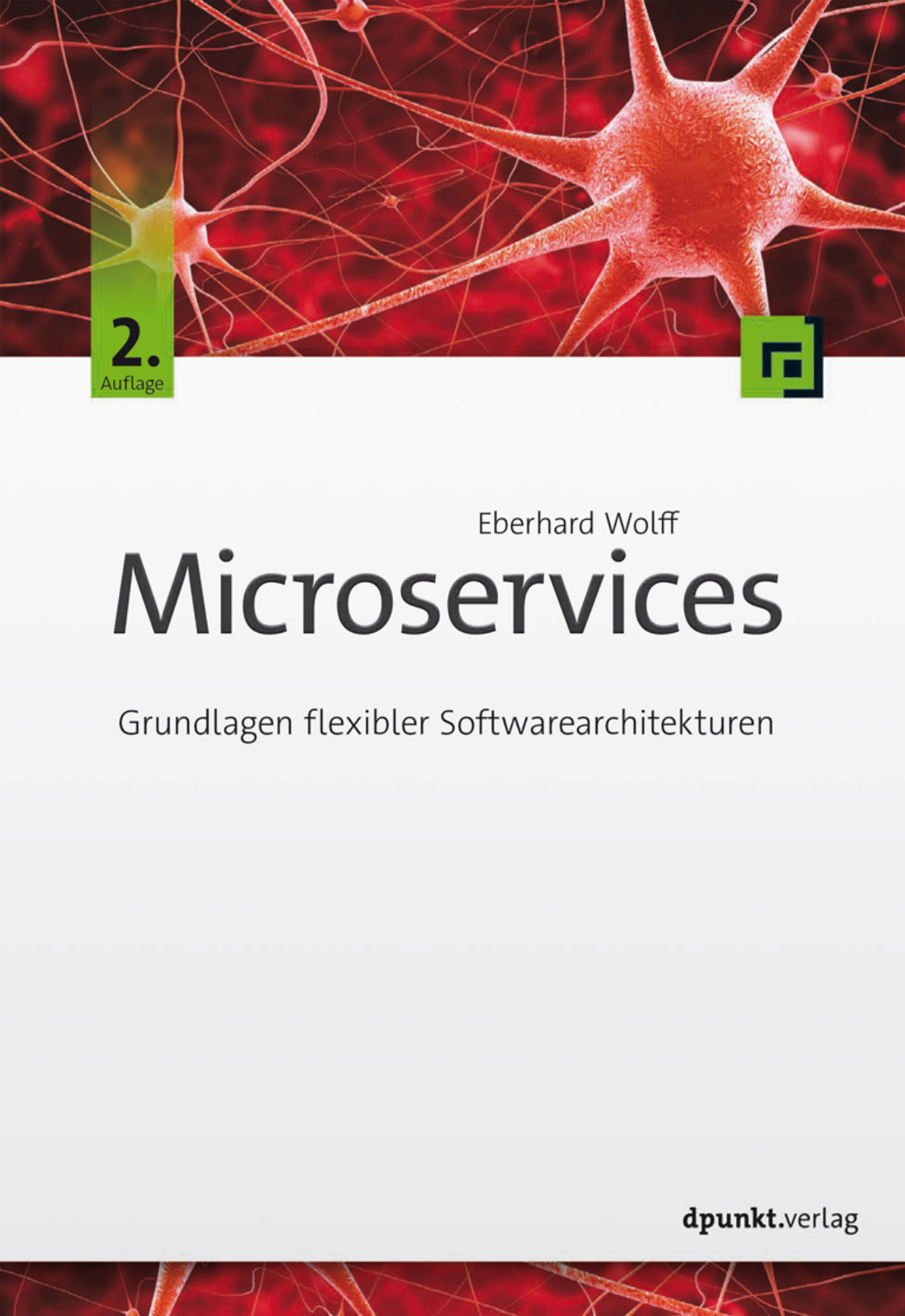


The background of the book cover is a detailed illustration of a neuron, rendered in a vibrant red color. The neuron has a central cell body with several long, thin, branching processes extending outwards, creating a complex, web-like pattern. The overall aesthetic is scientific and modern.

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 Architektursätzen – 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
Copyright © 2018 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

Inhaltsverzeichnis

1	Vorwort	1
1.1	Überblick über Microservices	2
1.2	Warum Microservices?	3

Teil I	Motivation und Grundlagen	7
---------------	----------------------------------	----------

2	Einleitung	9
2.1	Überblick über das Buch	9
2.2	Für wen ist das Buch?	9
2.3	Übersicht über die Kapitel	10
2.4	Essays	12
2.5	Pfade durch das Buch	12
2.6	Danksagung	13
2.7	Änderungen in der 2. Auflage	14
2.8	Links & Literatur	14
3	Microservice-Szenarien	15
3.1	Eine E-Commerce-Legacy-Anwendung modernisieren	15
3.2	Ein neues Signalsystem entwickeln	23
3.3	Fazit	26

