Aber es gibt ganz andere Anwendungsfälle, bei denen sich Microservices beispielsweise als verteilte Systeme schon fast aufzwingen.

Teil II

Microservices: Was, warum und warum vielleicht nicht?

Dieser Teil des Buchs erläutert die verschiedenen Facetten von Microservices-Architekturen, um die Möglichkeiten von Microservices darzustellen. Vor- und Nachteile werden verdeutlicht, sodass man abschätzen kann, welchen Gewinn Microservices bringen und an welchen Stellen bei der Umsetzung von Microservices-Architekturen Vorsicht wichtig ist.

Kapitel 4 klärt den Begriff »Microservice« genauer. Der Begriff wird aus verschiedenen Perspektiven beleuchtet, was für ein Verständnis des Microservices-Ansatzes essenziell ist. Wichtige Aspekte sind die Größe des Microservice, das Gesetz von Conway als organisatorischer Einfluss und Domain-Driven Design bzw. BOUNDED CONTEXT aus einer fachlichen Sicht. Dazu kommt die Frage, ob ein Microservice eine UI enthalten soll. Die Vorteile des Ansatzes stehen in Kapitel 5 im Mittelpunkt – und zwar aus technischer, organisatorischer und geschäftlicher Sicht. Die Herausforderungen in Kapitel 6 liegen in den Bereichen Technik, Architektur, Infrastruktur und Betrieb. Kapitel 7 grenzt Microservices gegen SOA (Service-Oriented Architecture) ab. Die Abgrenzung beleuchtet Microservices noch aus einem weiteren Blickwinkel, um den Microservices-Ansatz noch klarer darzustellen. Außerdem werden Microservices immer wieder mit SOA-Architekturen verglichen.

Der dritte Teil des Buchs zeigt dann, wie Microservices praktisch umgesetzt werden können.