Teil II	Microservices: Was, warum und warum vielleicht nicht?	29
----------------	--	-----------

4	Was sind Microservices?	31
4.1	Größe eines Microservice	31
4.2	Das Gesetz von Conway	39

4.3	Domain-Driven Design und Bounded Context	44
4.4	Self-contained Systems	54
4.5	Fazit	55
4.6	Links & Literatur	57
5	Gründe für Microservices	59
5.1	Technische Vorteile	59
5.2	Organisatorische Vorteile	67
5.3	Vorteile aus Geschäftssicht	69
5.4	Fazit	71
5.5	Links & Literatur	72
6	Herausforderungen bei Microservices	73
6.1	Technische Herausforderungen	73
6.2	Architektur	77
6.3	Infrastruktur und Betrieb	80
6.4	Fazit	81
6.5	Links & Literatur	82
7	Microservices und SOA	83
7.1	Was ist SOA?	83
7.2	Unterschiede zwischen SOA und Microservices	89
7.3	Fazit	94
7.4	Links & Literatur	96
Teil III	Microservices umsetzen	97
8	Architektur von Microservice-Systemen	101
8.1	Fachliche Architektur	101
8.2	Architekturmanagement	106
8.3	Techniken zum Anpassen der Architektur	111
8.4	Microservice-Systeme weiterentwickeln	120
8.5	Microservice und Legacy-Anwendung	127
8.6	Event-driven Architecture	137
8.7	Technische Architektur	138

8.8	Konfiguration und Koordination	141
8.9	Service Discovery	143
8.10	Load Balancing	146
8.11	Skalierbarkeit	150
8.12	Sicherheit	153
8.13	Dokumentation und Metadaten	161
8.14	Fazit	163
8.15	Links & Literatur	165
9	Integration und Kommunikation	167
9.1	Web und UI	167
9.2	REST	179
9.3	SOAP und RPC	182
9.4	Messaging	183
9.5	Datenreplikation	187
9.6	Schnittstellen: intern und extern	190
9.7	Fazit	193
9.8	Links & Literatur	196
10	Architektur eines Microservice	197
10.1	Fachliche Architektur	197
10.2	CQRS	199
10.3	Event Sourcing	201
10.4	Hexagonale Architekturen	203
10.5	Resilience und Stabilität	207
10.6	Technische Architektur	212
10.7	Fazit	215
10.8	Links & Literatur	216
11	Testen von Microservices und Microservice-Systemen	217
11.1	Warum testen?	217
11.2	Wie testen?	219
11.3	Risiken beim Deployment minimieren	224
11.4	Tests des Gesamtsystems	225
11.5	Legacy-Anwendungen mit Microservices testen	228
11.6	Tests einzelner Microservices	231

11.7	Consumer-Driven Contract Test	233
11.8	Technische Standards testen	237
11.9	Fazit	239
11.10	Links & Literatur	240
12	Betrieb und Continuous Delivery von Microservices	241
12.1	Herausforderungen beim Betrieb von Microservices	241
12.2	Logging	244
12.3	Monitoring	250
12.4	Deployment	256
12.5	Steuerung	262
12.6	Infrastrukturen	263
12.7	Fazit	269
12.8	Links & Literatur	271
13	Organisatorische Auswirkungen der Architektur	273
13.1	Organisatorische Vorteile von Microservices	273
13.2	Alternativer Umgang mit dem Gesetz von Conway	277
13.3	Spielräume schaffen: Mikro- und Makro-Architektur	281
13.4	Technische Führung	287
13.5	DevOps	288
13.6	Schnittstelle zu den Fachbereichen	293
13.7	Wiederverwendbarer Code	295
13.8	Microservices ohne Organisationsänderung?	299
13.9	Fazit	301
13.10	Links & Literatur	303

Teil IV	Technologien	305
----------------	---------------------	------------

14	Beispiel-Implementierung von Microservices	307
14.1	Fachliche Architektur	308
14.2	Basistechnologien	310
14.3	Build	314
14.4	Deployment mit Docker	316
14.5	Docker Machine	317
14.6	Docker Compose	319

14.7	Service Discovery	322
14.8	Routing	325
14.9	Resilience	327
14.10	Load Balancing	331
14.11	Integration anderer Technologien	332
14.12	Tests	332
14.13	Weitere Beispiele	338
14.14	Fazit	339
14.15	Links & Literatur	341
15	Technologien für Nanoservices	343
15.1	Warum Nanoservices?	344
15.2	Definition Nanoservice	346
15.3	Amazon Lambda	347
15.4	OSGi	350
15.5	Java EE	353
15.6	Vert.x	356
15.7	Erlang	358
15.8	Seneca	362
15.9	Fazit	364
15.10	Links & Literatur	366
16	Wie mit Microservices loslegen?	367
16.1	Warum Microservices?	367
16.2	Wege zu Microservices	368
16.3	Microservice: Hype oder Realität?	369
16.4	Fazit	370
	Index	371

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.

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.

Monolithen

1.2 Warum Microservices?

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

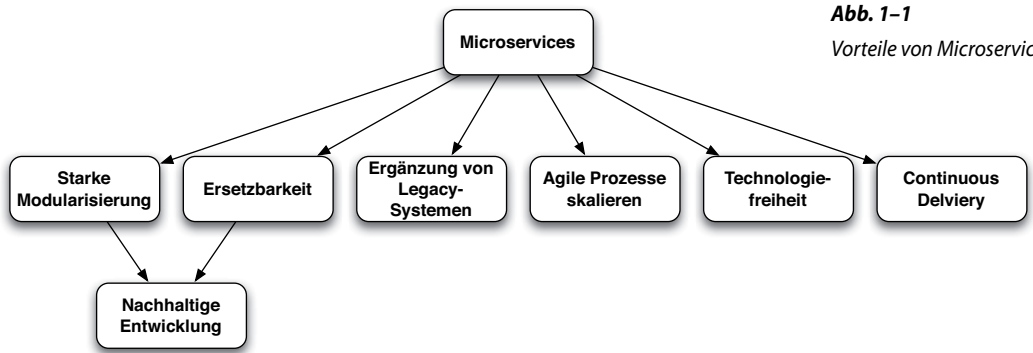


Abb. 1-1

Vorteile von Microservices

Microservices haben einige wesentliche Vorteile:

- 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.

Starke Modularisierung

- 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.

- 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.

Unabhängige Skalierung

- 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.

Technologiefreiheit

- 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.

Continuous Delivery

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:

- | | |
|---|--|
| <i>Beziehungen sind versteckt.</i> | ■ 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. |
| <i>Refactoring ist schwierig.</i> | ■ 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. |
| <i>Fachliche Architektur ist wichtig.</i> | ■ 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. |
| <i>Betrieb ist komplex.</i> | ■ 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. |
| <i>Verteilte Systeme sind komplex.</i> | ■ 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

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.

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

Teil IV

- 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.

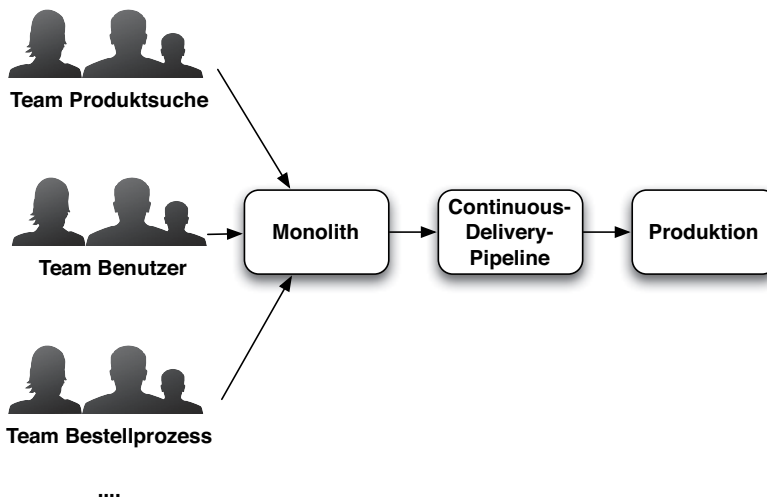


Abb. 3–1

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.

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.

Testen bleibt eine Herausforderung.

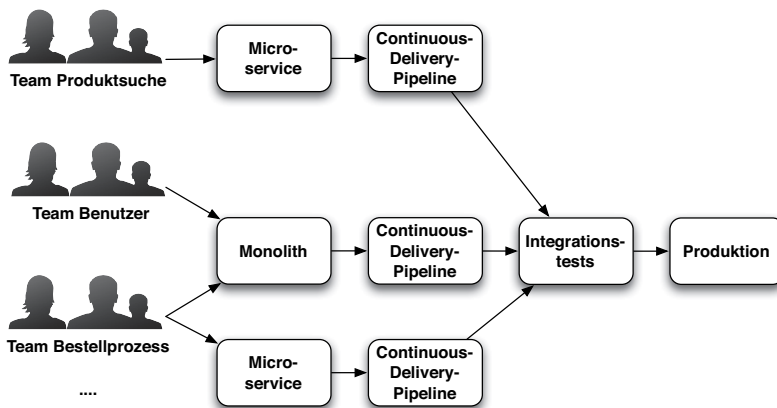


Abb. 3-2

Entkoppelte Arbeit durch Microservices

Abbildung 3-2 zeigt den aktuellen Stand: Die Produktsuche arbeitet auf ihrem eigenen Microservice und vollständig unabhängig vom Deployment-Monolithen. Eine Koordination 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.

Aktueller Stand der Migration

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 Netzausfällen. 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.